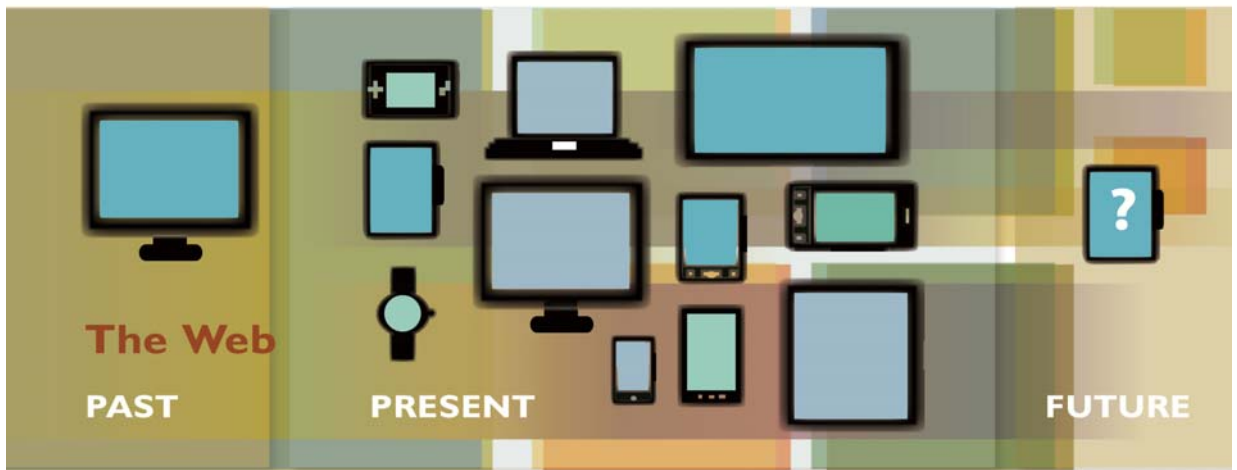


# THE RESPONSIVE WEB



■ ■ ■ ■ ■ ■ ■ Matthew Carver

 MANNING



**MEAP Edition  
Manning Early Access Program  
The Responsive Web version 12**

Copyright 2013 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# *brief contents*

---

- 1. Learning to work Responsively*
- 2. Design for mobile first*
- 3. Using style tiles to communicate design*
- 4. Responsive user experience design patterns*
- 5. Responsive layouts*
- 6. Adding content modules and typography*
- 7. Adding graphics in the browser with CSS*
- 8. Progressive enhancement with Modernizr*
- 9. Testing and optimization for responsive websites*

## 1

## *Learning to work Responsively*

Remember when people called the internet the “information superhighway?” It sounds cheesy now, but imagine that “superhighway.” Right now it’s full of people in sports cars, 18-wheelers, bicycles, family sedans, racecars, and pickup trucks. Some travel at hundreds of miles an hour, others go at a snail’s pace. Some legs of the highway have bike lanes, sedan lanes, and fast lanes. Every once and a while a traveler gets confused, and a sports car ends up in the bike lane, and a bike ends up in the sedan lane.

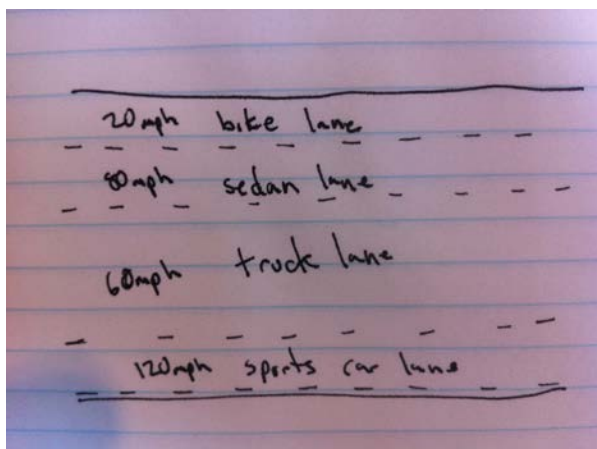


Figure 1.1 A cluttered “information superhighway.”

As the Department of Transportation created traffic standards, so the responsive web strives to standardize development patterns through a flexible and adaptive experience. For the

last decade or so, web design and development has remained in a fairly tight window. Websites have been a medium for desktop computers and laptops. Bandwidth and screen resolutions have stabilized, and most users engage websites with a traditional mouse and keyboard. The set expectations gave us an anticipated standard, and we played within the confines of our sandbox.

With the rise of handheld and tablet devices, web design and development is undergoing a phase of rapid and painful growth. The mobile website is nothing new. Mobile optimized websites have been around for over a decade. The problem lies in the architecture of these mobile sites.

Ethan Marcotte (co-author of *Handcrafted CSS* and *Designing with Web Standards, Third Edition*) wrote an article for A List Apart called “Responsive Web Design” which proposed a new technique for designing web pages to accommodate the needs of users with screens of all sizes, from mobile to desktop. The article became one of the most groundbreaking books in the history of web design. Ethan’s book struck a resonating cord among designers and developers worldwide, and the principles sparked a revolution.

This chapter serves as a quick introduction to the responsive web. It introduces the core concepts and gives you a base on which to get started. Once we get through the basic concepts, you’ll build your first responsive site!

---

### Designer / developer insights

In this book, the lines between what is considered “design” and what is considered “development” are blurred. Occasionally, these roles are occupied by one person, and other times multiple people take on these responsibilities. Either way, the responsive web requires harmony between the two skillsets.

The goal of this book is to teach designers and developers to not only learn the practices and executions that will produce successful responsive web sites, but to teach how to communicate and collaborate more efficiently. Responsive design is truly successful by working on how we work as well as what we produce. Look for these callouts to give insight on the specific roles.

There will be some sections in this book that will dive deeply into teaching design principles, and other parts will talk specifically about skillsets important to developers. Knowledge of both the design and development skills involved in responsive design are important to a balanced education on the topic, but at times it’ll be important to draw specific connections. Be sure to look for callouts like this one to draw out those connections and provide discipline-specific insights.

---

## 1.1 Meet the Responsive Web

I know you may be eager to start building your first responsive site; however, before we get there I want to make sure you have the basics under your belt. In this section I'll let you in on what the responsive web really is and its key features. Once through with the quick introduction you'll be ready to start building.

### 1.1.1 What is the Responsive Web?

- A responsive site is one that uses a single URL for mobile, tablet, and desktop sites. With about 10% of traffic (and growing) coming from mobile devices, and an increasing number of tablets and smartphones on the market, this is a crucial segment of all web traffic.
- A responsive site strives for consistency across devices. *By using a single URL, you ensure all inbound links to your site serve consistent content.*
- A responsive site delivers faster and heightens user experience. *By developing mobile sites first, an emphasis is placed on efficiency.*
- A responsive side is future friendly. *Every site will eventually need to be optimized for new technology, but by building responsively, you ensure that optimization won't entail a full site redesign.*

If you've been involved in designing or developing a website, you probably have a standard workflow. You have tools you use in certain ways to construct your work. In many cases, building a responsive site requires adjustments to these tools, and the use of entirely new tools all together.

Traditional web development takes a waterfall approach. The project follows a sequence, typically along the lines of Figure 1.2.



Figure 1.2 Commonly called the “waterfall” methodology, each phase in this process involves creating and passing a deliverable to the next person in the workflow.

The waterfall approach becomes inefficient and costly if the team needs to consider more variations of a project. Also, what if there are inconsistencies or performance issues when you get to development? Suddenly the entire project has to be changed, even potentially rebuilt.

The responsive web is about adaptation. With responsive web design, teams work closely together to build a site. Instead of passing off deliverables along the “website assembly line,” teams iterate and improve upon each other’s work (figure 1.3).

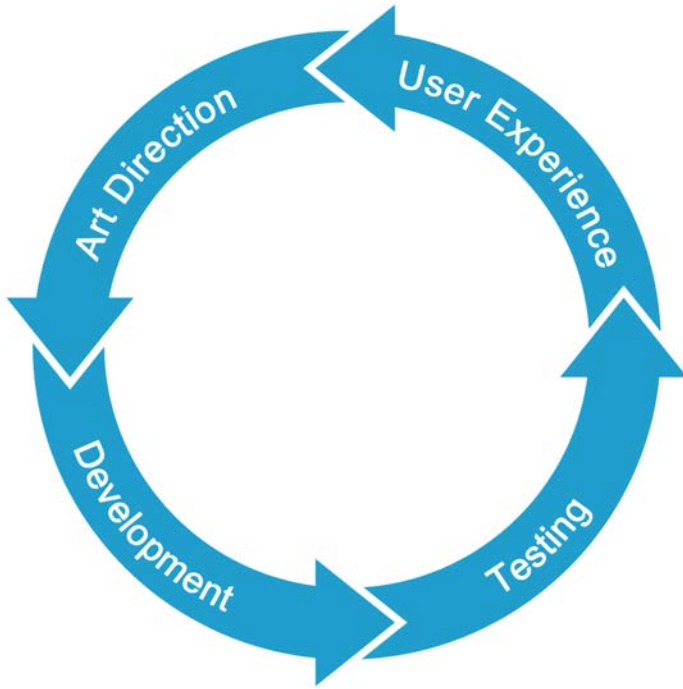


Figure 1.3 In this new, more agile approach, user experience and development happen iteratively. Deliverables are passed along and reviewed in an iterative cycle.

In the traditional pixel perfect web, the emphasis was on recreating the creative asset, but in this approach, the emphasis is on adaptiveness. Using the standard pixel widths and font sizes won't do anymore. We need something a little more fluid.

By focusing on adaptiveness and giving the site a fluid layout, your site stands to gain:

- A layout that adapts to variations in screen size technology. If a new web enabled product hits the market with an uncommon screen size (I'm looking at you, iPhone 5), then you are already prepared for it.
- By optimizing for mobile first, you prioritize load times from the beginning of development. Faster sites are always better.
- Cross browser layout issues can actually be easier to resolve with fluid CSS.

### 1.1.2 Key features

The responsive web couldn't exist as it is without a few key components: media queries, breakpoints, and the ability to serve varied CSS across various devices. These features are what create the cross browser ease and give our sites the ability to adapt to the user's screen.

## MEDIA QUERIES

Media queries are a type of CSS rule that qualify CSS based on factors defined by the query. Each media query contains a media type and a set of expressions that are checked by the browser. Some media types can be “screen” for digital screens, “print” for printed pages, or “all” for, well, all media types. Expressions are much more finite and include instructions such as “max-width” or “orientation”.

Media queries are at the very heart of the responsive web. They come from a specification in the 2001 working draft of the W3C CSS3 proposal that presented a solution to offering various CSS rules depending on browser size and device screen size. This ingenious little feature is the life and spirit of the responsive web, and without it mobile web development would be in a really tight spot.

The media query is simple and looks a little something like this:

```
@media screen {      #A
  p{ font-family: sans-serif }      #B
}
```

**#A The @media call initiates the query and then we can start declaring what media we would like to target, or our media type. In this case our media type is digital screens.**

**#B Between the brackets we can apply our usual CSS, but it would only effect browsers on digital screens.**

This line in a stylesheet will tell the browser to give paragraph tags the font-family of “sans-serif,” but only on screens. It won’t give the style to printed pages or e-readers. Now imagine something like this:

```
@media handheld{
  p{font-family: sans-serif }
}
```

With that little line, we are now targeting any user on a device that identifies itself as “handheld.” This would most likely be a device such as a smartphone or an iPod touch.

Media queries can be used to deliver CSS rules based on a number of factors, including screen resolution, orientation, and even color index!

Media queries can also be used to serve a relevant CSS file, based on the criteria laid out in the media query in a <link> tag. In this format, a media query is served within the head tag at the very top of a web page. These media queries would look like this:

```
<link rel="stylesheet" type="text/css" media="handheld" href="sans-
serif.css">
```

Using this method allows you to load this style sheet only for browsers that identify themselves as being “handheld.” The biggest difference between these two methods is that by serving the separate style sheet for mobile, we have additional HTTP requests for each required style sheet. I personally find the inline method easier to use and to teach, so we will go forward with that method in this book, but the <link> tag method is an equally efficient alternative.



The key to using media queries in responsive design lies in their ability to serve CSS based on viewport width. Viewport width is the width of the browser window. These are what are called “expressions” and they serve as the parameters that the browser checks against.

Some of the information a device relays to a server includes browser, the resolution of the device being used, and the size of the window viewing the page. In the responsive web, it’s important to note these factors and to understand their differences. Using media queries, you can serve CSS based on either device width or browser window width. To apply CSS based on a browser with a width of less than or equal to 400 pixels, you would use a media query like this:

```
@media (max-width: 400px) { ... }
```

Alternatively, if you needed to target only devices with a width smaller than 400 pixels, you would change our expression to something like this:

```
@media (max-device-width: 400px) { ... }
```

It’s important to note the differences between the two, because in some cases you might wish to serve the smaller size rules to a browser widow that’s been slimmed down by the user to avoid a horizontal scroll bar, or because of certain site functionality. You might wish to target only small screen devices, if you would prefer desktop users to be given the full screen version of a site, regardless of window size.

Another helpful distinction that you will want to be aware of is the difference between a “min-width” and a “max-width” media query.

@media (max-width:400px){...} targets a browser with a width of 400px or less, whereas @media(min-width:400px){...} targets a browser with a width of 400px or more. This is a minor but crucial distinction. With “max-width,” the rules affect every viewport below the set width, but “min-width” affects everything *over* the viewport width of 400px.

So how does one decide when to use a media query? That brings us to our next topic...

## BREAKPOINTS

The goal of responsive design is to avoid what Ethan Marcotte refers to as the “zero sum game” of redesigning a website for every possible device and viewport. To avoid this, we need to set bars of where we alter our layout to fit the needs of the changing context. As the site we’re working on goes from a mobile device width to a desktop width, at what point does it change, or “break?”

This is what we call a “breakpoint” in responsive design. Breakpoints are the points at which new rules are served to the responsive site.

Remember the resizing room analogy from earlier? Let’s say you have a 600 square foot room that starts to shrink. At about 550 square feet, things start rubbing against each other and the room is cramped, so you resize the furniture and adjust the room’s layout. The room then continues to shrink, and when it hits 500 square feet, you again have to adapt the room’s layout. In this metaphor, your room’s breakpoints are at 600 square feet, 550 square feet, and 500 square feet, because these are the points at which your layout starts to break.

Some like using standardized breakpoints, built specifically for mobile, tablet, and desktop variations. For me, I prefer starting from a mobile-first website and then growing the site from there. In my method, I expand the site gradually, and once the layout starts to look off, or has excessive space at the sides, I insert a breakpoint and start to adjust as gradually as possible.

#### **GRID BASED LAYOUT**

Xxxxxxxxxx

#### **IMAGES AND MEDIA**

Xxxxxxxxxx

## **1.2 Building your first Responsive Site**

Now that you have an understanding of what the responsive web is, it's time to dive into building a responsive site. In this chapter, we're going to walk through the fundamentals of a responsive site build. In this portion of the book it's important to have some basic understanding of HTML and CSS. For the absolute beginner, I'll try to break the concepts down a bit, but it might be helpful to do some research if you find yourself getting lost.

First we're going to create a prototype. When building a responsive site, we normally use rapid prototyping, because you can quickly view and arrange content in the browser. It differs from general prototyping in that Rapid prototyping is written in HTML, so it renders in mobile and tablet browsers as well as desktop browsers. This gives teams a distinct advantage once the actual design phase approaches.

Then we'll discuss how to interpret a traditional layout, like the one you might get from an art director in the form of a Photoshop, Fireworks, InDesign, or Illustrator file. We'll discuss how to take the components of a full site design and interpret them into the markup for a mobile site. After that, we'll cover how to use percentages to build the site layout, and I'll show you how to implement responsive images. Then we'll get hands on with our first breakpoint.

This is some exciting stuff, and this section will build some of the foundations that will carry us through the rest of the book. This is by no means all there is to the responsive web, but it is at least the tip of the iceberg.

### **1.2.1 Creating Prototypes**

When I was a teenager, I loved to work with my hands. I would build toolbox after toolbox in my high school shop class. I would build one and think to myself, "This is good, but it's not great," and immediately want to do it again. Every night I would snip, bend, and weld these little metal boxes, and each toolbox I made was better than the last, improving my technique and adding little tweaks here and there.

Creating prototype after prototype made it so that when I took my final exam I knew exactly how the toolbox should be built. The same is true for responsive sites. By prototyping before you build, you make sure that you're creating a site that communicates your vision clearly. For both the designer and developer, rapid prototyping is essential. Rapid prototyping is the process of building a site for exploratory purposes and while there are a few different approaches to rapid prototyping, we're going to use Foundation, by Zurb. Most prototyping

frameworks are relatively similar, and though I would discourage using prototype markup in the actual production-ready site build, if you absolutely have to, Foundation is relatively clean for production.

---

### **Developer Insight: Prototyping**

---

Rapid prototyping is your first line of defense in the war against bad ideas. Since the responsive web is constantly in flux and every element on the page needs to be agile enough to refactor itself, getting in ahead of design and identifying modules and page templates is the best way to guide a conversation. When prototyping it's common to find out that there are easier ways for users to accomplish goal, or that an element is completely unnecessary.

The most important part though, is that it gives you a common piece to talk about with your team, that can be interacted with across devices. Remember, until something is actually it's purely speculative. I'm constantly asked to attend meetings or review creative to try and find out whether or not ideas will work. Most of the time I make my best guess, but I can never truly know unless I have time to build a prototype and experiment. That's where some of the most innovative work comes from, in my experience, the sense of exploration that comes with building a prototype.

---

### **FROM WIREFRAME TO PROTOTYPE**

We are going to prototype a redesign for my site, [matthewcarver.com](http://matthewcarver.com). I'm using my personal site because it's one of the few kinds of site I can say with confidence every developer has experience with. Everyone who builds websites starts with their own blog or portfolio site. Even if you've never built a website, you've probably considered what your site needs to have on it. Since I want to keep my site extremely simple, we are going to work from a rough sketch.

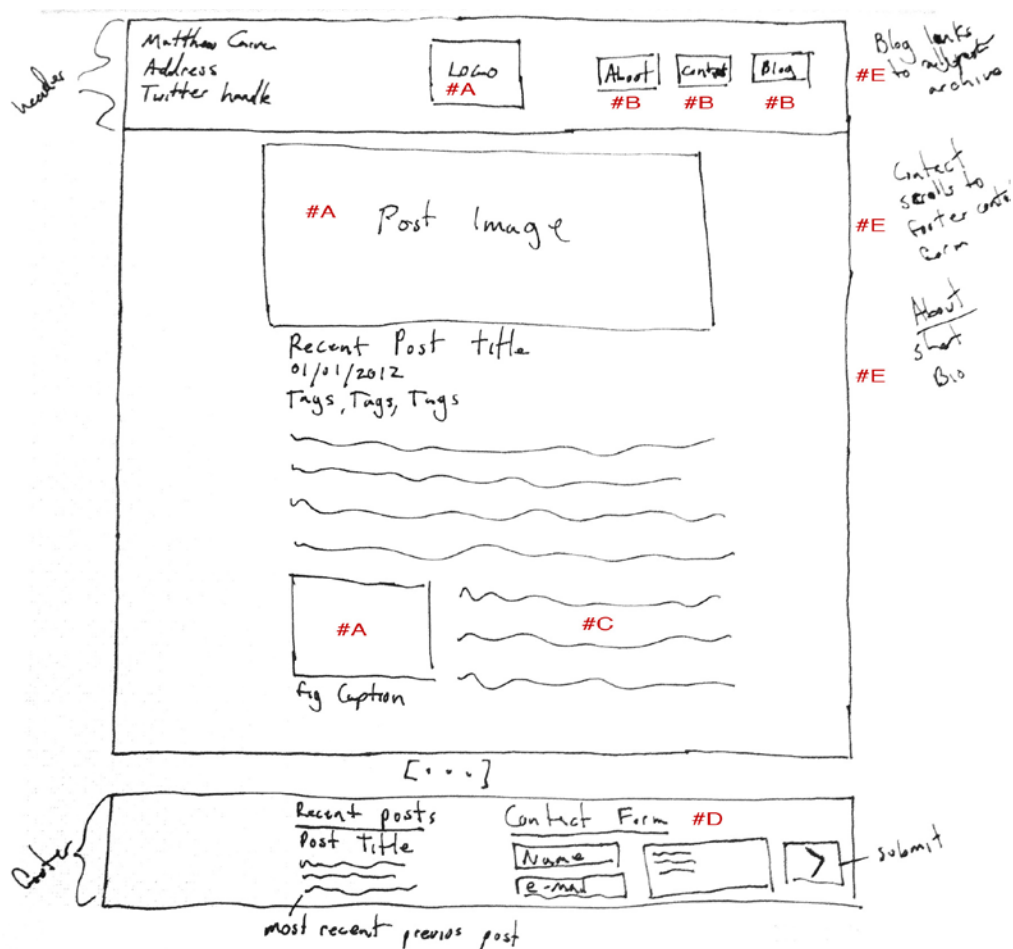


Figure 3.2 Rough wireframes give us some direction with our prototype.

**#A** We'll be using placeholder images to get a sense of image size and relationship. We don't want to spend a lot of time working through the creative aspects of these images just yet.

**#B** These buttons will be generated using preset elements in the Foundation framework.

**#C** We'll be using placeholder text to give us a sense of the copy.

**#D** Foundation features a handy framework for forms, which we'll be taking advantage of for this mini contact form.

**#E** Some scratch notes on the side can be useful reminders, if you can manage to read them.

As you can see, this rough draft gives us a model to build our prototype out of. We can clearly tell what elements we have on the page and get a general sense of their relationships. It's just enough to get us started.

## USING MARKUP TO CREATE A PROTOTYPE

90% of the CSS and JavaScript used in Foundation is already there for you, it's just a matter of writing your markup to fit it, and luckily the markup is straight forward. The first thing you should know about Foundation is that in its desktop view, it's based on a 12 column grid. Every row contains 12 columns, and rows can be nested within each other. This will start to make sense as we go along.

We'll start by writing the code for the header area. We need to set our header aside as our first row, and then work inside that row to separate areas out based on our 12 columns.

```
<div class="row">#A
  <div class="five columns"> #B
    <p>Matthew Carver<br/>
      New York, NY<br/>
      @matthew_carver</p>
  </div>
  <div class="two columns"> #B
    #C
  </div>
  <div class="three columns offset-by-two"> #B #D
    <a class="small button">About</a> #E
    <a class="small button">Blog</a> #E
    <a class="small button">Contact</a> #E
  </div>
</div>
```

**#A** We declare a row to start our section off.

**#B** By simply stating “five columns” or “two columns,” we can apply the desired spacing, as long as we end up with a total of twelve columns.

**#C** Serving placeholder images from placeholder.it is a simple way of showing where images belong and roughly what size they should be.

**#D** Here, we need to add a little white space, so we offset our column by three column widths.

**#E** The .button class gives us buttons to imply interface elements. These buttons can be accompanied by adjectives to describe their size (tiny, small, medium, large), as well as relationship to the rest of the interface (such as success, alert, or secondary buttons).

With this little snippet of code, we've produced a simple header for our prototype. Now that we have our header, we can add our footer before moving on to adding the content.

```
<div class="row">
  <div class="three columns">
    <h5>Previous Post</h5>
    <h6>Post Title</h6>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod</p>
    <a href="#">Read More</a>
  </div>
  <div class="seven columns">
    <div class="row">
      <div class="nine columns">
        <h5>Contact</h5>
        <input type="text" placeholder="Name"> #A
        <input type="text" placeholder="E-mail"> #A
      </div>
    </div>
  </div>
</div>
```

```

<textarea placeholder="Message"></textarea> #A
</div>
<div class="three columns">
  <h5>&nbsp;</h5> #B
  <input type="button" value="Submit" class="large button">
</div>
</div>
</div>
</div>

```

**#A Foundation has styles in place for form elements. These are extremely helpful in discovering usability issues.**

**#B This blank <h5> is there just for spacing purposes. Since this is just for internal use, we don't need to worry too much about making our markup pretty.**

In just a few lines of code, we've been able to produce a browser based prototype, which can be hosted in a development environment and shared with the rest of the team. As a designer, this can be a powerful tool in anticipating the layout of a site, and as a developer, having a model in the browser speaks directly to the responsive elements of a site.

If you scale your browser window down, you'll start to see how the site breaks down in smaller screens. If you followed along and built a prototype for yourself, you'll see that if we immediately jump into the mobile view, our header breaks down and gets a little clunky. Fortunately, the Foundation framework offers us control over what our prototype looks like as our viewport shrinks.

## USING RAPID PROTOTYPES TO CREATE CONTENT

One of the advantages of using a rapid prototype is the ability to quickly view and arrange content in the browser. For instance, if you're doing a redesign of a blog site, you can post existing blog articles, images, and videos into the prototype and view ways the content interacts. Through this sort of exploration, you have the opportunity to discover facets of the site that might remain uncovered in wireframes or comprehensive layouts.

Before you use rapid prototype you need to define what type of content to arrange. Defining content types is important in the responsive web for two reasons.

- 1) Defining content early can identify why the user is visiting the site and prioritize according to those needs.
- 2) Once you identify the content types in a site, you can start building a content well. A content well is a collection of assets, such as images, articles, and copy for the site. Preparing content before the actual build will give developers a chance to focus on the task at hand, while other teams can prepare and edit the content. As a developer, nothing is worse than spending the night before a site launch adjusting pages for new content. Hopefully, with a prototype and a good client, you can circumvent those long nights.

Looking at our wireframes we can see that we have two different content types, a large image and an article, with a headline, tags, and a date. We're going to want to replicate these in our prototype as they are in the wireframe.

```

<article class="row">
  <header class="twelve columns"> #A

```

```

<figure>
  
</figure>
<h1>Recent Post Title</h1>
</header>
<div class="twelve columns">
  <aside class="three columns">
    <div class="row"> #B
      <div class="eleven columns panel"> #C
        <h5>01/01/2012</h5>
        <p><span class="label">Tag</span> [...]</p> #D
      </div>
    </div>
    <p>Lorem ...</p>
    <p>Lorem ...</p>
    <p>Lorem ...</p>
    <p>Lorem ...</p>
  </div>
</article>

```

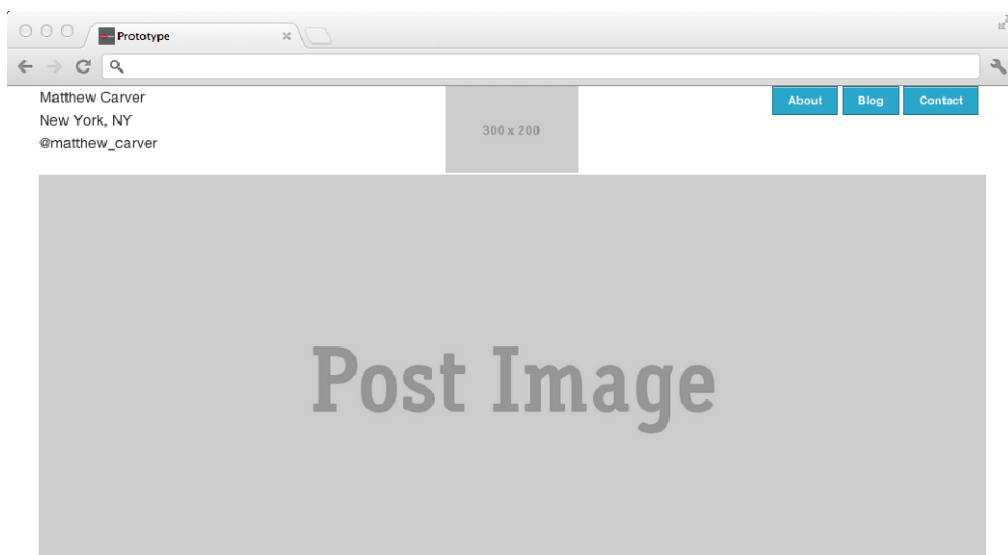
**#A** Twelve columns will fill end to end in the row, but will also ensure consistent padding throughout the rest of the page. Every column has a padding of 15px applied to the left and right.

**#B** We can nest rows within rows. This way we have a little more control over sizing and spacing of elements. A nested row is still composed of twelve columns.

**#C** The .panel class is used to give the element a little grey background. It's useful in distinguishing certain elements on the page.

**#D** Here is an example of the label class.

With the above markup, we have produced a rather respectable prototype (figure 3.7).



## Recent Post Title

01/01/2012

Tag Tag Tag Tag Tag

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate

Figure 3.7 With just a little markup, we are well on our way to a functional prototype.

You'll notice I've already slightly deviated from my wireframes. Once I got my content into place, I noticed that the date and tags look better floated to the left. This is an example of the kind of insight you can find by building a prototype.

### Designer Insight: Reading a prototype

When working with a browser prototype, it's hard not to get too hung up on the visual layout. A prototype should address a few questions about the site, mostly the idea of what will be on the page. By giving you a rough example of what content will be on a page, it frees you to design more of the "how". How things will come together, or how you can present a set of tags in a meaningful way.

Charles Eames once said "The details are not the details. They make the design". A prototype doesn't deal in details, it deals in the big, mammoth chunks that construct the



frame for the site. It gives the content a place to sit and simply lets the objects lie on the page but design is what brings these crude shapes and forms like and beauty.

When a designer is forced to deal in both the clunky chunks of a site, as well as the details, the details get lost and the chunks become obstacles. The design exists simply to satisfy the needs for chunks and their, not to make the chunks themselves something satisfying.

### 1.2.2 The basic Responsive layout

In the responsive web, we strive to circumvent the need to design multiple layouts for the various screen sizes. In advertising, a comprehensive layout (or comp for short) is a static image used to represent the final composition of a site. It's an element left over from the days of print advertising, and served its purpose well back then. Comps were also handy to have in the early days of web design because of the lack of variation in screen sizes.

Unfortunately, comps are too limiting for responsive design. They fail to speak to the scalability of a site. That's where a developer comes in. Developers have the ability to translate the design into the language of the internet, using HTML, CSS, and JavaScript for front end development. Later we'll talk about using rapid prototypes to help bridge the gap between a responsive front end and the design, but for now let's focus on how to take that full screen layout and turn it into a responsive website.

We can look at a website's composition and make some basic assumptions about what can stay, what can be refactored, and what needs to go to save space in a small screen (mobile) environment. Let's look at our example layout (figure 1.4), and make a few of these assumptions for how we are going to build this mobile-first.



Figure 1.4 Inspecting the design, we can start to reveal more about the site's intent.

**#A Obviously we're going to have to adjust the two-column layout for small screen devices. It would make sense to stack the header, left navigation, and article content vertically in this case. In later chapters we'll go over how to build off-canvas navigation, but for now we'll just stack everything.**  
**#B That masthead image could be a bandwidth hog. We are going to want to scale that down for mobile devices, so we don't kill the user's bandwidth**  
**#C Since there's a lot of type on the page, we are going to want to scale the type for mobile. It's important to focus on content on a smaller screen. Readability is of the utmost importance in the responsive web.**

With a few assumptions in place, we now have a plan for converting this full screen art direction into a mobile-first site.

### Developer's insight: the CSS box model

In my experience, I find it's important to keep the basic CSS box model in mind when making these layout assumptions. The page will flow by default from top to bottom, and then left to right (or right to left, depending on your use of CSS floats). You can easily float objects left or right once you expand the layout, but with your markup, make sure to write the first (and most important) elements at the top of the document, and then work down the page.

## 1.2.3 Mobile-first markup

We're going to focus first on the markup of the page. This way we can focus on writing our HTML in the most efficient manner possible. We'll approach the CSS a little later in the chapter, but since writing efficient code is so important to the responsive web, we want to spend a little time explaining practices that will make your markup as clean and semantically correct as possible. This will also be beneficial later, as you expand your site into the tablet and desktop versions. Since you need to go over the same code multiple times, make sure to take your time and write good, clean code.

The goal in writing the markup is simple: convert the content into HTML. We'll use some basic placement elements, but ideally we want to keep things as lean as possible. This is good practice in general, but especially important in responsive development. CSS and JavaScript are flexible, but our markup should stay consistent between viewports.

So let's start with some standard stuff. Since we're going to go through basic site structure and CSS, I'm going to throw in a doctype declaration and html, head, and body tags.

```
<!doctype html> #A
<html>#B
<head> #C

<link rel="stylesheet" href="screen.css"> #D

</head>
<body> #E
```

```
</body>
</html>
```

- #A Here, we declare our “doctype.” This lets our browser know the page is written in HTML5.**
- #B This is our html tag. It’s the base tag in any website.**
- #C Our head tag is where we enclose all of the information the browser needs in order to render the page properly.**
- #D This is the link to our “stylesheet,” which is written in CSS.**
- #E This is our body tag, which is where all of the displayed page content goes.**

From here we can start interpreting our markup into a responsive site. The first thing we want is a container div. This will prevent our site from being too fluid and becoming unstable between breakpoints. Note that I follow the closing tag with a comment identifying the div’s selector. This will give me a little flag to remind me what the closure applies to. Since responsive design is highly iterative, these comments can be life savers.

```
<!doctype html>
<html>
<head>

<link rel="stylesheet" href="screen.css">

</head>
<body>
<div class="container"> #A
</div>
<!-- end .container --> #B
</body>
</html>
```

- #A Using a container to wrap the page can be very helpful in scaling the site. In more complex designs, you’ll want to use multiple containers sharing the same class.**
- #B As the page grows, it’ll be incredibly helpful to have these comments to show us where our elements close.**

Next, we’re going to start creating our page elements. We want to model our DOM on a left to right, top to bottom structure. This will give us a logical flow when expanding into the wider view. Let’s start with our main layout structure.

```
<div class="container">
  <header class="main"> #A

  </header>
  <!-- end header.main -->
  <aside class="main"> #B

  </aside>
  <!-- end aside.main -->
  <section class="content"> #C
    <header class="masthead"> #D
    </header>
    <!-- end header.masthead -->
    <article> #E
```

```

        </article>
    </section>
    <!-- end section.content -->
</div>
<!-- end .container -->

```

- #A This header will serve as our site header. Here, we'll build the site wide header.**
- #B This aside will be our site navigation.**
- #C The section with the class of content will serve as our main content area. This is where the article copy will end up.**
- #D Here is our masthead for the content section. In this block we will call our article masthead.**
- #E This article tag is where we will directly control the written article for the page.**

There are a few things to note in this example. I use classes to define the areas I'm marking up. This will give me repeatable classes to use throughout the site and give me more streamlined CSS. I prefer to reserve ids for JavaScript selectors, using them in CSS only in cases where I need a very specific selector.

We've established our structure, so we can start placing content into the page. When writing your initial mockup, it helps to keep in mind that you need to write as little as possible. Be as direct and semantic as possible.

Now that we have our base template out of the way, let's get into the details of our markup. We'll start with `<header class="main">`:

```

<header class="main">
  <h1 id="logo">book-site.com</h1> #A
  <div class="social">
    <a class="icon twitter" href="javascript:void(0);">twitter</a> #B
    <a class="icon facebook" href="javascript:void(0);">facebook</a>
  </div>
</header>
<!-- end header.main ] -->

```

- #A We'll be using CSS to display an image in the site logo, but it's good form to keep write out the site name, just in case our CSS isn't loaded.**
- #B For these temporary links I'm using javascript void, instead of the more common #. This simply keeps the page from returning to the top when I click the link.**

Next we can start marking up the aside block, which will contain secondary elements such as navigation or advertisements:

```

<aside class="main">
  <nav> #A
    <a href="javascript:void(0);">books</a>
    <a href="javascript:void(0);">authors</a>
    <a href="javascript:void(0);">news</a>
    <a href="javascript:void(0);">blog</a>
  </nav>
  <div class="ads"> #B
    <figure class="ad">#C
      
    </figure>
  </div>
</aside>

```

```

    <figure class="ad">
      
    </figure>
  </div>
  <!-- end .ads -->
</aside>

```

**#A This <nav> tag will serve as our primary site navigation.**

**#B Here is our container for the ads.**

**#C The <figure> tag serves as a semantic wrapper for our responsive images.**

Notice that I'm wrapping the <img> tag on the page with a <figure> tag. The figure tag will be used as a responsive image wrapper to scale the image with CSS. It's a handy little trick, and I like to use the wonderfully semantic <figure> tag, as opposed to using a <div> with an image-wrap class. We'll cover responsive images in the next section, when we start talking about CSS.

```

<section class="main">
<article>
  <header class="masthead"> #A
    <figure>
      
    </figure>
  </header>
  <!-- end header.masthead -->
  <h2>Headline</h2> #B
  <p>Lorem ipsum dolor sit amet, [...]</p>#C
</article>
</section>
<!-- end section.main -->

```

**#A This header holds what is called the “masthead.” It's an old print term used to describe a graphical image at the top of a page. Since the masthead is important to the article below, it's included in the article tag.**

**#B The headline follows the masthead.**

**#C I used an ellipsis to shorten the placeholder copy, for the sake of brevity.**

So now we should have something that looks like figure 2.3.

**book-site.com**

[twitter](#) [facebook](#)  
[books](#) [authors](#) [news](#) [blog](#)

ad

ad

Masthead

## Headline

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 1.5 Our page in raw, unstyled markup.

This essentially functions as our raw material. It's unrefined, but now we have an idea of where to go. By throwing all of our content out there, we have a base to start from and can begin to style our page. It's not uncommon to find that you need to add some helpers to the page, but since we are taking a mobile-first approach, it's important to strive for as light a page as possible.

### 1.2.4 Using percentages in CSS

We can now start styling our page. This is where the rubber really meets the road in responsive design. Percentages give us a fluid model to base our site's structure around. With percentages, every element becomes relative to its parent element.

One of the wonderful things about building web pages is that when you assign a rule to an element, it takes on that property. When you build anything in this world, you assign it some values. Say you're building a birdhouse. You can make that birdhouse six inches long, seven inches wide, and five inches tall. When you're done, you're done. It's a 6x7x5 birdhouse, forever and always.

That birdhouse can never adapt to fit bigger birds, or shrink to fit in a smaller tree, but on a web page you can do exactly that. If you can build a flexible birdhouse that is 5% as tall as the tree it's in, then if the tree grows, the birdhouse grows with it, and then bigger birds can live inside it. So, how do we turn the very static blueprint we've been given into a magically expanding birdhouse?

## USING CONTAINERS

Let's start by defining the context. Because we are adapting a mobile-first approach, we'll start by defining our container element with a little CSS:

```
.container{
  width:240px; #A
  margin:0 auto;
}
```

**#A This width will be expanded later using media queries.**

With our container in place, we can start putting some other pieces into place. Our container will be the overall width of our expanding birdhouse. As the content needs to grow, this container will grow to accommodate it.

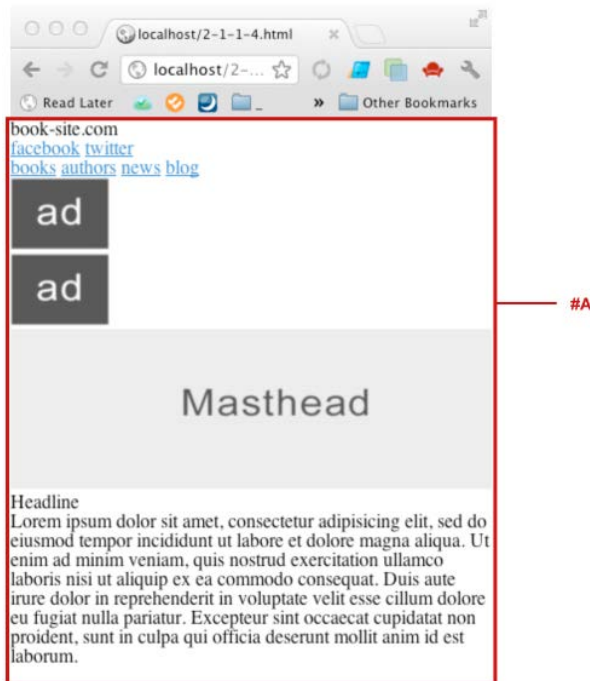


Figure 1.6 A container div will wrapper around all the content. The outline, #A, shows a container div. This will serve to wrap our content and rescale it at various viewports.

From here, we need to start making the contents of this container. We need to ensure that the containers are fluid, so that they can grow and shrink to meet our needs.

### WHOLE, HALVES, AND QUARTERS

At its core, the mobile grid is broken into wholes, halves, and quarters. This is a very simple grid and works extremely well for small screen devices. By using such a rudimentary grid, it becomes easier to understand content areas and how they interact. Important areas, or areas that serve as wrappers for quarantined content, get to be wholes, occupying 100% of their parent. Other areas can be halves or quarters, depending on how their priority and how they interact with the rest of the page.

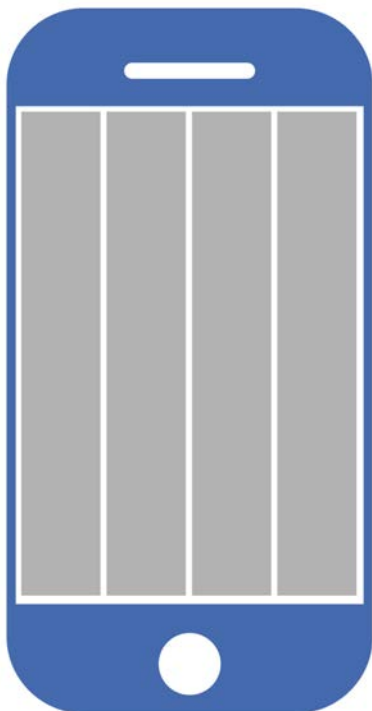


Figure 1.7 A four column grid for mobile web sites.

We know that the header, aside, and content sections are going to fill the entire width of the container by stacking vertically on top of each other, so they can take advantage of all the space allotted to them.

```

*.main{
  width:100%;#A
  padding:10px 0;
}

```

**#A By setting the width to 100%, every element with the main class will occupy all of the width within the parent element.**



With this, all of our main blocks will fill out the width and stack nicely. Now, let's add a little styling and jump ahead a bit.

```
header.main{background:#4d4d4d; height:16px;}#A
aside.main{background:#e4e4e4; height:16px;}#A

h1#logo{
  width:50%; #B
  height:20px;
  float:left;
  background:url(/images/logo.jpg) no-repeat top left;
  text-indent:-99999px;
}

.social{
  width:50%; #B
  float:left;
}

.icon{ #C
  width:25px;
  height:20px;
  display:block;
  float:right;
  margin:0 10px 0 0;
  text-indent:-99999px;
}

.icon.facebook{background:url(/images/facebook.jpg) no-repeat top right;} #D
.icon.twitter{background:url(/images/twitter.jpg) no-repeat top right;} #D
```

**#A** These rules simply assign colors to `aside.main` and `header.main`.

**#B** In these two places we are setting a width of 50%. This way the two elements each occupy 50% of their parent element, in this case, `header.main`.

**#C** Using the class of `icon` to give the rules for both of our icons saves us from repeating ourselves, since they both function similarly.

**#D** Using the unique `.facebook` and `.twitter` classes, we can specify images for each of the links.

There's a lot going on in this bit of CSS. First of all, we are giving two elements, `.logo` and `.social`, block percentage based rules. This ensures they will always occupy 50% of their parent. If the parent (`header.main`) occupies 320px, then `div.social` and `div.logo` will each occupy 160px. If the header scales up to 640px, then `div.social` and `div.logo` will each occupy 320px.

When you know that you have even halves or quarters, percentages can be easy. It gets more complicated when you have to make a percentage based layout based on fixed creative. Later in the book, we'll go into depth on how to do that.

We've already set our `<aside class="main">` to 100%. Now we'll add this to our stylesheet:

```
nav{
```

```

    width:100%;
}

nav a{
  display:block;
  float:left;
  width:25%; #A
  text-align:center;
  text-decoration: none;
  color:#333;
}

```

**#A This call of 25% will break our four navigation links into quarters, each occupying 25% of the page.**

Here the links are broken into quarters of the navigation area. This gives the links a good amount of interactive space.

```

.ads figure{
  width:50%; #A
  display:block;
  float:left;
  padding-top:10px;
}

.ads figure img{
  display:block;
  margin:0 auto; #B
}

```

**#A We split the two figure tags wrapping the ads into halves.**

**#B By setting margin:0 auto;, the images will remain centered in the parent figure tag.**

Using some simple, fluid CSS, we've managed to organize our site in an orderly manner and add some logical navigation.

### 1.2.5 Adding text and images

Now that we have our primary navigation and site header in place, we can start dropping content. Since this is a rudimentary mobile site, the content will just follow below the navigation, preserving the hierarchy from the comp.

This means we'll include a masthead, a headline, and a few paragraphs of placeholder text. Earlier we wrote the markup for this; now we're just adding the content. To review, the markup looked like this:

```

<article>
  <header class="masthead">
    <figure>
      
    </figure>
  </header>
  <!-- end header.masthead -->
  <h2>Headline</h2>
  <p>Lorem ipsum dolor sit amet, [...]</p>

```

```

</article>
</section>
<!-- end section.main -->

```

Content is king in the responsive web. Every site I've ever visited in my life has been visited because the site contained some information that I wanted. Because of our profession, sometimes that information relates to design choices or code snippets, but when we are browsing for information outside of our professional lives, we're often looking for some other bit of content.

The content we are scouring for could be video, images, or text. No matter what the content, the means by which we search are always changing, and in the responsive web we need to adjust our content accordingly.

In later chapters I'll focus deeply on this topic, since there is a lot to talk about. However, for now let's just focus on two important factors:

- Displaying images fluidly, so they can be easily resized with the browser window
- Applying scalable CSS to the text, so we can adjust font sizes for various devices

Before we can do this, we have to set the context of the article tag. This tag will serve as our wrapper for the text, headline, and images it contains. We can just apply some simple CSS to accomplish this wrapping:

```

article{
  width:100%;
}

```

## SCALABLE IMAGES

In the article's masthead, we have a large image that will need to be shrunk down:

```

.masthead{
  width:100%;
  padding-bottom:10px;
}

figure{
  width:100%; #A
}

figure img{
  width:100%; #B
  max-width:1000px; #C
  height:auto; #D
  display:block; #E
  margin:0 auto; #E
}

```

**#A** We start by setting the `<figure>` tag to have a width of 100%. We're going to use the figure tag because it's intended as a semantic wrapper for images. This makes more sense than using a div with a image-wrap class, since we should always strive to use semantic markup. Some sort of wrapper is necessary in order to properly scale the containing image.

**#B** Here we repeat our width of 100%. The `<img>` will then occupy 100% of the wrapper.

**#C The max-width will prevent the image from exceeding its native size. This way we won't lose any image quality from the image scaling above its native size.**

**#D The height:auto rule will maintain the image's aspect ratio.**

**#E Here we are setting display:block and margin:0 auto to keep the image centered if it scales beyond the max-width of 1000px.**

There's some important stuff going on here. This is how we scale images for responsive web development. Luckily for us, all browsers can scale images and preserve their aspect ratio. Aspect ratio is the image's proportional relationship between width and height. An image that is 1000px by 500px can scale down to 100px by 50px for a mobile device and still look great. In fact, for devices with higher pixel density, such as the Apple Retina Display, this can create a desirable effect by doubling the image's pixel density.

By giving the image a max-width of 1000px and centering it inside the figure tag, we ensure that the image doesn't look "broken" if the figure tag extends beyond 1000px. This is a safety measure to ensure that our site retains its wonderful layout in a wide screen monitor.

### **1.2.6 The fickle and mighty EM**

Now that we have a fluid masthead image, we need to focus in on our text. Adjustable text is another foundational element in the responsive web. People read copy differently, depending on the device they are using. A desktop monitor might be further away from a user than a handheld device.

Personally, I like text on all of my devices to be large, because I don't like having the device too close to my face. Because of this, my devices commonly have a higher default text size. This inevitably breaks layouts that aren't designed with fluidity in mind. If a site is built with an ebb and flow for content, using less rigid CSS, my text size preferences shouldn't affect the layout.

So how is this achieved? The answer is in EM values. EM values are a little hard to understand, because they are abstract and can change at the drop of a hat. Historically front-end development has had an emphasis on pixel perfection, and typically developers have used pixel control. Pixels are literal, stiff, and consistent. 12 pixel type is exactly 12 pixels (unless on a Retina display, which has quadruple the pixel density, so 1 pixel unit is actually 4 pixels). The EM, however, is a bit fickle; but once mastered, it's highly rewarding.

---

#### **Designer's insight: web typography**

Mobile web sites are great opportunities to focus on type. In small screen layouts, there are a few important factors to consider. First, the screen is obviously going to be smaller, so there's less room to spare. Type might also need to be scaled up in order to be readable.

It's important that you allow your site's written content to flow, and clear it of any distractions. Look at popular apps, such as Readability and Instapaper. The popularity of these apps is the result of a user's desire to strip down distractions and get to the heart of content.

The trick of EM is that they are a cascading size. Browsers have a set base font size by default, usually 16px. This would make 1EM equal to 16px by default.

Imagine you have an H1 that has a font-size of 2EM: that H1 would be 32px. Now, say you need a smaller block of text within that H1, so you wrote this:

```
<h1> Headline <small>Sub-Head</small></h1>
```

With CSS, you would apply this rule:

```
h1{
  font-size:2em;
}

h1 small{
  font-size:0.75em;
}
```

The `<h1>` takes the default font-size (16px) and multiplies it by its EM size (2em) to get the applied font-size (which would appear as 32px). The font size would then cascade to the child tag, which in turn sets the font size to 0.75em, so the parent font-size (which appears as 32px) is multiplied by the child selector (0.75em) to get the new size (32 x 0.75 = 24px).

Let's apply this to our demo. First we want to give our h1 its sizing. Let's resist the urge to reset the body type to 16px, as most resets are prone to do. We'll avoid this to respect the user's browser settings. Since it's a header, we'll make it considerably bigger than the rest of the type:

```
article h1{
  font-size: 1.5em;
}
```

So if the user has their default type set to 16px, this header will appear to be 24px tall.

We can leave the paragraph text set to 1EM, since it's a default setting for paragraphs, but let's suppose that we know we are going to be including some bold text inline that needs to be bigger. We could do this using by adding a rule of font-size: 1.1em to the `<strong>` tag, written in CSS as this:

```
p strong{font-size:1.1em}
```

We are going to discuss the details of EM in depth later in the book, where we compare using pixels and EMs in responsive design, and how to manipulate cascading EM sizes to your benefit. For now, just keep in mind that EM should only be applied directly to inline text elements, such as `<span>`, `<p>`, `<h1>` through `<h6>`, `<small>`, and `<big>`.

Avoid setting EM directly to block elements on the page. This can cause problems when the font-sizes cascade. If you need a particular size text, but only in a given element, say for instance this scenario:

```
<div class="promotional-area">
  <span>EXTRA BIG TEXT</span>
</div>
```

Instead of this:

```
span{font-size:5em; /* big text */}
```

Try doing this:

```
.promotional-area span{font-size:5em; /* big text */}
```

Or use a helper class, like this:

```
<div class="promotional-area">
  <span class="big-text">EXTRA BIG TEXT</span>
</div>

span.big-text{font-size:5em; /* big text */}
```

We should have a rough little demo site, but if you expand your browser you'll see that we only have a mobile site. If the entire point of the responsive web is to build sites that are fluid, this simply won't do. The responsive web is all about adjusting layouts for various devices. Now it's time to apply the core concepts of responsive web, media queries and breakpoints.

### 1.2.7 Your first breakpoint

As you get better at building responsive sites, you'll get a better feel for where to set breakpoints, but for now we'll simply set one at 600px. In this instance, 600px is kind of arbitrary, but a good rule of thumb with responsive sites is that you should insert breakpoints whenever the site breaks. If the layout starts to fall apart, you add a break point and fix it. In our workflow, we are striving to get this new mobile site to look like the desktop creative, so we want to incrementally adapt the site until it looks like the comp.

```
@media only screen and (min-width: 600px){#A
  #container{
    width:600px;#B
  }
}
```

**#A Here we declare the rules that have to be met before the CSS in our media query takes effect.**  
**#B All CSS within the brackets of the media query takes precedence over the existing rules.**

With this simple rule, our container div has taken a new shape. Now we can begin the work of refactoring our layout to match our original creative. The rules within the media query have a higher value than those outside of it.

Those with JavaScript or jQuery experience might recognize this as being similar to an IF statement. Basically, @media is like saying "If you're viewing this," and then you state the

conditions which validate the media query. In the above example, the conditions are “only screen” and “(min-width:600px)”.

The first condition is quite literal. “Only screen” means just that: only on a screen. The second condition, “(min-width:600px),” is slightly more complicated. This could be translated as “at a minimum width of 600px.” If we put the entire query together, it could be read as “If you’re viewing this only on a screen at a minimum width of 600px.”

Using the above query, if the browser size meets or exceeds 600px, the new rules take effect. There are some other subtleties involved in this, and several reasons for doing it this way. You are applying new rules only if the browser exceeds a size, as opposed to applying new rules if a browser is under a size as you might do with `@media (max-width:600px);`. This is beneficial, based on the assumption that a smaller screen means a smaller device, which means limited capabilities. This is a little complicated, and like much of this chapter, we’ll be discussing it more later.

---

### Developer’s insight: adding sibling queries

---

So this helps us change our conditions for 600px, but what happens if we need to expand for an even larger screen? We can add more rules on top of our existing rules. After the first media query, we can add another, for instance:

```
@media only screen and (min-width: 800px){#A

  #container{

    width:800px;#B

  }

}
```

**#A** Now at 800px wide, we apply our new rules.

**#B** Our rule here will override our previous style.

You know how you can override a style using an `!important` declaration in CSS? This functions in much the same way. By calling a media query later, you can stack the new rules on top of the previous ones. None of the rules for larger screens are loaded for smaller screens, but as you expand, the rules cascade down.

These are truly some cascading style sheets!

---

And there you have it, you’re first responsive website. Approaching the site in this way, with a prototype makes building the site that easy. Now we can get into adding details and style, but this will serve as our base.

### 1.3 *Summary*

In this chapter, we discussed what the responsive web is and its core concepts. You learned how to build your own responsive site. With this chapter, you have all the information you need to make a basic responsive site.

As we addressed earlier, collaboration is a huge part of your ability to succeed in responsive web design. This site example is extremely boiled down and with a designers eye and a coders hands, we can make something magical. Prototyping and giving each other meaningful deliverables is crucial to this process.



# 2

## *Design for mobile first*

This chapter covers

- Responsive navigation and design patterns
- Responsive grids
- Content design

There's a term I like to use to describe what happens when a project starts to get out of control: "gilding the lily." It's an idiom that is derived from Shakespeare, and it means to over embellish something. There's a lot of gliding of lilies that can go on when you assume everybody visiting your website is on a desktop browser. When you have a 1600 pixel wide canvas, you have a lot of space to fill, so you might add more buttons, animation, widgets, and images. If you were only designing for a desktop, great, but the whole idea behind responsive design is to be able to move through each device fluidly. You can imagine then if after you design for the desktop you need to make the site mobile you'd have to start hiding buttons and images and navigation. You'd stuff and scale and tuck until your 1600 pixel canvas was crammed into the frame. If you're not careful, the user will end up still loading all that hidden content on their cellphone, which slows the load time. So now, you've taken all this time for cramming, and your user with a 3G connection hits the back button because your page isn't loading. Not exactly the best user experience. If you're wondering if there is a better way, I'm here to tell you there is.

In 2009, Luke Wroblewski wrote a blog post and later a book titled "Mobile First," in which he laid out a philosophy of web design and development that asked people to focus first on a mobile experience. It's true that the canvas you begin with is much smaller, but by doing mobile first you retain control as the site expands, as opposed to losing control as the site shrinks.

In this chapter we are going to create a responsive website mobile first. We'll start out talking a bit more about mobile first and the pros and cons, plus specific challenges designers

face when designing for mobile first. Then we'll create a mobile first site showing how designers can solve the challenges of designing a responsive site.

### **Developer's insight: responsive web design is a team sport**

Though this chapter focuses on the design process, it's important to remember that the principles behind the process are important even to somebody who only codes. Even if you have no interest in design whatsoever, knowing about some of the topics covered here will help you speak to art direction, and help give you some ideas that you can share with designers.

As a developer, you might not think of yourself as creative, but the entire site experience depends on you. Clean and efficient code is of the utmost importance in a responsive site because without it, pages load slowly and the interaction can feel clunky. Make sure to take your time and develop sites mobile-first, but with the end goal in mind. This means using semantic markup, object oriented CSS, and efficient JavaScript.

## **2.1 Why mobile-first design**

One of the biggest challenges in responsive web design is how to build a site that evolves and scales consistently. The responsive approach seems too limiting to a lot of designers because the canvas they begin with is so much smaller. However, by working inside of the limitations that exist, you're able to exercise control over those limitations and master the small screen first to then scale consistently.

In this section we're going to look at the benefits of starting small and also the challenges, then once that's behind us we'll get started with our mobile first site.

### **2.1.1 Benefits of Mobile First Design**

Designing for mobile first can feel like a huge step and maybe a bit of a departure from your current workflow. When embracing a new way of working it's important to vet the idea and have a good sense of its rewards. Aside from creating a mobile site, designing mobile first has a wealth of value.

#### **PRIORITIZES CONTENT**

Mobile first also allows you to prioritize content and focus on the most important parts of your site early. A mobile screen only has room for the most important content and makes it so that you make a decision on what is crucial to your site in the early design phases. By loading the essentials first you'll be able to add functionality, as it's needed.

#### **WORK IN NEW SPACE AS SITE EXPANDS**

A while ago I changed apartments. I was living in a small studio and upgraded to a one bedroom. In my new apartment I found that I had a lot of open space because of the increase in size. After a few trips to the store I filled the space out pretty well. I moved from Texas to

New York City a few years later. After the move I realized that apartments in New York are incredibly small compared to what I had in Texas. When I moved everything in, I quickly ran out of space and ended up having to get rid of a lot of furniture to make space.

The point is this: It's much easier to fill new space than to cram the same things into a smaller space. By designing for mobile first you work in new space as the site expands, instead of cramming the same content into an ever-tightening space.

### **2.1.2 The challenges of designing for mobile first**

When designing for mobile screens first, you will be presented with some new challenges. As a designer it's easy to get accustomed to having a large canvas to work on. Web design is a constantly evolving medium so a constantly changing canvas is nothing new but identifying mobile first challenges within the same canvas as the desktop can be a bit of a learning curve.

#### **CLUTTERING THE SCREEN**

With desktop site's it's been easy to let the content get lost in a sea of secondary information. On some sites content gets shoved to the side to make room for ads, links to other articles, related products, social media integration, or any number of sidebar distractions.

Mobile is a medium of singular focus. This is evident by some of the most popular apps in mobile platforms, such as Twitter or Instagram. The applications put the content front and center and integrate secondary content at intuitive points along the linear content path.

In designing mobile first you have an excuse to strip away these secondary, confusing elements and focus directly on the content. Take advantage of it. Mobile first may as well be synonymous with being "content first" and at its core a "mobile website strategy" is a content strategy. Mobile web design is 90% content design and 10% decorative design, as a result of its size, power, and bandwidth limitations.

#### **LIMITATIONS IN INPUT**

Another major challenge in mobile first design is the limitations in input. In traditional mouse and keyboard computer usage the inputs are very precise tools, a mouse and a keyboard. Touch screens only have one input type, a finger. When the first iPhone came out one of the early complaints was its lack of a physical keyboard. A software based keyboard can be prone to latency issues and for people with larger hands it can lead to a lot of errors while typing. These issues have been largely forgotten because the need for a software keyboard on the device screen offered more benefits than problems.

The lack of a mouse adds a new complication to the mix though, and that is that the finger lacks the precision of a mouse. A mouse can give pinpoint accuracy on a clickable object, interacting with a small button on the page isn't difficult, but a finger has the challenge of obscuring the object that the user is attempting to click. Because of this all critical elements need to be larger in a mobile screen. Now you have the challenge of less screen space and the need for bigger elements. We'll explore a few ways to overcome these issues in this chapter.

---

### Developer's Insight: Coarse or Precise Pointers

In a current draft of upcoming additions to the CSS specifications there is a level four media query for coarse or precise pointers. This would function much like some current media queries, but offer unique styling for pointer types. The media query would classify pointers with limited accuracy, such as touch screens, as coarse and devices with accurate pointing, such as a mouse, as precise.

These media queries would be called as such:

```
@media (pointer:coarse){}
```

Currently there is no adoption for this specifically, but with some luck it might be. Currently developers can identify touch devices singularly by using Modernizr. For more information on how to do this, look to the chapter on Modernizr in the appendix.

---

A lot of what holds designers back from mobile first web design is that we're being hit with a smaller screen and fewer resources first. Although this is true, with a bit of practice it soon becomes second nature.

In the next section, we'll take the prototype and wireframes we built for our responsive site in chapter 01 and start designing mobile first. Because site level header and footers are the primary interface for our site, we'll start there.

---

### It's all in the name: High Fidelity Wireframes

In presenting wireframes to a client, you might find the idea of a prototype to be confusing to a client and other members of a team. One easy way to contextualize prototypes is by referring to them as "high fidelity wireframes". In doing this it can be reassuring and communicates that a rapid prototype is still a rough sketch, even though its written in code and viewable in the browser.

---

## 2.2 *Designing Headers and Footers for Small Screen*

The header in the prototype calls for the name, location, and twitter handle. To the right of that are the navigation elements where users can look for specific content. To display these elements on a smaller screen in a way that provides an optimal user experience we'll look at the off canvas design pattern.

Design patterns are reusable solutions to address a recurring problem. With off canvas navigation we solve the problem of optimizing user experience, making the screen less cluttered, by hiding the elements to the left or right side of the page, and when cued, the

navigation slides out. This is the pattern used by Facebook and Path for hiding the main navigation and allowing the user to focus on the immediately important information.

In this section we'll be designing headers and footers for the small screen, so let's get started.

### 2.2.1 *Creating headers and footers*

We'll start out by designing our header's unaltered state, and give it two buttons that we can later use to cue the off canvas navigation (figure 2.1).



Figure 2.1 An initial design for our header. Notice the two icons in the corners. To the left is one to denote “Information,” and to the right is an icon to denote “Site Navigation.”

We have two simple icons in the header, both about 44x44 pixels, and a logo to reinforce that the visitor is in the right place. To the right, we have a “three bars” navigation icon. This is becoming a standard icon for expanded navigation. On the left is an information icon. We'll be using these two buttons to give our users a way to navigate deeper into our site.

Since we are using “off canvas” navigation, we will need to show what that navigation looks like, so let's get a sense of the spacing, color, and design of the navigation expanded (figure 2.2).

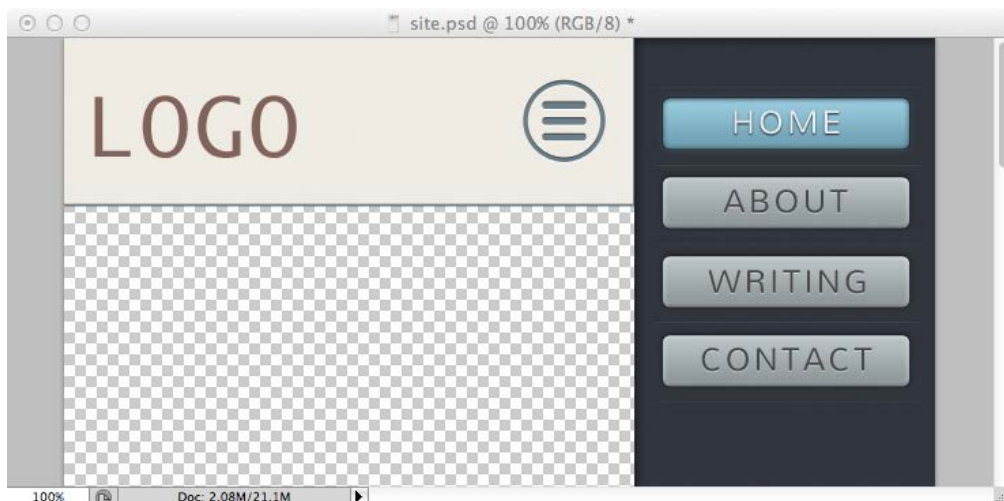


Figure 2.2 The navigation will be expanded on the press of the three bar navigation icon.

Now we have some obvious site navigation. For this site, there are only a few sections to access, but in a large scale site we can scroll down to see more options. We could also potentially slide the header off further to the left and leave more room for the navigation on the right. We can easily apply the same style to the information content on the left side of our page to display some more information (about me, what the visitor can expect on the page (figure 2.3).

### Developer insight: off canvas navigation

Off canvas navigation can be a complicated challenge for developers. There's a few different ways to approach the challenge. One is by simply applying various states as classes on the body element (such as "left-nav-exposed" or "right-nav-exposed") and adding and removing these classes with jQuery. This is a relatively simple way of accomplishing this task, but we'll cover some more in-depth ways to approach it later.

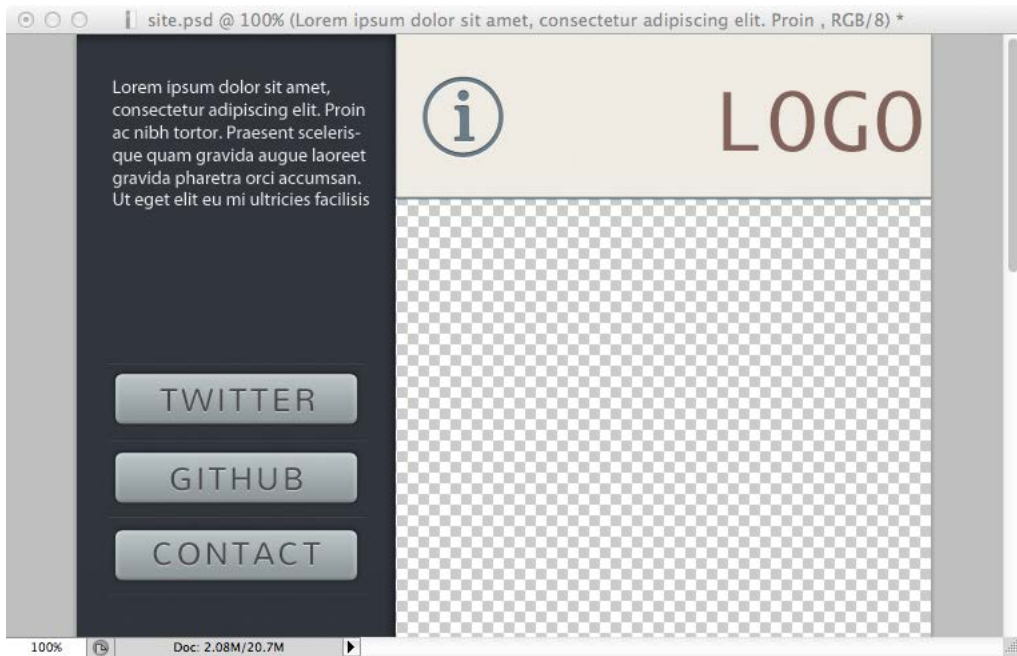


Figure 2.3 On the opposite side, we have more information. We repeat a few of the links from the other navigation, which is fine since these links are to further inform the user about the subject of the site.

## 2.3 *Designing for a touch interface*

As I mentioned earlier, a 44x44 pixel size should be minimum for any clickable target. This is because the user will be engaging the site with a thumb or forefinger, and obscuring the link or button they are trying to tap on. In a desktop browser, the mouse or trackpad is usually the exclusive direct interface. Think about when you use a small computer in your living room for watching television. It's probably rare that you need the keyboard, because most of your browsing is casual and involves clicking on links to cue up videos on a streaming service, or browsing through links on a news site.

On touch screens, you end up obscuring the screen in order to explore it. That's why it's crucial that at all times your user has a safe area to tap and drag, so they can explore the site, and the featured content is large and in the front and center. This way if a user is reading an article or browsing links, the flow of content isn't interrupted by the need to navigate. You can flow through the website like a drop of water.

---

### Developer's insight: tap as a hover

For years, hovering over an element has been a way of interacting with a page. Users could hover over a page element with their mouse to reveal supplemental content, such as navigation elements or secondary information. In theory a tap would register as a click, rendering navigation hidden with a hover state unusable.

Fortunately, most mobile browsers have resolved this issue by requiring a second tap on links or elements with a hover state. This double tapping on mobile browser is an effective way of ensuring users can navigate websites, but it's important when developing responsive sites that you keep this feature in mind.

---

In designing our body content for our website, we want to keep this in mind. We want to make sure to use a font that is easily legible on a digital screen, and we want to be careful not to crowd the screen with links. Large text in a mobile environment is great because it ensures that the text gives the user plenty of safe area, and it allows them to comfortably read the page without having to hold the phone too close to their face.

Another great option is subtle background patterns, as opposed to large images, which will slow page load. While it's easy to believe that users want the same things on desktops as they do on small screen devices, it's also important not to ignore the importance of implied context by a smaller viewport.

---

### Developer's Insight: Patterns and Base 64 encoding

Later in this book we discuss methods of improving site performance. One of those ways is to use Base 64 encoding on images. When implementing small, subtle patterns, try using the tips on base 64 encoding directly in the CSS to avoid having to make an additional server request for additional assets. If done properly the addition of the image data directly to the CSS should have a net zero total page size, but will improve site load slightly. We'll cover this topic in greater detail later in the book.

---

#### 2.3.1 *The simplified small screen grid*

For smaller screens, we need a simplified grid. In chapter one I mentioned braking sites into small grids. I prefer to break things into quarters or thirds. Historically, grids have included up to 12 columns, which works in a desktop environment where you have a big width to cover, but on mobile anything smaller than that starts to be unmanageable and too tight for screens smaller than 320 pixels wide.



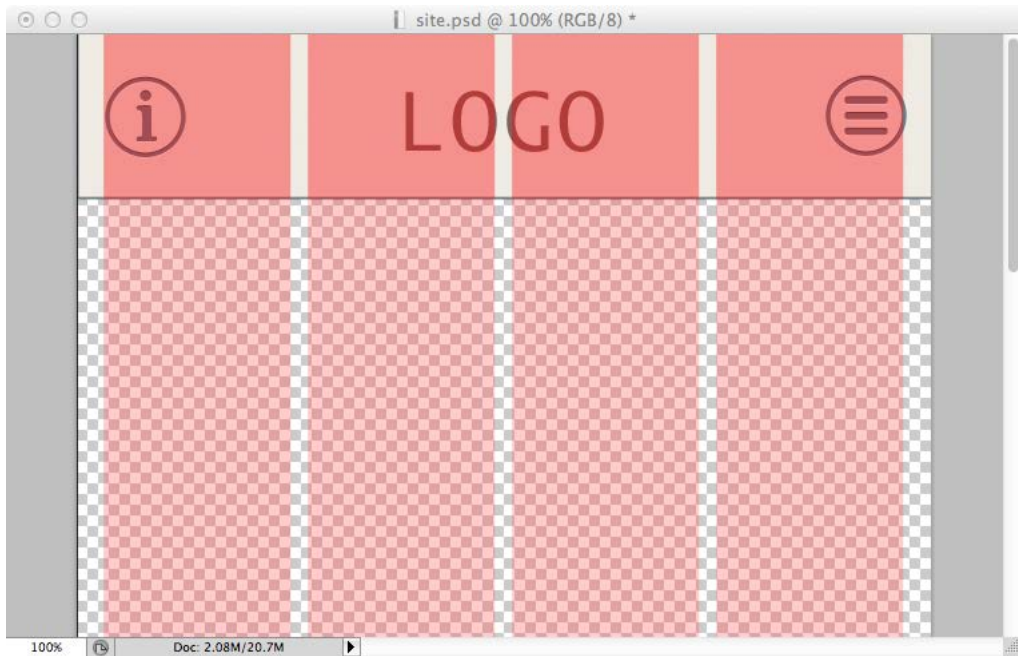


Figure 2.4 The grid in place

When working in a grid, it's important to keep in mind how the site elements will scale and float. It's best to think of the responsive grid as going from left to right, then top to bottom. This helps to create a clear and consistent flow of information.

On a small screen, you might have a one column grid with four blocks in a small-screen viewport. The four blocks stack on top of each other in the mobile browser.

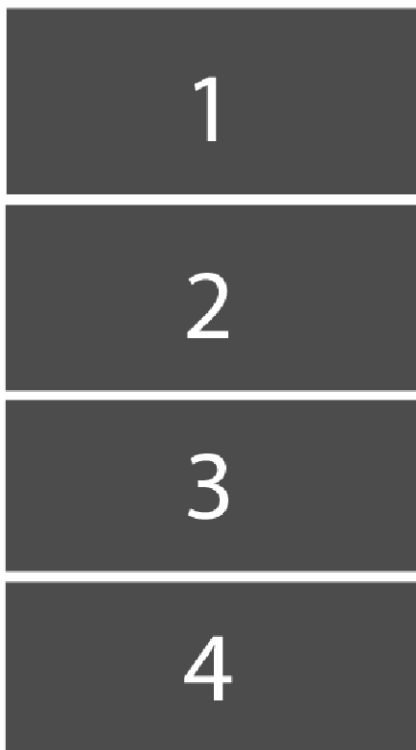


Figure 2.5 Four elements stacking on a small screen

This vertical stacking encourages the user to scroll, and prevents any content from cluttering up the page. As the viewport expands, there is more room for this content and the grid can change.



Figure 2.6 The vertically stacked elements are able to sort into two columns in a mid-sized screen.

As more screen real estate becomes available, the elements can adjust their placements and separate into two columns. The content occupies relatively similar volume; it's only the placement that's changed.

This can be extrapolated further as the viewport gets wider and the elements have more room to fill.

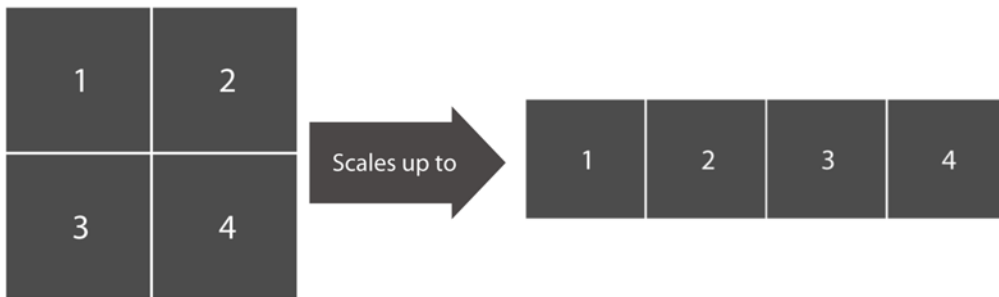


Figure 2.7 The same grid again, now for larger screens

The new grid works wonderfully for moving from a two column grid with four blocks, and it scales up nicely to four columns with four blocks. As a designer, you should always anticipate the grid flowing from top to bottom and right to left.

This logic can apply to all elements on the page, and grids can even be nested inside grid blocks. It's important to understand and acknowledge this early in the design phase so you can create sites that take advantage of the way CSS natively works. This is another reason why prototypes are crucial to responsive design, and why designing in the browser is ultimately the best way to go.

## 2.4 Designing content for a small screen

Now that we have our grid and navigation, let's give the site its content. We first want to ensure that there is a background that will contrast the body copy. The content of the page is the reason the user is viewing the site, so we want to give it priority, and make the site as easy to read as possible.

In figure 2.8, I'm using a full width image to enhance the article and set a tone for the article. The full width will be easy to manipulate later, but it also just looks nice.

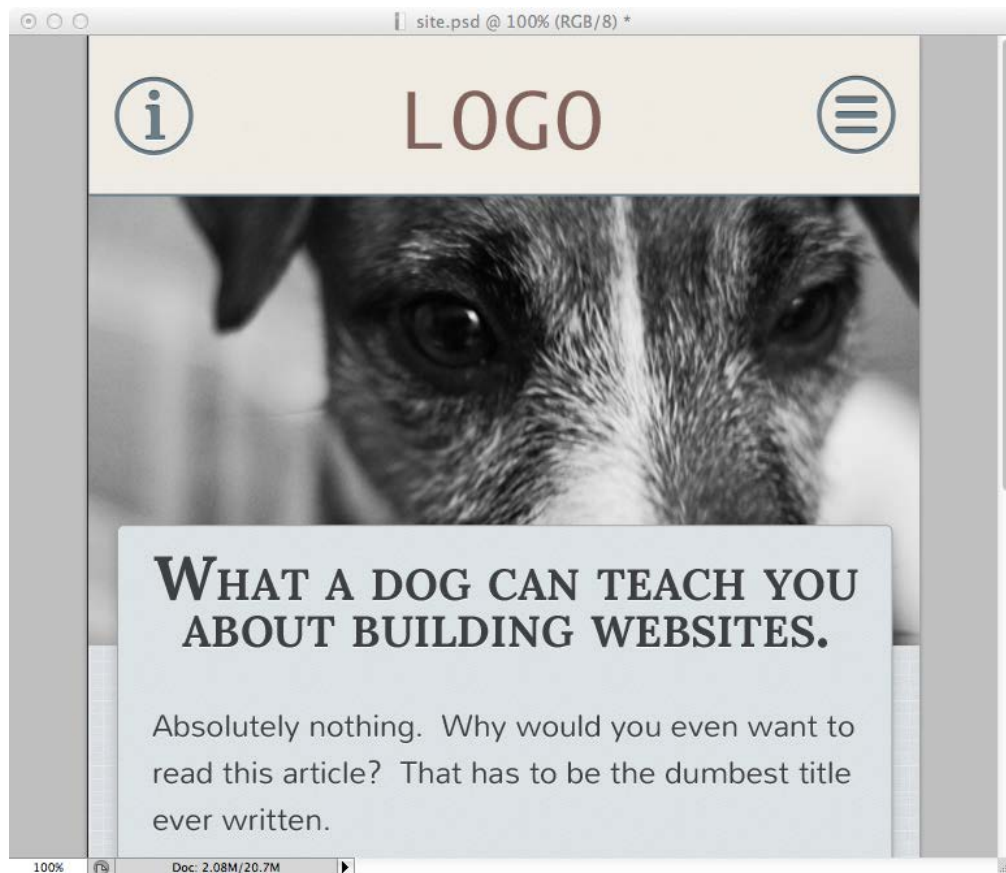


Figure 2.8 Our designed mobile site

We can also see how the design will work with the off canvas navigation deployed. We want to make sure to not design these elements in a silo, because we need everything to play nicely together and create an atmosphere for the user.

Designing for content is tough, and one of the biggest reasons that I advocate for the use of prototypes. By prototyping we can gather content and build a site that meets the client's needs, instead of forcing the client's needs to fit our design.

### 2.4.1 *Using web fonts in layouts*

With small screen design, much of the screen real estate is occupied by type. One of the best ways to give a site a unique look and feel is with web fonts. Using CSS, we can import web fonts into a style sheet and use them in our document. The fonts are hosted on the site's server, just like an HTML file or an image, and loaded into the document. Using web fonts can also bog down load time, but it's an efficient and simple way to add a lot of character to a site.

Historically, designers were limited to using fonts that they could assume visitors had installed on their computers. There were only a handful of typefaces that a designer could confidently design around. Any custom font that was required in a site design had to be implemented with image replacement, which was a huge burden and made editing and updating sites troublesome and time consuming. The use of web fonts is more limiting than in print creative, because you need to have licensed fonts for use online. Font foundries have to make the typeface available for use online, or the site's owners can be fined.

However, there are several services available to designers to help them use fonts online, most notably TypeKit, fonts.com, and Google fonts.

#### **TYPEKIT**

Typekit was one of the first major font hosting services available. They were bought by Adobe recently, and as a result their library has dramatically expanded. Typekit offers a great library and offers reliable cross-browser consistency and the ability to host custom fonts. The downside is that you can't use the fonts locally, and there is a small cost.

#### **Advantages:**

- High quality and highly optimized fonts.
- Fast to render on the page.
- Affordable plans.

#### **Disadvantages:**

- Lacks some of the most popular fonts.

#### **Fonts.com**

Fonts.com offers a service similar to Typekit and has exclusive rights to several popular font families. The biggest benefit of fonts.com is that it allows you to download fonts for use in comps, which is nice if you absolutely have to rely on comps. You can also host your own fonts with a fonts.com subscription, a feature that is unavailable through Typekit. Fonts.com also offers the ability to self typefaces with certain packages. There is also a cost for using fonts.com.

#### **Advantages:**

#### **Disadvantages:**

- Has exclusive rights to some very popular font families.
- Option to download fonts for local use.
- Self hosting is available.
- Lacks the speed and reliability of Typekit.

### GOOGLE WEB FONTS

Google also hosts a web font service, which is free to use and allows you to download the fonts for use locally. Unfortunately the selection through Google is limited, and the quality of fonts available isn't as high as the subscription sites.

#### Advantages:

- Free,
- All fonts are available for download.
- Self hosting is available.

#### Disadvantages:

- Limited Selection.
- Fonts are generally of a lower quality.

In addition to web font services, there is also the option of self hosting fonts on your own server. This works well if you can acquire and maintain the rights to a font, but purchasing the rights to a web font can be very expensive and require a secured hosting environment.

### SELF HOSTING

With self hosting, there's the issue of cost but with it comes the flexibility to use the exact font you want. With a service, you are limited to the number of fonts they have available and sometimes there's a great reason to use a specific. Once you have the client on board to cover the cost there are a few tricks involved in self hosting. You need to be sure you have the bandwidth to serve the typefaces, after that you can embed them into your CSS and you're ready to go. The files themselves need to be optimized for web use as well. This means the type needs to be converted into three different formats (WOFF, TrueType, and EOT) in order to cover the preferred formats by the major browsers. Conversion can compromise the quality of the fonts as well. Ultimately, self hosting is a much more difficult technical hurdle than you might assume and can cause some design issues.

#### Advantages:

- High level of control.
- Not limited to available libraries.

#### Disadvantages:

- Often fonts are not optimized for web performance, so there are issues with aliasing and artifacting.
- Can include a very expensive one time or recurring fee.

## 2.5 Summary

In this chapter we discussed the beginning phases of designing a site, mobile-first. The role of a designer, strictly speaking, can be the most difficult position on a project, because as a

designer you have to communicate a brand into an artistic medium that you might not have the highest degree of control over. This is the main reason why a designer with little or no experience in CSS is at a huge disadvantage. Web design will always feel like a limiting and confining medium if you don't take the time to learn the details, and ultimately that means designing with CSS.

Now that we've created a small screen site, we need to expand it to mid-sized and large screens. Remember, the goal of responsive design is to not create comps for every single view and every single page. To avoid doing this, in the next chapter we'll learn how to create and communicate design using a style guide.

# 3

## *Using style tiles to communicate design*

This chapter covers

- What a style guide is and why it's important
- The importance of meaningful client deliverables
- Introduction to Style Tiles
- Building a Style Tile

Interior designers use swatches and fabric samples to create a palette to work from. In interior design this is an important part of the process, because painting and furnishing a room is costly in material and labor, so setting color and style ahead of time can help to alleviate those expenses.

Style guides function in much the same way, offering clients a palette and theme before the hard work of drafting the front-end of a website begins. This is important in responsive design because the website requires a lot of moving parts. Every design element needs to be scalable, flexible, and natively built in CSS, in order to maintain a small load burden and efficient architecture.

In order to bridge the gap between mobile, tablet, and desktop sites, we need to visualize the design of a site without implying dimensions, size, or format. Style guides give us the ability to abstract our design and break it into manageable chunks.

In this chapter we'll learn how to articulate the parts most important to a design and show how you can establish a creative visual style for a website without forming the entire sites execution in a graphics editing program. This will let you take a collaborative approach to the actual implementation and create a more vivid and lively website. In the end by breaking from



traditional ways of creating a visual language for a site's design, you open up a design to the fluidity required in responsive design.

### **3.1 Visualizing design with style guides**

When visualizing the design in traditional web development, working with comprehensive layouts (comps) was the norm. As discussed in Chapter 1, comps are static images used to represent a single state of the final coded site. The term, comp, much like the deliverable itself, is a leftover from the days of print advertising.

The problem with designing in fully laid out comps is that it assumes too much and fails to communicate the scalability required in responsive web design. Comps are rigid and only represent a single state in a site's design. They are also costly to produce and adjust in responsive design. In addition they present a false sense of security in the site designs effectiveness. They portray to a client a sort of "best case scenario" that rarely reflects the actual use of the site. In a comp all the content is controlled, but when a site is published and the user or client begins inserting their own content, a lot of times the visual design suffers.

Say for instance a comp is designed for a list of recipes. Once a client has the keys to the site turned over to them maybe they start adding or removing content, so a design that called for 4 recipes to be displayed on a page now has 8. Or perhaps there's a recipe title that is 80 characters long and suddenly the client has uploaded a title that is 200 characters long and it causes the text to overlap. There are a myriad of ways a developer can fix these problems, but wouldn't it be more effective if we changed the way we worked so our site's had the fluidity they needed to accommodate these sorts of problems before they arise?

In a sense the responsive web is about building for these sorts of factors as well. It's about building a web that is versatile, agile, and driven by purpose. A purpose of a site is the content it contains, and a well-designed site is one that beautifully contextualizes and provides accessibility to that content. That's why style guides are so important. They allow you to talk about the visualization of the site without having to solve the problem of contextualizing the content at the same time. How many times have you been designing a page and had a client change their mind over what certain navigation items should be called? Those problems shouldn't be solved when you're trying to establish a visual style for a site, they can be more easily solved with context.

#### **3.1.1 What is a style guide?**

The term "style guide" is used to communicate the design standards in place for the site being developed. It needs to communicate layout, branding, typography, color, and navigation to the team. By giving the actual design a starting point and by collaborating closely with developers, designers can create a high quality responsive site quickly and easily.

Most front-end developers spend the majority of their time looking for patterns to increase efficiency. If a developer can become skilled at finding and capitalizing off of visual patterns, he or she can dramatically reduce the time it takes to build a site. A style guide is a type of deliverable that gives developers these patterns in no uncertain terms.

When I was a kid, I loved Highlights magazine. It was a little paper bi-monthly magazine that had little cartoons and games in it. One of the games was called “Spot the Difference”. You would be presented with two different drawings and be asked to identify the difference between them. This is a lot of what the experience of being a front end developer becomes. Spot the difference between comps, or between the site and the comp. By giving a sort of bible to follow, designers can give developers a proper tool to execute a site.

Let’s look at what makes up a style guide.

#### **LAYOUT**

When drafting a style guide, you want to make sure to specify spacing with a grid system. The grid may need to be altered for small and mid sized screens, but it should give a general sense of proportion and spacing. Defining a layout structure lets the developer know that grid blocks should have a certain width, padding, and margin.

#### **BRANDING, PATTERNS AND COLOR**

Your style guide should say something about the client’s online brand. This is more to show patterns and themes than anything else. Setting a color palette gives the team certain colors to work with, but more importantly it gives colors for a client to comment on, as opposed to abstract combinations of colors and textures throughout the design. This leads to more meaningful feedback.

#### **TYPOGRAPHY**

The typefaces used on a site are key to the overall site design. You’ll want to indicate all typographic decisions as clearly as possible. Font sizes are relative, but font faces, weights, and styles are important information to communicate up front. Most clients have an established typeface, which is important to specify, but there are also other typefaces that may fit within the project well. It’s also important to specify typographic decisions such as how large base header font sizes or paragraph text should be and any inline text styling such as italics or inline link styles.

#### **NAVIGATION**

Navigation applies specifically to user interface patterns and shapes. This means designing a button, link styling, and iconography. Again, by pulling the design for these individual elements outside of a specific use case, it opens the design up for consistency and lets you focus on the element as opposed to the whole picture.

Most of the time a style guide can be reverse engineered when a site is being developed, which can be a bit backwards. It’s much like building a house so you can draft a blueprint. While handing over a direct style guide will empower developers to do their best work, a framework for building style guides has also emerged in recent years. It’s a sort of starting off point for designers to hand over to clients and developers so that the entire team can have a meaningful conversation about a site’s visual design. Where as a mood board, or collection of images might be too abstract and a full page comp might be too absolute, Style Tiles gives just enough information without restricting a project.

### 3.1.2 *Style Tiles: Creating A Visual Language*

Samantha Warren, the creator of Style Tiles, was in search of a way to provide her clients with a better deliverable. Traditional web design is done in Photoshop and can be, at times, an unsatisfying experience.

The problem with comps is that they tell too much about how a site is supposed to look, while not actually doing much to satisfy some of the biggest parts of a design. As important as what a site looks like is what it does, how it loads, what the experience feels like. To design a comp and then reproduce it is to apply your creativity to what I call a “dead end deliverable”. A comp has no use once it’s been sliced into pieces and put back together by a developer. Instead, a Style Tile can become a target, a set of visual rules that define how a site should look, feel, and interact.

With responsive design, a site can be built with meaningful deliverables. A rapid prototype guides a site’s backend structure. A style tiles influences front end design and can be used to offer design guidance as a site expands.

In her article in *A List Apart*, “Style Tiles and How They Work,” Warren says that by relying on Style Tiles to inform design she’s able to have meaningful conversation with stakeholders about a site’s visual elements earlier, without having to flesh out the entire layout of a site. This provokes thought and dialogue about what the client likes and how the client envisions the site looking. The point is to achieve the results quickly without investing a large amount of time and energy too early in the process. Figure 3.1 is an example of what a Style Tile could look like.

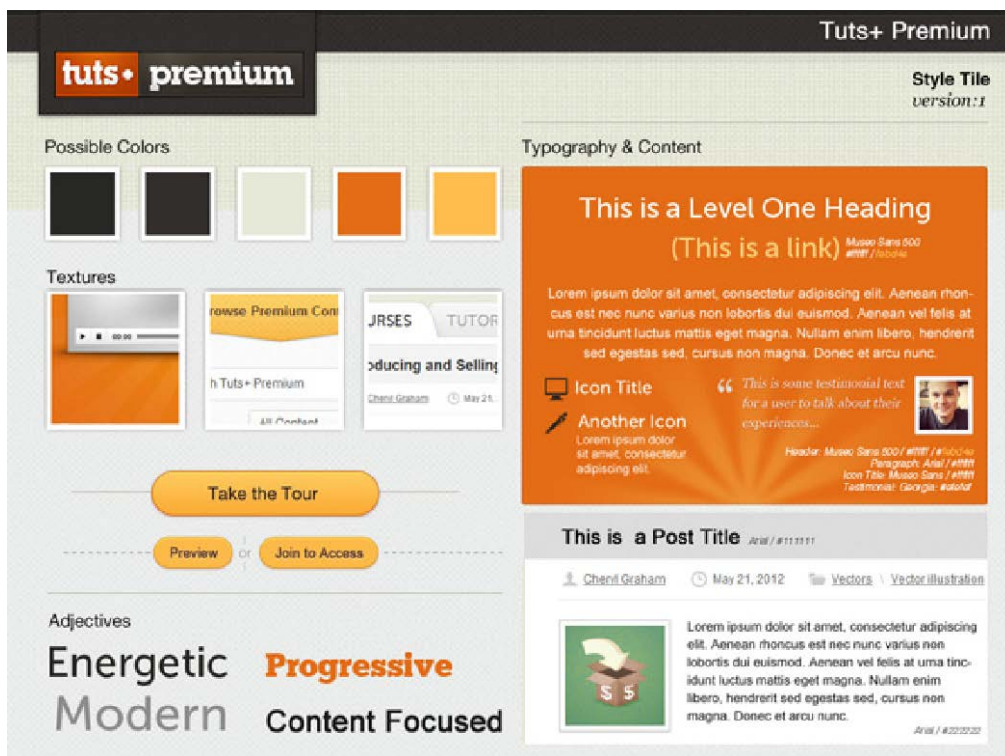


Figure 3.1 An example of a Style Tile for the tuts+ website

### 3.2 How to create a style tile

Using style tiles as an early deliverable requires a huge amount of adaptation from every member of the teams. It's not a burden that falls alone on the shoulders of the designer, but rather it requires the entire team to shift the way it discusses the project and the projects visual identity.

Steve Jobs once said of Apple's design process that design isn't just how a thing looks, but how it feels. In order for internal teams and clients to find success is something like a style tile, all the stakeholders need to understand that you are ultimately building towards the goal of how a product is going to feel.

So how do you build to something that conveys not just a look, but also a feeling? It starts by focusing in on your client and having meaningful conversations with them. You first need to get feedback and learn from your client. Next you create a pallet based on the feedback you received. This pallet includes not just colors and typefaces, but adjectives and emotive terms to describe the projected end product.

### 3.2.1 *Get Feedback*

Before starting on a style tile, it's crucial to get feedback on how a client defines their visual brand. Much of the design work with a style tile depends on feedback and direct interaction with the client. This might begin as a mood board or a collection of images that convey the brand's mood in the browser, but ultimately these mood boards are too vague to be used as a deliverable. Start with a series of questions to get at what your client is looking for as part of a design kick off survey. Some good examples of questions would be:

- If your site was a cola, what would it be? (For example: Coca-Cola Classic, Mountain Dew, Dr. Pepper?)
- Is your site modern or traditional?
- If your site was a city, what city would it be?

Ideally, you would want to collect a series of adjectives to describe the site design. In our example site if I was the client how would I describe what I want out of my site? Some adjectives I might use are "Rugged," "Industrial," "Artisan," or "Hand-Crafted." I would say that if my site was a beer, it would be Shiner Bock, because I want it to have a vintage, unique feel, but still appeal to a large audience. I would probably describe it as simple and modern, but also somewhat rustic.

Once you know the direction your clients wants to go it's on to the pallet.

### 3.2.2 *Design the style tile*

As with all frameworks, the format for designing style tiles is intended as a starting point. The base file can be downloaded from <http://styletil.es>, but can seem a bit bland (figure 3.2).



Figure 3.2 The base style tile, downloaded from [styletil.es](http://styletil.es)

This base gives us a roadmap to the asset we will be creating. In it we have spaces for colors, textures, buttons, some type treatment, and a set of adjectives to describe the online brand of the client. There's also space to define a logo treatment, and document the name and version of the creative being worked on.

You'll also notice the cyan vertical rules in the template. This is for defining the site's grid. The default is a 16 column grid, but feel free to adjust it to fit your site's needs. In the case of our running demo, we'll be designing a blog. Because of this, it's going to be a type-heavy site, so 16 columns might be too fine of a standard. I'm going to adjust this to be a little simpler, and work in an eight column grid (figure 3.3).



Figure 3.3 An eight column grid, using a semi-transparent red layer.

This layer is simply to give a sense of the basic grid layout and won't directly translate into the creative work being done within the style tile. It's most useful as a guide for the grid system in development. Most of the work being done here is to offer talking points and direction, so keep that in mind.

In the previous chapter, we walked through the design of a mobile site. In doing so, we made a few creative decisions that we'll want reflected in this style tile. With our grid in place, we can start adding some of our already defined assets, such as our logo and some of the colors and typefaces. This is as simple as taking the existing swatches and placeholder text in the template file you downloaded from [styletil.es](http://www.styletil.es) and changing the swatches and default typefaces to ones that are brand appropriate and reflect the creative direction of the site.

In this format you are free to conceptualize the brand identity and interactive elements without having to create layout and infrastructure at the same time. This produces a focus on the important parts of design and leads to better feedback from clients.



### 3.2.3 Creating the Style Tile

Textures and patterns are nice to include in a style tile. Texture can be a useful design tool in separating levels of content or separating elements on a common plane. It's not a must have, especially with flat design becoming more popular, but it's something that I like to plan with in some of my designs.

Say for instance you have a site with a main content area and within that sections of callouts or elements that tangent on the central contents focus. Having a texture available gives you a plane to distinguish between these elements. You also might want a texture to fill your site's negative space and add more personality to the site.

In the below example I've added some texture, selected a typeface, added accent images, buttons, and some adjectives to describe the site

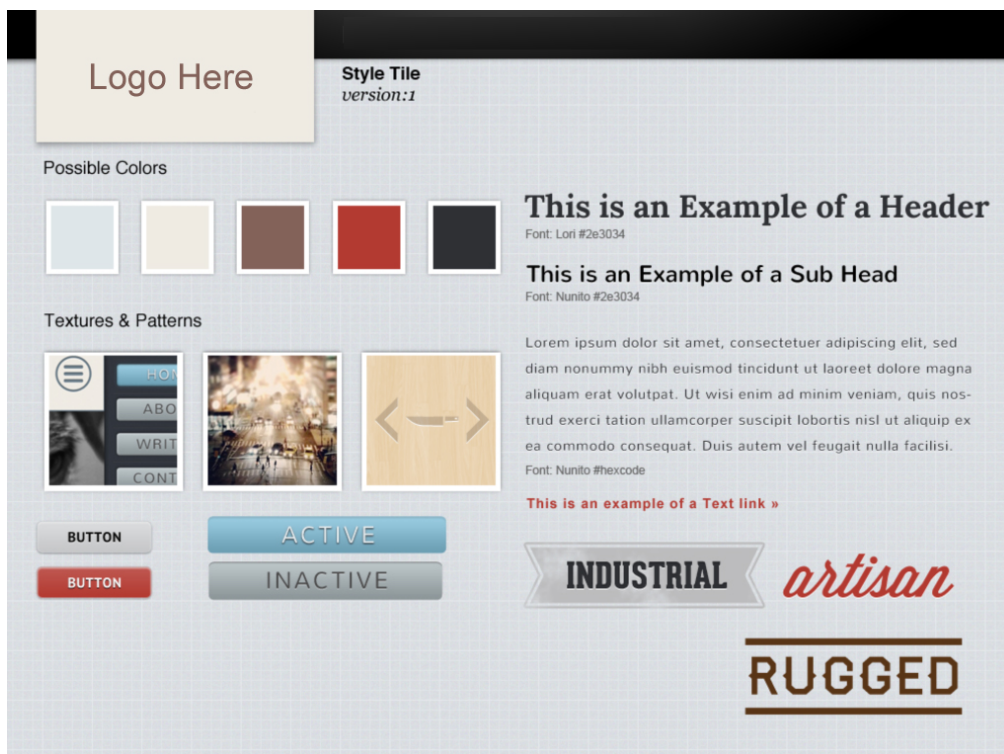


Figure 3.5 Here we have an early initial pass at a style tile.

In order to make a style tile that speaks to the design of an end site we want to add some important elements. You should feel free to use your creativity to make the style tile something unique to your project, but I like to add a few layers of design.



## TYPOGRAPHY

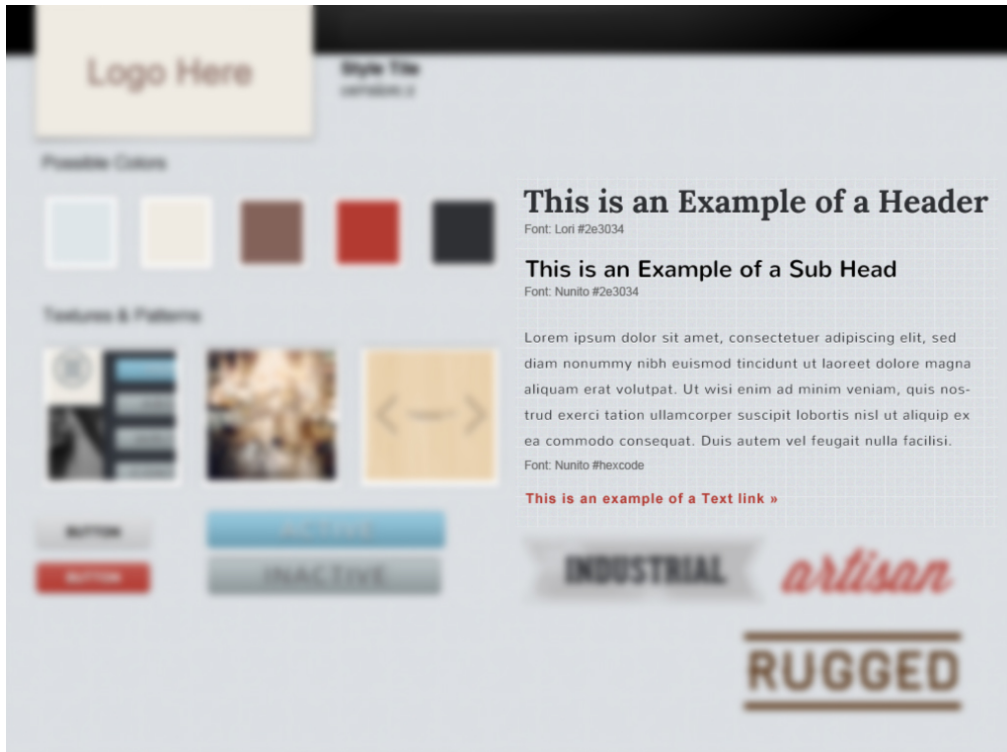


Figure 3.6 The typography portion of our style tile.

Adding type to the left side gives us a place to identify typography style. Font sizes, colors, weights, leading and kerning can all be played with here to get a nice balance and articulate an idea.

## COLORS

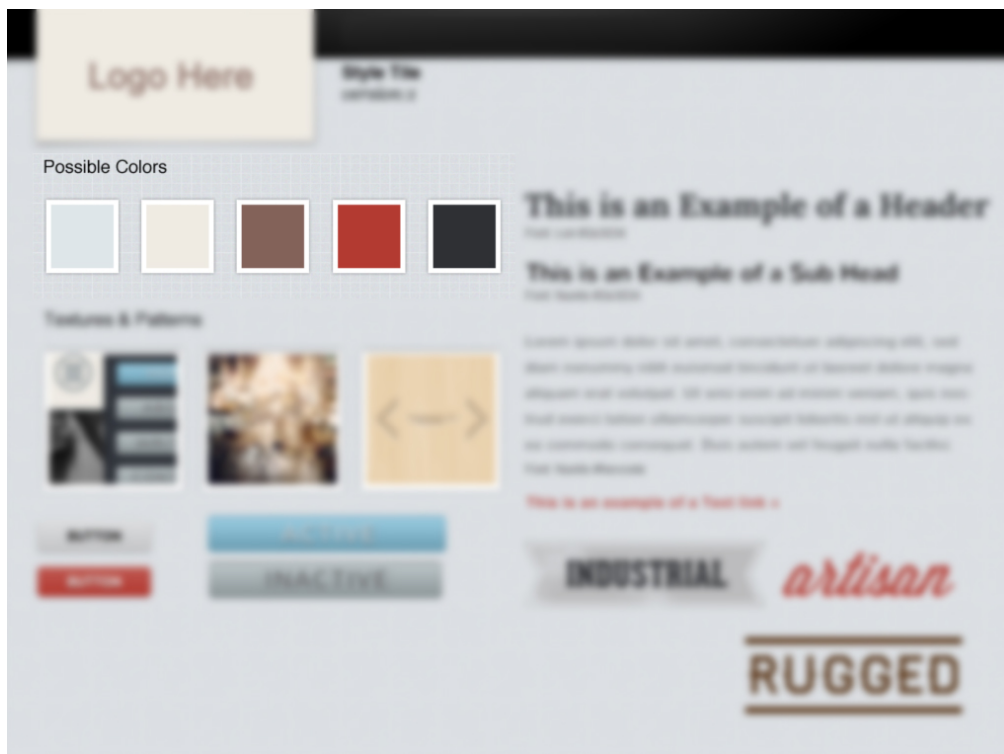


Figure 3.7 An example color swatch

Here we have a small color palette that can be used in the site. These colors can be for text, call to action buttons, divider lines, accent points, and things like that. It's a base of colors for use in the site design. By identifying the colors outside of the comps, you can better ensure consistency and find a unified palette. A site's color palette can become fractured if its reverse engineered out of a comp, but with a style tile all colors are established before applied to the final site.

## TEXTURES AND PATTERNS

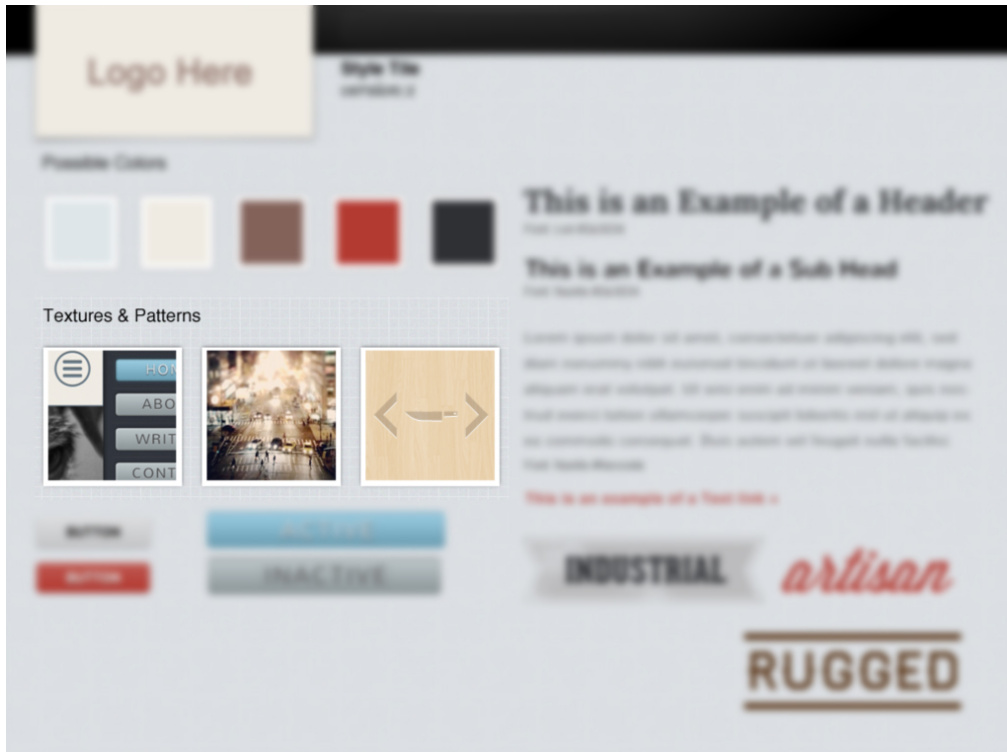
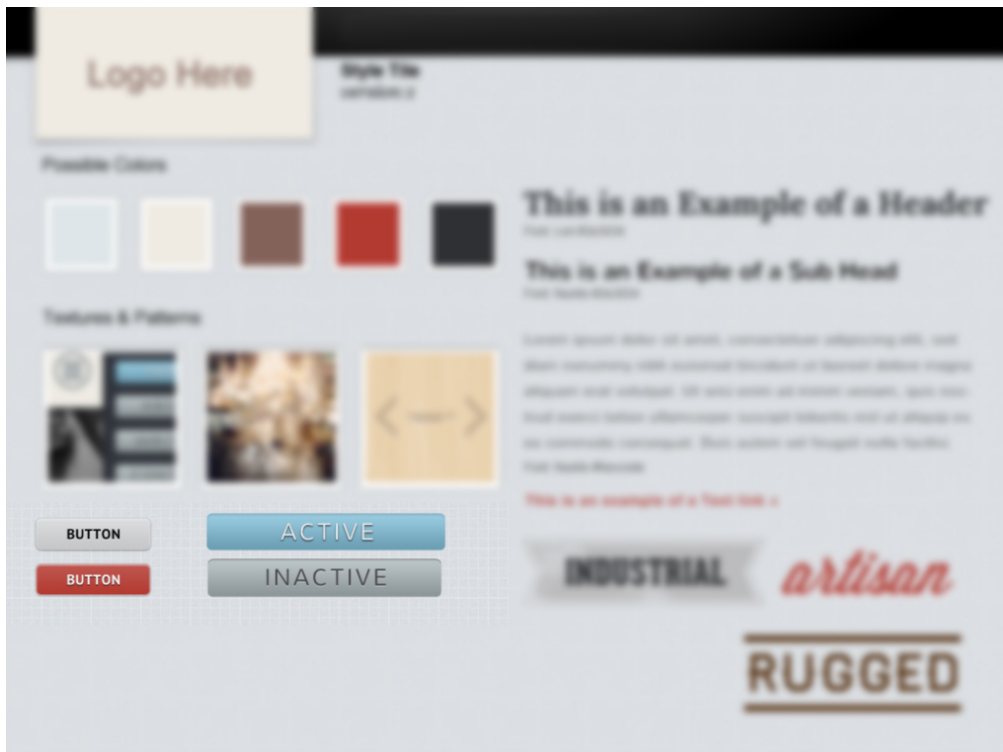


Figure 3.8 The “textures” here are little artifacts that capture some of look and feel we want in the final product

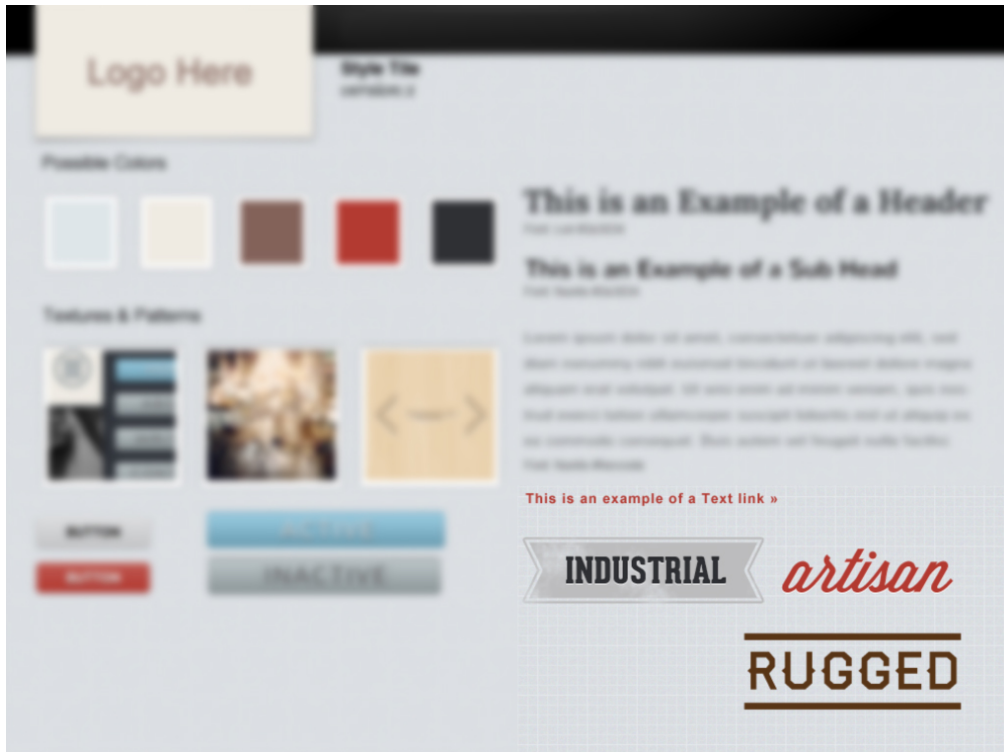
With textures, you’re able to create a swatch of some media styling. This can be patterns you want in the site, or a mood board of images and their treatments. Here you are free to open up some free form concepts that should end up in the final site. For me it’s more of a mood board than anything else. The treatment of the images would translate into the final site, and the images themselves would inform the look and feel of the media used on the site.

## BUTTONS



For the buttons, we want to design something that looks clickable and follows some accepted user interface standards. Your buttons don't always have to look glossy and rounded; what's important is that they contrast the background and are obviously buttons.

## ADJECTIVES



The adjectives section should be more or less a playground for ideas. Use a few words to describe the visual brand and some typefaces to accentuate those points. The type and colors don't necessarily have to be implemented in the final design; they are more or less to help articulate the end goals of the site to the client.

Now we have a base to start with in our style tile. Keep in mind that the style tile is being built as a deliverable along side the prototype. This reduces the need to create what I call "snapshot comps," which are only useful in showing one website state at a time.

### 3.2.4 Iterative design with a style tile

Another great thing about style tiles is that they are easy to integrate over. You can start with multiple style tiles each with unique design directions and easily combine, revise and add to them to create the final look and feel of the site.

Because they focus on the parts of the design that matter they are quickly produced, so a few designers can each work on different directions independently and each direction could reflect a different treatment of the user interface elements, mood boards, or colors. Each tile

reflects a different impression of the client's brand and each one could lead to different and fruitful conversations.

In rounds of revisions and as you process though the final design direction, a style tile can be improved upon to create the final design. I would definitely set a limit to the amount of iterations though, since criticism is very easy to give to visual elements and ultimately the design will be integrated into a prototype that might cause the client and design team to want to rethink some of the decisions.

### **3.3    *The death of the mockup***

Within the industry, there has been a lot of talk about what responsive web design does to creating Photoshop comps or mockups. If the creation of a mockup for every viewport is too labor intensive, and we use a style tile to identify visual style, where does this leave the traditional mock up?

This is a hard question to answer, and one I struggle with on a daily basis. With some, traditional clients, mockups are a requirement. They are afraid of doing anything related to “coding” before getting a feeling of familiarity with what they are buying. Many clients find technology intimidating, but when they get a PDF full of JPEGs, they feel like they can understand that easier. Other clients want everything in the browser as soon as possible. A delicate balance is required.

Most of the problems on the client's side boils down to context. Even traditional clients can be won over to the side of browser-based prototypes if given the proper context. It's all about using familiar terms to describe new ideas. For instance, I like to refer to rapid prototypes without any visual style as “high fidelity prototypes” or “browser prototypes”. I explain to the client that we want them to engage the site concept in the browser so we can get accurate feedback from the way they interact with it. Since everyone interacts with their devices a little differently, this is a great way to discover how your client interacts with their sites.

Mockups or comps themselves are a poor deliverable to development team and force a number of poor decision on the client and designers. The truth is that there is no true way to simulate a responsive environment (currently) with graphic design tools. The biggest problem is that touch screen environment is deeply personal, and touch interactions must be very sensitive in order to feel natural. In some projects you could spend hours trying to figure out minor problems like how a swipe should behave or how an object should follow a touch.

Something as simple as binding an interaction to the start or the end of a touch can make or break the way an interaction feels. Unfortunately it's not possible to simulate that in a static design.

But what does this mean to designers? Should every designer learn to code? No. Coding and designing are two different things all together and there should be a division of labor between code and comp, if only because it's more economically viable. Having an entire site built by one person in both design and code is just too subjective and can get risky.

What works best is to collaborate closely with developers and designers. On a current project I sit next to the designer and we offer each other feed back as we create elements. He'll mockup a module, or component for the site, and I'll build it into the design system. Now I build everything modularly and as the site is built we encourage discovery of how the modules work together.

So, is the mockup dead? Yes and no. I think clients and design still need some degree of mockups, but ultimately how a site feels in the browser needs to supersede everything. A mockup can serve as a good target and a style tile is great for filling in gaps. I would just advise teams to not spend too much time building mockup to figure out a site's details in the browser. Homepage or critical page mockups can be helpful, but until you've interacted with the site, you'll be exploring every element of the design.

### **3.4 Summary**

Style guide based design is a completely different approach to traditional web design. As we've been discussing up to this point, responsive web design makes us rethink a lot of the ways we go about building and designing websites. By abstracting the visual design from the layout and function of a site, we can produce a more manicured deliverable that focuses on the immediate needs of a client. It also offers us a reprieve from repeating the same elements over and over again through various compositions, and having to maintain consistency between them. By using a style tile, you focus on what's most important to you as a designer: the design.

In the next chapter we'll discuss some design patterns to accomplish simple navigation struggles in responsive environments.

# 4

## *Responsive user experience design patterns*

This chapter covers

- The origins and importance of design patterns
- Using two different design patterns to solve the same problem
- Two popular navigation design patterns

The architect Christopher Alexandra is famous for theories on the use of patterns in design, and his work helped inform early software developers in the 60's on the benefits of establishing design patterns. In architecture, a design pattern is a way of documenting reoccurring problems and their solutions. Similarly, in responsive design, design patterns help us avoid mistakes we've made in the past and establish familiar solutions to the repetitive problems of web development.

It would be impossible to list and create examples for every responsive design pattern there is, primarily because these solutions are still being discovered, but also because it would be an entire book unto itself. Instead, I've opted to show two design patterns for navigation, specifically to show how multiple solutions can achieve similar results. I'm focusing on navigation since we've already established a navigation design pattern in the previous chapter, and also because it is a complex problem with multiple solutions.

So before we go forward with building our site in HTML and CSS, which we will begin in chapter 7, it's important for us to explore the flexibility of the web and find an accessible solution to the specific challenges raised by our site.



---

### Designer Insight: user experience and site navigation

This chapter covers some elements directly related to the user experience of a site. In some workplaces a lot of this would give consideration to having somebody that specializes in user experience design. No matter what, though, user experience is ultimately the responsibility of everyone involved in the project.

Navigation design is a very broad subject, and though we present a few ways to approach it in a responsive site in the following chapters, educating yourself on the discipline of user experience is a good idea. A great starting resource is a book called “Don’t Make Me Think,” by Steve King.

---



---

### Developer Insight: Why build from scratch

Earlier I used Foundation to build a navigation prototype, but here we are building one from scratch. The reason for this is simple, every site is unique.

When I was a kid in shop class my teacher taught me to use every saw in the room. Each saw had it’s own uses and added it’s own flair to a cut. He taught me that while you could make a table with a band saw, you can only make one kind of table with any one kind of saw. By learning my tools and their uses I expanded my ability to create.

It would be easy to use a framework and make it work for the design I’ve been given, and I’ve had to do that on multiple projects, mostly when time is an issue, but in most of my work clients have a unique problem or need and they are coming to me to solve it. This doesn’t leave much room for using frameworks.

---

## 4.1 Single level navigation

Typically, site navigation is done through a series of links or buttons that sit at the very top of a web page (figure 4.2). Top level navigation commonly serves as a portal throughout the site map. The below shows a single level top-level navigation



Figure 4.1 An example of a site's top level navigation

Single level refers to the fact that there is no hidden or secondary navigation, there is only one level of navigation: Home, Products, and Contact

The issue we're faced with here is displaying the content in a mobile and tablet view. The problem with this in a responsive site is that the links and logos could become broken up and jumbled. In Chapter 2 we talked about off-canvas navigation to solve this sort of problem, but that isn't the only solution. Off-canvas navigation is a very common pattern and gives us some flexibility in adding layers to our horizontal spacing, but it can also be limiting and hiding all of the navigation for a site behind a toggle button introduces a single point of failure.

### **Designer insight: taking advantage of vertical space**

One of the obvious reasons for using off-canvas navigation is to have some additional functionality to the very top of a page. This necessity is birthed by the lack of horizontal space in a mobile environment. This is one of the defining characteristics of small screens, to the point that as in development we commonly set media queries strictly against the site's width.

There's an old publishing term that used to get thrown around a lot in web development called "above the fold". It's a reference to newspapers, where the top story and most expensive ad space is above the fold in the front page, so if your article was really good or an advertiser paid the right price their content would be "above the fold".

In web design this manifested itself as being the space on a page that is visible without scrolling. This screen real estate has been prized and led to the trend of assigning this screen space to a carousel module, so that more content could claim to be above this metaphorical fold. Unfortunately, the fold is a myth and in reality content below it is viewed dramatically more than content in the second slide of a carousel.

Web design offers many unique opportunities, but one of the less exploited is the availability of vertical space. A site has, literally, unlimited vertical space, but horizontal space is always relegated to the width of the browser and is always a finite resource. Take advantage of this horizontal space. Users will scroll if your site looks good and loads quickly. Every meaningful visitor is a visitor that's there for a reason and they will be happy to explore your site for the content they need if there's good content and beautiful structure.

To solve the problem of top-level navigation in responsive design let's elaborate on a couple design patterns: toggle navigation and select menu,. After we look at the patterns we'll discuss the benefits and challenges of these two approaches and then move into multi-level integration.

#### **4.1.1 Toggle navigation pattern**

The toggle menu pattern is a method of displaying all of the top-level navigation behind a menu button on smaller screens. The navigation is hidden and shown by tapping the menu

button. The navigation has an “active” and “inactive” state, determined by toggling the menu button. A good example of this is used on Starbucks.com.

In a single level navigation, the menu would mirror our earlier example on the desktop. In a small screen environment, our navigation will appear simply as a logo and a menu button, with the content immediately below. Once the menu button is tapped, the navigation will expand below and push the content down. In figure 2.3, you’ll see a mockup comparing the “default” and “expanded” states. The default state is the one the page is in when the user enters the site’s URL. The expanded state is the one that the user sees after tapping on the “menu” button.

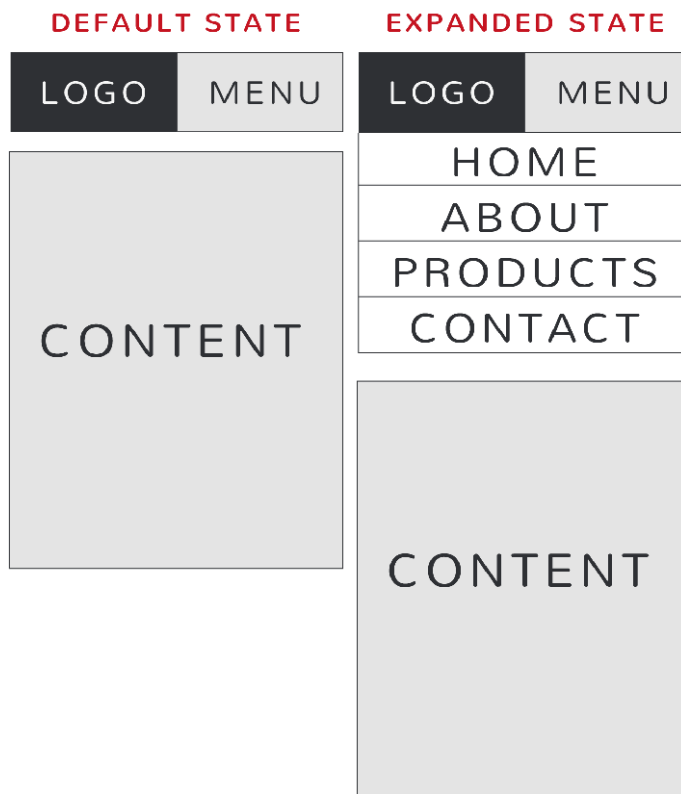


Figure 2.3 A small screen toggle menu. On the left is the default, collapsed state, and on the right is the expanded, active state.

This design pattern has advantages in design as well as in development. As a design feature, this navigation has a consistent feel with the desktop version. The user looks to the same area

for navigation, relative to the desktop layout, but because the navigation is toggled behind a menu button, it has a smaller footprint on the screen real estate.

### CODING THE TOGGLE NAVIGATION

To accomplish this, we'll use some basic markup, CSS, and a little jQuery to toggle the expanded state. First, our HTML.

```
<header>
  <div id="logo" class="l-half grey">Logo</div>
  <div id="menu" class="l-half light-grey">Menu</div> #A
</header>
<nav> #B
  <ul>
    <li>Home</li>
    <li>About</li>
    <li>Products</li>
    <li>Contact</li>
  </ul>
</nav>
<article>
  <h2>Headline</h2>
  <p>Lorem ipsum ...</p>
</article>
#A The menu button will toggle the navigation from hidden to displayed.
#B Here we have the navigation that will be hidden and shown.
```

This is strictly some bare bones markup. By default we'll hide the navigation, and then we'll reveal it if the class of "is-expanded" is applied. We'll even animate it a little with a CSS transition.

```
*{text-transform: uppercase; font-family: Helvetica; text-align: center;}
header, nav{width:100%;}
ul{list-style: none; margin:0; padding:0; float:left; width:100%;}
li{border-bottom: 1px solid #2e3034; margin:0; padding:20px 0; float:left;
width:100%;}

.l-half{width:50%; float:left; padding: 20px 0;}
.grey{background:#2e3034; color:#fff;}
.light-grey{background:#e4e4e4; color:#2e3034; outline:1px solid
#2e3034;}

nav{height:1px; overflow: hidden; width:100%; outline:1px solid #2e3034; -
webkit-transition: height 1s;} #A
nav.is-expanded{height:235px;} #B
#A By default the navigation is 1px tall with the overflow hidden.
#B This class will make the navigation visible.
```

Now add a media query to remove the menu button and display the navigation.

```
@media(min-width:700px){
  #logo{width:200px;}
  #menu{display: none;}
  nav, nav.is-expanded{height:60px;width:500px; float: right}
  ul{width:500px;}
```

```

    li{width:25%;}

    article{float:left;}
}

```

This toggle navigation helps move relevant content up to the top of the page on an initial page load, so the page's content takes priority and the user doesn't have to scroll through navigation items. In a desktop view, the navigation returns to being fully visible and accessible.

This approach has the benefit of keeping the navigation at the top of the page and isolated, so if more elements are added, the layout doesn't break. It relies on a little bit of jQuery, but it could be converted easily to plain Javascript to eliminate the need for a framework.

Using jQuery:

```

$("#menu").click(function(){
    $('nav').toggleClass('is-expanded');
});

```

**A demo of this can be viewed here: <http://cdpn.io/kjztF>**

JavaScript Equivalent

```

document.getElementById('menu').addEventListener('click', function(){
    var el = document.getElementById('nav');
    el.className = el.className + "is-expanded";
},false );

```

The demo for this example can be found here: <http://cdpn.io/Gbhrx>

---

### Developer Insight: JavaScript versus jQuery.

---

jQuery is nearly synonymous with JavaScript for many front end developers. This is with good reason, it makes JavaScript much more accessible and is much easier in cross browser environments. Unfortunately this comes at an expense, mostly in the page load burden.

jQuery clocks in at 28.32kb when gzipped and minified. This isn't an outrageous file size, but that's not the real problem. According to some tests JavaScript performance can be 4 to 5 times slower than on a high end laptop. In general it's wise to use as little JavaScript as possible in mobile site's and to try to avoid bulky frameworks and plug-ins whenever possible.

---

One of the biggest benefits to the toggle navigation is that it easily scales up for a second level navigation. Imagine you have a complex site that requires a level of nested navigation, for instance you might have a site with a section that has sub pages within that section. Later

in this chapter we'll show an example of how to build a multilevel navigation with a toggle navigation design pattern.

A toggle navigation is one way to fix the problem of responsive navigation. The toggle navigation is great because it maintains a single code base and can be rearranged to fit mobile and desktop environments. The biggest shortcoming is that it is limiting in the number of elements you can have in the top levels. I find this pattern to be most useful in sites with a small CMS, like a Wordpress blog or a small personal site.

It's a good idea to keep your top-level navigation pretty direct and simple, but in a very complex site, such as an e-commerce site or a large scale online magazine, the toggle navigation becomes too limiting. In these cases a combination of an off-canvas navigation and a toggle navigation can be successful, but in cases of large scale sites it's crucial to evaluate the needs of the specific site before making a decision.

Another design pattern that helps in solving the problem of responsive navigation is the Select menu.

### **4.1.2 Select menu**

The select menu is one of the earliest responsive navigation design patterns that came about because it provides simple interaction in a small space. This solution benefits the user in the same way the toggle navigation does, by reducing the vertical space the navigation occupies. When the navigation is triggered, the user engages in the browser's default select menu interface.

#### **HOW TO CREATE A SELECT MENU NAVIGATION**

To refresh, we want to create a single level navigation in a mobile environment. Now, when we scale down to the mobile viewport, we're going to see something a little different from our previous example. Here we'll have a `<select>` dropdown in place of a menu.

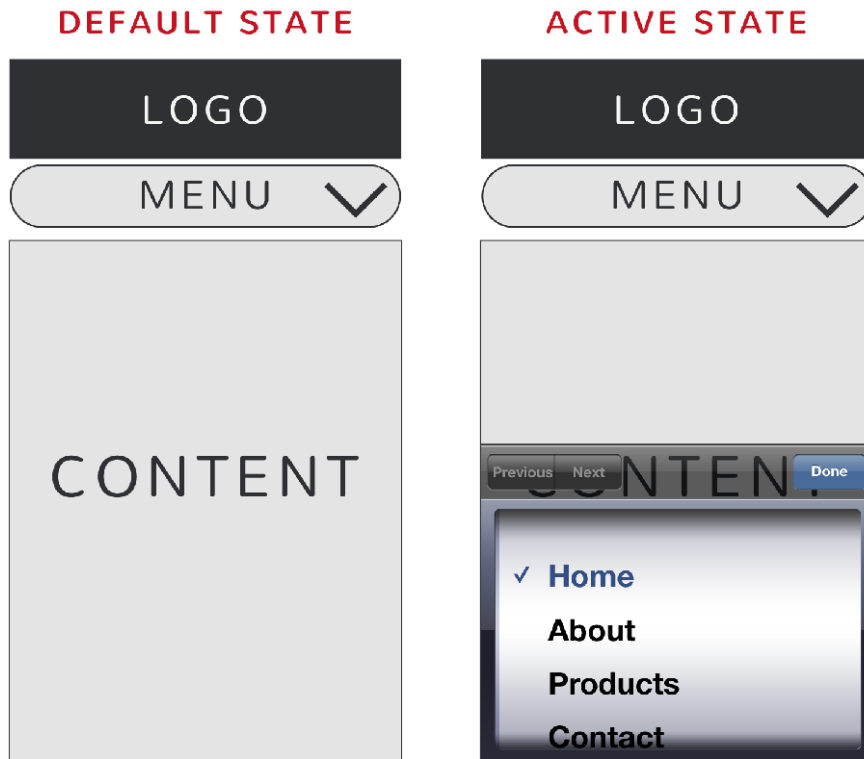


Figure 2.7 The menu in this example is simply a placeholder for the browser default. In the active state, I'm using the iOS 5 default for the select menu interaction.

The code for this version is also going to be similar to our previous example, but instead of changing the format of the navigation code and hiding the menu, we'll hide and show a select menu. Our header markup will look like this:

```
<header>
  <div id="logo" class="l-half grey">Logo</div>
  <select id="menu">
    <option value="">Menu</option>
    <option value="/home">Home</option>
    <option value="/about">About</option>
    <option value="/practice">Products</option>
    <option value="/contact">Contact</option>
  </select>
</header>
```

In menu, the value attribute will be the destination URL. Imagine if each option tag was an anchor, then the href in the anchor would be the value in the option menu. In the browser this will do nothing by itself, so we need a little jQuery to make this work.

```
$("#menu").change(function() {
    window.location = $(this).find("option:selected").val();
});
```

With this little touch of jQuery, we will navigate the user deeper into the site. The .change method listens for the menu to change status and runs the function once the event has taken place.

The core benefit of this pattern is that it uses a very minimal amount of JavaScript. It saves space and leverages the built in operating system user interface to display a navigation. This pattern is useful in longer navigation elements.

### 4.1.3 Toggle navigation versus select menu

With these two design patterns, there's a good amount of benefits and drawbacks to each version. The toggle navigation is simple and effective at masking navigation elements and saving space in a small screen, but it suffers from lack of versatility. Meanwhile the select menu is also compact and to the point but is, to be honest, a huge hack.

The immediate drawbacks of the select menu are:

- It depends 100% on the browser's support of JavaScript. No JavaScript means the user will be stuck completely.
- Select menus can be brutal to style cross-browser, and the creative can be compromised by the default styling. You're very limited in your ability to control the presentation of the navigation.
- It can not be refactored for desktop views. If the user scales up to a desktop view the select menu is completely ineffective and should be replaced with a desktop menu. All of this is excessive and should be avoided as a rule in responsive design.

Meanwhile, the drawbacks of toggle navigations are:

- The toggle navigation favors simplicity and can lack the depth required for complex navigation patterns. If your site has multiple tiers of navigation, it might be difficult to maintain in a toggle navigation.
- The toggle navigation takes advantage of vertical space. In the case that you are using a fixed header (you shouldn't be, by the way. I'll explain in a second) the toggle navigation could end up getting cut off.

---

#### Developer Insight: Why you should avoid sticky headers

*"The navigation should follow the user down the page, that way if they need to move to another page they can without scrolling up to the top" – This UX guy I used to know.*



Here's the problem with sticky headers. They take up too much space in mobile. I mentioned earlier how vertical space is an unlimited resource. That's only untrue if things are being fixed on the page. Since you have a fixed width, when elements start reducing the available screen real estate. The same problems start to arise when you apply a fixed position to anything on the page. When something is fixed, it's not responding to it's environment. Avoid fixed elements at all costs. Modal pop ups, I'm looking at you.

Both of these navigation patterns serve varied purposes and can be useful for different needs. Ultimately my personal opinion is that the select menu option is best to use in a pinch until you figure out something better. When ever possible, though, I would try to use a toggle navigation or an off-canvas navigation that we showed earlier.

## 4.2 Multi-level toggle navigation

We've covered a few basic responses to navigation patterns in this chapter, but sometimes you need to have multiple tiers of navigation. In a select menu, these can simply be listed along with the other navigation elements, so an example isn't very exciting, but as I explained earlier, a toggle navigation can be expanded to accommodate a second tier.



Figure 2.X A desktop example of a multi-level navigation.

With a second tier in your navigation, you need to find ways to accommodate groupings of navigation elements. In a common desktop version a navigation element is hovered and it reveals the second level of navigation. In a mobile version the second tier can be called on top of its parent element.

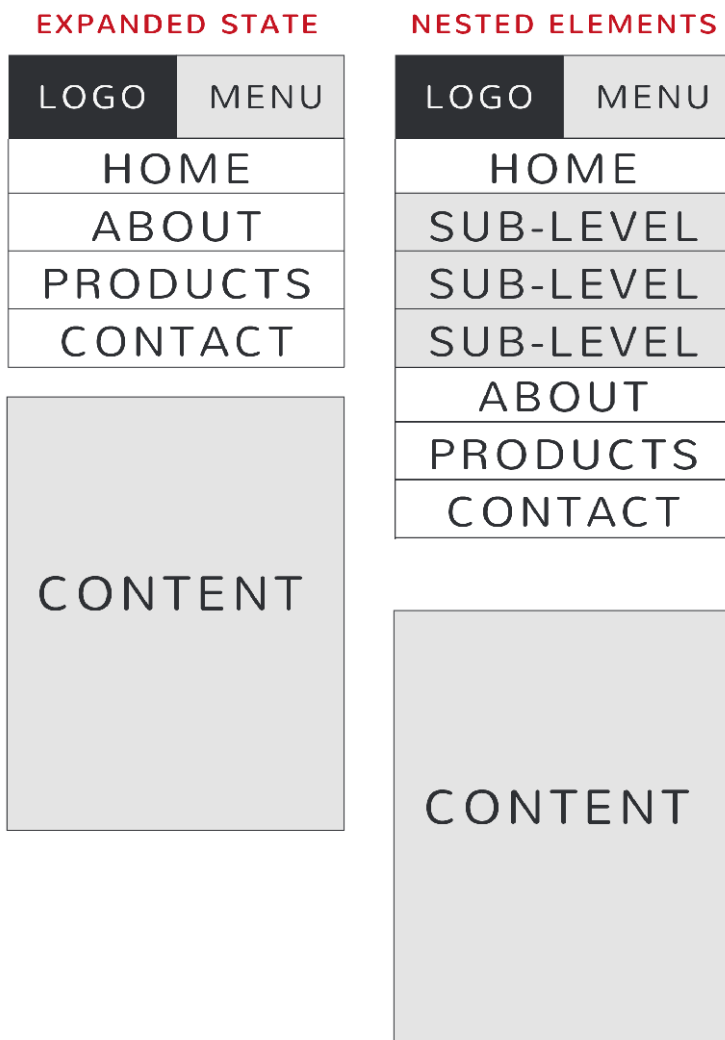


Figure 2.4 Toggle navigation with nested elements.

On the left, our navigation is one level deep, meaning that the user has expanded the menu using the menu button. On the right, the user has expanded the navigation one level deeper by tapping on home. This provides direct access to a subsection of pages.

## HTML

```

<header>
  <div id="logo" class="l-half grey">Logo</div>
  <div id="menu" class="l-half light-grey">Menu</div>
</header>
<nav>
  <ul>
    <li class="has-subnav">
      <a href="#">Home</a>
      <ul class="subnav">
        <li><a href="#">sub-nav link</a></li>
        <li><a href="#">sub-nav link</a></li>
        <li><a href="#">sub-nav link</a></li>
        <li><a href="#">sub-nav link</a></li>
      </ul>
    </li>
    <li>
      <a href="#">About</a>
    </li>
    <li>
      <a href="#">Products</a>
    </li>
    <li>
      <a href="#">Contact</a>
    </li>
  </ul>
</nav>
<article>
  <h2>Headline</h2>
  <p>
    Lorem ipsum [...]
  </p>
</article>

```

## CSS

```

*{
  text-transform: uppercase;
  font-family: Helvetica;
  text-align: center;
}

header, nav{width:100%;}

ul{
  list-style: none;
  margin:0;
  padding:0;
  float:left;
  width:100%;
}

ul.subnav{

```

```

    height:0px;
    overflow:hidden;
}

li.is-expanded ul.subnav{
    height:236px;
}

ul.subnav li a{
    background:#e4e4e4;
}

li a{
    border-bottom: 1px solid #2e3034;
    margin:0;
    padding:20px 0;
    float:left;
    width:100%;
}

.l-half{
    width:50%;
    float:left;
    padding: 20px 0;
}

.grey{
    background:#2e3034;
    color:#fff;
}

.light-grey{
    background:#e4e4e4;
    color:#2e3034;
    outline:1px solid #2e3034;
}

nav{
    height:1px;
    overflow: hidden;
    width:100%;
    outline:1px solid #2e3034;
    -webkit-transition: height 1s;
}

nav.is-expanded{height:235px;}

nav.is-expanded.is-subnav-expanded{
    height:470px;
}

```

## JS

```

$('#menu').click(function(){
    $('nav').toggleClass('is-expanded');
});

```

```

$('.has-subnav a').click(function(){
    $(this).parent().toggleClass('is-expanded');
    $('nav').toggleClass('is-subnav-expanded');
});

```

This code can be viewed here: <http://cdpn.io/kjztF>

Here we have some HTML, which has a second level of an unordered list that is revealed when its parent link is clicked.

### 4.3 Summary

In this chapter, we've discussed responsive navigation patterns, and I've outlined two different approaches to top level navigation in responsive designs. This is just the tip of the iceberg. There are patterns for page layouts, text elements, and modules, with new patterns being discovered and implemented every day.

It's important to mention that these two patterns are simply to show how there can be multiple solutions to the same problem. By exploring the variety of possibilities we can build not according to trends but we can build to meet the specific needs of our site.

When attempting to solve the problem of responsive navigation its important to keep in mind:

- What are the needs of this given site? How is going to be updated and how often?
- How can I take advantage of the space available to me in small screen environments?
- Am I prioritizing the content and making sure the user has immediate access to what they came to this site for?

In the next chapter we're going to dive into building site layouts using percentages. This chapter will teach you the fundamentals of working in percentages as opposed to fixed pixel layouts and how to avoid making rigid websites. The next chapter will include tips on making sure the site content is kept at the top of mind.

# 5

## *Responsive layouts*

This chapter covers

- Using percentages for layout
- Creating an off-canvas navigation
- Adjusting the layout for varied screen sizes

Building a responsive layout is, quite possibly, the easiest task in building a responsive website. All a layout has to do is gracefully refactor at given breakpoints. In the first chapter of this book we demoed some media queries to enable this refactoring. We showed how selectors can override each other when applied from within a media query, now it's just a matter of applying this logic. An element in a small screen should be one width and then another width in a larger screen.

However simple this task should be, things always find a way of getting more complicated. In the case of responsive layouts, things get more complicated because layout is still being determined in fixed terms. You see, when a layout is created in a graphics editing program, it uses an assumed absolute sizing and that sizing is dictated using measurements like pixels. This is why it's important that in building responsive sites, we determine layout in the browser, not in a comp.

It's very easy to adjust sizes and share screen space with CSS, but doing it in graphics programs is a recipe for inconsistency and extraneous coding. This is because there is an information breakdown in the way these two technologies render their visual layers.

In this chapter we'll cover the most effective ways of crafting a responsive layout. This means building a fluid grid system using percentages instead of pixels. We'll also cover some of the tricks you can use to make this process a little easier.

### Designer's insight: a designer's role in layout

Though much of the hands-on responsibility here lies on the shoulders of the developer, an art director's sense of space is absolutely crucial to prevent the layout from looking too static or uniform.

Quick sketches and regular, short check-ins are crucial for the designer to retain input in this phase of the process. Naturally, this is going to require some understanding and trust on both sides, but the product that results from working like this is a site that everyone can be proud of.

## 5.1 Fluid Layouts

In responsive web design you'll notice the use of percentages for layout widths. These percentages create what's called a "fluid layout". A fluid layout is one that adjusts to the available space. In order to understand how this works and why it's important you need to know how percentages work in CSS.

Historically a web design would base all of its sizing on pixels. These pixels represent screen pixels, although it's not a 1 for 1 representation. High DPI screens will convert what's set as a "px" or "pixel" size into its appropriate grouping of screen pixels. This is a complicated equation and not entirely important, what is important is understanding that pixels are fixed units of measurement on digital screens. A pixel on a screen is like a centimeter or an inch on a ruler, it's simply a unit to measure sizing with.

The problem with pixel sizing in responsive web design is that a fixed size doesn't lend itself to scaling well. Once set, it's set. The elements set with pixels can have their sizes adjusted, but this would require changing everything on the screen. To effectively build a responsive site we need to set sizes relative to something. This is where percentages come into play.

Percentages work in CSS as the name implies, they occupy a percentage of space. That percentage of space is determined by the parent element. This means if there's an element that is 1000 pixels wide and inside it there are two elements. One is 20% wide and the other is 80% wide, the element that's 20% wide will be rendered 200 pixels wide and the one that's 80% wide will be rendered at 800 pixels wide, totaling the 1000 pixel wide parent element.

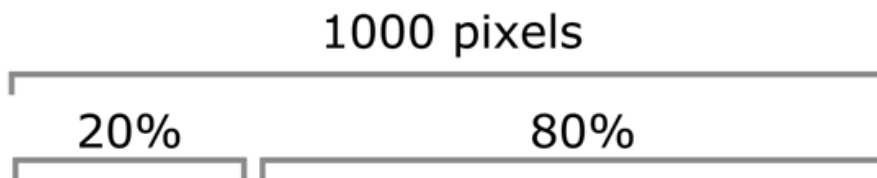


Figure 5.1 Widths represented in percentages.

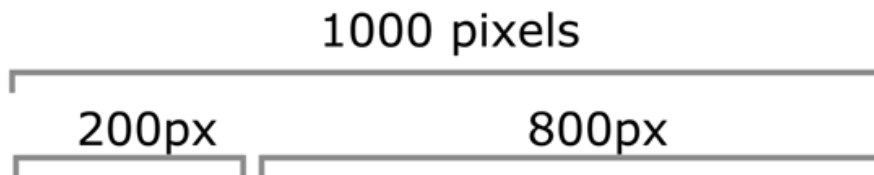


Figure 5.2 The same widths in their pixel equivalent.

If this diagram was to be represented in code you could think of it like this:

```
<div class="parent">
  <div class="twenty-percent">
    </div>
    <div class="eighty-percent">
      </div>
</div>
<style>
.parent{
  width:1000px;
  height:20px;
  outline:1px solid green;
}
.twenty-percent{
  width:20%;
  height:20px;
  float:left;
  background:red;
}
.eighty-percent{
  width:80%;
  height:20px;
  float:left;
  background:blue;
}
</style>
```

This example is viewable here: <http://cdpn.io/IIKet>

This example produces this 20 pixel tall line:



When we scale this down it can retain its same scaled size. The layout remains “fluid” because it’s relative to a fixed point on the page, in this case the parent (.parent) element. By using percentages we’ve detached the layout from a singularly useful fixed pixel width. We can demonstrate this by changing the percentages to pixels in our example. So now, let’s change the above rules to this:

```
.twenty-percent{
  width:200px;
  height:20px;
```



```

        float:left;
        background:red;
    }
    .eighty-percent{
        width:800px;
        height:20px;
        float:left;
        background:blue;
    }

```

Which still creates this line:



This example is viewable here: <http://cdpn.io/tweBD>

Nothing has changed, yet, but watch what happens with we shrink the `.parent` element to 800 pixels wide.

```

.parent{
    width:800px;
    height:20px;
    outline:1px solid green;
}

```



This example can be viewed at: <http://cdpn.io/faBci>

The layout is now broken. The 800px wide element is too big to be in line with the 200 pixel element and has returned to a second line, but if we change the 200px and the 800px back to percentages, we'll get a different result entirely:

```

    .twenty-percent{
        width:20%;
        height:20px;
        float:left;
        background:red;
    }
    .eighty-percent{
        width:80%;
        height:20px;
        float:left;
        background:blue;
    }

```



This example is viewable at: <http://cdpn.io/EiGat>

The line is now back in place, except it's 800 pixels wide instead of 1000 pixels, because the two objects have kept their widths relative to their shared parent. You might say at this point, "but there's still a fixed pixel size, the `.parent` element!". You would be right, but this is just an example. In practice that `.parent` element could be the browser's viewport. The point here is to keep everything relative, flexible, and **fluid**. Just like how liquids fill the space their given, so do fluid layouts.

### Developer's Insight: What about height?

If you're paying attention you might have noticed that I've been using fluid widths, but even in the example above I use a pixel height. This is for good reason. In responsive environments height is often malleable. In the examples I'm setting height so the elements are visible, but often fix heights can be troublesome. One way of overcoming this is to let the child elements determine height automatically. In the case that you need to set a height that is relative to the width of an element, for instance to maintain aspect ratio, an effective way I've found is to set height to 0, then using padding-bottom to set the proper percentage. This is because padding is set relative to the width of the parent element, just like width, and that even applies to padding on the top and bottom. Most browsers will still display the content if it's contained in the padding of an element.

### 5.1.1 Box sizing

One of the things that makes using percentages to create fluid layouts is the difficulty in maintaining paddings and margins. Say you have the above 20% and 80% examples and you wanted to put some margin between the two elements. This would be easy if you had nice round numbers, like 1% or 2%. In a perfect world you could just draw the 20% wide box as 19% and add a margin-left of 1% and be done with it.

Unfortunately this is rarely the case. In the real world you'll find yourself searching for percentages that aren't that safe and rounded. You could end up trying to use a margin-left of 1.5436%. You could easily do some math, subtract the padding from width and adjust. This becomes a real burden when you need padding-left and padding-right. Then you find that the padding needs some tweaking and you have to go change things over and over. Fortunately there's a better way. There's a rule in CSS called "Box-sizing".

Typical box model width is determined like this:

$width + padding + margin = rendered\ width$

By setting `box-sizing: border-box` the width of an element is determined like this:

$width = rendered\ width$

Lets look at this another way.

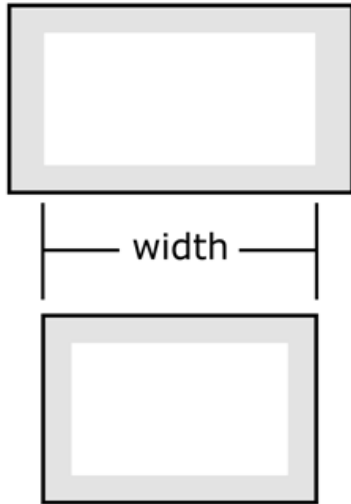


Figure 5.3 Imaging the grey area as padding. In the bottom example we see an element with border-box applied

Let's look at this another way. Here's a simple grid of four 25% width elements:

```
<div class="grid"></div>
<div class="grid"></div>
<div class="grid"></div>
<div class="grid"></div>

.grid{
  width:25%;
  height:0px;
  padding-bottom:25%; /* See the developers insight above */
  float:left;
  outline:5px solid red; /* so the grid is visible */
}
```

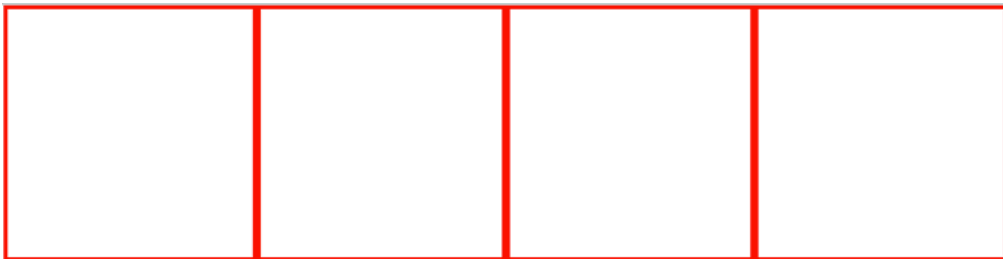


Figure 5.4 A simple grid. Code is viewable here: <http://cdpn.io/geKbJ>

Everything is going just fine, but lets put some padding between these elements:

```
.grid{
  width:25%;
  height:0px;
  padding-left:1.5678%;
  padding-right:1.5678%;
  padding-bottom:10%; /* See the developers insight above */
  float:left;
  outline:5px solid red; /* so the grid is visible */
}
```

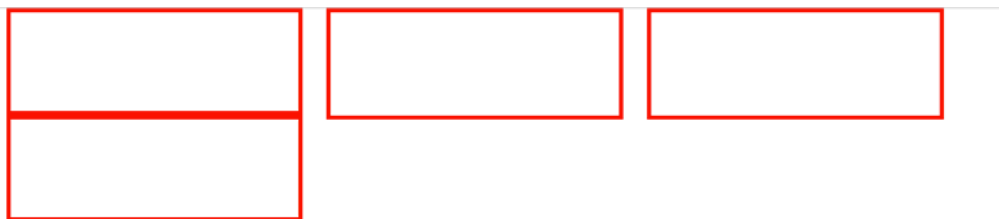


Figure: 5.5 A broken grid. Applying box-sizing will fix this.

Now all our spacing is off. Our grid is broken, but we can fix it by adding this one line:

```
*{box-sizing:border-box;}
```



Figure 5.6 Now with box-sizing added. View here: <http://cdpn.io/nBHve>

Now we can change our paddings to be whatever we want without affecting our grid! This will make building responsive sites dramatically easier by ensuring that sizing is predictable.

## BROWSER SUPPORT

One concern with box-sizing might be browser support. Fortunately, box-sizing is available to all modern browsers and Internet Explorer down to IE8. There is no fall back to IE7 and under and unfortunately if supporting IE7 and lower is a priority, then you might be better served avoiding box-sizing altogether.

### 5.1.2 Fluid Grid Systems

With our understanding of how fluid layouts work in a purely structural sense, let's talk about grid systems. Grid systems in web design are completely common. They became a hot topic years ago when layouts were set to specific grid sizing. Have you ever wondered why a lot of web sites seem to be about the same width? This is a result of fixed width grid systems that were established in the early days of modern web design.

Around the time that "Web 2.0" became a thing, the change from table layouts to div based layouts created the need for a div based grid system. The grid system was popularized as a method to apply vertical structure to a webpage. It arranged the site into columns to give more sense to the structure of the content.

In the early 2000's about 50% of user's had a screen resolution of 800 x 600. Through the 2000's that grew to 1024 x 768. In these times most of the advancements in personal computing came in the forms of processor speed and in the later half of the decade graphic processing (which governs screen resolution) and desktop screen sizes began to become of higher and higher priority.

Because of the predictable nature of screen sizes in this time, layout grids were set at a predictable size, like 1024px wide. Obviously this predictability is no longer available and an alternative is required. In this case we need a fluid grid system.

#### OUT OF THE BOX GRIDS

There are many frameworks that include grid systems out of the box. Twitter Bootstrap and Foundation both include their own unique grids. The difference between them is near non existent, and when picking a front-end framework, the included grid systems are near identical enough that they shouldn't be a deciding factor.

There are also dozens of other self contained grid systems, such as "skeleton" or "golden grid system". These too also amount to a matter of taste and again are nearly identical. I refrain from using the "out of the box" grids, most of the time, simply because in production I only ever use a small part of their features and generally a custom grid system is needed anyways with sizes that reflect the needs of the site.

#### BUILDING A GRID

In order to teach how a basic grid system works, it's easier to teach how to build a grid than it would be to show how to use an "out of the box" fluid grid. We'll simply build a single "row" class to define a grouping of grid pieces, a "grid" class to define a grid piece, and a few classes for how many columns each grid should occupy. First we need to identify how many columns we want each row to contain. For the sake of simplicity we'll start at an 8 column grid in desktop and a 4 column grid in mobile.

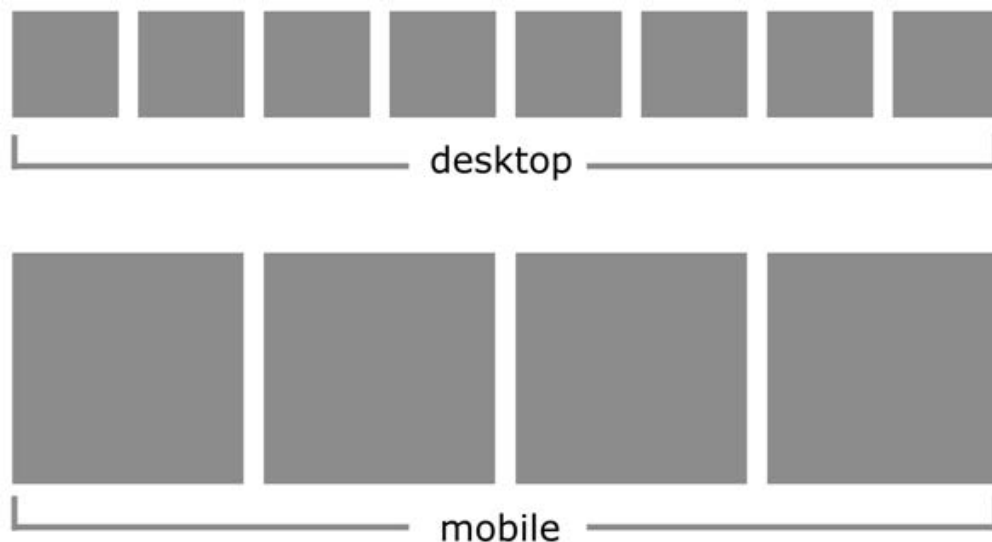


Figure 5.4 Here we see the desktop and mobile grids. It's important that the desktop grid is divisible by the mobile grid, so that rows remain even.

Now to create a simple 8 column grid we'll start with some boilerplate HTML and simple CSS. We'll do an 8 column row that breaks into 4 columns for mobile. We'll set the sizing against the grid elements as opposed to the row itself (which is a popular method in some "out of the box" grids). This gives us the flexibility to adjust the grid elements within the row.

**HTML:**

```
<div class="row">
  <div class="grid m-grid-1 d-grid-1">
  </div>
  <div class="grid m-grid-1 d-grid-1">
  </div>
  <div class="grid m-grid-1 d-grid-1">
  </div>
  <div class="grid m-grid-1 d-grid-1">
  </div>
  <div class="grid m-grid-1 d-grid-1">
  </div>
  <div class="grid m-grid-1 d-grid-1">
  </div>
  <div class="grid m-grid-1 d-grid-1">
  </div>
  <div class="grid m-grid-1 d-grid-1">
  </div>
</div>
```

**CSS:**

```

.row{
  width:100%;
  max-width:960px;
  /* for desktop view */
  margin:0 auto;
  outline: 1px solid blue;
  /* to visualize our element */
}

/* clearfix set against the row class for conviance */
.row:before,.row:after {content: " ";display: table;}
.row:after {clear: both;}

.grid{
  height:20px;
  /* to visualize */
  float:left;
  margin:1%;
  outline:1px solid red;
  /* to visualize */
}

.m-grid-1{
  width:23%;
}

@media (min-width:960px){
  .d-grid-1{width:10.5%;}
}

```

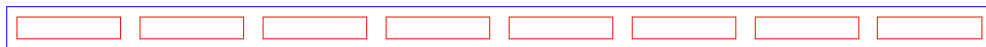
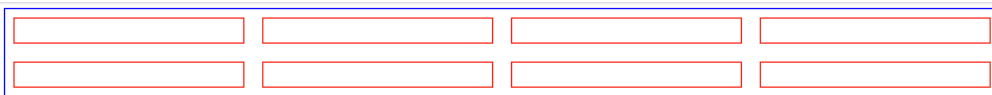
**Desktop:****Mobile:**

Figure 5.5 The start of a simple grid system with mobile and desktop views. Code is viewable here: <http://cdpn.io/lamFi>

Here we have the beginnings of a custom fluid grid system. The objects here simply represent one column width elements in our side. The classes prefixed “m-” are for our mobile grid and the classes prefixed “d-” are for our desktop grid. Now each object has its width applied by its “m-” or “d-” class.

We can expand this grid system by adding more grid sizing classes.

```

.m-grid-1{width:23%;}
.m-grid-2{width:48%;}
.m-grid-3{width:73%;}
.m-grid-4{width:98%;}

@media (min-width:960px){
  .d-grid-1{width:10.5%;}
  .d-grid-2{width:23%;}
  .d-grid-3{width:35.5%;}
  .d-grid-4{width:48%;}
  .d-grid-5{width:60.5%;}
  .d-grid-6{width:73%;}
  .d-grid-7{width:85.5%;}
  .d-grid-8{width:98%;}
}

```

Code can be viewed here: <http://cdpn.io/KFALc>

With this we now can build layouts with a very bare bones grid system. I would recommend adjusting this to fit your needs, but this should give you the start in logic to build something more robust. The class word like this: X-grid-Y. X is either m for mobile or d for desktop and Y is the number 1 through 12 which represents the number of columns. We could potentially break this down further by adding a "t" prefix for tablets and a second breakpoint.

This sort of fluid grid is helpful in creating basic fluid layout, but lets dig into some trickier components.

## 5.2 *Building a fluid layout*

With the background out the way, it's time for us to dig in to building the layout. With a fluid layout we'll be building parts of the page that refactor themselves as they change viewports. In this section we'll build a fluid, responsive header for our site that adapts to a fixed width in the desktop size.

### 5.2.1 *Interpreting the prototype*

If we were pressed for time, we could develop straight from our prototype, but we aren't so what we want to do is break the prototype into pieces so we can identify all the parts and then combine them into a whole. Building a layout out of a prototype is simply a matter of trying to find efficiencies and applying the given design to the code you have.

Given that a prototype can be wildly different from one another, there is no set way to interpret a prototype. There are, however a few tips I can give you to make it easier:

1. You should use a prototype to communicate ideas. Nothing in the prototype form is final. Remember this when interpreting what's in a prototype. Perhaps with the design system you've been given, there's new ways to interact and animate elements on the page.
2. Identify groupings of objects on the page to identify major layout components. This will help you determine your necessary templates and give you the layout options you need to accommodate.



3. The design elements you have at this point should serve to inform typography and user interface. It would be helpful to write the code for these UI elements and typographical design separate of building the actual page. This can serve as a style guide for the site and encourage a consistency in

After you've identified your first layout elements it's time to start coding. For our site, we'll start with a header.

### 5.2.2 Start coding

We want to make sure to write HTML that is streamlined. Browsers parse HTML first to determine DOM structure, so it's important to keep your markup as tidy as possible while writing code that is maintainable. Try to remember that when you are writing markup, it's going to be interpreted by browsers as well as people, and a good front end should appeal to the interests of both. Dissecting how browsers work is too big of a topic to cover here, but any time you can spend reading up on the inner workings of web browsers is time well spent.

#### Designer insight: inside the browser

Learning about how browsers render a web page can be especially challenging for a designer who doesn't spend time writing code and seeing how HTML and CSS interact. To read more on the subject, Manning Publications offers a great book : "Manning: Hello! HTML5 & CSS3".

The following in figure 5.x is what our header will look like after we've written the markup and added CSS.

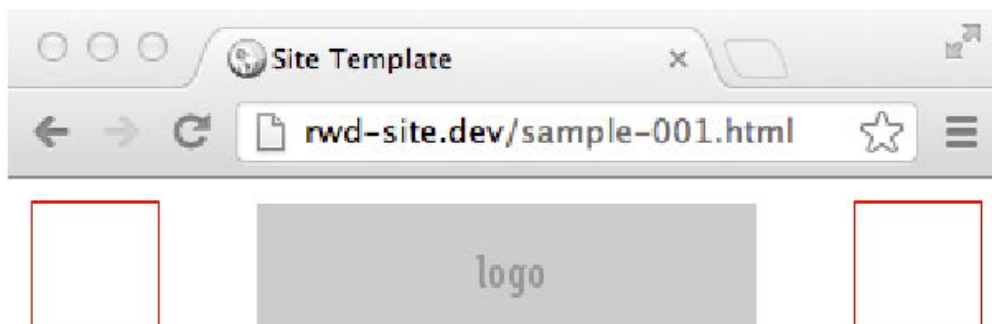


Figure 5.x our header after we've written the markup and added CSS

Here we want to add some markup for the required elements. We'll need two content areas, one for some supplemental information and one for navigation. In the markup we

also want to keep in mind that as the viewport expands, we want our site to retain the same basic structure.

```
<div class="wrapper">
  <header id="topHeader">
    <aside class="left-tray">
      <p>Brief bio [...]</p>
      <nav>
        <a href="#">Twitter</a>
        <a href="#">Github</a>
        <a href="#">Contact</a>
      </nav>
    </aside> #A
    <span id="infoTray"></span>
    <div class="logo"></div>
    <span id="navTray"></span>
    <aside class="right-tray">
      <nav>
        <a href="#">Home</a>
        <a href="#">About</a>
        <a href="#">Writing</a>
        <a href="#">Contact</a>
      </nav>
    </aside> #A
  </header>
  <article></article>
</div>
```

**#A These two aside tags will be used for our off-canvas navigation in the mobile view.**

The markup for this section is pretty straight forward. We've taken the substance of our prototype and added some content and design patterns to improve communication and ease of use. Now we need to add some CSS for layout.

### Developer's insight: layout versus style

CSS is used to add both styles and positioning to page elements. In this workflow, we want to handle those two tasks separately. The reason for this is that we are using two deliverables to articulate our ideas. The prototype is for layout, and the style guide is for style. This way we can achieve the layout with input from UX designers, and the style with input from art directors, without the two disciplines offering conflicting feedback.

We want to hide our off-canvas navigation first, and make our button elements visible. To show the buttons on the page, I'll use 1 pixel red outlines to temporarily makes them visible. To see this in detail, check out sample-001.html and sample-001.css in the code samples.

```
.wrapper{
```

```

width:100%;
position:relative;

-webkit-transition:all 1.0s ease-in-out; #A
-moz-transition:all 1.0s ease-in-out; #A
-o-transition:all 1.0s ease-in-out; #A
transition:all 1.0s ease-in-out; #A
}

[...]
```

```

#topHeader .left-tray{left:-50%;} #B
#topHeader .right-tray{right:-50%;} #B
```

**#A** We'll use these CSS transitions to animate our canvas.

**#B** I'm using negative positioning to hide the off-canvas elements. We'll use jQuery to add state classes to hide and show these later.

With these pieces in place, we simply need to add a little jQuery magic to make our layout move around.

### 5.2.3 Animating the off-canvas elements

In this case, we're just going to add classes to the body tag on our page to communicate page states. When writing these classes, we'll avoid using terms like "slide," so that they are logical to reuse in larger viewports.



Figure 7.7 The left off-canvas element expanded.

With a little touch of jQuery, we're adding and moving classes.

```
$( "#infoTrayBtn" ).click(function() {
```

```

    $("body").toggleClass('info-active');
  });

  $("#navTrayBtn").click(function(){
    $("body").toggleClass('nav-active');
  });

```

Next, we need some CSS to support our states.

```

.info-active .wrapper{left:50%;}
.nav-active .wrapper{right:50%;}

```

Now we have a simple off-canvas design pattern. Since we're focusing on layout styles, it's going to look completely unstyled; but our base is there, and we can start adding design elements.

We have our header relatively in place, let's make this element responsive. We're going to do this per element, so we can focus and make each modular element on the page function appropriately.

### 5.2.4 Making the element responsive

In responsive web design, something is only done after it's been blown out to work for a range of site widths. Although we have a nice starting point in our website now, we need to expand for wider views, or else we'll just have a gigantic mobile website.

#### ADDING MEDIA QUERIES TO MAKE THE HEADER RESPONSIVE

When building a responsive site, it's all about using the same objects on the page to create a varied layout. You want a single HTML base and nearly no hidden objects on a page. What we want to do is avoid the trap of changing elements at assigned breakpoints simply because a certain device has a certain viewport. Because new devices with new resolutions are coming out all the time, the goal is to make something fluid that works without dependency on a few set viewports.

In our example our first breakpoint is at about 720 pixels. This is a good example breakpoint because while the elements still seem to work fine, there is definitely some open space to play in. At our first breakpoint we need to consider the amount of space we now have, and take advantage of it according to our site's priorities. Since the content on the left is supplemental and the content to the right is navigation, the content in the right tray takes priority.

#### Developer insight: tracking viewport with console.log

One thing that makes my life much easier is logging the current viewport in the browser console. It gives me an absolute number to base my media queries off of. Another little tip is that developing in Chrome lets me use the developer tools to make my life easier. If you open the settings in the developer console, you can dock the tools to the right.

The browser will retain its viewport width, minus the size of the inspector tools, so you can preview a mobile site and have a large console at the same time.

To log the viewport, simply add `console.log(document.body.clientWidth);` to your JavaScript file, and it'll log the viewport width on site load.

What we want to do is open up some of the space on the right to give us room for our navigation. To do this we can move our logo to the left, and then we can pull our left navigation over. We also need to hide the navigation button on the right, since we won't be using it.

```
@media (min-width: 720px){
    #navTrayBtn{display: none;}

    #topHeader .right-tray{
        width:auto;
        padding:10px;
        right:0;
    }

    #topHeader .logo{
        left:70px;
        margin-left:0;
    }
}
```

With just a few minimal tweaks, we've changed the layout of the page completely (figure 7.9).

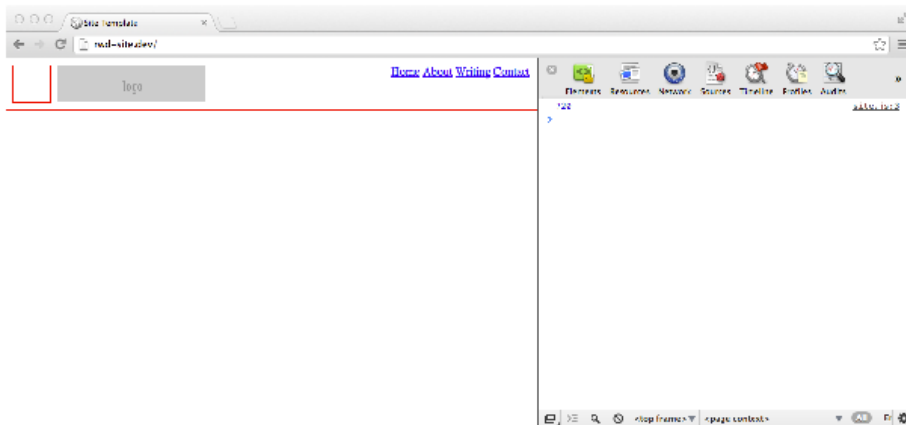


Figure 7.9 Our updated layout

This is an absolute success in setting up a break point. Without changing our markup at all, we've adapted our header's layout to fit its new context. While the right navigation looks great, our left navigation is now a little off. First off, it might look better to have the logo completely left justified. There's also the issue of its expanded state (figure 7.10).

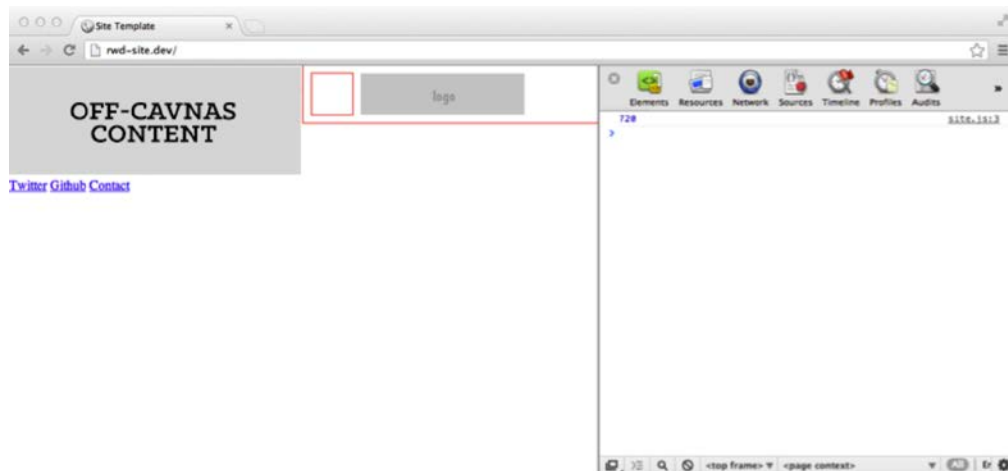


Figure 7.10 Our expanded information panel on the left of the page is looking a bit out of place in the larger viewport.

Here, the content within the off-canvas navigation doesn't seem to justify its screen real estate. Let's change the way this button works, while keeping as much of its core intact as possible. The easiest way to do this is to change the CSS. The most elegant way of doing this is to not change how it functions, but where the interaction takes place. Instead of hiding the off-canvas elements to the left, let's move them to the top and pull them down into the user's view.

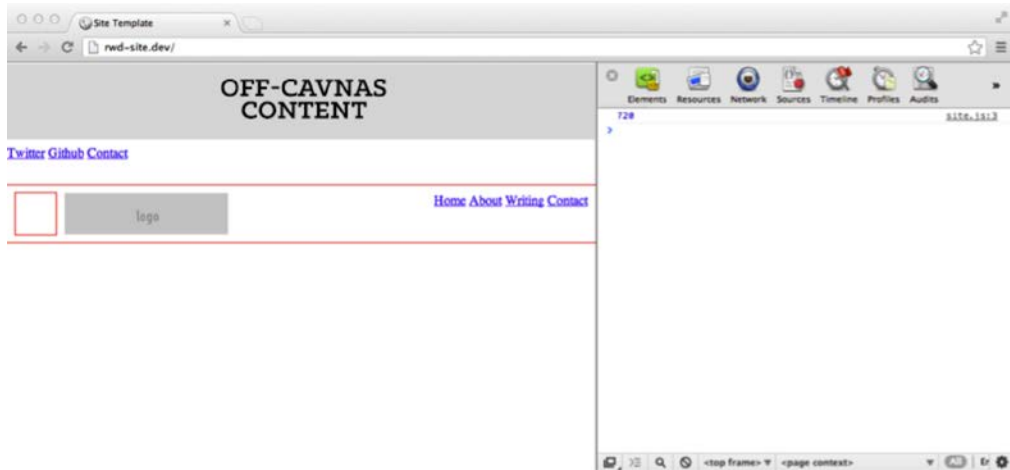


Figure 7.11 Our nav now drops in from the top down.

We now have a header in place, with a single breakpoint at 720 pixels. From here we can start expanding our page for larger screens. This is going to mean adjusting not just the positioning, but also some of the logic behind our elements.

### 5.2.5 Expanding into the wider views

When going from tablet to desktop, you can start opening up your layout a bit. You have room to move your elements around a little more. In our case, we want to keep things mostly the same but increase sizes a little. The easiest way to set this is with a max-width rule on the .wrapper class. Because everything within the wrapper is percentage based, the site should retain its structure just fine. We just add the following rules to .wrapper in our site.css file:

```
max-width: 1024px;
margin: 0 auto;
```

And our site will be framed in a 1024px wide wrapper (figure 7.12).



Figure 7.12 Our framed site header

Our header now has a lot of empty white space, which is okay for the most part. Once we get into designing the page a little later, we'll be glad we have that extra space, because it'll give us more space to expand creatively. One thing we can do now in the layout phase is increase the size of the logo and decrease the size of the information button. The reason we can decrease the information button is because we can assume our user is using a mouse/trackpad and keyboard as their inputs at this point.



Figure 7.13 A smaller information button

With this new layout, our site functions nicely for a desktop browser with some nice interactive touches to engage the user. We've created a header module that reacts responsively, both in its interaction and in its layout. We've used very few fixed width elements, instead relying entirely on percentages, giving us a nice degree of fluidity for small and mid sized screens.

## 5.3 Summary

In this chapter we covered the core fundamentals of building a responsive layout. In it we learned the basics of using percentages to create a fluid layout. We discussed how fluid layouts vary from pixel based layouts, as well as how they can be changed with media queries.

We also touched on the use of the box-sizing to affect how the box model renders on a page and what the benefits and drawbacks of it are. We also touched on fluid grid systems, building off canvas navigations, and how to hide content off the page.

There was a lot of ground covered in this chapter, but with this bit out of the way we're ready to proceed and start building responsive layouts. In the following chapter's we'll build on what we've learned and get a little more detailed.



# 6

## *Adding Content Modules and Typography*

### Summary

- Add an adjacent content module to our layout.
- Build thorough demo content
- Add web fonts using an external CDN
- Easily scale typefaces for various breakpoints

A few years ago, the American public broadcaster NPR began to expand its digital channels. At the time they had a few iOS and Android apps, as well as a website to serve content to. In serving content to these channels, they created an API that would feed content to a server, which the websites and applications could request articles from and serve them to the users as an XML file.

Using their API, NPR only had to change the presentation layer to meet the needs of the platform and could rely on the XML feed for content. As a result NPR has developed a robust library of applications that listeners can use to consume content in the format and on the device that they prefer.

Although this example is related to developing a suite of applications, the lesson remains equally important. Content is the substance behind everything we do online. Layout is the foundation for the contents display, but it's the content itself that the user is visiting for. The formats might change, it might be video, audio, text, or even some sort of experience, but ultimately every visitor is looking for some sort of content.

---

In our example site, our content is pretty clear. We'll be creating a site to host blog articles. Sometimes a luxury like this isn't available. In my experience a client will come to me looking for "a website". They aren't really sure of what they want to say or how they want to say it, so the obvious first step for me is to start building a website and eventually figure out where we need content and start plugging something in.

I now realize how counter intuitive this process is and how much harm it can really do to creating a quality experience. Even if a site appears simple it's crucial to identify why somebody might be there and build around that purpose. At one point I was discussing with a friend a recent site I was building for a nearby restaurant. There was a lot of discussion around what the experience of the site should be, what images and interactions to use. I made the point that 90% of people just want to find a nearby restaurant and look at the menu.

"Then that should be your site," he told me. I argued that it would be too simple and that there should be more to it than that. "Why?" he asked me, "you just said that 90% of the visitors just want the menu and the location. Why can't it just be that?". My friend had made a point that changed how I looked at the content of the pages I build. If you know what people want, why waste time and money building something nobody will use?

---

What does our user want from our site? The focus of the site is an outlet for writing and to share ideas on web development so we want the most recent article front and center when the user visits. A typical post includes an image, a headline, publishing date, and tags. The site is centered around written content, so this is our focus. In this chapter we'll create a content module in a mobile view and then scale up the site and discuss typographical concerns like sizing and styles that we'll encounter as the site expands. All the techniques in this chapter can be applied to any responsive site you're working on, because most all sites have and need content.

## 6.1 *Adding a content module*

In Chapter 1 we discussed the benefits of designing sites mobile first. When taking this approach it's important to identify the necessary components of a site and how their needs are adjusted as the width of a site increases. In the previous chapter we discussed how width is a fixed resource in the web. This is one of the most visible driving factors in the need for responsive web sites, as it has the deepest effect of how we curate the content of our site.

I have a close friend who is a talented painter and artist. When she starts an art exhibit she always goes to the space the exhibit is being held so she can choose the right work to display and get a general sense of what will be shown. It's crucial to the work that she curates her exhibits in context of the space it occupies. This is similar to the task we as

developers take on in building and designing our sites. Often our role is that of a curator, but in order to properly curate our sites, we have to first be aware of what we are curating. It's more than simply making the content of the site bigger or smaller, but what is important and how it is best consumed.

Each of our content modules is like a piece of art. They have their unique properties and consideration. Our content might be a video, a block of text, an image, or an experience, but more often than not it's a combination of all four. Let's get started.

### 6.1.1 *Creating useful placeholder content*

In our prototype, we included a few paragraphs of placeholder text, but at this point we want to add a set of commonly used HTML elements into our placeholder article, so we can start building a typographic base for our written content. We want to see how inline elements, such as bolded text, links, and italicized text, as well as block elements, such as unordered lists and headlines within the article use standard HTML tags.

By doing this we begin to create a typographic standard for our site. Although we don't want to start defining typefaces yet, what we're looking to do is have the elements in place so we can see how they look as the site expands and the layout changes.

The mark up will look something like this (and is included in sample-004.html):

```
<section role="main">
  <article>
    <figure class="masthead-image">
      
    </figure>
    <hgroup>
      <h1>Article Headline Sample - Character count of 47</h1>
      <h2>Article reenforcing sub headline - character count of 56</h2>
    </hgroup>
    <aside>

    </aside>
    <p>Lorem ipsum dolor sit amet
    <b>inline bold element</b> [...]
    <a href="#">inline text link</a> [...]
    <i>inline italics</i> [...]
    dolore magna aliqua. [...]
  </p>
  <ul>
    <span>Unordered List:</span>
    <li>List Item</li>
    <li>List Item</li>
    <li>List Item</li>
    <li>List Item</li>
  </ul>
  <p>Lorem ipsum [...] </p>
  <ol>
    <span>Ordered List:</span>
    <li>List Item</li>
    <li>List Item</li>
    <li>List Item</li>
    <li>List Item</li>
  </ol>
```

```

<p>Lorem ipsum [...].</p>
<p>Lorem ipsum [...] </p>
<h1>In Article Headline 1</h1>
<h2>In Article Headline 2</h2>
<h3>In Article Headline 3</h3>
<h4>In Article Headline 4</h4>
<h5>In Article Headline 5</h5>
<p>Lorem ipsum [...] </p>
<figure>
  
  <figcaption>Example caption</figcaption>
</figure>
<p>Lorem ipsum [...] </p>
<p>Lorem ipsum [...]</p>
</article>
</section>

```

This covers all the elements included in the prototype, as well as a good number of base level typographical elements. There's a lot of base level content in here and I've kept the use of classes to a minimum because, again, we're only really looking for the site's core at this point. The reason we want to focus on this core is it gives us a large set of elements that we can anticipate using in our final product.

If we modeled our core around, say, just a sample article, we couldn't possibly anticipate all the content types we would need as the site grows. For instance, one article might only need some paragraphs and an image, but another might need unordered lists and inline bold elements. When building a core for a site, it's best to anticipate the various use cases early.

Now we have a lot of unstyled markup without any CSS to govern its layout. Without CSS in place, our content is going to look pretty rough. Take a look for yourself in figure 6.1.

As it stands, our page looks pretty raw, which is fine, but stuff overlaps and the layout needs to be put in place. We're going to use a little CSS to get everything in place and then we can start making some typographic decisions.

```

section[role="main"]{                                #A
  padding: 70px 10px 0;                                #B
}

figure{width: 100%;}                                #C
figure img{width:100%; height:auto;} #C

```

**#A** Here we are using an HTML attribute of role because the content of this section is the "main" content of the document. Using role has value in accessibility. It's also useful as a CSS selector because it holds a higher value than a class and its stands out from IDs and Classes in the document, making it easier to identify.

**#B** We're giving a little padding to the top and sides of the section wrapper. The padding on top is to off set the header, which is positioned absolute and removes it from the DOM order. Elements positioned absolutely are always positioned relative to their parent element and according to assigned top, bottom, left, and right coordinates.

**#C** I like to set this as a base rule in my CSS, because figure only makes sense semantically as a wrapper for images. Since we want fluid images all img tags within figure tags should scale to their

parent. I can use classes on the figure to change whether these are half width or full width, or floated left or right.

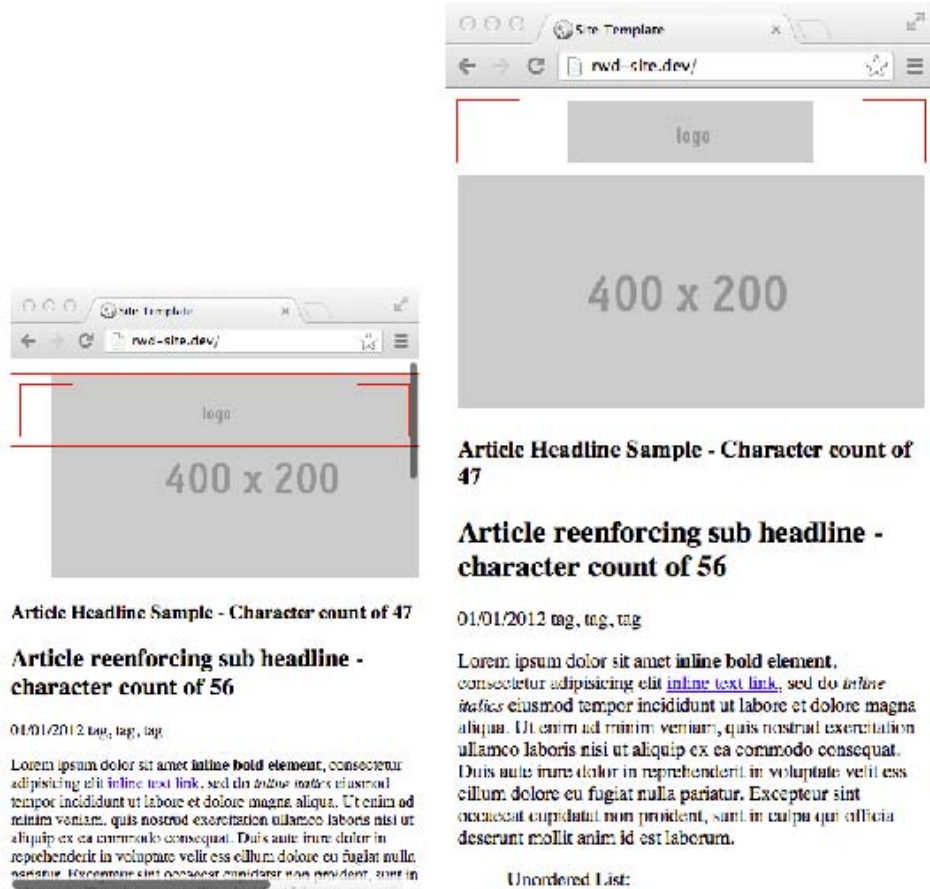


Figure 6.1 (Left) Our content module is full and in place (Right) Our content is now in place.

Now that we've put our content in place and the header and content is no longer overlapping, let's apply some rules so we can govern its positioning. We're going to give it a light grey background and then apply some sizing rules.

To review, our markup looks like this:

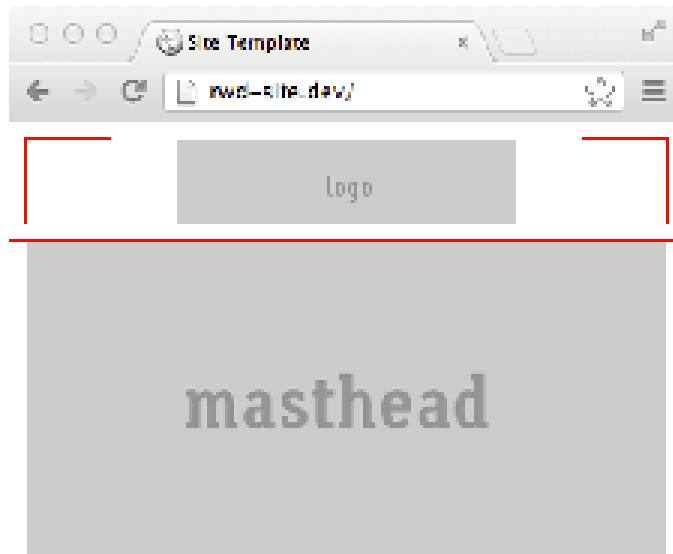
```
<aside class="article-data">
  <time>01/01/2012</time>
  <span class="tag-cloud"><a href="#">tag</a>, <a href="#">tag</a>,
<a href="#">tag</a></span>
</aside>
```

So to position the elements, we'll give it this:

```
.article-data{
```

```
width:30%;  
padding:5%;  
margin:0 5% 5% 0;  
  
float: left;  
  
background:#ccc;  
}  
  
.article-data time{  
    display: block;  
}
```

This produces this in the layout:



**Article Headline Sample - Character count of 47**

**Article reenforcing sub headline - character count of 56**

<p>01/01/2012</p> <p><a href="#">tag</a>, <a href="#">tag</a>, <a href="#">tag</a></p>	<p>Etiam ipsum dolor sit amet, <b>inline bold element</b>, consectetur adipiscing elit. <a href="#">inline text link</a>, sed do <del>ut</del> <del>ut</del> <del>ut</del> eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.</p>
--	--

Figure 6.x Our sidebar in place. Notice how the sidebar floats off to the left side of the content block. As we expand this view we can expand that sidebar, but having it float in line gives us room for flexibility in the mobile view.

We now have our content module laid out and we're ready to start moving our scale up. This process will be similar to the ones used in previous chapters, when we adjusted the layout elements for large screens, or like in the previous chapter where we adjusted the grid layout, so we won't go back through the layout tweaks for bigger viewports. What is important here is how the content looks in the scaled up viewports, specifically the typography.

## **6.2 *Typography in responsive design***

Typography is an absolutely crucial part of all of web design. More of the web's content comes in the form of the written word, and more often than not the final written content is unpredictable.

One of the most common problems is that a site is designed to house a very finite amount of content. If design takes precedence over function and content, a site can be prone to breaking. For instance, a headline is mocked up in a layout within a set box. A designer might draft that box around some placeholder copy and mock it up for mobile, tablet, and desktop views, but the placeholder copy is always the same size. Often times this leads to having to adjust the layout to accommodate 3 lines of text where only 2 were mocked up in the art work, or 1 line of text leaving a big open space where the following two lines should go.

When designing content areas, this is absolutely crucial to keep in mind. One of the ways to combat this is to set a character count, but this can be cumbersome to maintain. Imagine CMS builds where every text input had a minimum and maximum text allotment to keep the page consistent. It could take days of work and in the end the client may find these constraints "too limiting".

Because content is always being generated and therefore always evolving, producing prototypes and style tiles, then layouts, then applying the design is so much better. In this section we're going to go over a few ways you can design content for the web and start to design the visual identity of the site in its early forms.

### **6.2.1 *Embedding typefaces***

Earlier, we set some typefaces in our style tile, "Nunito" for body copy and subheads and "Lori" for headlines. In order to render these fonts on the page, let's take advantage of Google's Web Font's service. It's free to use and hosts a wide variety of typefaces. There are other services available that charge for use, but for our purposes we'll stick with Google.



## Designer Insight: Web fonts

Web fonts offer a lot of options for designers and can make a site look great, but are also easy to abuse. There's a habit to use too many fonts, or fonts that aren't available or licensed for web use.

First, we should find our fonts in Google's library. This is as simple as visiting [google.com/webfonts](http://google.com/webfonts) and inputting the font name into the search field on the left.

Once we've found the proper typeface, we want to include style variations. View the variations by clicking "See all styles" and selecting the styles you want to include. In this instance, we want to include "normal" and "bold" styles. Once we've included the styles, we simply click the "Add to Collection" button.

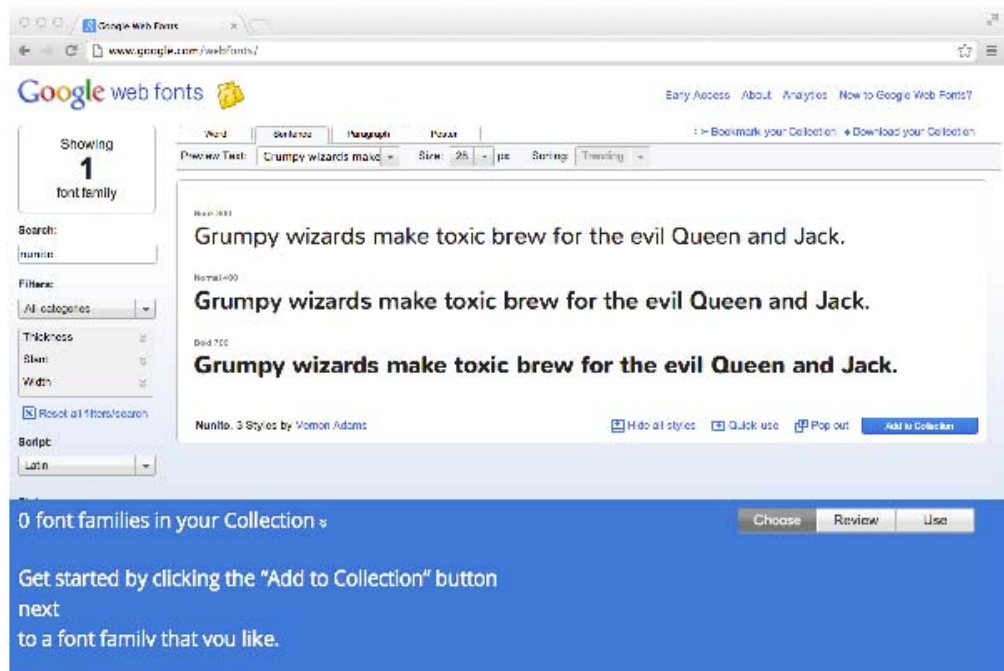


Figure 6.x Selecting styles to use. Be careful to only add the styles you need. Browsers do a good job of italicizing text, but I do recommend using a bold style if it's available, since font weight can render awkwardly in the browser.

Do this process again for the "Lora" typeface and we should have our collection. With Lora, we only want the bold style to be added, since we will only be using it for headlines.

Once we're done, click "use" and proceed to get the code needed. In this screen we can adjust the typefaces in use to reduce page burden. A little farther down we find the link to include the fonts on our page. It should look like this:

```
<link
href='http://fonts.googleapis.com/css?family=Lora:700|Nunito:400,700'
rel='stylesheet' type='text/css'>
```

With this our fonts are ready to be used on the page. We can now call the fonts in our CSS and start setting the typographic core for the site.

### Developer Insight: CSS file structure

In the code samples all of this work is done in a single stylesheet. This is simply a result of keeping things in a single place for your reference. Personally, I like to use separate files for my typographic base and core layout styles. I use SASS (which you will learn more about in a later chapter about CSS preprocessing) to merge these files into a single, minified, stylesheet.

I also like having all my CSS required for the mobile site to be in a single style sheet, which is the only stylesheet I serve to mobile devices in order to keep the mobile load time as efficient as possible.

## 6.2.2 Setting a typographic base

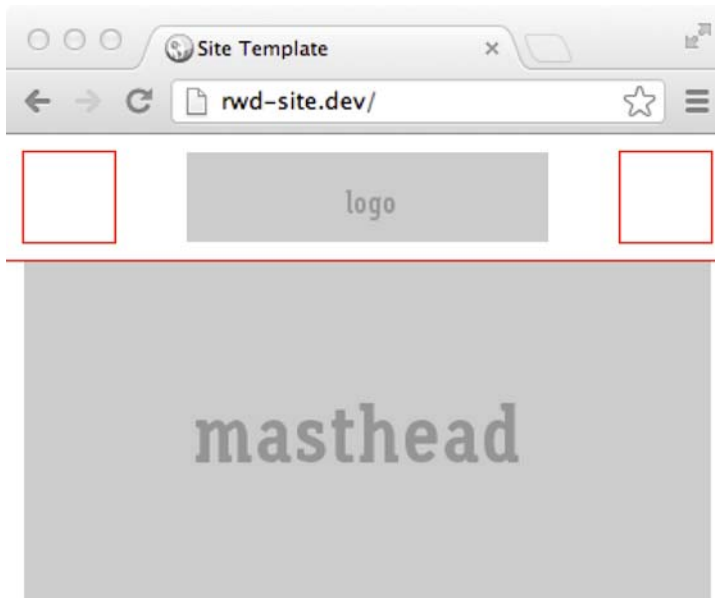
Since content is the absolute core of our page setting the base for our sites typography is a crucial step. In some cases you might find that your core content is video, or images, but even in these cases the web requires a lot of written content. Setting the type is like setting the base melody in a symphony. Once you have your melody in place, you start building upon that until the final work reveals itself. The base may change at times or require some tweaking, but it serves as a nice little foundation on which the site grows.

Let's start with adding the appropriate typefaces to the high level selectors. Since almost all of the copy will rely on the "Nunito" typeface, we can set that as our body font with this line of CSS:

```
body{font-family:'Nunito', arial, sans-serif ;}
```

Now, we simply override this style on our H1 tags with this:

```
h1{font-family: 'Lora', Times New Roman, serif}
```



**Article Headline Sample - Character count of 47**

**Article reenforcing sub headline - character count of 56**

<p>01/01/2012  <a href="#">tag</a>, <a href="#">tag</a>, <a href="#">tag</a></p>	<p>Lorem ipsum dolor sit amet  <b>inline bold element</b>,  consectetur adipisicing elit  <a href="#">inline text link</a>, sed do <i>inline italics</i> eiusmod tempor  incididunt ut labore et dolore magna aliqua. Ut  enim ad minim veniam, quis nostrud exercitation  ullamco laboris nisi ut aliquip ex ea commodo  consequat. Duis aute irure dolor in reprehenderit in  voluptate velit ess cillum dolore eu fugiat nulla  reusitatur. Excepteur sint occaecat cupidatat non</p>
--	--

Figure 6.x Our web fonts in action. As you can see in the above visual, the site is beginning to take on a little more personality.

Now that we are using our fonts from Google, we need to set our sizes. Earlier in Chapter 1 we discussed the use of em instead of px for font sizing. Let's put this knowledge to use here.

We want to find an easily readable size for our type. 20px is a good starting point. We can set the font-size on the body tag to 20px, this way we have a base to start with and we can adjust from there.

```
body{
  font-size: 20px;
  font-family:'Nunito', arial, sans-serif ;
}
```

We also know that our h1 tag should contain our biggest font on the page (this could change according to different content modules, but just bear with me on this one). Remember, when setting em sizes, 1em is always relevant to the parent elements font-size. In this case 1em is 20px. After some dabbling in our mobile view, I've found 1.75em to be a nice starting point.

```
h1{
  font-size: 1.75em;
  font-family: 'Lora', Times New Roman, serif
}
```

From here, we can start balancing out our headers. The header tags should get incrementally smaller, ending with an H6, which I like to make smaller than the body copy.

```
h2{font-size:1.6em;}
h3{font-size:1.4em;}
h4{font-size:1.25em;}
h5{font-size:1em;}
h6{font-size:0.8em;}
```



## In Article Headline 1

## In Article Headline 2

## In Article Headline 3

## In Article Headline 4

### In Article Headline 5

Lorem ipsum dolor sit amet, consectetur  
 adipiscing elit, sed do eiusmod tempor  
 incididunt ut labore et dolore magna  
 aliqua. Ut enim ad minim veniam, quis  
 nostrud exercitation ullamco laboris nisi  
 ut aliquip ex ea commodo consequat.  
 Duis aute irure dolor in reprehenderit in  
 voluptate velit esse cillum dolore eu

Figure 7.11 Our headlines is now easily readable and size down nicely.

With that our typeface have scaled down, but with only one minor adjustment. It's important to keep in mind as you go about building your site that if, say, you want a smaller headline, you resist the temptation to simply scale down the tag using CSS and try using the proper tag for the size you want.

This also applies to creating classes for modules. Once you have a base, you might find that in another module you would want a smaller typeface, say for instance some thumbnails at the bottom of the page. We can keep our site styles consistent and change the details of some copy using a class applied to the parent element.

## 6.3 Summary

In this chapter we added an adjacent content module and propagated it with content. We also discussed how to build a base typographic palette and include some sizing and styles. In doing this we've prepared our canvas for our design.

We started with a simple prototype and a style tile and now we've got something that's starting to look like a website. With a little more work we'll have something that looks even more like a website and after that it'll get even closer to looking like a web site. In our next chapter we'll start adding graphics and apply more of our visual brand to our page.

# 7

## *Adding graphics in the browser with CSS*

In this chapter we will learn how to:

- Learn to use CSS as a design tool
- Maintain proportions in a responsive site
- Add responsive media
- Use SVG in modern sites

It was tempting to title this chapter “Designing in the browser”. This is the phase where we would execute what we might commonly understand as being the “design” phase, but as we’ve discussed, design is a huge oversimplification of all that needs to go on here. We’ve already designed the site. We’ve discussed the user interface, content, visual brand and identity; everything is ready for us to start crafting the presentation layer.

In Chapter 2 we used Photoshop to create a presentation layer for our site. This was to show how to take a familiar process and apply it to mobile design. In this chapter, though, we’re going to execute a lot of those ideas, but with one huge variation, we’re not going to use Photoshop. What we’ll use? Cascading Style Sheets (CSS)..And with CSS3 we can add gradients, round corners, drop shadows (on text as well as elements), color models for not just RGB colors but also alpha transparency, and opacity. In addition to features that help enable beautiful design, CSS offers the ability to animate between states, and because it renders natively in the browser it also looks sharp and vibrant in high DPI screens, such as Retina Displays. By using this to your advantage you can avoid the messy business of detecting high DPI displays and serving alternative files in those browsers.

## 7.1 Using CSS to implement design

In getting started, let's first refer back to our Style Tile. We'll use our style tile from Chapter 3 (Figure 7.1) as a guide in making decisions regarding color and visual appearance.

### Developer Insight: Designing in phases

For most developers designing web pages is a boring and intimidating task. An understanding and respect of web design basics is crucial to success in front end development, but it's important to not get too hung up in trying to perfect a web design yourself. The process described in this book is intended to be highly iterative and collaborative.

In my daily work I find myself relying on other people for their expertise and they rely on me for mine. This is the nature of collaboration and it's important to keep that in mind. Building web sites and applications is more of a team sport than ever.



Figure 7.1 The style tile from chapter 3 serves as a guide in our design.

Using the style tile as a guide we can start with the basics, such as a color scheme and some patterns or background texture. While we will still use some small images to create background patterns with just a little CSS we'll produce this:



Figure 7.2 The beginnings of a designed page.

Here we've started to give a few small tweaks with some basic CSS, but already the design is starting to come together. A lot of the techniques used here are common like using background images for the textures and patterns and adding color to the typography, but there are a few tricks hidden in the CSS that are extremely helpful in building responsive websites.



In this section we'll learn a few of these helpful tips, such as maintaining aspect ratios in images and videos as well as using sprites or icon font families for a user interface. The current state of the CSS in the example can be found within `sample-001.css`.

The first issue you may face is how to display an image with a consistent proportion or aspect ratio, let's see how CSS can solve the problem.

### ***7.1.1 Maintaining proportions in a fluid structure.***

In a scalable and responsive web site, a lot of times images or media elements height and width need to maintain a relationship. A 3 by 4 image might need to be displayed at a width of 200 pixels on one device and 400 pixels wide in another, but still stay the same size. One way of doing this with images is by declaring a height of auto, but what about elements that don't have an automatic height, such as an element with a background image?

There's a helpful trick in CSS to help with this. In our current sample we're using it on the logo. In the mobile screen size we want the logo to display at a width of 100 pixels wide, but as the screen affords more size we want the logo to increase in height as well as width. This can be done in a similar way to the fluid images example we discussed in Chapter 1.



Figure 7.3 Our logo should maintain a consistent relationship between width and height.

We'll start with the mark up:

```
<div class="logo-wrapper">
  <h1 id="logo">Matthew Carver</h1>
</div>
```

We have an h1 tag that we'll be using to display our graphic logo with a wrapper to constrain our proportions. The logo being shown in the h1 should be able to scale proportionally with the width of the parent element

```
.logo-wrapper{
  position:relative; #A
  width:100px;
```

```

    left:50%;
}

#logo{
    position:absolute; #A
    overflow:hidden;
    width:100%;
    height:0px;
    padding-bottom:44%;#B
    font-size:0px;
    background: url(/images/logo.png) no-repeat;
    background-size:cover;
}

```

**#A Positioning is the key to making this trick work. Positioning relative is essentially a default positioning, but the child's positioning (#logo) is given in relation to it's parent.**

**#B The percentage based padding of an element is always relative to the width of it's parent, which is how it can be used to draw a relationship between width and height.**

The trick we're using here is a few layers deep. First, we set the parent to the desired width. In this case we want a set 100 pixels, but we could just as easily use an em or percentage value. We also need to set the positioning to "relative". This is important because we need the ability to set absolute position to the child element in order to maintain the sizing.

The trick we're exploiting here is one in which padding is relative to the width of a parent element. Percentage based height is always relative to other heights. Percentage based padding is always consistent, whether top and bottom, or left and right, and always relative to the parent element. Therefore a padding bottom is relative to the width of a parent element. We can exploit this to our advantage in responsive websites.

Another place this CSS trick comes in handy is with video. Videos almost always require a set aspect ratio (see figure 7.4).



Figure 7.4 A responsive video in place.

This is easily applied to a video by changing the HTML tags used.

```
<div class="video-wrapper">
  <iframe width="560" height="315"
src="http://www.youtube.com/embed/9bZkp7q19f0" frameborder="0"
allowfullscreen></iframe>
</div>
```

```
.video-wrapper{
  position: relative;
  padding-bottom: 56.25%;      height: 0;
  overflow: hidden;
}
```

```
.video-wrapper iframe{
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
}
```

Using the above HTML and CSS, the YouTube video will scale nicely between views. The video iframe will scale fully from right to left and maintain a proper aspect ratio.

Another handy way to scale images is with a single CSS rule called `background-size`. With this rule an image applied to the background of an element is scaled to fit the element's size. It works much like `background-position`, in that you can specify unit sizing, but it also enables the declaration of "auto" with scales to the proportions of the source element.

Say, for instance you had a 100 pixel by 200 pixel background image. If you applied it as a background image and used `background-size` to scale it down, you could declare the following:

```
background-size: 50% auto
```

The 50% would be applied to the image's native horizontal dimensions and then the height would be scaled automatically. Therefore, the image would be displayed as 50 pixels by 100 pixels. With the logo now maintaining a proper aspect ratio with the ability to scale between viewports, let's look at some ways we can add the user interface graphics to the page.

## 7.2 Using icon fonts in your design

With CSS3 we can load custom typefaces into our designs, as we discussed in Chapter 6. Leveraging this ability for typography has obvious advantages in creating unique designs in the browser, but it can also be exploited to our advantage to create beautiful user interfaces.

### 7.2.1 User interface sprites

Typically user interface design is accomplished by creating an image called a *sprite*, which is a collection of images put into a single image, and displaying the part of the image in that sprite as it's needed. A single image is used as the background for an element, and the portion displayed is aligned by using `background-position` (see figure 7.5)

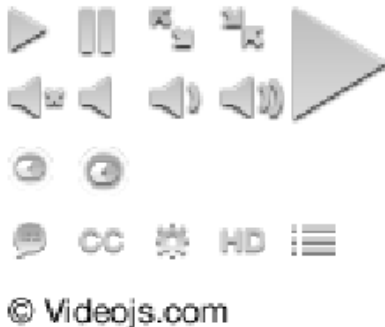


Figure 7.5 An example of a commonly used image sprite, this one from Videojs.com

This is an effective method traditionally, but in a responsive web site it can be slightly more nuanced. High DPI displays will display the image fuzzier than normal displays and in smaller screens we might want to use smaller or larger buttons, which would require a new sprite for various breakpoints.

The best way around this is to use an icon font face to replace the need for a sprite. Because it's a custom typeface the icons are vector based as opposed to the raster based images in a typical .png sprite. The font face also scales elegantly for various size requirements.

### **Designer insight: Vector images versus rasterized images**

The difference between vector images and rasterized images is an important distinction in responsive web design due to the need to accommodate varied pixel densities. The easiest way to describe the difference is that Photoshop produces rasterized images and Illustrator produces vector images.

Rasterized images are files that contain pixel data and create an image by assigning each pixel a color. This becomes challenging when pixel sizes are skewed by changing requirements. An image that is created at 100 pixels wide looks "pixelated" when it's stretched to 200 pixels wide.

Vector images, on the other hand are created by setting points within a file and mathematically drawing lines and curves between them. Because there is no assumed "pixel" data being communicated, the vector images can scale fluidly. Fonts are by default vector based images, but there are other ways of creating vector art for use on the web, most notably the SVG format which we will discuss later in this chapter.

## **7.2.2 Font based user-interface graphics**

In our example site, we're using an open source font library for user interface icons, called Font-Awesome. The library of icons is available for free use and is also available through public a Content Delivery Network (CDN) such as the Google CDN we used earlier for our custom typefaces. The CDN I'm using for Font Awesome is through cdnjs.com.

Here I've used Font Awesome to create the information icon and the four bar navigation icon (figure 7.6).



Figure 7.6 Our site's UI, at the top of the page, is generated using a typeface instead of images.

This is very simple thanks to the hard work of the people behind Font Awesome. The first thing you do is import the CSS library into our page.

```
<link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/font-awesome/3.0.2/css/font-awesome.min.css" type="text/css" charset="utf-8">
```

After that add a few classes to the buttons.

```
<span id="infoTrayBtn" class="icon-info-sign btn"></span>
<div class="logo-wrapper">
  <h1 id="logo">Matthew Carver</h1>
```

```

</div>
<span id="navTrayBtn" class="icon-align-justify btn"></span>

```

Then use the .btn class to style the buttons in the same way you would style any other type on the page.

```

.btn{
  font-size:2.2em;
  color:#2e3034;
  text-align:center;
}

```

In just a minute or two you've created a simple user interface that will scale nicely and look gorgeous on high-density displays. This is mostly thanks to the fact that font-faces are rendered on the page using scalable vector graphics, or SVG, but what if we wanted to leverage this advantage but we can't find an icon font to meet our needs?

Scalable vector graphics or SVG for short can currently be used in place of traditional background images in most browsers now and can offer us the same high quality graphics while maintaining a custom design.

### ***7.3 Scalable Vector Graphics***

Scalable vector graphics are an XML based image format that has been an open standard since 1999. It hasn't been until recently that the file format has been accessible in modern browsers but it came in at the perfect time.

SVG gives designers the opportunity to create images that remain sharp and beautiful at any scale and in any display. They can be zoomed in on and retain their crispness or scaled down and retain their detail. Take a look at figure 7.7.





Figure 7.7 This star was created with a raster based imaging program, zoomed in at 400% original size.



Figure 7.8 The same star created in a vector based imaging program, also zoomed in at 400%.

Creating an SVG can be accomplished through Illustrator or Photoshop or a myriad of other programs. I use an application called “Sketch” to create vector images. Any vectors created in Photoshop will have to be exported to Illustrator because Photoshop lacks the ability to save for web as an SVG. Once you’ve created your graphics you can save the files as SVG for use on your page.

Another great feature of SVG is that Illustrator can open and edit an SVG and save it without having to export every time, meaning changes to graphics on a page can be made quickly and easily by designers without having to update any CSS.

### 7.3.1 Adding an SVG to a page

SVG can be added to a web page one of two ways, either as an object or as a CSS background image. An object is displayed on the page itself, a CSS background image is applied as a style of an object. This is a minor, but important distinction.

To display our star on the page, it's as simple as linking to the source file on our page. In the index.html file, simply drop a little mark up and the image shows up in the browser (Figure 7.9).

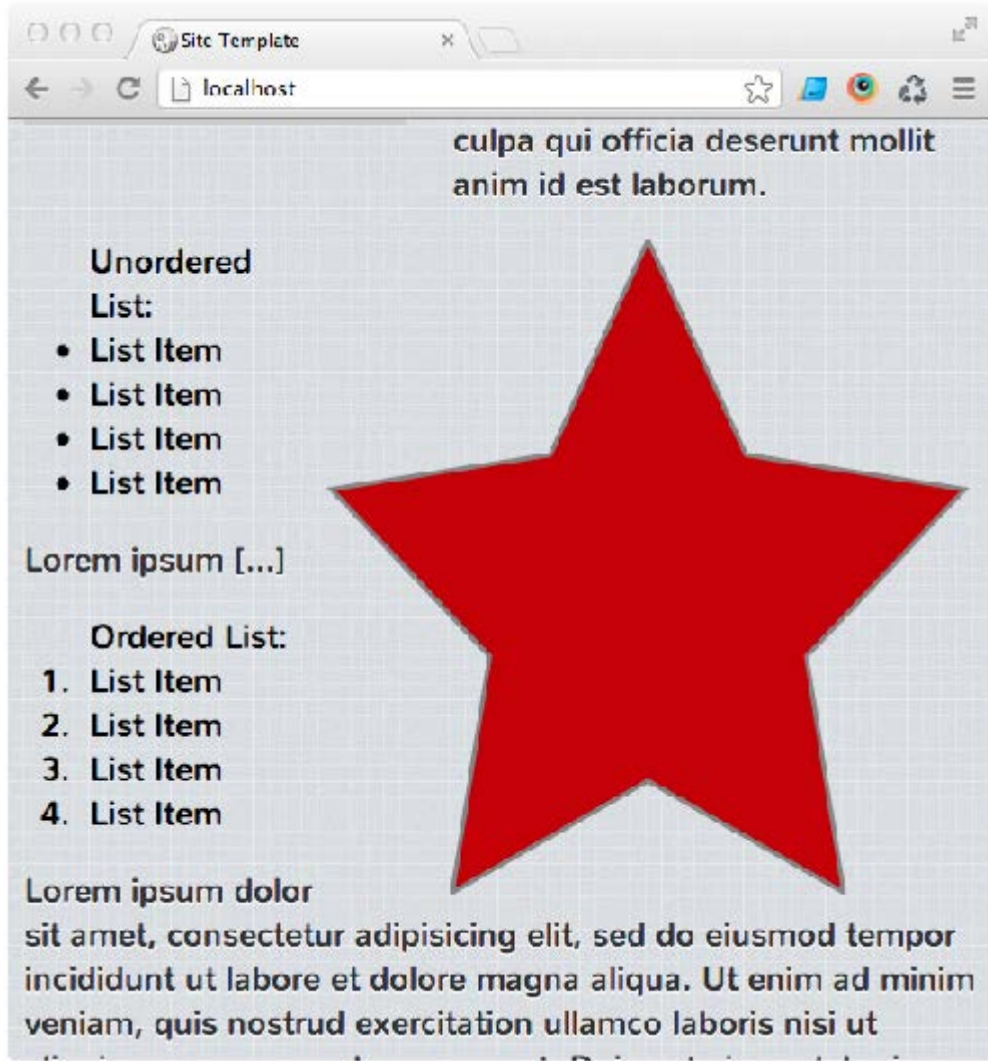


Figure 7.9 Our SVG on the page

Embedding an SVG is as easy as using the `<object>` tag and applying a few attributes. When using an object tag to embed an SVG, you use the "data" attribute to assign the svg file. This will display the SVG within the object tag.

```
<object type="image/svg+xml"
        width="400" height="400" style="float:right"
        data="images/star.svg">
</object>
```

As you can see in the above example, the image retains transparency much like a .png file and is scaled up to 400 pixels by 400 pixels, as specified in our object tag. If I wanted to scale this image down, I could simply change the width and height and the image would still look beautiful (figure 7.10).

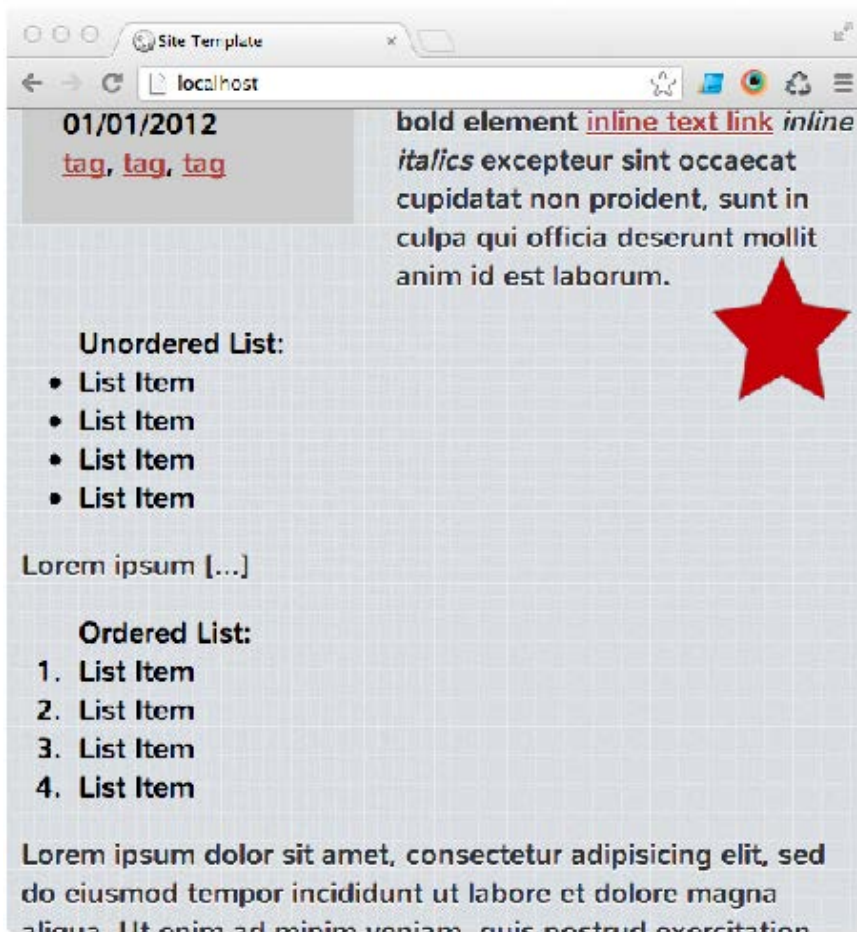


Figure 7.10 The star scaled down.

The sizing of the SVG can be scaled using the width and height attributes. It can also be scaled with CSS.

```
<object type="image/svg+xml"
        width="100" height="100" style="float:right"
        data="images/star.svg">
</object>
```

While this is helpful, it's not nearly as impressive as when I scale the image up in figure 7.11.

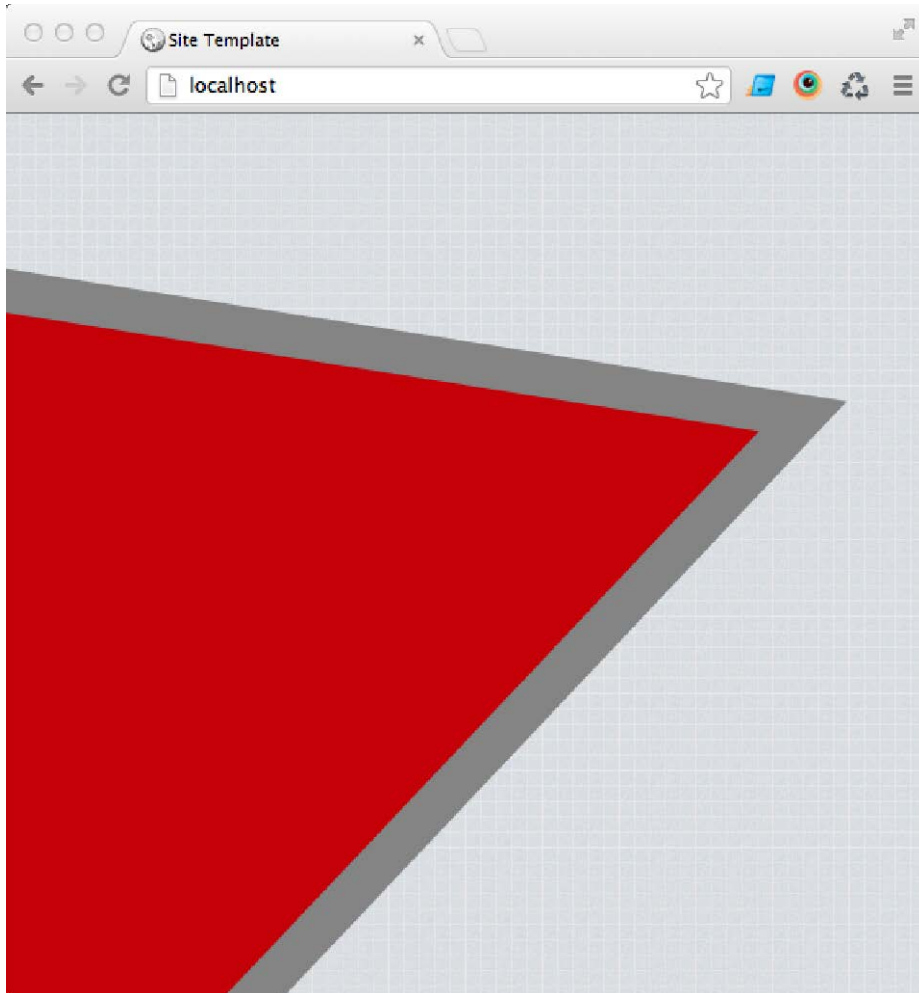


Figure 7.11 An intensely zoomed in SVG star.

Here, we've increased the size of the SVG using the width and height attributes.

```
<object type="image/svg+xml"
        width="4000" height="4000" style="float:right"
        data="images/star.svg">
</object>
```

Now the image is scaled to a ridiculous 4000 by 4000 size and still looks wonderful. Notice that the file hasn't changed, so the file size is the same, meaning the user isn't burdened with a bigger file to get the bigger image.

### **7.3.2 Implementing SVG with CSS**

Being able to add SVG to the document is helpful, but using them in CSS can be much more useful at times, such as when you want the svg to scale with an object on the page. SVG should be the graphic standard of choice when designing a user interface because of its crisp appearance and its scalability. This is easiest when done in CSS.

A great example of how to do this practically is by replacing our logo with an SVG. In order to do this let's create an SVG file using Sketch and export it into the images folder. This could also be done with Illustrator or any vector editing program. All we do is replace the PNG with an SVG file.

```
#logo{
  position:absolute;
  overflow:hidden;
  width:100%;
  height:0px;
  padding-bottom:53%;
  font-size:0px;
  background-image: url(/images/logo.svg);
  background-size:cover;
```

And what we get is figure 7.12.



Figure 7.12 Our logo is now an SVG

```
}
```

Let's say, hypothetically we could use just a small form of the logo for this site, but in larger views we may want the full logo. This can be done without changing the background image, and simply using background-size.

```
#logo {
  position: absolute;
  overflow: hidden;
  width: 131px;
  height: 0px;
```

```
padding-bottom: 53%;
font-size: 0px;
background-image: url(/images/logo.svg);
background-size: 242px auto;
}
```

We've now scaled up the logo without compromising image quality. This is how you might want to handle using a sprite with SVG. Background-size changed the scale at which the image is rendered without affecting the element the image is rendered within (figure x.x)

### 7.3.3 Limitations of the SVG format

Working with new technology always has its consequences and the SVG format is no different. While it gives designers and developers a much higher degree of quality, it comes at an expense and requires some foresight to overcome these expenses.

#### FILE SIZE

The most obvious of the consequences is file size. The star example comes in at just a handful of bytes, but the logo is 70KB, which is pretty large considering its PNG equivalent is only 9KB, but these figures are a little misleading. This issue can be overcome with server side gzip compression, but it's still important to keep note of how many and how large the files being loaded are.

#### SINGLE COLOR FILES

Another complication is that SVG only truly works well with monochromatic, or single color, files. It's great at applying gradients and colors to shapes, but complex images can come at an expense in file size. This is why SVG is most useful for user interface elements or site iconography.

#### THE SERVER

There is also the issue of ensuring that your server will serve .svg files. This is as simple as adding a few lines to the servers .htaccess file.

```
AddType image/svg+xml svg
AddType image/svg+xml svgz
```

This ensures that your server is configured to read the SVG file format.

#### BROWSER SUPPORT

The other issue, and probably the most problematic is browser support. SVG is supported by all modern browsers, except Internet Explorer. This can be troublesome, but is easily solved by creating a .png file as a fall back when exporting your .svg file. Instead of simply serving the fallback through the typical browser detection, I recommend using feature detection to accomplish this. In the next chapter we will cover feature detection at a greater length.

## 7.4 Summary

In this chapter we discussed using modern techniques such as CSS3, font-faces, and SVG to create and implement beautiful responsive websites.

We talked about creating scalable background images and videos. Using these techniques, images can scale fluidly in a responsive environment. We applied the same principles to creating video elements that scale with a page.



In the case of designing the user interface of a site, we discussed using icon based font families to create our buttons and UI elements. By using an icon font we can scale up beautiful vector based graphics using CSS.

Finally, we learned about the benefits and implementation of scalable vector graphics, or SVGs. We learned about the features of SVG and how to display them on a page and use them in CSS. We also discussed the inherent limitations that come with a progressive format.

In our next chapter we'll discuss ways to overcome some of cross browser issues that arise with using modern CSS by discussing Modernizr, a javascript library that thwarts this issues by basing our CSS on feature detection as opposed to assumptions based on browser detection.

# 8

## *Progressive enhancement with Modernizr*

In this chapter, we:

- Introduce responsive experiences.
- Introduce progressive enhancements.
- Dive deeper into Modernizr.
- Use Modernizr to progressively enhance an element.

Building responsive web sites is a practice of building in an extra dimension of depth to your pages. As we've discussed previously this additional depth is initially communicated and immediately evident as screen size, but as is obvious to anyone who uses multiple devices to access the internet on a regular basis there is more variation than that. We've also discussed the nuances between touch interfaces and mouse pointer interfaces, which requires new thinking in creating interface elements.

In so much as the old way of creating websites relied on the a dependable screen size, it also relied on dependable input. User's input data with keyboards and a mouse. It's easy to forget that the iPhone was not the first "smart phone" or internet enabled phone. There were many predecessors, from the Sidekick to the Blackberry. They had limited success, but the notion of a cellphone as a small computer was not unique before the iPhone, it just wasn't as successful. What made the iPhone a success was the software keyboard and multi-touch screen. When people talk about the legacy of Steve Jobs and how the iPhone was a revolutionary invention, its not because Apple released a phone with a web browser, it's because the interface was actually usable. The device just worked.

While the introduction of touch interfaces and small screen devices is really interesting and can be fun to design and develop for, it's merely the tip of the responsive iceberg. Beyond building sites to work with the latest cellphones, the beauty of the web is not just it's ability to give you sports scores while you wait in line at the coffee shop. The beauty of the web is its ubiquity and its ability to serve people content and information at any time. While in the United States we tend to have access to modern cell phones and tablets, building a responsive website means also supporting those old devices too as well as making content accessible to sight impaired users or users with crippled hardware.

---

### **The future of interfaces**

Currently touch screen has been a leap forward in computer interface technology. On a consumer level, it's become common and a part of daily life. Windows 8 is an entire OS built on the concept of the laptop with a touch interface. While this is incredible and poses huge challenges for us as developers, it's merely the beginning.

Recently I was in a brainstorming meeting for a marketing campaign for a large scale web content channel and we were trying to come up with ideas for an installation. One that we came up with is the idea of building a "Tony Stark" like interface using three dimensional space and projections to create an interface that responded to a users gestures, voice commands, and personal preferences. The user would experience the interface in full stereoscopic 3D with the use of a VR headset. When asked if this is something that could actually be built my answer was a confident yes.

Right now there are products such as the Leap Motion that can produce interactions based on special awareness, making three dimensional gesture controls a reality. There is also devices like the Oculus Rift, a virtual reality headset that uses two high resolution displays worn on the face in a form factor similar to ski goggles.

While nothing is certain and it's hard to say what exact impact this will have on the web, the technology is getting close to being a reality and we may find that in 6 or 7 years there will be a need for media queries to detect VR headsets.

---

In order to serve these users we first need a workflow that gives us the proper precedence for tending to all of the needs of our users. We've talked about designing and developing for mobile first and building up from there, but let's take that a little further. What if instead of just throwing HTML, JS, and CSS at a browser until we get what we want out of it, we build from a foundation and work our way up?

So far we've talked about starting with prototypes, then style tiles, then content, then building the site, now the final part of our responsive site is a technique called "progressive enhancement"

## 8.1 *What is progressive enhancement?*

Progressive enhancement is a technique of enhancing a site from it's lowest point into something that is more technologically advanced. This process is kind of a counter point to mobile first design. Mobile first design deals directly with form, progressive enhancement deals with function. When dealing with the functions of a site, we first have to conceptualize a model for figuring out how we can enhance our site. This way we've discussed up to this point is by using media queries, but let's break that down and see how we can take this mobile first concept a step further using progressive enhancements to create a responsive experience.

As Mark Boulton described in a blog post titled "A Responsive Experience", these sorts of experiences are achieved by use of three conceptual components working together: sensors, systems, and actuators.

- **Sensors** are components that sense the environment. In short, our browser. The browser gives us valuable information about our user and their current capabilities.
- **Systems** are the responses to the information provided by the sensor. In previous chapters we've discussed media queries. In this context, media queries are a form of system. We could also use JavaScript to build other systems to tell us other information, such as the user's time or geographic location.
- **Actuators** are the responses to the system instructions. This could be CSS applied within a media query, a JavaScript plugin that gives the page improvements based on the user's capabilities. In previous chapters our actuators have been the enhancements or adjustment's we've made for larger viewports.

Previously we've talked about using the browser as a sensor and basing our systems off of one piece of feedback and adjusting accordingly, but we know that there is a lot more to building responsive websites than width. In the example we've been working with, we've built around the changes in the width of a site. This is just the first and easiest to understand way to understand the differences between mobile, tablet, and desktop sites.

In addition to media queries for sensing the form of a site, we also have a very base level sensor we can use, and that is the lack of JavaScript. The overwhelming majority of web traffic uses JavaScript to render webpages, but with that being said, whether or not JavaScript is enabled should be the first sensor you use when progressively enhancing a site, that is to say every site you build should be able to be navigated, even to a simple degree, without JavaScript enabled.

Beyond this first sensor of whether or not JavaScript is enabled, we need sensors that give us feedback to adjust sites based on other factors.

Historically in front-end development developers would adjust a site to fit the constraints of the layout engine being used by a particular browser. There were only 3 browsers, Netscape, Internet Explorer, and Opera in 2003. Firefox, Safari, and the first mobile browser Opera Mini, were released by 2005 and Chrome wasn't released until 2008.

Currently there are 5 major browsers, each with it's own mobile version. Across that array of browsers there are also older versions that user's haven't upgraded. In the same way that

creating multiple layouts for multiple screen sizes eventually becomes a zero sum game, so does building multiple front-ends for multiple browsers.

We've talked a lot about how to build a site for new, cutting edge mobile browsers. We're using responsive web design to accommodate these browsers, but while accommodating new browsers it's important to not do so at the expensive of older browsers.

Progressive enhancement is one strategy to cope with browsers failures to support up to date features. As we've been discussing in this book, there is a temptation to build for the most up-to-date features available to us but in a responsive web, a design is contextual. The design of a site is contextual to the frame that it's being viewed through. Responsive web design has become popular because it resolves the most obvious changing context, the available screen real estate, but the context of a browser runs much deeper than it's viewport size.

In the previous chapter we discussed using SVG images in a site's design. SVG makes for a great solution for high resolution displays, but what about it's support in older browsers? It's not supported in IE 8 or lower, so we have to build in a fall back if we support that browser. We could identify the browser and serve alternative styles against that browser but then you would have serve those same alternative styles for every browser that doesn't support SVG.

Wouldn't it be easier if you could just write a style once that would be used against every browser that didn't support SVG? That way you wouldn't have to keep updated about every browsers feature set or be surprised when a user reports a bug from an older version of a browser you didn't think anyone used. You could just set the fallback once and forget it. This is where Modernizr comes in handy.

### **Designer Insight: Progressive enhancement in design**

Progressive enhancement is implemented in development, but it's important to consider ways to design solutions around feature support. In some cases this might require a module on a site to be completely replaced and supplemented with a module that accomplishes the same task. Be ready to provide design support for the fallback modules.

Traditionally, feature detection has been accomplished by detecting the browsers user agent. This is done through Javascript, using the navigator object. The navigator object dates back to the Netscape days and used to be a developers best tool in cross browser compatibility.

If you have Chrome or Safari available, try opening your browsers web inspector by right clicking on a page and selecting "Inspect Element". After you've opened the web inspector, click on "Console" and after the caret type "navigator.userAgent" and hit enter.

This will return your current browser's user-agent, which is a string of text used to identify the browser in use. For example, I'm using Google Chrome, which returns the following:

```
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.35 (KHTML, like Gecko) Chrome/27.0.1443.2 Safari/537.35"
```

In a lot of ways, the navigator object is the one of the best sensors we have available to us to inform our system about what our user is capable of, however it's not very future friendly.

It bases your site's actuators on a lot of assumptions about what the browser does and does not support. It is also unreliable because it can be configured by the user specifically in order to access sites that the browser might not be able to support.

### Developer Insight: The Navigator object

Although navigator falls apart when trying to make assumptions about the user's feature support, it can be an incredibly handy sensor. In addition to the user agent, it can be used to detect geolocation, whether cookies are enabled, and even where the user is in the world. If your console is still open, try inputting this:

```
navigator.geolocation.getCurrentPosition(function(position) {
  console.log(position.coords.latitude, position.coords.longitude);
});
```

It should return two values. This is the longitude and latitude of the server your ISP is currently using. Copy the values into a google search and you can see a Google Maps location of where your user agent is broadcasting your location. This can be used as an additional sensor which you can use to create a richer responsive experience.

For example, take my example personal blog. I might want to detect if a visitor is from New York and see if they want to take me out for a beer or two. I could potentially use the navigator's geolocation feature to spring an alert encouraging the user to contact me directly.

Detecting user agents has been the traditional method of applying progressive enhancement. Developers would know that a certain feature was unavailable to a particular browser and build in a fall back or a hack to make the feature work. This was a manageable workflow historically but currently is simply unrealistic. There are just too many browsers to keep track of and new operating systems and browser versions being added daily.

What if instead of supporting browsers, we supported features? Then we could create fallbacks for particular features and browsers that don't support that feature universally get served the fallback while browsers that do will get the more modern implementation. This is the exact use that Modernizr was created for.

## 8.2 What is Modernizr?

Modernizr is a javascript library used to detect features in the browser. It's loaded in the head of your page and runs during a page load. Adding it to your site is as simple as adding any other javascript library, such as jQuery, but once added Modernizer gives an incredible amount of control in rendering your page and ensuring that every user is served a quality experience.

On load of the library, Modernizr runs a series of checks against the users browser to determine what features the browser supports and creates a JavaScript object that you can use

to test against. Modernizr doesn't create support these features, it simply gives you a way to provide fallback support for modern features.

Modernizr also applies a set of classes to the HTML tag, giving you the ability to modify the pages CSS using the corresponding CSS classes. These CSS classes allow you to use CSS systems to build actuators that will adjust your pages to allow the progressive enhancements available for a page.

As opposed to using the User Agent, Modernizr detects features directly by running a series of JavaScript tests that return boolean (true or false) values. This dictates the classes that are set to the html tag, and gives you the ability to use JavaScript to detect whether a feature is available. This is all done out of the box by installing the Modernizr library.

### **8.2.1 Installing Modernizr**

Installing Modernizr is as simple as linking to the JavaScript library in the head of your page, but where the installation process gets complicated is creating the version that you need. Modernizr is available for download through their website (<http://modernizr.com/>) in one of two versions, the development version and the production version.

The development version is a full 42kb, uncompressed file. This version is great if you're well versed in JavaScript and want to make some tweaks to the tests it performs, this is the version for you. Because it's uncompressed, it's easy to read and augment but it's best left to developers with a firm understanding of JavaScript.

For those of you who might not be completely adept at JavaScript, or would like to quickly build a customized version of Modernizr, this is where the production version of the library comes into play. The production version building tool on the site gives you the ability to create a version with only the tests you require.

This comes in handy when you know you only need a certain set of test's. For instance your site might not take advantage of CSS box-shadows, but it might need to support CSS gradients. Using the build tool, so can include the tests you need and exclude the ones you don't, keeping the source code trim and your users total load time down.

For our example, I'll be working off of the development version. I find that when I'm building a site it's best to work with the full version and then once you know what features you'll be using in your site, trim the version down. In the previous examples we've installed the development version by default as well.

Installing Modernizr is as simple as linking to the source .js file.

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

After linking you can simply add a class of "no-js" to the html tag on the page like this:

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

And with that you're ready to go. The no-js class is removed by Modernizr and replaced with the test classes. This way you can add CSS for pages with JavaScript disabled.

### **8.2.2 Using Modernizr for cross browser CSS**

With Modernizr installed we have everything we need in place to start doing some progressive enhancements. We'll start with a raw sample site.

```
<!doctype html>
```

```

<html lang="en">
  <head>
    <meta charset="utf-8">
    <script type="text/javascript"
src="//cdnjs.cloudflare.com/ajax/libs/modernizr/2.6.2/modernizr.min.js"></scr
ipt>
  </head>
  <body>

  </body>
</html>

```

In the last chapter, we used SVG graphics, so let's use this small test to detect whether or not our browser is capable of supporting SVG. For the sake of simplicity we'll just add two span tags to the page to detect support.

```

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <script type="text/javascript"
src="//cdnjs.cloudflare.com/ajax/libs/modernizr/2.6.2/modernizr.min.js"></scr
ipt>
    <style type="text/css">
      .yes{color:green;}
      .no{color:red;}
      .svg span{display:none;} #A
      .svg .yes{display:inline;} #B
      .no-svg .no{display:inline;} #C
    </style>

  </head>
  <body>

    <div class="svg">
      <span class="yes">Huzzah! You have SVG support.</span>
      <span class="no">BOO! You don't have SVG support.</span>
    </div>

  </body>
</html>

```

**#A** Here both spans are given the `display:none`; rule to hide both.

**#B** On page load Modernizr adds its feature detection classes to the html tag. If Modernizr detects that svg is supported, it'll add the "svg" class to the html tag. At that point this span will have the `display:inline` rule.

**#C** If Modernizr detects that svg is not supported, then the html tag gets the no-svg class and this span is visible.

If you test this in a browser that supports SVG, you'll see the message "Huzzah! You have SVG support." whereas if you have a browser that fails to support SVG, you'll find the "BOO! You don't have SVG support." message. You can find this example in the sample-001.html file.

This example is pretty rudimentary, but it displays the core idea of using Modernizr to fix cross browser issues. Modernizr adds the classes to the html tag, offering a way to override styles based on Modernizr's feature detection. If we were doing this same fix using the old user agent method, we would have to have a style sheet for each browser that doesn't support SVG



and change our CSS for each one (for anyone interested, the only major browsers lacking SVG support are Internet Explorer 7 and under).

By adding the `svg/no-svg` class to the `html` on the page, your CSS now has a selector that can be used to override existing CSS rules. Because it's on the parent-most tag it can be used to override other classes on the page. So in this case, both `span` tags are given `"display:none;"` and if there is no SVG support the `"display:inline"` declaration on the `span` tag with a class of `.no` overrides the `"display:hidden"` thanks to the `"no-svg"` rule on the `html` tag.

Let's try a more useful example of the same idea. We might want to have an SVG background image on the page, which falls back to a PNG if the browser doesn't support SVG. By default we'll use the PNG image. This way the SVG isn't served unless it's needed and becomes a progressive enhancement.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <script type="text/javascript"
src="//cdnjs.cloudflare.com/ajax/libs/modernizr/2.6.2/modernizr.min.js"></scr
ipt>
    <style type="text/css">

      .skull{
        width:300px;
        height:300px;
        background-image:url(images/skull.png);
        background-size:100% auto;
        background-repeat: no-repeat;
      }

      .svg .skull{background-image:url(images/skull.svg);}

    </style>

  </head>
  <body>

    <div class="skull"></div>

  </body>
</html>
```

Now we have an awesome SVG skull that will look awesome and crisp for user's with High resolution displays, and still look good for users with older browsers. It does the work of cross-browser capability without having to remember or maintain a running list of which browsers support SVG.

This is excellent for supporting features with CSS, but what about testing a user's capabilities and offering deeper support for progressive enhancements. For that you need to take advantage of some of Modernizr's other capabilities.

### 8.3 Javascript feature detection with Modernizr

Giving you a some CSS hooks to alter the design of a page based on the features a browser supports is a pretty handy feature and, in my opinion, justifies the use of Modernizr in most every project I work on. Ultimately though, this leads to a bit of site bloat as you add progressive enhancements. In many cases site functionality and interaction might also need to be progressively enhanced and JavaScript is the best way to do that.

As I mentioned earlier, in addition to applying a series of classes to the html tag on the page, Modernizr creates a Javascript object with Boolean values applied to it for each of the tests ran. This requires some base level JavaScript ability, but is deeply rewarding and can offer a huge degree of control over what your page loads.

If you have the example page open, open your browsers console and type "Modernizr". This will return the full object in your browser with all of the tests ran and each of their corresponding true or false values.

These boolean values are easiest to weigh against by using a simple javascript if statement. You can access each of the tests as properties of the Modernizr JavaScript object with dot notation. In your example page, try adding the following in your head tag:

```
<script type="text/javascript">
  if(Modernizr.opacity) {
    alert('opacity is supported');
  }
</script>
```

Reload the page and if your browser supports CSS opacity, you'll get an alert saying "opacity is supported". This is a pretty common CSS property, but let's try using a sensor that is a little more relevant to our responsive goals, touch.

#### 8.3.1 Detecting touch support

With responsive sites, it's often not enough to detect viewport. Many times you'll find that site's will assume that a smaller viewport means you have touch screen capabilities. This isn't always the case. Often I'll find myself browsing in a smaller window (say for instance, to hide my personal browsing from any of my co-workers prying eyes).

If I come across a page that assumes I have a touch screen, some elements on the page might be hidden behind touch based systems. Likewise, I might be on a desktop computer that has a touch interface as it's primary pointer. In this case I might have trouble tapping small links on a page. It would be nice to be able to get feedback for these use cases, and Modernizr's touch property give us that.

Try replacing the previous script example of opacity with the following:

```
<script type="text/javascript">
  if(Modernizr.touch) {
    alert('touch is supported');
  }
</script>
```

After you refresh the page, nothing happens. Where did our alert go?

The if statement returned false, therefore the alert contained within it wasn't run. If we write an "else" statement after the "if" statement, we can run a command in the event that the if statement fails.

```
if(Moderizr.touch) {
    alert('touch is supported');
}else{
    alert('touch is not supported')
}
```

This way, our bases are covered in both events.

Obviously, simply alerting the user to whether touch events are available doesn't have a wealth of practical applications. It's great for testing, but useless outside of that. What would really be helpful is if we could load a one Javascript plugin or CSS file if the test passes and another if the test fails. Luckily Moderizr is extended using yepnope.js to support this ability.

### 8.3.2 Using Modernizr.load and Yep Nope

Another of the advantageous features we find in Modernizr is its resource loader. Using Modernizr.load we can load a set of files, depending on whether or not a test is passed. This is great because it gives us a chance to only load the assets we need for the user that is visiting the website.

As a syntax it's completely straight forward and written in plain English. Here's an example of a simple test.

```
<script type="text/javascript">
    Modernizr.load({
        test: Modernizr.touch, #A
        yep: 'stylesheets/touch.css', #B
        nope: 'stylesheets/no-touch.css' #C
    });
</script>
```

**#A Here we choose the test we want to run**

**#B In "yep" we offer the files to serve if the test passes**

**#C In "nope" we offer the files to serve if the test fails.**

Let's break this down a little bit. Modernizr.load runs on page load. In this example it runs the tests specified. If the tests are passed the assets after "yep" are loaded. If the tests fail, the assets after "nope" are loaded. If you ran this test in the example on a computer without touch support, the background should be red. If there is touch support it should be blue.

We can also specify multiple files by enclosing the assets in brackets like this:

```
Modernizr.load({
    test: Modernizr.touch,
    yep: ['stylesheets/touch.css', "touch.js"],
    nope: ['stylesheets/no-touch.css', "no-touch.js"]
});
```

You can also run multiple sets of tests by enclosing the tests in curly brackets and adding comma separation, like this:

```
<script type="text/javascript">
    Modernizr.load([
        {
            test: Modernizr.touch,
            yep: ['stylesheets/touch.css', "touch.js"],
            nope: ['stylesheets/no-touch.css', "no-touch.js"]
        }
    ]
);
```

```

    },
    {
      test: Modernizr.opacity,
      yep: ['stylesheets/opacity.css', "opacity.js"],
      nope: ['stylesheets/no-opacity.css', "no-opacity.js"]
    },
  ],
});
</script>

```

As I'm sure your quickly noticing, there is a huge depth to the degree with which you can customize a user's experience with Modernizr. It gives you the ability to fine tune a website specifically for the user's needs.

## 8.4 Summary

As is the case with much of the subject's we've talked on, there is a great wealth of information to be learned about Modernizr. It functions exceptionally as a system with which to serve your user actuators to create speedy and highly functional websites.

Take some time and read up on the libraries documentation at [modernizr.com](http://modernizr.com). I've found it to be an incredibly helpful resource and invaluable in my personal workflow.

# 9

## *Testing and optimization for responsive website*

In this chapter, we'll learn:

1. Why optimization is crucial to building responsive experiences
2. How to use web inspectors to find out ways to improve a site performance.
3. Tip on improving performance.

As we've gone through our example site we've used HTML and CSS to build a responsive website that adapts to our changing viewports. We've discussed how to add images and media to a responsive site and ways to prevent failure in older browsers. All of this work has led to us adding a lot of extra work and content to our pages though, and in this comes one of the biggest challenges with responsive web development, how do you add the functionality of responsiveness without bloating your site's load time. If we simply add a few CSS techniques to our workflow and create pages that scale down, we've succeeded in making our websites visually scale, but potentially at the cost of performance.

One of the biggest cons people give towards responsive design is that responsive sites perform slowly, and it's true. Generally responsive web sites mean adding the responsiveness like a feature. Sometimes this means loading extra images or hiding elements. Showing and hiding content is one of the ways sites can get out of control, but there are some small steps you can take to ensure your site performs optimally.

If a chair is pleasing to the eyes, but gives people back pain, what good is the chair? In the same way, if a responsive site is beautiful and scales nicely but comes at the expense of load times, what good is the site? With the recent inclusion of testing tools for both desktops and mobile devices, it's easier NOW than ever to make an incredibly fast website. In previous chapters we've built our sites using a resized browser window. This works fine when we're simply concerned with the basic scaling and architecture of a site, but there is no substitute for

working directly in a device. By testing in a device we can view the actual environment the user will end up with.

In the previous chapter we discussed using Modernizr to add support for various features for browser support and it's usefulness in progressive enhancement. In order to enact progressive enhancements though, we have to test then enhance to support. Testing in native environments is the best way to anticipate the experience your users will have.

## **9.1 *What is responsive testing?***

Responsive web design requires a workflow that accounts for testing during the build process. In building a site and progressively enhancing into the final product, we ensure that the site is responsive in not just size and layout, but in capabilities and features. It's a common misconception that a website has to look identical from browser to browser. Just as in responsive design there is a difference in layout between breakpoint, there is a need to also a need to have a difference in capabilities between browsers. Since older browsers have CSS limitations, it's important to contextualize pieces of a site that require certain capabilities and provide fallbacks for those pieces using Modernizr.

In responsive web design it's important to remember that there is more to being responsive than simply anticipating browser width. This is something we've discussed throughout this book. The only way to get a sense for how a site works in various devices is to experience it first hand. Since most developers can't exactly justify the expense related to building their own browser labs, there are also a number of environment simulators that are available for use.

I personally find it's best to draft a testing plan based on a singular starting point and to add progressive enhancements through various devices, operating systems and environments. This "starting point" can be a real scenario

### **9.1.1 *Simulated testing environments***

Simulated environments are the easiest way to test for various operating systems. There are a number of software simulators available for use. These simulated environments are simple emulators built to test operating systems outside of their native environments. There are a number of ways to install these virtual environments and a few different types of environments you can install.

There are a few programs on Macintosh that are available to simulate Windows and Android environments as well as iOS. Unfortunately, because of Apple's software philosophy, OSX is unavailable for emulation in environments outside of Apple hardware. For this reason, OSX is my preferred development environment.

One application to assist in building simulated environments is Parallels. Parallels is an application for Mac which allows you to build simulated environments based on Operating Systems and hardware standards that you set.

### USING PARALLELS FOR MAC

Once installed you can open Parallels and you are greeted with an installation wizard. This wizard will walk you through the process of setting up a new virtual machine. Here you are given a series of options. You can install a new copy of Windows, migrate an existing copy of Windows, download Chrome OS, Ubuntu, or Android or you can build a virtual machine from a OSX recovery disc.



Figure 9.1 On launch Parallels gives you some options in configuring your virtual environment.

Once installed Parallels will run with its desired operating system in its own window. This window will function like an OS inside of your OS. You can share assets and test against your local host as well as anything online.



Figure 9.2 Running Windows 8 on a Mac. With this set up I have access to a simulated environment for most web users. IE10 still has the developer tools to allow you to test previous versions of IE, so the bulk of cross browser bugs can be identified.

The settings for each virtual are configurable and will give you a chance to simulate variations in your testing environment. This offers you the chance to intentionally cripple your website and see how it looks on older computers, so you can make sure to accommodate those users. While this might seem a little over the top, it's incredibly helpful. My family comes from a rural area in Tennessee, and my mother doesn't have access to the type of bandwidth I can get in my apartment in Manhattan. Because of this I often find that when I call her to celebrate a site launch, she might say something like "I'm sure it's great, it's just taking a long time to load."



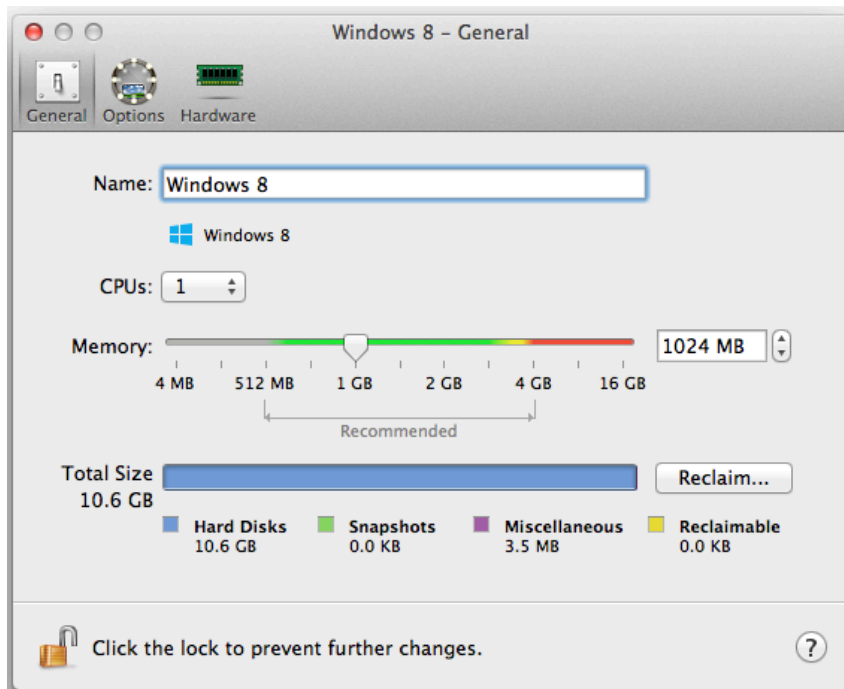


Figure 9.3 The settings for my Windows 8 virtual machine. Here we can reduce the amount of memory devoted to our virtual machine. This is located under Virtual Machine > Configure. Under the Hardware section we also have an option to limit video memory, which is helpful in debugging animations.

### THE PROS OF PARALLELS

There are a lot of benefits to using Parallels. It's efficient and consistent and because it's a paid program it offers a lot of support and premium features. For the most part, it lets you set up almost any configuration and combination that you might want. Generating simulated environments gives you insight on various browsers and can help identify potential problem areas before they get too serious.

### THE CONS OF PARALLELS

In spite of all it can do, there are a few obvious drawbacks. First of all is the price tag. Purchasing Parallels still requires a copy of Windows, which when purchased alone is around \$90 for the pro pack. It also can be resource intense, but no more so than a lot of other virtual machines. There is a free alternative to Parallels released by Oracle called VirtualBox, which gives you the same ability to install virtual machines, but is slightly more cumbersome and I've had a lot of issues with. It also lacks support for Chrome OS and Android. That being said, in a pinch it works great.

## INSTALLING IOS VIRTUAL ENVIRONMENTS

Aside from Parallels, you can simulate iOS environments using the iOS Simulator included in Xcode. Xcode is an IDE, compiler, and SDK kit for iPhone development, but it also includes a set of simulators which are handy for web testing. Xcode is downloaded in the App Store and installed with a click of the install button.

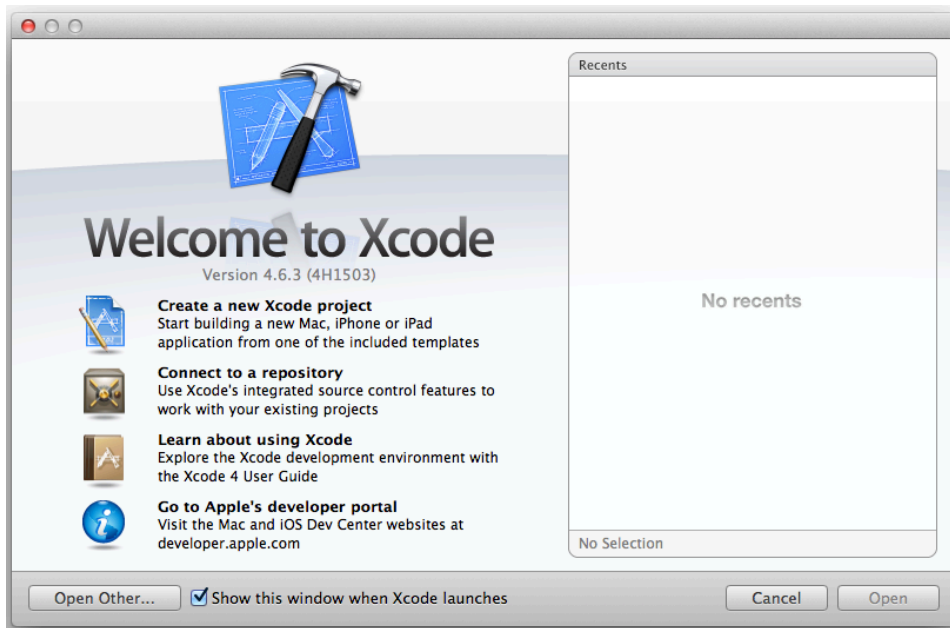


Figure 9.4 The Xcode welcome screen.

To access the iPhone simulator, simply click on Xcode > Open Developer Tools > iOS Simulator. Here you have a functioning local version of iOS which you can use to test your sites in Mobile Safari.



Figure 9.5 The iOS Simulator screen. This application is incredibly useful for testing responsive websites in their native environments.

The simulator is very useful, but it's important to note that while this will give a good sense of layout and overall look and feel, you will still be lacking the native feel of touch interactions. You also have the ability to test using a variety of devices and operating systems. You can test on any iOS that you can install locally and Apple provides emulators for iPhones and iPads dating back to the first iPhones and iPads. Since Apple has control over the hardware in those devices, it's very easy for them to accurately emulate their performance.

## 9.2 *Browser Tools for Testing*

When you go into developing a responsive site, if you can afford it, building a browser lab will make for an incredible testing lab. While the actual devices are hard to recommend since they are constantly changing, it's a good idea to have a device for each major OS. A high-end Android, low-end Android, an older iPod Touch and an up to date iPhone or iPod touch, a Windows phone, a Blackberry, and a tablet or two is a good starting point. It's a good idea to spread out the screen sizes as much as possible.

Generally a good spread of 2 or 3 phones and 2 or 3 tablets, each with a different OS is a good selection. Don't worry about buying new devices, since they are strictly for testing and having older devices forces you to work within tighter parameters. A site built to work well on a crappy phone with a slow connection will perform even better on a faster phone.

When in a pinch, there are also a number of emulators available for testing. These can show you how the software will render the page and are great for testing against software errors, but nothing compares to the actual experience of interacting with a site on a device.

## 9.3 *Using web inspectors*

In both mobile, tablet, and desktop browsers there are a variety of inspectors available to you. Earlier we used the Chrome inspector to investigate a page's markup and CSS. The web-kit inspector is single handedly the best tool at your disposal for web development. As I've gotten more adept at writing code, I've found the web inspector to be one of the absolute fundamental tools in my toolkit. Internet Explorer has had a developer toolkit available since IE6. Likewise, Firefox has had a plugin called "Firebug" available for a few years and has recently included its own developer tool set. These tools offer relatively the same toolsets as Chrome developer tools, but for the purposes of this book, we're going to stick with the Chrome tools.

Aside from markup and a JavaScript console, the inspector lets you view how a site loads, what is loaded, and how fast it loads. In Chrome, Google even offers a plugin called PageSpeed which can test your page and help you find problem areas in your site.

The key to mastering the web inspector is knowing what each of the tabs at the top do and how to use them:

- **Elements:** This gives you a full view of the rendered DOM and the ability to inspect each object associated with CSS. This is incredible for tweaking designs here and there and giving feedback on potential CSS bugs. This is the default Developer Tools panel and often my most used.
- **Resources:** this panel shows the resources being used by a page. This includes

JavaScript, CSS, HTML, as well as the storage methods used by the browser, such as local storage and cookies.

- **Network:** In the network panel you can view what is loaded, how it's applied, and it's overall effect in page rendering. It will show a time line of how the page behaves. you can even break down to the millisecond how various assets effect the page load.
- **Sources:** This panel is used for debugging JavaScript and can help you track down the root issues in JavaScript applications. It's used to
- **Timeline:** Within timeline you can record a page load and dive deep into how the page is loading and how the actual load time is effected.
- **Console:** this is useful for viewing and testing javascript objects and its also great for logging messages when debugging.

Two of these features, resources and network, are especially helpful in creating a responsive web site. These two panels will help you optimize your site and reduce the user's over all bandwidth burden.

### 9.3.1 Elements

In the elements tab you have access to the full DOM. Here you can see everything as you wrote it, after it's been affected by JavaScript and even while JavaScript runs on the page. Once you select an element to inspect you have a full view of it's CSS properties, including all selectors effecting the object and their computed styles. We discussed the inspector back in Chapter 3, so there's no need to go in depth here.

### 9.3.2 Resource

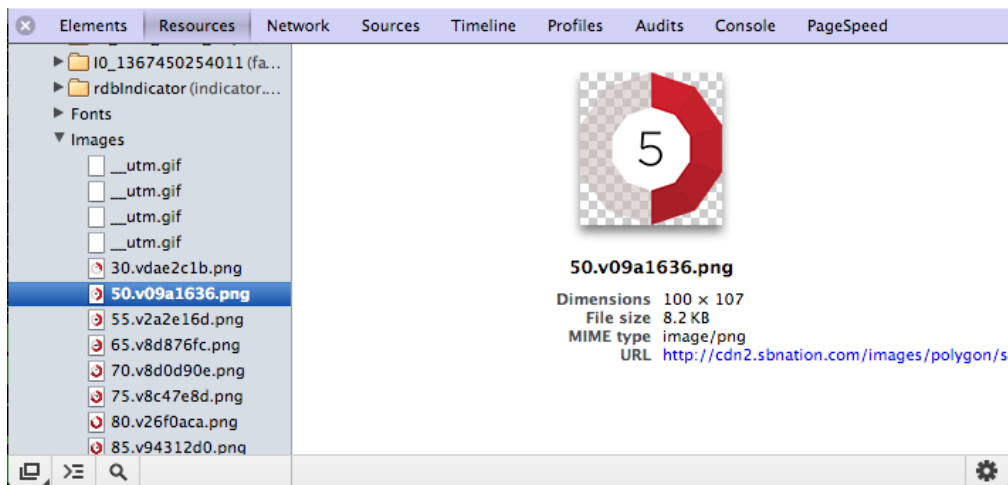


Figure X.1 Inspecting an image in the resources panel

In the resources panel you can view the elements being queried by the server on page load. These are seen under “frames” and can be used to identify what CSS and JavaScript is being loaded, as well as images and other assets. It can become quickly apparent what you might be loading that you don’t need or places you can be more efficient.

When clicking on a asset you can view what the source asset is and see its size, both in terms of data and relative pixel measurements. This gives you a sense of every elements weight in the load process.

Name	Path	Method	Status	Type	Size	Time	Timeline
50.v09a1636.png	cdn2.sbnation.com/images/polygon/scores	GET	200 OK	image/png	8.6 KB	177 ms	
30.vdae2c1b.png	cdn3.sbnation.com/images/polygon/scores	GET	200 OK	image/png	8.5 KB	192 ms	
1367440914	www.sbnation.com/chorus_images/125379	GET	301 Moved Permanently	text/html	601 B	286 ms	
1367439671	www.sbnation.com/chorus_images/125369	GET	301 Moved Permanently	text/html	601 B	339 ms	
100.v0482512.png	cdn3.sbnation.com/images/polygon/scores	GET	200 OK	image/png	9.6 KB	184 ms	
017click-http3AN2F92Fox-d.sbnation.c	view.atdmt.com/MRT/view/437674800/did	GET	200 OK	text/html	3.1 KB	296 ms	

161 requests | 2.7 MB transferred | 1.0 min (includ: 2.04 s, DOMContentLoaded: 1.15 s)

Figure X.2 The network panel

### 9.3.3 Network

The network panel is of particular interest in trying to maximize a site’s efficiency for responsive web sites. Using this tool helps you to get a visual sense of every request and it’s effects on you sites load time, because it shows the requests a page makes and the time it takes to execute that request.

A client (or web browser) makes a request for every asset used on the page. This means every image file, CSS file, JavaScript file, requires a request to a sever. These requests take time, called a “Round-Trip Time” or RTT.

A request breaks down a few ways. First the page has to make the request and then a server has to send the response. After this handshake between servers the site can begin to load. This handshake actually breaks down into three total round trips from the client to the server. The first to find the file (called DNS name resolution), the second to establish a connection (a TCP connection), and a third round trip to begin the transferring of a file.

#### ACCESSING DEVELOPER TOOLS ON IOS

In iOS 6 Apple began including the ability to access Safari’s developer tools within the desktop version of Safari. This gives you the ability to inspect element on your iPhone. This is accomplished by attaching your iPhone with it’s USB cord and opening Safari. Once there you can access the tools by clicking Develop > Devices and selecting you device. This window is similar to the Chrome developer tools. This works with iPhones as well as iPads.

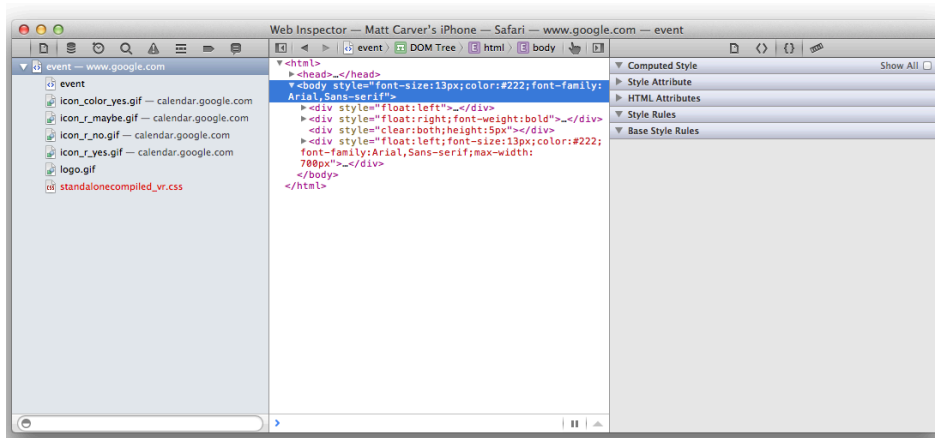


Figure 9.X Safari developer tools running from an attached iPhone.

### 9.3.4 Sources

In the sources panel you can inspect specific source files used in the project. This gives you the ability to investigate a single file. This is mostly useful in debugging JavaScript, but can also be helpfully in inspecting your CSS files in the developer tools.

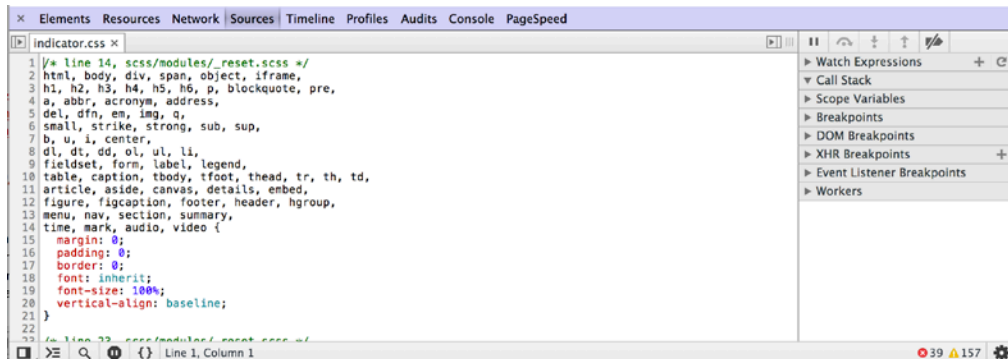


Figure 9.X The Sources panel in Chrome developer tools.

### 9.3.5 Timeline

The timeline panel gives you the opportunity to see on a microscopic level the loading of a page. This gives you a visual representation of all the requests made on page load, the time taken to return those requests and the assets total load time. When trying to reduce total load time it's important to first look at the number of requests being made and see if there are any ways to consolidate that. You can also investigate to see if any files are taking an extraordinary time to load.

It's also important to remember the load order of a page. Pages start with the initial DOM load, then CSS, at that point the page is displayed and JavaScript is executed. In the timeline you can watch how your page loads all of these assets.

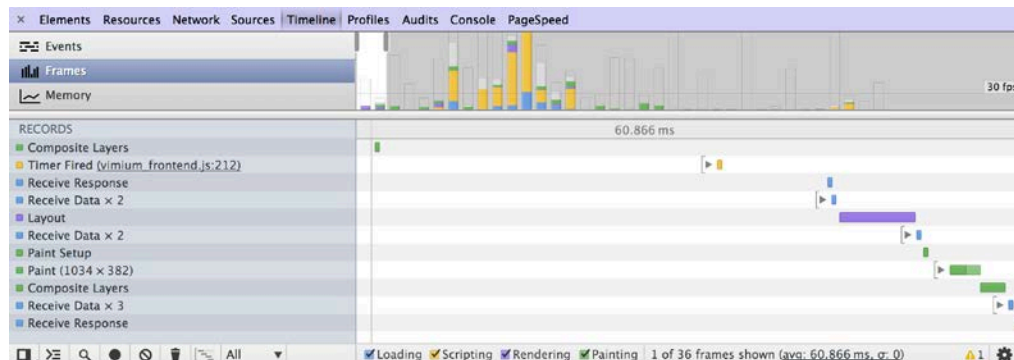


Figure 9.X The timeline panel feeds real time load processing information.

The time will show loading, scripting, rendering, and painting of the page in real time once you hit the record button. The recording can be stopped then to analyze the results and zoom in on parts of the process to get some in depth feedback.

## 9.4 Tips on reducing request times

Even as bandwidth speeds go up, HTTP requests still create a latency problem in a majority of sites, since then are highly dependent on not just the user but also the server. This is commonly considered an issue in mobile development because mobile users in the United States tend to be on cellular connections which tend to be slower and feature bandwidth caps, meaning the assets transferred need to be minified to improve performance in mobile.

While that's true it's only part of the story. Reducing HTTP requests is also of major importance across all sites, but it's not just load time that is the concern. Most users are found to be impatient when first experiencing a new site and if a site doesn't load in a matter of seconds, a user will leave. The initial few HTTP requests can add to the time before the site begins rendering, increasing the amount of people leaving before the site is done loading. Reducing these requests can give you a dramatic advantage over your competition. There is also more and more coming out about the effect of load time on Search Engine Optimization.

One of the easiest ways to reduce request times is to host as much of the site's data on the same server. This is because it circumvents the first part of the RTT is DNS look up, and the browser can cache this DNS lookup, so it's already done. Every DNS lookup a browser has to do comes at a cost of total load time. This lookup is between name servers and the further away the servers, the long it takes.

Another big way to improve performance is to ensure there are no bad request on your page. These are wasteful and can bog the site down as an asset searches for a non existent



asset. This issue is way more common than most people think. Sometimes it's a result of trying to pull in an old JavaScript file that no longer exists in the head tag, other times its a bad path to a rarely used image in a CSS file. It's important that you ensure on every page load the page can find all the required assets. This means making sure to write correct paths to assets and maintaining those same paths in the live environment.

Once you minimized the page's requests it's time to start tending to your site's assets. Compressing JavaScript and CSS is one excellent way to do that, but its well covered territory. Image compression is another big one.

### **Developer's Insight: Compressing CSS and JS**

Compressing CSS and JS files can save a lot of extra bites, but sometimes it can cause other developer's headaches. When compressing files, make sure to document how you compressed the assets so anyone else working on the project knows how to make the files manageable again.

#### **9.4.1 Increasing image efficiency with base 64 encoding**

Images are another major point to hit on when improving site performance. There are applications to help you compress images and reduce their over all burden in the site. One way to go about optimizing images is to use what's called "base64 encoding". This means breaking down an image into it's data form and inserting it into your CSS that way.

I find this method to be extreme effective in using textures on a site, you can easily save yourself some HTTP requests by simply including the texture as a base64 image. The data will be about give or take the same, but you'll spare yourself a request by including the image in the css.

You can convert an image into base64 by using any number of tool online. A quick search will bring several sites that offer services. Once you have the string, you can easily insert it into css with the following code:

```
body {
    background-image: url(data:image/png;base64,
        [ long string of Base64 data here ]
    );
}
```

This seems completely foreign and complicated, but it works exactly like a normal background image.

## **9.5 Summary**

In this chapter we learned the importance of site optimization in responsive web design. We learned how to use web inspectors to view the document structure and learn valuable information on how our page loads and what assets are being used in a site load.

We learned a few less commonly used tips on how to optimize a page load and how to reduce server requests. By optimizing our pages, we ensure that a page will work well in the

constraints of mobile and we can increase page load time in spite of having added additional code to create a responsive web site.