Community Experience Distilled

Learning Modernizr

Create forward-compatible websites using feature detection features of Modernizr

Adam Watson

[PACKT] PUBLISHING open source* community experience distilled

Learning Modernizr

Create forward-compatible websites using feature detection features of Modernizr

Adam Watson

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Learning Modernizr

First published: December 2012

Production Reference: 1171212

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-022-9

www.packtpub.com

Cover Image by J.Blaminsky (milak6@wp.pl)

Credits

Author
Adam Watson

Reviewers
Chetankumar Akarte
Ben Fhala
Michelle Williamson

Acquisition Editor
Wilson D'Souza

Commissioning Editor
Meeta Rajani
Maria D'souza

Technical Editor
Nitee Shetty

Copy Editors
Vrinda Amberkar
Alfida Paiva

Project Coordinator
Shraddha Bagadia

Proofreader
Aaron Nash

Indexers
Hemangini Bari
Rekha Nair

Graphics
Aditi Gajjar

Production Coordinator
Prachali Bhiwandkar

Cover Work
Prachali Bhiwandkar

About the Author

Adam Watson began life as a guitar maker but quickly transitioned into tech after landing a firmware-testing gig at Hewlett Packard. After getting a taste of the tech life, he began designing and coding websites in early 2000. Now, he works for BUZZMEDIA, a leading pop-culture company developing a mobile face for the company network of more than 40 music and pop-culture websites. He has also helped in writing the official Facebook plugin for WordPress, re-factoring portions of it a week before its official launch.

BUZZMEDIA is the leader in pop-culture and has millennial audience reach. It's the world's fastest-growing digital media company, with more than 100 million monthly unique visitors.

Over 35 BUZZMEDIA brands including SPIN, Celebuzz, Stereogum, Buzznet, The Superficial, Idolator, Just Jared, Just Jared Jr., PureVolume, The Hype Machine, AbsolutePunk, What Would Tyler Durden Do?, SocialiteLife, Go Fug Yourself, Pink is the New Blog, Gorilla vs. Bear, RCRD LBL, Videogum, TheFrisky, Egotastic!, Concrete Loop, Brooklyn Vegan, Crunktastical, Punknews.org, and others are category-leading, authentic, influential voices.

I'd like to thank Kelly, M.J., and Maddy for their support in helping me to make this book possible.

About the Reviewers

Chetankumar Akarte is an Electronics Engineer from Nagpur University located in central India. He has more than 6 years of experience in the design, development, and deployment of Web, Windows, and mobile-based applications, with expertise in PHP, .NET, JavaScript, Java, Android, and more.

He likes to contribute on the newsgroups' forums. He has written articles for Electronics For You, DeveloperIQ, and Flash & Flex Developer's Magazine. In his spare time, he likes to maintain his technical blog at `http://www.tipsntracks.com` to get in touch with developers community. He has been a technical reviewer for two books published by Packt Publishing. He has released some Marathi and Hindi e-book applications on the Android market (`https://play.google.com/store/apps/developer?id=Chetankumar+Akarte`).

He lives in the hilly Kharghar area of Navi Mumbai with his son Kaiwalya and wife Shraddha. You can visit his websites at `http://www.xfunda.com` and `http://www.tipsntracks.com`, or get in touch with him at `chetan.akarte@gmail.com`.

> I would like to thank my wife Shraddha and my parents for their consistent support and encouragement. My lovely son Kaiwalya who allowed me to use his playtime with me to dedicate towards this book. I would also like to thank Packt Publishing for giving me the opportunity to do something useful and especially the Project Coordinator, Shraddha Bagadia for all the valuable support.

Ben Fhala is a leading voice in the Technology Advertising world of New York. He is the owner of the online video training school `02geek.com`, an Adobe ACP. He enjoys spending most of his time in learning and teaching, and has a love for visual programming and visualization in general.

He has had the honor to develop applications for US Congress members, Prime Ministers, and Presidents around the world. He has built many advertising experiences for companies such as Target, AT&T, Crayola, Marriott, Neutrogena, and Nokia. He has technically directed and led many award winning projects and has so far helped three agencies receive an Agency of the Year Award (not just helped but led the technical side to produce award quality work).

He has recently written his first book, *HTML5 Graphing and Data Visualization Cookbook*, *Packt Publishing*, available from November 2012.

> Thanks for giving me a chance to be a part of this book. I've enjoyed reading it and putting my few small pointers. It's been honored to be included in this project and I want to thank the readers for joining in and wish them a great read.

Michelle Williamson began her journey with computers in 1994 with a traumatizing mishap involving a 15-page graduate class paper and an unformatted floppy disk. She spent 5 years as a staunch luddite before becoming obsessed with web development and technology in general. She has been a freelancing web developer since 2000, starting out in Microsoft platforms, then drinking the open source Kool-Aid in 2008 and has since devoted her time primarily to Drupal development. She's an incessant learner, addicted to head-scratching challenges, and looks forward to experiencing the continued evolution of mobile technology.

Table of Contents

Preface

While we eagerly wait for the days to come where a single piece of code works across all browsers. Cross-browser strategies have arisen to meet these challenges in the meantime. You may already be familiar with some of these strategies; for example, detecting the browser version and manufacturer, and serving different pieces accordingly. More recently a new technique has emerged known as feature detection, targeting the look and feel of features as opposed to specific browsers. For example, creating a look for an element such as a div element for browsers that support and do not support box-shadow. We'll dive into this and much more in the chapters that follow.

What this book covers

Chapter 1, *Getting Started with Modernizr*, describes the concepts of feature detection compared to more traditional User Agent sniffing.

Chapter 2, *Installing Modernizr*, installs the library and establishes a basic HTML template.

Chapter 3, *Using Modernizr the Right Way*, covers multiple methods of feature detection and implements designs and functionalities around them.

Chapter 4, *Customizing to Your Unique Needs*, builds a custom-tailored version of the Modernizr library as well as exploring a few of the extras that come with it.

What you need for this book

You will need a text editor such as Sublime Text, and a modern WebKit browser such as Google Chrome.

Who this book is for

This book is for web developers with intermediate to advanced knowledge of cross-browser development with CSS and HTML5.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The `no-js` CSS class method isn't something exclusive to Modernizr".

A block of code is set as follows:

```
<div id="main">
<div id="frame-1" class="frame"></div>
<div id="frame-2" class="frame"></div>
<div id="frame-3" class="frame"></div>
<div id="frame-4" class="frame"></div>
<div id="frame-5" class="frame"></div>
</div>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#navbar{
  width: 100%;
  height: 50px;
  text-align: center;
  position: fixed;
  top: 0;
  padding: 0;
  background: url( images/stripe-header.png ) 0 0 #333;
  border-bottom: solid 1px white;
  z-index: 10;
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Simply select **Production** as the download option".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Modernizr

Are you tired of approaching the way you build your websites as though it was two or maybe three years ago? By that I mean not building the experience as if it were for the browser of today, but instead building it for the oldest browser your users are most likely to be navigating with. Are you also tired of building different versions of the same website for an abundance of browsers, by version, and then stuffing them full of CSS hacks? What if there was a better and more manageable way?

What if you could build something not only for three years ago, but also for today? What if you could even build it for as far into the future as three years from now? What if you could do this without having to do any sort of browser sniffing at all? I'm very happy to say that this is possible today, thanks to the concept of feature detection and a lightweight, customizable detection library named **Modernizr**.

In this chapter we will cover the following topics:

- What is feature detection?
- What is Modernizr?
- The Modernizr namespace; storing the test results.
- CSS selector test results.
- Designing for the feature and not the browser.
- Why UA sniffing is bad.

Detect and design with features, not User Agents (browsers)

What if you could build your website based on features instead of for the individual browser idiosyncrasies by manufacturer and version, making your website not just backward compatible but also forward compatible? You could quite potentially build a fully backward and forward compatible experience using a single code base across the entire UA spectrum. What I mean by this is instead of baking in an MSIE 7.0 version, an MSIE 8.0 version, a Firefox version, and so on of your website, and then using JavaScript to listen for, or sniff out, the browser version in play, it would be much simpler to instead build a single version of your website that supports all of the older generation, latest generation, and in many cases even future generation technologies, such as a video API, `box-shadow`, first-of-type, and more.

> Think of your website as a full-fledged cable television network broadcasting over 130 channels, and your users as customers that sign up for only the most basic package available, of only 15 channels. Any time that they upgrade their cable (browser) package to one offering additional channels (features), they can begin enjoying them immediately because you have already been broadcasting to each one of those channels the entire time.

What happens now is that a proverbial line is drawn in the sand, and the site is built on the assumption that a particular set of features will exist and are thus supported. If not, fallbacks are in place to allow a smooth degradation of the experience as usual, but more importantly the site is built to adopt features that the browser will eventually have. Modernizr can detect CSS features, such as `@font-face`, `box-shadow`, and CSS gradients. It can also detect HTML5 elements, such as `canvas`, `localstorage`, and `application cache`. In all it can detect over 40 features for you, the developer.

Another term commonly used to describe this technique is "progressive enhancement". When the time finally comes that the user decides to upgrade their browser, the new features that the more recent browser version brings with it, for example `text-shadow`, will automatically be detected and picked up by your website, to be leveraged by your site with no extra work or code from you when they do. Without any additional work on your part, any text that is assigned `text-shadow` attributes will turn on at the flick of a switch so that user's experience will smoothly, and progressively be enhanced.

> What is Modernizr? More importantly, why should you use it? At its foundation, Modernizr is a feature-detection library powered by none other than JavaScript.

Here is an example of conditionally adding CSS classes based on the browser, also known as the User Agent. When the browser parses this HTML document and finds a match, that class will be conditionally added to the page.

```
<!--IE only conditional comments http://www.quirksmode.org/css/
condcom.html -->
<!--[if lt IE 7]> <html class="lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]> <html class=" lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]> <html class="lt-ie9"> <![endif]-->
```

Now that the browser version has been found, the developer can use CSS to alter the page based on the version of the browser that is used to parse the page. In the following example, IE 7, IE 8, and IE 9 all use a different method for a `drop shadow` attribute on an anchor element:

```
/* IE7 Conditional class using UA sniffing */
.lt-ie7 a{ display: block; float: left; background: url( drop-shadow.
gif ); }
.lt-ie8 a{ display: inline-block; background: url( drop-shadow.png );
}
.lt-ie9 a{ display: inline-block;  box-shadow: 10px 5px 5px
rgba(0,0,0,0.5); }
```

The problem with the conditional method of applying styles is that not only does it require more code, but it also leaves a burden on the developer to know what browser version is capable of a given feature, in this case `box-shadow`. Here is the same example using `Modernizr`. Note how `Modernizr` has done the heavy lifting for you, irrespective of whether or not the `box-shadow` feature is supported by the browser:

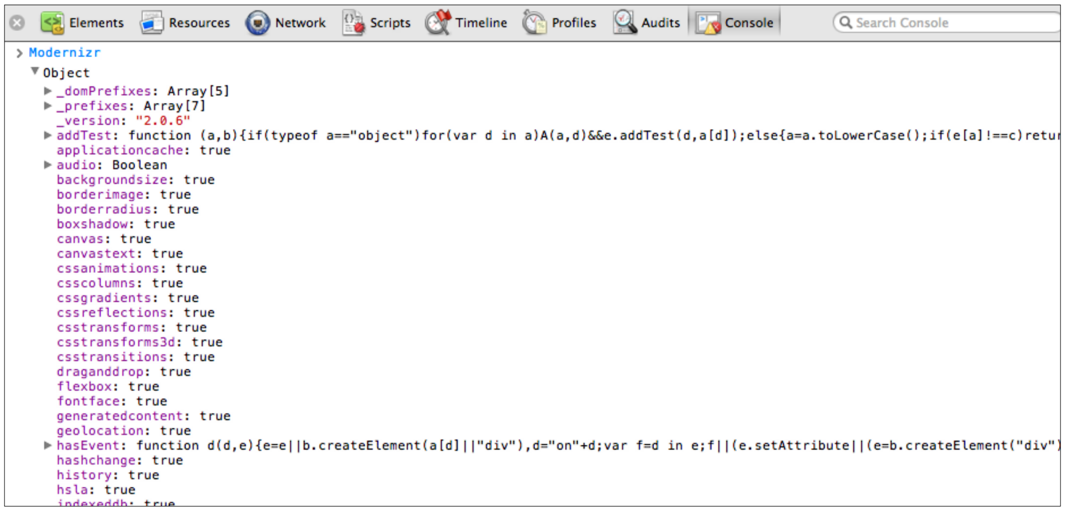
```
/* Box shadow using Modernizr CSS feature detected classes */

.box-shadow a{ box-shadow: 10px 5px 5px rgba(0,0,0,0.5); }
.no-box-shadow a{ background: url( drop-shadow.gif ); }
```

The Modernizr namespace

The Modernizr JavaScript library in your web page's header is a lightweight feature library that will test your browser for the features that it may or may not support, and store those results in a JavaScript namespace aptly named `Modernizr`. You can then use those test results as you build your website.

From this point on everything you need to know about what features your user's browser can and cannot support can be checked for in two separate places. Going back to the cable television analogy, you now know what channels (features) your user does and does not have, and can act accordingly.

The following screenshot shows the `Modernizr` object from the console:



```
> Modernizr
  ▼Object
    ▶ _domPrefixes: Array[5]
    ▶ _prefixes: Array[7]
      _version: "2.0.6"
    ▶ addTest: function (a,b){if(typeof a=="object")for(var d in a)A(a,d)&&e.addTest(d,a[d]);else{a=a.toLowerCase();if(e[a]!==c)retur
      applicationcache: true
    ▶ audio: Boolean
      backgroundsize: true
      borderimage: true
      borderradius: true
      boxshadow: true
      canvas: true
      canvastext: true
      cssanimations: true
      csscolumns: true
      cssgradients: true
      cssreflections: true
      csstransforms: true
      csstransforms3d: true
      csstransitions: true
      draganddrop: true
      flexbox: true
      fontface: true
      generatedcontent: true
      geolocation: true
    ▶ hasEvent: function d(d,e){e=e||b.createElement(a[d]||"div"),d="on"+d;var f=d in e;f||(e.setAttribute||(e=b.createElement("div")
      hashchange: true
      history: true
      hsla: true
      indexeddb: true
```

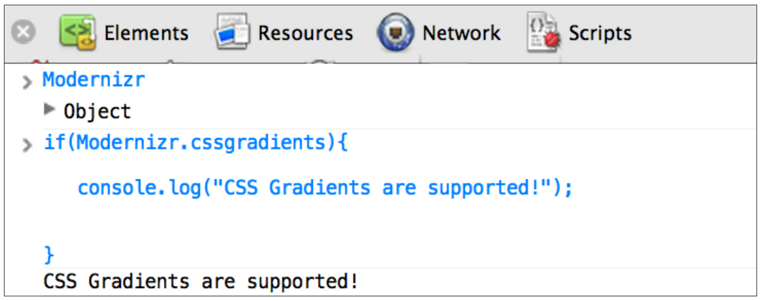
`Modernizr` is storing either `true` or `false` properties inside its namespace based on the features that the browser supports, which can in turn be used to tightly and easily control the user's experience.

The first place you can use to check for support is, of course, the `Modernizr` JavaScript object now available on the page. Let's say, for example, you wanted to know if the viewer of your web page has support for 3D CSS transformations within the browser (user agent) they are using. Checking for this is as simple as a conditional in your JavaScript code, as shown in the following code snippet:

```
//JavaScript test for 3D CSS Transformations.
if( Modernizr.csstransforms3d ){
  console.log('3D CSS Transformations are supported!');
}
```

`Modernizr` has stored this as a property of either `true` or `false`, and by checking the result of this test, we are able to ascertain if the browser that any particular user is accessing your web page with has that feature supported. If they do, great! If they don't, then we can degrade the experience a bit so they aren't left viewing a big empty box on the page, and the best part is that once they do inevitably upgrade, all of the really cool features will be there, ready and waiting.

After the `Modernizr` library feature tests have run on your page, there will be a great number of test results now stored as properties in the `Modernizr` JavaScript namespace. In fact, at the time of this writing, over 40 feature tests were performed by this lightweight, lightning fast library, not counting the additional extensions available in the user community for even more feature tests, for example, CSS3 media queries. The following screenshot shows an example from the JavaScript console that is testing for CSS gradients and logging a success message if they are supported:



```
> Modernizr
  ▶ Object
> if(Modernizr.cssgradients){

     console.log("CSS Gradients are supported!");


  }
  CSS Gradients are supported!
```

Modernizr version 2.5 tests over 40 next-generation features, which means that there are over 40 applications you can build for today that aren't even available yet in many of the current browsers.

Supporting features with CSS selectors

It doesn't stop at JavaScript either. There is a second way to leverage the feature tests from Modernizr. You see that what Modernizr has also done behind the scenes is added a series of CSS selectors to the `HTML` element of the web page. By inspecting the element in the browser and viewing the **Document Object Model** (**DOM**), you'll see that Modernizr was also hard at work on the DOM tree in addition to the JavaScript, and has added a selector to the `HTML` element for all of the feature tests that were performed.

If I inspect an element in Google Chrome 19 for example, I get the following code snippet added to the HTML element of my DOM tree:

```
<html class="js flexbox canvas canvastext webgl no-touch geolocation
postmessage websqldatabase indexeddb hashchange history draganddrop
websockets rgba hsla multiplebgs backgroundsize borderimage
borderradius boxshadow textshadow opacity cssanimations csscolumns
cssgradients cssreflections csstransforms csstransforms3d
csstransitions fontface generatedcontent video audio localstorage
sessionstorage webworkers applicationcache svg inlinesvg smil
svgclippaths"...
```

The whole mess of selectors that you see are the feature tests, which `Modernizr` ran and has now stored as properties within its namespace. Now that they are additionally inside the DOM, there is a second capability available for use. We can now create style conditions around these functionalities from the stylesheet as well.

It's not limited to pass the tests either. Let's see what happens when I run that exact same `Modernizr` script in Internet Explorer 8 using Windows 7 as my environment:

```
<html class="js no-flexbox no-canvas no-canvastext no-webgl no-
touch geolocation postmessage no-websqldatabase no-indexeddb
hashchange no-history draganddrop no-websockets no-rgba no-hsla no-
multiplebgs no-backgroundsize no-borderimage no-borderradius no-
boxshadow no-textshadow no-opacity no-cssanimations no-csscolumns
no-cssgradients no-cssreflections no-csstransforms no-csstransforms3d
no-csstransitions fontface generatedcontent no-video no-audio
localstorage sessionstorage no-webworkers no-applicationcache no-svg
no-inlinesvg no-smil no-svgclippaths"...
```

Now that's a whole lot of selectors and a whole lot of no's! Let's look at some key differences between the two examples. You see `Modernizr` not only adds in its own classes for the features that are supported but also for the features that are not supported.

For example, Internet Explorer 8 does not support the CSS 3D transformations (and won't until version 10), previously tested for and supported in Google Chrome 19, and as such `Modernizr` has added a `no-csstransforms3d` class to the HTML element.

All other CSS features that are supported and not supported have been added as well. For example, CSS gradients are also unsupported by this version of Internet Explorer, and therefore a class of `no-cssgradients` has been added, and a fallback can be created via the stylesheet. Using this you can do something as extreme as hiding elements that use this feature, or as mild as providing an alternate view to the user.

By taking advantage of the `no-cssgradients` class that was added for us by our lovely feature-detection library, we can enhance the experience and do it all once from one stylesheet if we like; more on that later.

Here is an example of using CSS to progressively adapt the experience for the user:

```
//No support for 3D transforms.no-cssgradients .header{
background-image: url('images/bg-gradient.png');}
//Supported 3D transforms.cssgradients .header{
background-image: linear-gradient(bottom, rgb(13,25,248) 36%,
rgb(39,53,255) 68%, rgb(67,80,255) 84%);
}
```

As you can see in the preceding example, users with browsers that support CSS gradients will have the browser render a CSS gradient, and browsers that do not will instead have a more resourceful and HTTP request-hungry PNG image as the background image.

Focusing on features, not browsers

Hopefully by now you're starting to get the idea of how feature detection differs from the traditional way of browser (User Agent) sniffing, and how you can use that to your benefit.

Modernizr doesn't ever ask the question, is this Internet Explorer 8, or is this Firefox 13? Instead it asks the question, is there support for a particular feature? Is there CSS gradient support? Is there multiple background image support?

Features can be predicted in parts because although not a finalized specification, HTML5 has been adopted as a new standard by all the major browser manufacturers. That's right, *all* manufacturers, even Internet Explorer. You can bet that feature detection is a world more reliable than UA sniffing as well.

What's wrong with browser sniffing?

So what exactly is wrong with browser sniffing, the process of checking the browser manufacturer and version with which the user is visiting your site with the help of JavaScript? After all, it's gotten us all this far, right?

Consider the amount of browsers in the market today and contrast that with the browsers available five years ago. The market has grown; for example, a particular browser named Google Chrome came from virtually nowhere and quickly grabbed a huge chunk of the market share. Not only has the number of manufacturers for browsers grown, but the devices that they support and are deployed on have grown as well. Where there was once just the Opera browser for instance, Opera at the time of this publication now makes a version of their browser for Windows, Mac, Linux, Mobile, and Tablet browsers.

> Skip the middleman and quit wasting time using unreliable UA sniffing to determine the browser version and manufacturer, and then use that information to deduce what can and cannot be done. Instead go straight to the source and have Modernizr tell you what is and isn't possible.

That sure is a whole lot of browser sniffing to do! Plus that's only one manufacturer. We still haven't gotten around to Mozilla Firefox, Google Chrome, Apple Safari, or Microsoft Internet Explorer. We haven't even come near newer devices coming out every year, for example, Amazon Kindle Fire. Do you really want to have to detect and route for potentially every one of these devices and add in new code each time another one comes to market? I sure don't!

An example of User Agent (browser) sniffing the old way using JavaScript is shown in the following code snippet:

```
<script>
/*** Logging User Agent to the javascript console.
*/console.log( navigator.userAgent );
</script>
```

Doing this from Google Chrome 19 will return the following:

```
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5
(KHTML, like Gecko) Chrome/19.0.1084.56 Safari/536.5"
```

Yikes! So is this Mozilla, Chrome, or Safari? This return code hasn't even told us if we can use an HTML5 or CSS3 feature. We would need to further consult another resource such as `caniuse.com` to check what is and isn't available to us based on the type of browser we discovered. The amount of if-else statements could really get out of hand quickly.

Conversely let us look again at `Modernizr` running a feature test:

```
<script>
if(Modernizr.geolocation){
//That's it! Look how simple and easy that was.
}</script>
```

User Agent sniffing – a big headache and a little payoff

User Agent sniffing has and will always be nothing but an unreliable headache with very little pay off. It won't be getting simpler or easier, only more elaborate. In fact, the very popular and arguably most widely used JavaScript library in the world, jQuery even warns against using UA sniffing and favors feature detection for its own internal usage. In fact, don't be too surprised if jQuery eventually removes UA sniffing methods from its core library altogether in the future. I can almost guarantee you that the moment it isn't needed anymore, it will go the way of the Dodo.

If you are still very much tied to this method for building your web pages, now is the time to wean yourself away from browser sniffing and over to feature detection. We are in a very good place right now because, at long last, no single browser owns nearly the entire market share.

As of April 2012, no single browser had more than 25 percent of the market share, as seen in the following screenshot:

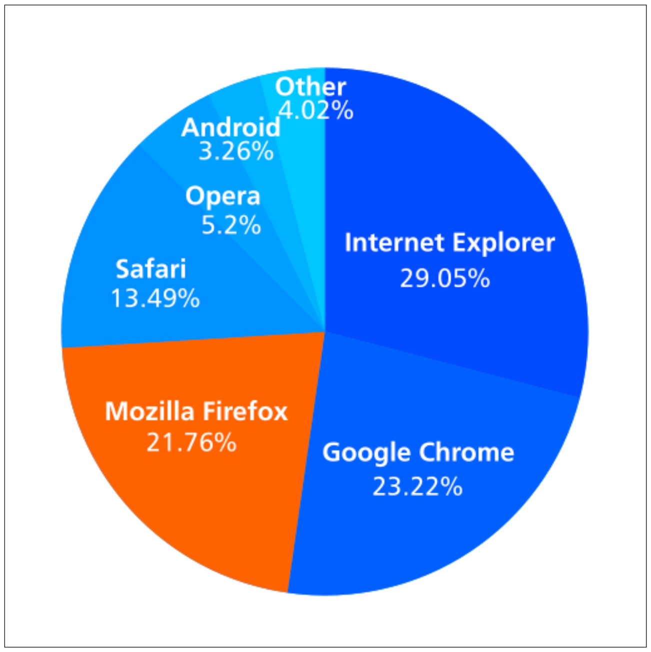


Image courtesy of Wikipedia (http://commons.wikimedia.org/wiki/File:Browser_Usage_on_Wikipedia_February_2012.svg)

Whereas before we were tied to constantly worrying and stressing over support for a particular browser—because although very antiquated it may have accounted for 85 percent of your users (I don't really need to name names here)—no longer is that the case.

In actuality, it makes things much easier for the vendors themselves to not have to deal with UA detection, as they can now focus less on reporting (and occasionally spoofing their information) and more on the supporting and even inventing of new features. It's a win for everyone!

Summary

By now I hope you can see how feature detection allows you to do so much more by using so little. The need to go back into the existing code for new devices is reduced, and with infinitely less worry than we ever had during browser sniffing. In the next chapter, we'll create a foundation for the use of the `Modernizr` library, as well as get a bit more in-depth with how this JavaScript feature-detection library works.

2
Installing Modernizr

We've covered the principles behind Modernizr and its feature detection. We are also aware of what feature detection actually is and how this library can save you a great deal of time as well as headache. Now we are in a good place where we can start putting these concepts into practice. To do so, we'll set up a bit of code to act as our feature detecting HTML foundation, and then add in Modernizr for our feature detecting to really take action.

In this chapter, we'll be diving a bit deeper into Modernizr by:

- Setting up a code base for feature detection based on HTML5 Boilerplate
- Downloading the Modernizr library
- Discussing blocking versus non blocking
- Creating sectioned frames and animating navigation with jQuery
- Spicing up the fonts using the Google font API

Creating the foundation

In a new HTML page, we'll start with a clean HTML5 skeleton, or foundation. I'm a huge fan of HTML5 Boilerplate, so I'm going to use a bit of markup borrowed from the `index.html` file that is included with the download package. I'll be removing a few elements; however, for the sake of staying within scope, I'll also explain the things I am intentionally leaving in. Don't worry about needing the full package for this book. I'll be sure to thoroughly cover every thing we need and you'll have all the code for the lessons here. That being said, I cannot recommend HTML5 Boilerplate highly enough for getting started with HTML5 web pages and it's worth checking out. In fact, some of the Modernizr team is also on the Boilerplate team. HTML5 Boilerplate, which includes Modernizr, is available for download at `http://html5boilerplate.com/`.

For these exercises, I'm going to use a simplified, scaled down version for the sake of clarity to focus more on Modernizr, and less on the many other goodies it contains, such as loading a Chrome frame plugin for legacy versions of IE.

HTML5 ★ BOILERPLATE

In a new directory, using a code editor of your choice, create an `index.html` file and add the following code. I'll explain this as we go as well so don't feel deterred if it feels a bit overwhelming at first.

For my code editor I will be using Sublime Text 2, which is available as a full feature, free trial download at `http://www.sublimetext.com/`, but of course, use whatever text editor you are most comfortable with. I'll be naming my folder `modernizr`, and in my folder my new `index.html` file will contain the following HTML code snippet:

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

```
<!doctype html>
<!-- paulirish.com/2008/conditional-stylesheets-vs-css-hacks-answer-
neither/ -->
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7" lang="en">
<![endif]-->
<!--[if IE 7]>    <html class="no-js lt-ie9 lt-ie8" lang="en">
<![endif]-->
<!--[if IE 8]>    <html class="no-js lt-ie9" lang="en"> <![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js" lang="en"> <!--<![endif]-->
<head>
  <meta charset="utf-8">
<!--  Force IE to use the latest version of its rendering engine -->
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>Getting Started with Modernizr</title>
  <meta name="description" content="A Mondernizr test page.">
  <!--  Modernizr will be included in the head of the page. We'll need
it do do some light lifting before the DOM tree renders load feature
detection and shimming  -->
<script src="modernizr-2.5.3.js"></script>
</head>
<body>
```

```
<h1>Modernizr is great!</h1>
</body>
</html>
```

Now for a quick run-through of what we have so far. In the first line, I am declaring the `doctype` attribute as `html`. The `doctype` attribute, also known as the **Document Type Definition** (**DTD**), lets the browser know that the page being rendered is HTML.

Earlier type definitions in the previous version of HTML4 were much more elaborate, as shown in the following code snippet:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
```

Thankfully in HTML5, declaring DTD is much simpler.

Now onto the next bit. This isn't using Modernizr just yet, but it is doing something slightly Modernizr-esque, in that, it is adding conditional CSS class selectors to the page. These are referred to as CSS conditional comments. IE has supported these conditional comments since Version 5. We can rest assured they will work until IE 10, which has removed support for them. Where they differ is that it is in a way UA sniffing, which is not ideal. As long as you don't rely on this heavily, you should be okay for now. We'll go ahead and leave it in for illustrative purposes of what we're solving with Modernizr and feature detection.

```
<!-- paulirish.com/2008/conditional-stylesheets-vs-css-hacks-answer-
neither/ -->
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7" lang="en">
<![endif]-->
<!--[if IE 7]>    <html class="no-js lt-ie9 lt-ie8" lang="en">
<![endif]-->
<!--[if IE 8]>    <html class="no-js lt-ie9" lang="en"> <![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js" lang="en"> <!--<![endif]-->
```

What is going to happen when the browser is loaded in Internet Explorer is that the conditional comments will be parsed and rendered. If for example, a person is using IE8, the HTML element will show up on the page as the following:

```
<html class="no-js lt-ie9 lt-ie8" lang="en">
```

You could therefore in theory, control certain aspects of these separate browsers by way of conditional class styles, if you wanted to, which would look something like the following code snippet:

```
/* A background for ie7 users */
.lt-ie7 body{
  background: url( upgrade-already-cmon-seriously.gif );
}
/* Everyone else */
body{
  background: url( super-happy-rainbows.png );
}
```

Using conditional comments

That being said, I wouldn't recommend using conditional comments unless absolutely necessary. I like to view conditional styles the way I view laundry day. I put it off as much as I can until it's absolutely necessary. If you'd like to learn more about conditional comments then pay a visit to the quirks mode website at `quirksmode.org/css/condcom.html`.

The no-js class

Also, you must be sure to add a `no-js` CSS class to the HTML element. Modernizr will look for and replace this after it has run along with adding in all the other CSS feature support tags that come from the results of its feature tests. Odds are that the browser will have `js`, unless your user has JavaScript disabled on his/her browser as he/she surfs your web page from the basement wearing a homemade tin foil hat to keep JavaScript from reading his thoughts. That's just a joke of course, but kidding aside, unlike the justice system, all browsers are assumed, and loaded guilty of being featureless; meaning, support for `js` is assumed to be missing and Modernizr is the detective on the case to prove otherwise. In the Modernizr world, browsers are guilty of being featureless until proven innocent. The `no-js` CSS class method isn't something exclusive to Modernizr; you'll see it in a few other places such as the WordPress backend.

```
<html class="no-js lt-ie9 lt-ie8" lang="en">
```

The next thing we do is set the newer, much simpler than before HTML5 metacharacter to `utf-8` for the character encoding for the HTML document:

```
 <meta charset="utf-8">
```

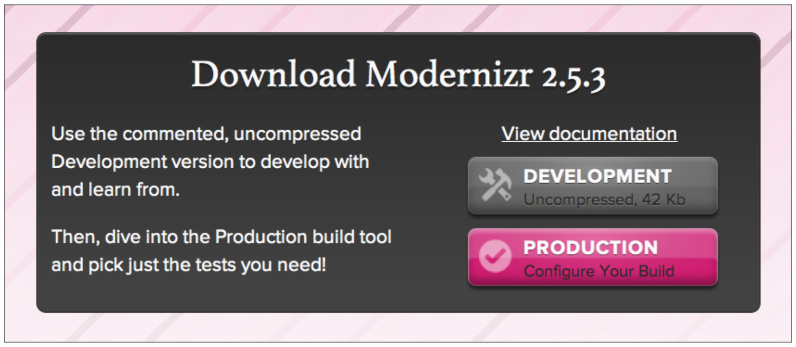
Then we set the compatibility mode for Internet Explorer to use the highest compatibility mode available. There's more on that at `http://msdn.microsoft.com/en-us/library/cc288325(v=vs.85).aspx`.

```
<!--  Force IE to use the latest version of its rendering engine -->
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

By telling IE to use the latest version of its rendering engine in our page, we're doing our part to eliminate as many IE hiccups as possible.

Downloading the Modernizr library

Now that the head of our document contains most of the important bits we come to the Modernizr library loading. I've coded this as though it is in the root of the site folder for now. Of course we haven't actually downloaded it yet so what better time than now to do that. I'm going to go to `www.modernizr.com` and download the full, uncompressed development version in its full glory, which will include more than 40 of the library's feature tests, as well as nicely documented code. Again, save this to the same folder that holds the HTML file we just made. The following screenshot gives us a peek of the website where we can download Modernizr:



Download Modernizr 2.5.3

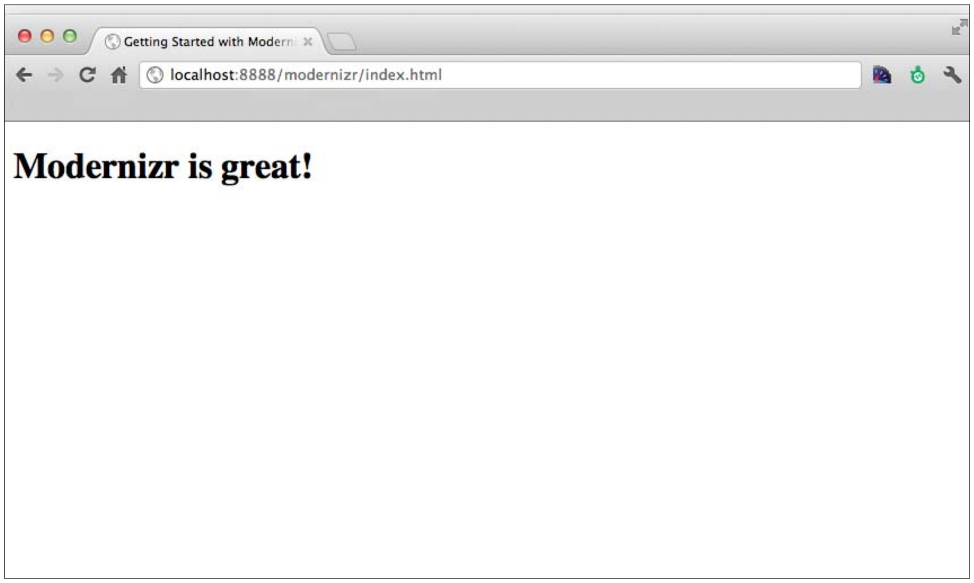
Use the commented, uncompressed Development version to develop with and learn from.

Then, dive into the Production build tool and pick just the tests you need!

There are a couple of other options available to you if you wanted to load this from a CDN instead of downloading it. While this isn't recommended for production environments, it is perfectly suitable for development. The detraction from using this for production is that the development version is probably much heavier than you need. I would recommend figuring out the features you need once your site is built, and then doing your own customized and minified build for your production websites, more on that later in the book. Where the CDN could come in handy would be if you had a series of development sites and didn't want to have to fuss with a download of the library for each site. I would advise that you do a custom build for production sites each time, and use only what you need. This is a very lightweight library but any overhead you can cut away is always going to help the overall performance of your page. This is especially important for blocking scripts. The following links are to some publicly available CDNs in case you're interested:

- From the Microsoft Modernizr CDN, we can visit:
  `http://www.asp.net/ajaxlibrary/cdn.ashx#Modernizr_Releases_on_the_CDN_7`
- From the CDNjs CDN, we can visit:
  `http://cdnjs.com/index.html`

The rest of the code is pretty standard HTML and doesn't really vary at all from HTML4 so I won't go into further detail on it. Go ahead and include it in your page to finish the HTML document. We now have a fully framed setup. Let's go ahead and preview it in the browser and see what we have so far. The following screen will appear:


Getting Started with Modern
localhost:8888/modernizr/index.html

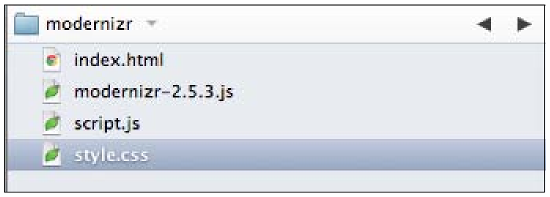
Modernizr is great!

We have the basic skeleton of an HTML5 page now. Let's sprinkle in a couple more important bits to make it more of a page. First off, we'll need a stylesheet so I'm going to make a `style.css` file and include that into the page header. Also, lets give this page some batteries with some jQuery compliments of the Google CDN, and then an empty script file that will be created in the same root as the rest of the site files and included in the page after jQuery. I'm going to name the empty JavaScript file `script.js`.

I won't go into detail on how to link a stylesheet since you can probably already do that with your eyes closed, so just before the ending body tag place the following code. We'll place this code snippet at the end of the script so the DOM tree won't be blocked by it:

```
<!-- jQuery from the CDN -->
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.
js"></script>
<!-- A page for all of our js -->
<script src="script.js"></script>
```

That's a pretty good start. Let's take a moment to recap. Modernizr is loaded into the head of the document along with jQuery being loaded at the end of the document to prevent DOM blocking. After the jQuery library `include` in the footer of the page we also call `include` from `script.js`, which will hold all of the custom JavaScript documents. This means that now is a good time to create that file in the same folder as all the other scripts. So at this point we have an `index.html` file to hold our code, a `style.css` file to contain all the paint, or styling for the page, the Modernizr library included in the header, an include of the jQuery library via the Google CDN, and a `script.js` file to place all of the JavaScript pages. These options can be seen in the following screenshot:



Now we have a really great blend of JavaScript, CSS, and HTML for us to get up and running. jQuery will help us do a lot of the JavaScript heavy lifting and save us even more cross browser JavaScript headaches, and Modernizr will do all the shimming and feature detection for us. Now that everything is connected, we can move on to the foundation code for our project. First we will verify the connectivity of the `js` file, add in the navigation, create five section frames that will have smooth jQuery animation-driven transitions, and then we'll style the whole thing and add in some custom fonts.

Verifying the script connection

Let's make sure everything's included correctly before moving ahead. In the `script.js` file, I'm going to add in a quick statement to be logged into the console. I like to do this to ensure that everything is hooked together before proceeding any further. As simple as this is, it is useful because it helps in any debugging later on when we can rule out that a script isn't properly linked in the HTML document.

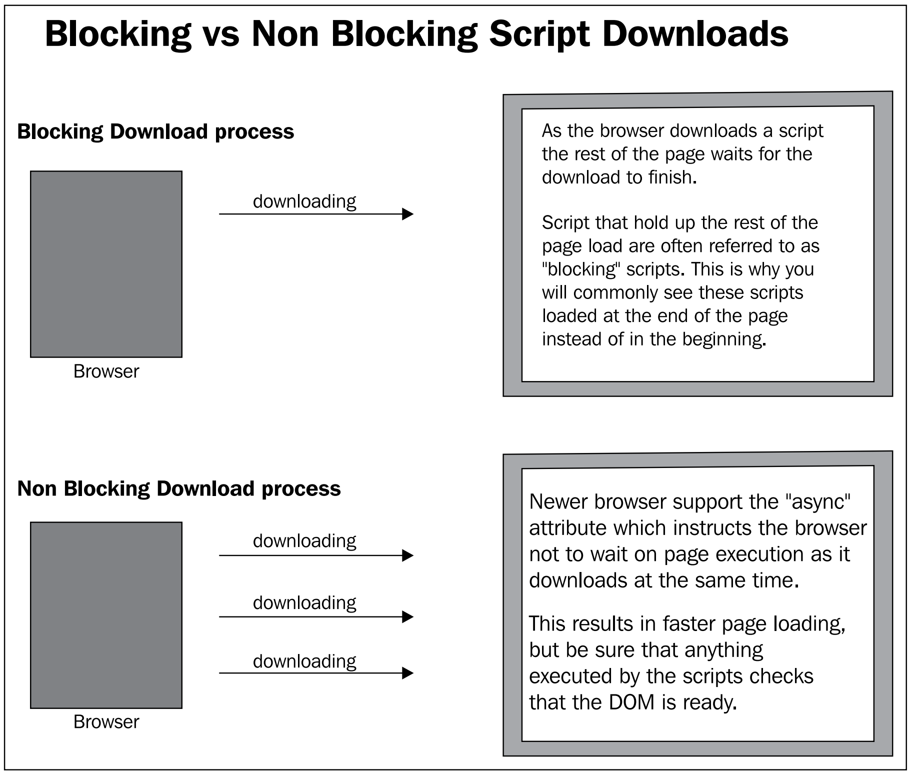
In `script.js` add the following code snippet:

```
<script>
//check that Modernizr is loading by printing out version.
console.log("Modernizr is present and version is: ",Modernizr._
version);
//check that jQuery is loading by printing out version.
console.log("jQuery is present and version is: ",jQuery.fn.jquery);
</script>
```

Blocking versus non blocking

The reason we've put this library into the head of the HTML page and not the footer is because we actually want Modernizr to be a blocking script; this way it will test for, and if applicable create or shim, any elements before the DOM is rendered. We also want to be able to tell what features are available to us before the page is rendered. The script will load, Modernizr will test the availability of the new semantic elements, and if necessary, shim in the ones that fail the tests, and the rest of the page load will be on its merry way.

Now when I say that we want the script to be "blocking", what I mean by that is the browser will wait to render or download any more of the page content until the script has finished loading. In essence, everything will move in a serial process and future processes will be "blocked" from occurring until after this takes place. This is also referred to as single threaded. More commonly as you may already be aware of, JavaScripts are called upon in the footer of the page, typically before the last `body` tag or by way of self-construction through use of an anonymous or immediate function, which only builds itself once the DOM is already parsed.


Blocking vs Non Blocking Script Downloads
Blocking Download process
downloading
Browser
As the browser downloads a script the rest of the page waits for the download to finish.
Script that hold up the rest of the page load are often referred to as "blocking" scripts. This is why you will commonly see these scripts loaded at the end of the page instead of in the beginning.
Non Blocking Download process
downloading
downloading
downloading
Browser
Newer browser support the "async" attribute which instructs the browser not to wait on page execution as it downloads at the same time.
This results in faster page loading, but be sure that anything executed by the scripts checks that the DOM is ready.

The async attribute

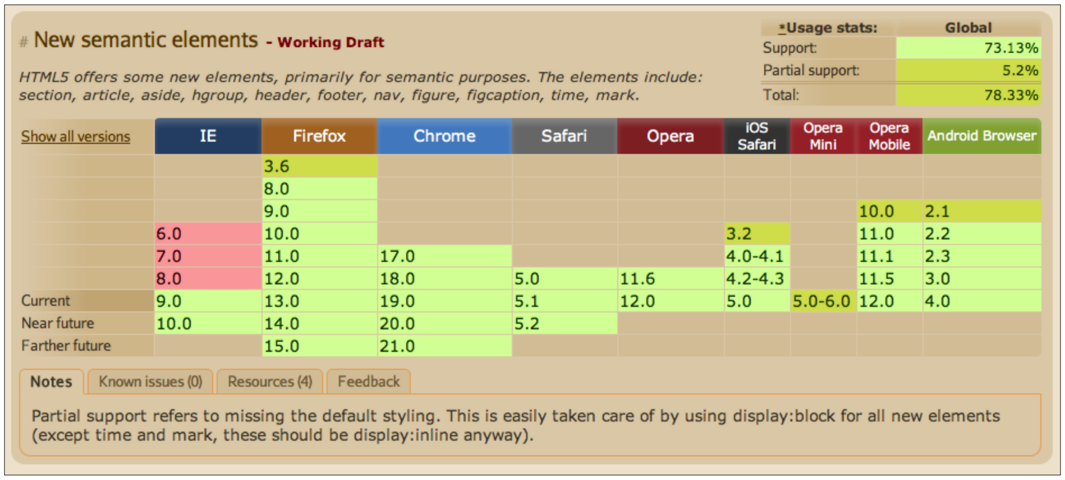
Even more recently, included page scripts can have the `async` attribute added to their tag elements, which will tell the browser to download other scripts in parallel. I like to think of serial versus parallel script downloading in terms of a phone conversation, each script being a single phone call. For each call made, a conversation is held, and once complete, the next phone call is made until there aren't any numbers left to be dialed. Parallel or asynchronous would be like having all of the callers on a conference call at one time. The browser, as the person making all these calls at once, has the superpower to hold all these conversations at the same time. I like to think of blocking scripts as phone calls, which contain pieces of information in their conversations that the person or browser would need to know before communicating or dialing up with the other scripts on this metaphoric conference call.

Blocking to allow shimming

For our needs, however, we want Modernizr to block that, so that all feature tests and shimming can be done before DOM render. The piece of information the browser needs before calling out the other scripts and parts of the page is what features exist, and whether or not semantic HTML5 elements need to be simulated. Doing otherwise could mean tragedy for something being targeted that doesn't exist because our shim wasn't there to serve its purpose by doing so. It would be similar to a roofer trying to attach shingles to a roof without any nails. Think of shimming as the nails for the CSS to attach certain selectors to their respective DOM nodes. Browsers such as IE typically ignore elements they don't recognize by default so the shims make the styles hold to the replicated semantic elements, and blocking the page ensures that happens in a timely manner.

> Shimming, which is also referred to as a "shiv", is when JavaScript recreates an HTML5 element that doesn't exist natively in the browser. The elements are thus "shimmed" in for use in styling. The browser will often ignore elements that don't exist natively otherwise.

Say for example, the browser that was used to render the page did not support the new HTML5 section element tag. If the page wasn't shimmed to accommodate this before the render tree was constructed, you would run the risk of the CSS not working on those section elements. Looking at the reference chart on `http://caniuse.com`, this is somewhat likely for anyone using IE 8 or earlier:



# New semantic elements - **Working Draft**

HTML5 offers some new elements, primarily for semantic purposes. The elements include: section, article, aside, hgroup, header, footer, nav, figure, figcaption, time, mark.

*Usage stats:	Global
Support:	73.13%
Partial support:	5.2%
Total:	78.33%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser
		3.6							
		8.0							
		9.0						10.0	2.1
	6.0	10.0				3.2		11.0	2.2
	7.0	11.0	17.0			4.0-4.1		11.1	2.3
	8.0	12.0	18.0	5.0	11.6	4.2-4.3		11.5	3.0
Current	9.0	13.0	19.0	5.1	12.0	5.0	5.0-6.0	12.0	4.0
Near future	10.0	14.0	20.0	5.2					
Farther future		15.0	21.0						

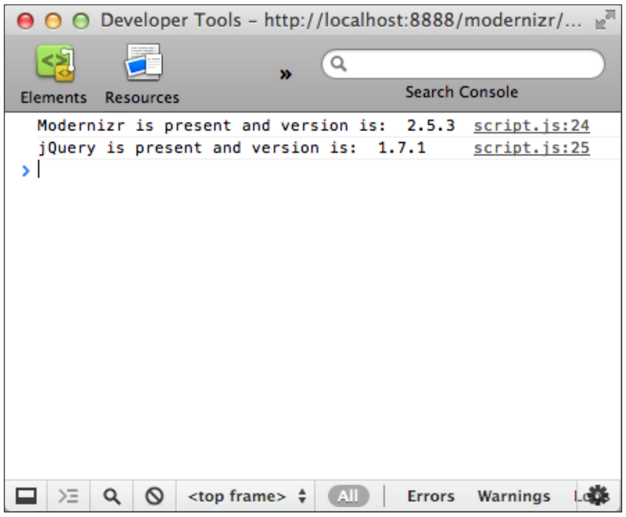
Partial support refers to missing the default styling. This is easily taken care of by using display:block for all new elements (except time and mark, these should be display:inline anyway).

Now that we've adequately covered how to load Modernizr in the page header, we can move back on to the HTML.

Adding the navigation

Now that we have verified all of the JavaScript that is connected, we can start adding in more visual HTML elements. I'm going to add in five sections to the page and a fixed navigation header to scroll to each of them. Once that is all in place and working, we'll disable the default HTML actions in the navigation and control everything with JavaScript. By doing this, there will be a nice graceful fallback for the two people on the planet that have JavaScript disabled. Just kidding, maybe it's only one person. All joking aside, a no JavaScript fallback will be in place in the event that it is disabled on the page.

If everything checks out as it should, you'll see the following printed in the JavaScript console in developer tools:



```
Modernizr is present and version is:  2.5.3  script.js:24
jQuery is present and version is:  1.7.1     script.js:25
```

While we're at it let's remove the `h1` tag as well. Since we now know for a fact that Modernizr is great, we don't need to "hello world" it. Once the `h1` tag is removed, it's time for a bit of navigation. The HTML used is as follows:

```
<!-- Placing everything in the <header> html5 tag. -->
<header>
<div id="navbar">
<div id="nav">
<!-- Wrap the navigation in the new html5 nav element -->
   <nav>
   <a href="#frame-1">Section One</a>
   <a href="#frame-2">Section Two</a>
   <a href="#frame-3">Section Three</a>
   <a href="#frame-4">Section Four</a>
   <a href="#frame-5">Section Four</a>
</nav>
</div>
</div>
</header>
```

This is a fairly straightforward navigation at the moment. The entire fragment is placed inside the HTML5 `header` element of the page. A `div` tag with the `id` field of `navbar` will be used for targeting.

I prefer to use HTML5 purely for semantic markup of the page as much as possible and to use `div` tags to target with styles. You could just as easily add CSS selectors to the new elements and they would be picked up as if they were any other inline or block element.

The section frames

After the nav element we'll add the page section frames. Each frame will be a div element, and each div element will have an id field matching the href attribute of the element from the navigation. For example, the first frame will have the id field of frame-1 which matches the href attribute of the first anchor tag in the navigation. Everything will also be wrapped in a div tag with the id field of main. Each panel or section will have the class name of frame, which allows us to apply common styles across sections as shown in the following code snippet:

```
<div id="main">
<div id="frame-1" class="frame"></div>
<div id="frame-2" class="frame"></div>
<div id="frame-3" class="frame"></div>
<div id="frame-4" class="frame"></div>
<div id="frame-5" class="frame"></div>
</div>
```

Styling the page

Now is also a good time to add some styles for the elements that are new to the page. In the styles.css file we'll add the following to start shaping things into place:

```
/* Basic navigation styling */
#navbar{

  width: 100%;
  height: 50px;
  text-align: center;
  position: fixed;
  top: 0;
  padding: 0;
  z-index: 10;
  background: url( images/stripe-header.png ) 0 0 #333;
  border-bottom: solid 1px white;
}
/* Add a decorative border to the top of the #main element */
#main{
  border-top: solid 2px #333;
}

/* General styling for the websites frames */
.frame{
  height: 700px;
```

```
    position: relative;
    border-bottom: dotted 5px #fff;
    background: url( images/grey-stripe.png );
  }
```

The navigation bar will remain in place and therefore has a fixed setting and a z index of `10` so that it may stay above all the content moving beneath it when any of the links in the navigation are clicked. The general styling for the frames gives each one a height of 700 pixels, and each will have relative positioning applied in preparation of the elements we're going to be placing inside them. I've added a dotted white border as a way to see where the boundaries of each frame reside.

Now I'm going to style the navigation links as well. Nothing over the top at this stage, but enough to get us started using the following code snippet:

```
  /* Apply some very basic styling to the anchor tags in the navigation
  */
  #nav a{
    padding: 0 10px;
    margin: 0 1px;
    background: #eee;
  }
```

This will allow us to see, albeit very crudely, where each navigation item ends and begins.

Smoother transitions with jQuery

There is not a lot happening yet, but we're really getting up and running with a great foundation so let's continue on that streak. If we click the navigation links you can see how the default browser behavior is to scroll to that particular section with the `id` field matching that of the `href` attribute. While not very exciting, I think it is important from time to time to stop and view the experience from as many levels as possible. Feel free to take a moment and preview this all in the browser. Clicking each link in the navigation should sort of knee jerk the page down to the relative section.

Now that's great and fully functional, but let's write some JavaScript that will, thanks to a little help from jQuery, create a more graceful scrolling transition via the jQuery's animation method.

In the `scripts.js` file, I'll be writing the following immediate function to be attached to the jQuery object:

```
  /**
   * Immediate function to disable the default browser behavior for the
   * navigation anchor tags and replace it with a much smoother animation
```

```
    * using jQuery.animate
   **/
    ( function($){
      //Bind a function to the 'click' event of the anchor tag
     $( '#nav a' ).click(function() {

         //The href attribute which matches the id of the div element.
          var destination = $( this ).attr( 'href' );
          //compensate for the height of the navigation bar
          var navBarHeight = $( '#navbar' ).height();
         // Calculate the amount using the distance of the element from
         the page top,
         // and the height of the navigation bar.
          var amount = $( destination ).offset().top - navBarHeight;
          $( 'html,body' ).animate( { scrollTop: amount }, 'slow' );

         // Cancel the default browser behavior by returning false.
         // which will disable the hash in the address bar from appearing.
         return false;

   });

    } )( jQuery );
```

I also removed the previous bits that were used to check the connectedness of our earlier JavaScripts since we don't need them anymore now that we know everything is hooked up correctly.

The script is fairly straightforward and uses jQuery to do the heavy lifting. On page load, all of the anchor elements that are children of the `nav` element will be bound with a function that is triggered by the mouse via a click event. When any one of these links is clicked, the page will scroll to the `div` element that has an `id` field matching the `href` attribute of the link. So the first link, which has an attribute of `#frame-1`, will smoothly scroll down the page until the `div` element with the element containing the matching `id` field of `frame-1` is reached. The uppermost part of the `div` element will then be positioned at the uppermost part of the page. The only thing is, we have a fixed element currently occupying some real estate at the top of the page, so we'll need to compensate for that. This is done by using jQuery to keep track of how high the navigation is and then subtracting that from the position the page would normally animate to. The very last part, which is returning `false`, is removing the default activity that the browser would normally do, which would have been to add the `href` attribute up into the top address bar, before jumping down to that section of the page, similar to the way the `name` attribute would work as an anchor in the page's content.

That's a bit undesirable, and using the returning `false` tactic will allow for a smoother experience, although you could leave it in if you wanted to remember the state of the page. By that I mean if you wanted a particular section of the page to be displayed on the initial page load minus any action of the user. That is to say, if you wanted to set up a bit of functionality that would read the `id` field from the address bar and move the page to that state without any clicking by the user. For example, if you wanted to share that section of the page with another person. By keeping the `id` field, also known as a hash, or `hashbang`, in the URL and combining it with something to listen for, perhaps the newer `hashchange` event and handle it based on parameters, you would accomplish a somewhat crude way of remembering the browser state.

If you refresh the page now, you'll see that the default and rather jumpy browser behavior has been replaced by a nice jQuery powered smooth transitioning effect using only a few simple lines of code. Each panel should now gracefully animate into view. Much better.

Determining the base experience

Now that we have the bones of the page in place, it would be a good time to stop and determine what the base, or core-level experience will be. Based on this determination, any necessary fallbacks or degradation can be put into place as well, as enhancements that the page will inherit and pick up on as the browser is updated to support such features.

Our base experience will use and assume support of the following:

- Images for icons
- Opacity
- Hex colors
- New HTML5 semantic elements
- Custom fonts
- Fixed positioning

Images for icons

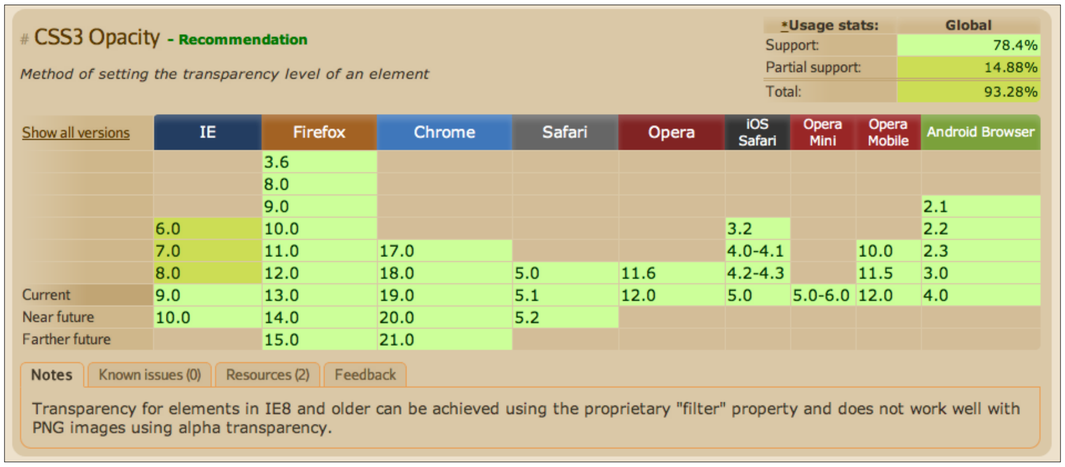
For the core experience, we're going to assume that our users aren't supporting the latest and greatest features used to make icons. These being, but not limited to, `border-radius`, `css-gradients`, `box-shadow`, and so on. The core experience is more or less a CSS2 leaning experience, but don't worry, we'll be folding in some really cool CSS3 stuff as well later on. If we were UA sniffing, the level of support we'd be looking at would roughly be IE 8, Safari 5, Opera 11.6, Chrome 17, Firefox 11, or future versions of these browser applications. For the purposes of this project, we won't be concerning ourselves with the mobile versions of these browsers but generally the newer mobile and tablet browsers are some of the most cutting edge technologies, so aside from device width considerations this experience would in all likelihood work in those as well. Maybe, having to buy a new cell phone every couple of years has its benefits after all. So the next time you see somebody rushing out to get the latest gadget, you can smile to yourself knowing it's probably packing some bleeding edge technology, such as WebKit, in addition to its shiny new high retina display that packs 4 pixels into the space that previous phones only occupied with a single pixel.

Images in this chapter will be minimal and consist of tabs for the navigation, a logo image, and a couple of background images as well. In *Chapter 3*, *Using Modernizr the Right Way*, we will see about doing away with all of them and replace them with CSS versions and even do a little bit of animating with the logo. The benefit of this is they will be vector-based, instead of pixel-based, carry fewer overheads with bytes, and eliminate the need for the HTTP requests that currently serve them up. Just imagine how happy that's going to make all of those performance evaluation browser tests at the reduction of HTTP requests and page size.

Opacity

The first thing you are probably thinking is, what about IE 8? After all, IE 8 doesn't actually support the opacity parameter despite all the rumors that it would. Well, this is where the degraded experience considerations come into play. If we again go back to the good folks at `caniuse.com` and view the chart on CSS opacity, we'll see that while it is not supported, there is partial support by way of filters. So if we use a little, actually a lot of vendor prefixing and steer clear of trying to get crazy on some transparent PNGs, which can end up looking like white haloed blobs when faded in and out, we'll be in a pretty good shape.

Here's what the vendor prefixing will look like:



CSS3 Opacity - Recommendation

Method of setting the transparency level of an element

*Usage stats:	Global
Support:	78.4%
Partial support:	14.88%
Total:	93.28%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser
		3.6							
		8.0							
		9.0							2.1
	6.0	10.0				3.2			2.2
	7.0	11.0	17.0			4.0-4.1		10.0	2.3
	8.0	12.0	18.0	5.0	11.6	4.2-4.3		11.5	3.0
Current	9.0	13.0	19.0	5.1	12.0	5.0	5.0-6.0	12.0	4.0
Near future	10.0	14.0	20.0	5.2					
Farther future		15.0	21.0						

Notes | Known issues (0) | Resources (2) | Feedback

Transparency for elements in IE8 and older can be achieved using the proprietary "filter" property and does not work well with PNG images using alpha transparency.

If you don't want to be bothered trying to remember the vendor prefix naming conventions, there are many sources on the Internet widely available and happy to do that for you in a matter of milliseconds. One I like in particular is Prefixr, which can be found at `www.prefixr.com`. Simply paste in your CSS, click to generate and you're just a copy and paste away from having a fully vendor prefixed stylesheet:

```
/* Vendor prefixed, and IE appeased opacity */
.watermark{
  /* MS Filtering to fill in gaps in opacity support */
  -ms-filter: "progid:DXImageTransform.Microsoft.Alpha(Opacity=30)";
  filter: alpha(opacity=30);
  opacity: .3;
}
```

We'll use this CSS class to simulate a watermarking effect in our page example in the next chapter.

Hex colors

Despite there being a few new color options to the game, we'll stick with the traditional plain toast RGB hexadecimal values for the core user experience for the time being. New to the field are `hsla` and `rgba`, which in addition to color, allow for luminosity and opacity.

HTML5 semantic elements

Yes, they are new and cool, and the good news is that even if your browser is older than your wine collection, and there's a moderate possibility the new elements aren't supported out of the box, shimming has our back, so these make it into the core experience.

Custom fonts

Font face slips in, albeit just barely. It's been supported since IE 8 with a little massaging; Firefox has had it since Version 3, and Chrome and Safari have had support woven in for a while as well. Plus, thanks to native CSS fallbacks, we can put in an additional system-available fonts as a stack to fill in any gaps.

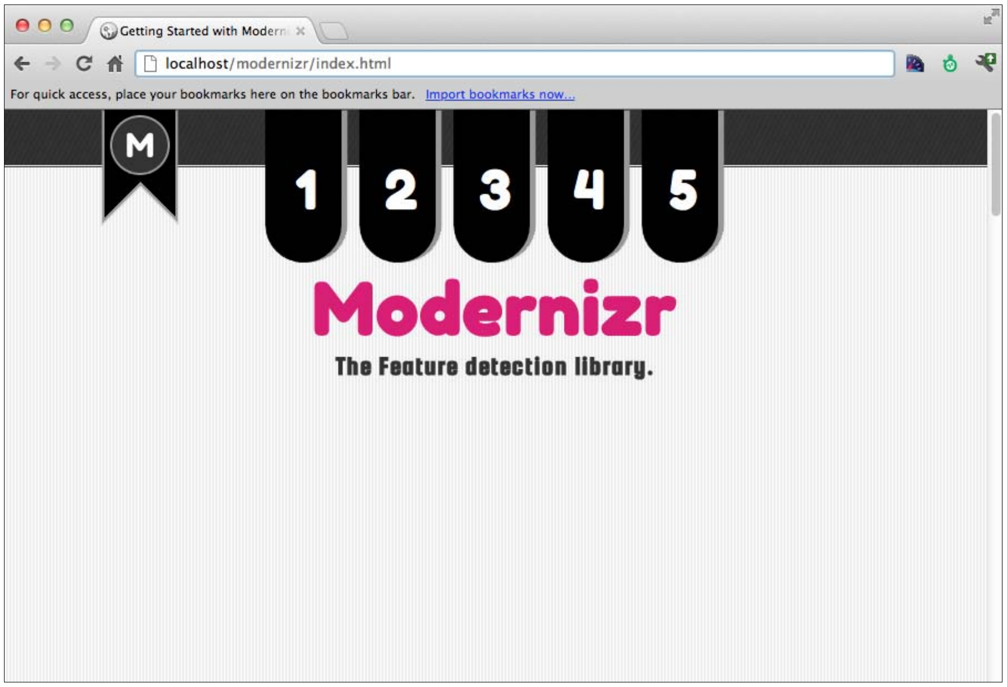
Fixed positioning

This title refers to the position of elements in the page. Our `nav` element will have a fixed position and will stay there oblivious to page scroll. Fixed positioning isn't exactly the new kid on the block, after all it's been supported since IE 7. Nevertheless, it's something to be aware of and is included in the core experience.

Time to put this into practice, and by practice I mean HTML, so let's piece in some new bits and bobs and then cover what's new. I'm going to build off the previous skeleton we made.

The core HTML

The new code we're going to add will result in the following core experience that we will fold in all sorts of CSS3 features in the next chapter:


Getting Started with Modern
localhost/modernizr/index.html
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...
M
1
2
3
4
5
Modernizr
The Feature detection library.

The first bits of the following code will be the conditional styles we mentioned earlier, as well as declaring the new HTML5 document type:

```
<!doctype html>
<!-- paulirish.com/2008/conditional-stylesheets-vs-css-hacks-answer-
neither/ -->
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7" lang="en">
<![endif]-->
<!--[if IE 7]>    <html class="no-js lt-ie9 lt-ie8" lang="en">
<![endif]-->
<!--[if IE 8]>    <html class="no-js lt-ie9" lang="en"> <![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js" lang="en"> <!--<![endif]-->
<head>
```

Now, we declare the character set for the page and also tell IE to use the latest version of its rendering engine available. Furthermore, we give our page a brief meta description as shown in the following code snippet:

```
<meta charset="utf-8">
<!--  Force IE to use the latest version of its rendering engine -->
<meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>Getting Started with Modernizr</title>
  <meta name="description" content="A Modernizr test page.">
```

Linking to the stylesheet and Modernizr library, as well as making a call to include Google web fonts in the document, is shown in the following code snippet:

```
  <link href='http://fonts.googleapis.com/css?family=Fredoka+One|Squad
a+One' rel='stylesheet' type='text/css'>

<link rel="stylesheet" href="style.css">
<!--  Modernizr will be included in the head of the page. We'll need
it do do some light lifting before the DOM tree renders load feature
detection and shimming  -->
<script src="modernizr-2.5.3.js"></script>
</head>
<body>
   <header>
  <!---  The page navigation  -->
  <div id="navbar">
   <div id="logo"></div>
      <div id="nav">
       <nav>
      <a href="#frame-1">1</a>
      <a href="#frame-2">2</a>
      <a href="#frame-3">3</a>
      <a href="#frame-4">4</a>
      <a href="#frame-5">5</a>
      </nav>
      </div>
    </div>
    </div>
  </header>
  <div id="main" role="main">
```

Now we set up a series of frames beginning of course with frame one, and increasing until five frames are reached. Each frame will hold a new series of features, often building off of the previous frame's contents. For the first frame, a title and subtitle are what we will begin with. The other frames will be elaborated on a bit later in the book. For now, the focus is on getting a structure into place. Beyond the frames, there is the footer and a call out to include the jQuery library on the page as well as a call out to custom `script.js` page where the heavy lifting for JavaScript will occur as shown in the following code snippet:

```
  <div id="frame-1" class="frame">
  <hgroup>
   <h1 class"title">Modernizr</h1>
   <h2 class="subtitle">The Feature detection library.</h2>
  </hgroup>
```

```
    </div>
       <div id="frame-2" class="frame ">Section Two</div>
       <div id="frame-3" class="frame">Section Three</div>
       <div id="frame-4" class="frame ">Section Four</div>
       <div id="frame-5" class="frame">Section Five</div>
     </div>
     <footer>
     <div class="foot">Footer</div>
    </footer>
    <!-- JavaScript at the bottom for fast page loading →
    <!-- jQuery via Google CDN -->
   <script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.
  js"></script>
   <script src="script.js"></script>
  </body>
  </html>
```

This naturally wouldn't be complete without the CSS paint, so here's that as well. This first part handles the basic styles for all of our header tags. Nothing too complex, yet beyond the custom Google font. The HTML is as follows:

```
/* Zero out page margins and pad the top of the page to compensate for
the header */
body{
  padding-top: 50px;
  margin: 0;
  background: #eee;
}

/* Header Elements */
h1, h2{
  display: block;
  margin: 0 auto;
  text-align: center;
}

/* A title for the first section as well as styling with the Google
web fonts */
h1{
  font-family: 'Fredoka One', cursive;
  padding-top: 10%;
  font-size: 4em;
  color: #D91E76;
}
```

```
/* Style the subtitle as well, also using a google web font */
h2{
  font-family: 'Squada One', cursive;
  color: #333;
}
```

Next we style the navigational elements. The navigational elements consist of the navigation bar, logo, and anchor elements. We will also add in a hover state style for the section links using the following code snippet:

```
/* Navigation */
#navbar{

  width: 100%;
  height: 50px;
  text-align: center;
  position: fixed;
  top: 0;
  padding: 0;
  background: url( images/stripe-header.png ) 0 0 #333;
  border-bottom: solid 1px white;
  z-index: 10;
}

/* The logo png, in the next chapter we'll enhance this and make it
entirely css based */
#navbar #logo{

  position: absolute;
  width: 70px;
  height: 110px;
  background: url( images/logo.png );
  top: -5px;
  left: 10%;
}

/* Apply some very basic styling to the anchor tags in the navigation
*/
#nav a{

  margin: 0 5px;
  line-height: 3em;
  color: white;
  font-size: 3em;
  text-decoration: none;
```

```
    font-family: 'Fredoka One', cursive;
    background: url( images/link-tabs.png ) no-repeat 0px 0 transparent;
    display: inline-block;
    height: 138px;
    width: 75px;
    text-align: center;
  }

  /* Hover states for the nav links. */
  #nav a:hover{

    color: black;
    background-position: -75px 0;
  }
```

Last, we style the main page frame that wraps the sections of the page. We also give each frame a general styling including the height and the bottom footer a black background using the following code snippet:

```
  /* Decorative border for the top element */
  #main{
    border-top: solid 2px #333;
  }

  /* General styles for all the sections, i.e. frames */
  #main .frame{
    height: 700px;
    position: relative;
    border-bottom: dotted 5px #fff;
    /* Image fallback, to be replaced by CSS3 hotness in chapter 3 */
    background: url( images/grey-stripe.png );
  }

  /* Footer , basic black for now*/
  .foot{
    background: black;
    height: 50px;
    border-top: solid 5px #333;
  }
```

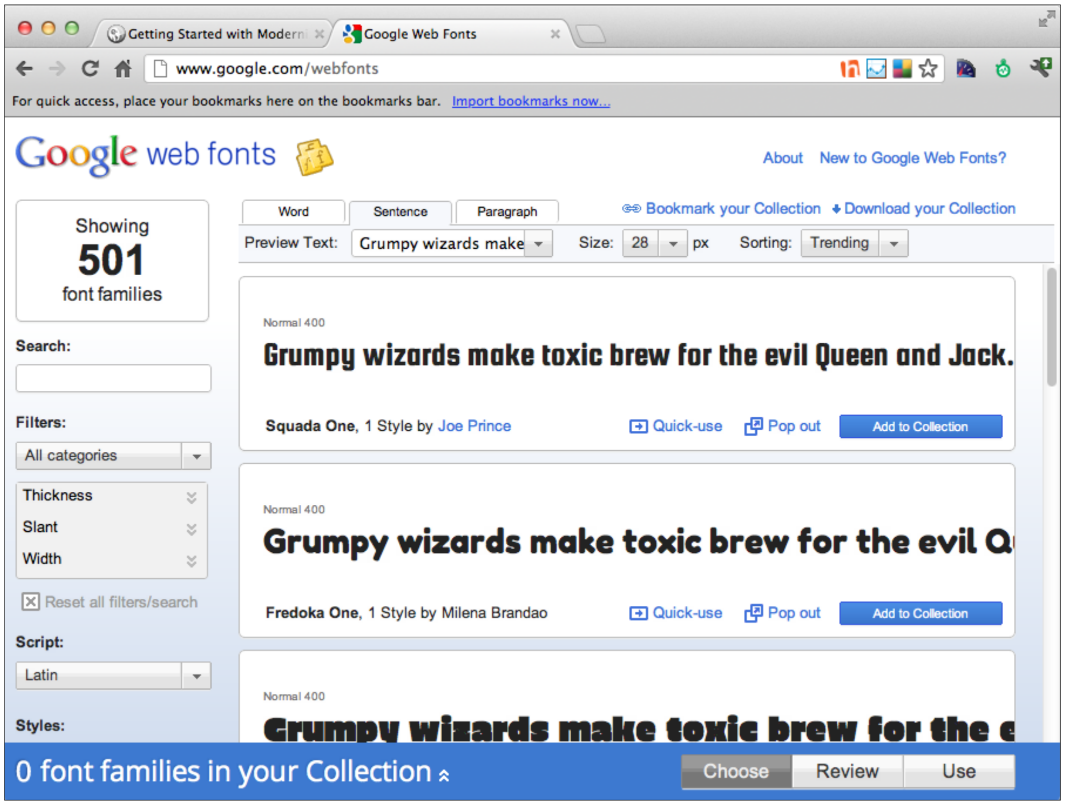
The JavaScript will remain the same for now. Let's sum up the fresh code. The majority of work has been done in paint, which is what I call the CSS, but there are some new things in the HTML.

Google fonts API

Google has an ever growing gold mine of spectacular and guilt-free open source fonts at our disposal. I've included a couple of fonts that I really like that spruce up the `h1` and `h2` tags for the first section. The code can even be generated for you on the website; in general the call is made as an HTTP request and each font is separated by a pipe. I've added it to the header of our web page using the following code snippet:

```
<link href='http://fonts.googleapis.com/css?family=Fredoka+One|Squada+
One' rel='stylesheet' type='text/css'>
```

The Google web fonts can be seen in the following screenshot:



Preview Text: Grumpy wizards make
Size: 28 px
Sorting: Trending

Normal 400
Grumpy wizards make toxic brew for the evil Queen and Jack.
Squada One, 1 Style by Joe Prince

Normal 400
Grumpy wizards make toxic brew for the evil Q
Fredoka One, 1 Style by Milena Brandao

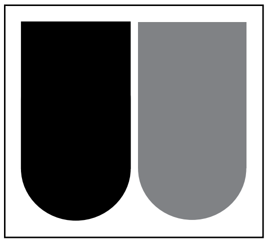
Normal 400
Grumpy wizards make toxic brew for the e

Apart from the web fonts, the rest of the additions are your everyday, run-of-the-mill HTML elements.

The CSS

The CSS for now is more or less a bunch of ordinary CSS2 with some background images at this stage in the game. The bulk of the look is with the header so let's cover that a bit.

The links are displayed as inline-block elements, which allows them to stay centered and also have a width and height applied to them. This is crucial if we want to be able to use a background image with them. The background image is a combination of an off state and a hover state. This means that the image has the visual look of the normal and hover state combined as one image which will cut HTTP requests in half. The hover style in the CSS moves the position of the background image to fulfill the look for the visual hover.

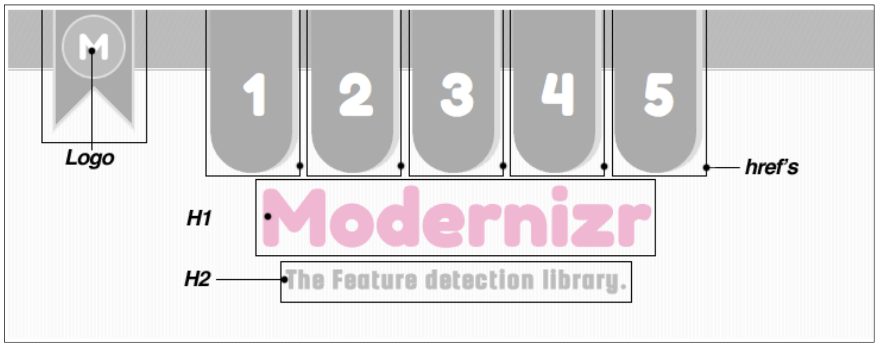


The preceding image is a look at the `nav` section of CSS again. Notice how the `hover` state swaps the position of the background image and changes the text color. Also notice that the font family is using one of the families included by the Google Font API. The code snippet used is as follows:

```
/* Apply some very basic styling to the anchor tags in the navigation
*/
#nav a{
  margin: 0 5px;
  line-height: 3em;
  color: white;
  font-size: 3em;
  text-decoration: none;
  font-family: 'Fredoka One', cursive;
  background: url( images/link-tabs.png ) no-repeat 0px 0 transparent;
  display: inline-block;
  height: 138px;
  width: 75px;
  text-align: center;
}
```

```
#nav a:hover{

  color: black;
  background-position: -75px 0;
}
```

We will accomplish the following identical look using no images at all in the next chapter, using CSS3 in conjunction with some CSS classes provided by Modernizr:



The header that spans across the top of the page in a fixed position uses a striped PNG image for the background, again in the next chapter we'll duplicate that with CSS3. The important points are the position set to fixed, the background URL, which is a PNG image (as small as possible about 115 bytes) with a fallback background color of `#333` as shown in the following code snippet:

```
/* Navigation */
#navbar{

  width: 100%;
  height: 50px;
  text-align: center;
  position: fixed;
  top: 0;
  padding: 0;
  background: url( images/stripe-header.png ) 0 0 #333;
  border-bottom: solid 1px white;
  z-index: 10;
}
```

The logo is a PNG image as well, again rather straightforward. It's a background image of its `div` element. The logo will be absolutely positioned inside the `navbar` element using the following code snippet:

```
/* The logo png, in the next chapter we'll enhance this and make it
entirely css based */
#navbar #logo{
  position: absolute;
  width: 70px;
  height: 110px;
  background: url( images/logo.png );
  top: -5px;
  left: 10%;
}
```

The final visual piece is the stripe background of each section panel. Again, this is a background image PNG with the overhead size as small as possible. Our PNG background image weighs in at only 81 bytes, as seen in the following code snippet:

```
/* General styles for all the sections, i.e. frames */
#main .frame{
  height: 700px;
  position: relative;
  border-bottom: dotted 5px #fff;
  /* Image fallback, to be replaced by CSS3 hotness in the next
chapter */
  background: url( images/grey-stripe.png );
}
```

There we have it. Each section panel will have a striped background, a thick bottom dashed border, and is ready for some fresh visual CSS3 content! I have also pre-emptively set its position to relative in preparation for any content we place inside that may end up being absolutely positioned. This is so the items inside each frame will be positioned absolutely to the frame itself as opposed to the page.

The header is placed inside the new HTML5 `h group` block and has been set to be a block element; typically this element is set as an inline element by default, which is common for text. By setting these `h1` and `h2` elements as block-level elements, we get to take advantage of centering them with the left and right margin being set to auto. Additionally, they have some of the Google font sprucing up as well, and are on their way to even more CSS3 being added in.

Summary

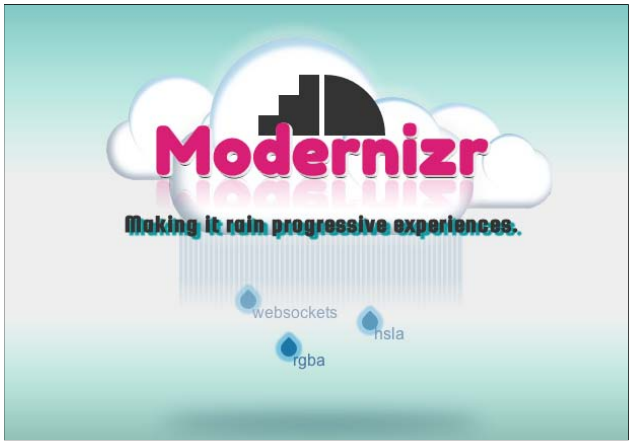
That's it for our base experience! We've built an HTML5 skeleton, included the full Modernizr library, the jQuery library from the CDN, and with our own native JavaScript created a smooth animation. We created an image light page header that remains fixed throughout the page scrolling and frames for the page navigation to animate to. Our first section, or frame, for now contains a group of headers that are styled by some custom fonts compliments of the Google Fonts API.

What's next? Since we have the core experience ready to go, we can move onto some CSS3 gravy for these mashed potatoes. In the next chapter, we will enhance with CSS3 using Modernizr as our feature compass. This means vector-based icons with border radius and gradients, fonts that pop off the page, animated logos, and a whole lot more!

3
Using Modernizr the Right Way

The Modernizr library is a tool that is built using JavaScript. Similar to the way that JavaScript can be used in a "wrong" and a "right" way, the same rules apply to Modernizr. By "wrong" and "right" I mean that it is very forgiving and will work whether implemented correctly or incorrectly. This chapter will be broken into phases or frames, and each frame will add something new to progress the experience. We'll focus on how a core experience can be enhanced by progression into more cutting edge features available in current cutting edge browsers.

I've always been a fan of the old 8-bit and 16-bit video games. So, I thought that would be a great look for something fun we can build. We'll start out with the first frame, building on our foundation of code that we made in the previous chapter. In the first frame we'll keep the look pretty close to that of the previous chapter, but this time round we'll feature detect to pull out the extra bits that can be done by more modern browsers. The following screenshot gives you a peek at our final video game built:



By the end of this chapter when we reach frame five, we will have achieved the following:

- A CSS cloud will be raining every feature that tested true
- All images will have been replaced so that only CSS remains for the visuals

Frame 1 – swapping images for CSS

In this section, we will be doing the following:

- Replace all core design images with CSS-only versions
- Recreate the logo using CSS border radius, box shadow, and web fonts
- Spice up the header logo using CSS gradients
- Recreate the navigation tabs using CSS border radius
- Replace the frame background using CSS gradients and background size
- Begin building the video game's look and feel

The first set of feature testing we'll be doing is for CSS gradients and also background size. This will allow us to achieve identical look but free from the shackles of PNG images. Cutting down on the amount of images used on the page will free up resource overhead and allow faster page loads. We won't remove the need for images entirely just yet, but the main site elements will all be converted.

Keeping it WebKit, for now

Everything that we will be doing right now will be using the WebKit vendor prefix of `webkit where needed`, which will work in WebKit browsers such as Safari and Chrome. Then towards the end of the chapter, we'll make sure all the vendors are addressed and we'll fill in those as well. Sticking with just one vendor for now will keep things simple as we work to get to the point of progressive experiences.

The main elements—navigation, background, and logo—currently consist of stripes and curves. Both of which can be recreated with CSS.

The stripes

The stripes can be replicated using CSS gradients in tandem with background size. Before we can use something like that we'll want to know that it's available to be used. We have two options to do this: either use the HTML selectors added for us or use the JavaScript to test a combination of features and on success add new classes to the DOM elements.

The following code snippet gives a small example of how that would be accomplished with CSS in our stylesheet:

```
/* Method 1: Using the seasoned html elements and CSS. */

#frame-1{
  ... Image version of striped background
}

.cssgradients #frame-1{
  ... CSS gradient version of striped background
}
```

The following code snippet gives an example of how the same effect can be accomplished in our JavaScript:

```
// Method 2: Using JavaScript to add a CSS class.
if( Modernizr.cssgradients ){
  //Use jQuery to season with the css class.
  $( "#frame-1" ).addClass("stripes");
}
```

We will then add the following `stripes` CSS class containing the gradient instructions for the earlier example:

```
/* CSS Stripe class to be added by JavaScript*/
.stripes{
   CSS Gradient info
}
```

With the new class added to the frame, the all new CSS version of the `stripes` will gracefully replace the original PNG image version.

Time to put this to practice. First we'll cover the navigation and then move into the first frame. Once that's completed we can move into adding fun with some visual elements.

In the stylesheet, I'll remove the background from the original `frame` class:

```
/* Structure class for each frame panel */
.frame{

  height: 700px;
  position: relative;
  border-bottom: dotted 5px #fff;
  background-color: #eee;
}
```

Also I'll create a CSS class called `vert-stripe-img` that will contain the background image removed from the frame, using the following code snippet:

```
.vert-stripe-img{
  //The original background image relocated to it's own class.
  background: url( images/grey-stripe.png );
}
```

And while we're at it, let's also create a similar class with the near identical look but using the CSS gradients and background size this time and naming it `vert-stripe-gradient`:

```
.vert-stripe-gradient{
  background-image: -webkit-linear-gradient(0deg,transparent 50%, #fff
  50%);
  background-size:5px;
}
```

I'm going to make a slight modification to the earlier `frame` CSS class by removing the background image. I'll be doing this so that `frame` acts more like a structure class and not a structure and styling combination. Now all instances of the `frame` class in the code will be for the structure only.

As you can see by checking the results for gradient and background size support via JavaScript, we can toggle what type of background is displayed. If the support is available the PNG background is exchanged for the purely CSS gradient version. I've made the stripes of the CSS version a little bit wider, as shown in the following code snippet, so it's a little easier to distinguish which is in use for the purposes of these exercises:

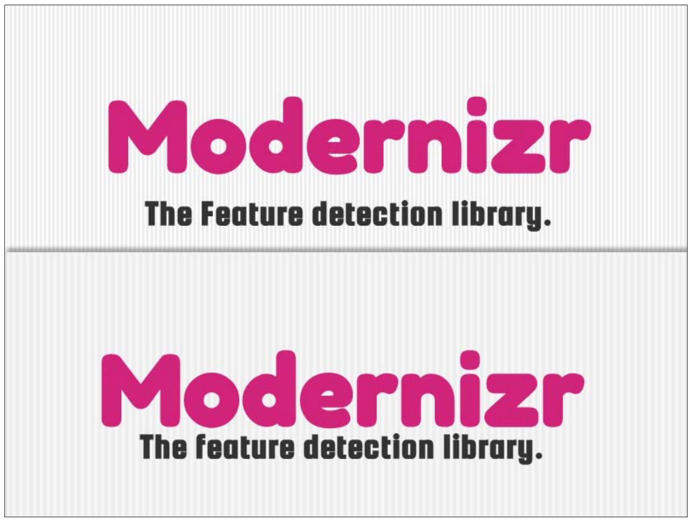
```
$(document).ready(function(){
//Check for CSS Gradient and Background Size support.
// If both are present then switch to using the gradient version.
if( Modernizr.cssgradients && Modernizr.backgroundsize ){
  $('#frame-1').removeClass('vert-stripe-img').addClass('vert-stripe-
gradient');
  }
});
```

From this point forward, I'm going to automatically add the `vert-stripe gradient` class to the subsequent frame panels in the exercises ahead. This is intended more as a proof-of-concept and in the future frames we will assume that this feature exists. My goal is that by the end of this chapter the gears in your head will be turning, thinking about all the possibilities available to you with this library. Hopefully, this gives an idea about how to use feature detection to toggle based on this feature set. The preceding example uses JavaScript.

However, the same could be achieved using only CSS selectors and chaining them together, as seen in the following code snippet:

```
//The same effect achieved with CSS only.
#frame-1{
  background: url( images/grey-stripe.png );
}
//The background gradients achieved by chaining together the CSS
classes.
.backgroundsize.cssgradients #frame-1{
  background-image: -webkit-linear-gradient(0deg,transparent 50%, #fff
50%);
  background-size:5px;
}
```

Doing this would achieve the same effect. By chaining together the two CSS classes, the browser will pick up and apply the gradients only if those two classes exist inside the HTML element. The effect of the preceding code snippet where we widened the stripes can be seen in the following screenshot:


Modernizr
The Feature detection library.
Modernizr
The feature detection library.

Now that we have stripes, let's do the same thing but using slanted stripes for the top navigation header on the page using the following code snippet:

```
//Image class for the striped navigation.
.nav-stripes-img{
 background: url( images/stripe-header.png ) 0 0 #333;
}
```

```
//Gradient class for the striped navigation.
.nav-stripes-gradient{
  background-size: 10px 10px;
  background-image:
-webkit-gradient(linear, 0% 0%, 100% 100%,
  color-stop(25%, rgba(0, 0, 0, 0.1)),
  color-stop(25%, transparent),
  color-stop(50%, transparent),
  color-stop(50%, rgba(0, 0, 0, 0.1)),
  color-stop(75%, rgba(0, 0, 0, 0.1)),
  color-stop(75%, transparent),
  color-stop(100%, transparent));
  background-color: #333;
}
```

And naturally we need to add the controller that enables this gradient striped in the JavaScript. It will be virtually identical to the previous statement, as seen in the following code snippet:

```
<script>
$(document).ready(function(){

//Check for CSS Gradient and Background Size support.
// If both are present then switch to using the gradient version.
if( Modernizr.cssgradients && Modernizr.backgroundsize ){

// Remove the navigation header image based
// stripes and replace with the CSS as well.
$('#navbar').removeClass('nav-stripes-img').addClass('nav-stripes-
gradient');
  }
}); //end document ready
</script>
```

With the JavaScript and CSS in place, the browser should now be showing the CSS gradient version of the stripes, which again are just a little bit wider than the image-based ones so they are visually easier to detect while we are testing features. That should about do it for the striped elements, which brings us to the curvy elements.

The curves

By curves I refer to the rounded parts, whether they be a full circle, like in the logo or a rounded edge, like without top tabs, which remind me a bit of cartoon piano keys. All of these are done with images for now but can be completely replicated using `borderradius`.

As this is a single feature dependency and not multiple, the previous example needed both, gradient support as well as background size support. We can do this much more easily using just the stylesheet we want without having to worry about the order of these special element declarations in the HTML tag. This is done by simply placing `borderradius` before whatever element we are targeting. I prefer to target with just CSS as much as possible because it keeps things simple and it's all done in a single file so it's easier to debug. Let's first start with the navigation tabs. I'm going to prepend a class of `no-borderradius` to the original, and also add in a border radius supported version, as seen in the following code snippet:

```
/* Navigation tabs using the current image based structure */
.no-borderradius #nav a{
  margin: 0 5px;
  line-height: 3em;
  color: white;
  font-size: 3em;
  text-decoration: none;
  font-family: 'Fredoka One', Arial;
  background: url( images/link-tabs.png ) no-repeat 0px 0 transparent;
  display: inline-block;
  height: 138px;
  width: 75px;
  text-align: center;
}
.no-borderradius #nav a:hover{
  color: black;
  background-position: -75px 0;
}

/* Navigation now using only CSS if borderradius is supported */
.borderradius #nav a{
  padding: 50px 20px 30px 20px;
  margin: 0 10px;
  background: #333;
  height: 100px;
  border-bottom-right-radius: 10em;
  border-bottom-left-radius: 10em;
  box-shadow: 6px 0px 0 #999;
  line-height: 3em;
  color: white;
  font-size: 3em;
  text-decoration: none;
  font-family: 'Fredoka One', Arial;
}
```

```
.borderradius #nav a:hover{
  color: #eee;
  background: black;
}
```

Last but not least, let's recreate the grey underneath the tabs with `box-shadow` using another CSS declaration, as seen the following code snippet:

```
/* Progressively add a box-shadow to the navigation tabs */
.boxshadow #nav a{
  box-shadow: 6px 0px 0 #999;
}
```

And as simple as that we have a default version of the navigation tabs and a progressive version using border radius support. If you want to see the difference between the two versions, simply edit the element in your browser console, prepend `no-` to the `borderradius` and `box-shadow` properties, and that's it!

Speaking of borders, before we convert the circle in the logo to CSS, let's first convert the logo banner to CSS as well. We can achieve this with a little trickery using the `border` property. However we'll need to do a bit of adding of some `div` elements, as seen in the following code snippet, to make this possible:

```
<!--  The HTML markup for CSS version of the logo  -->
<div id="logo">
<div class="logo-inner">
  <div class="circle">M</div>
  </div>
 </div>
```

This varies a bit from the original that was simply a single `div` element with an `id` field of `logo`. Again we have one of two options. We can either add the additional HTML and use only the stylesheet or add the new HTML via JavaScript and then use CSS to target everything.

I'm going to use JavaScript this time because I don't like having extra DOM nodes in my markup if I don't need to. Plus, the logo will look just fine should the feature be false, or even if the JavaScript was disabled (although that would be a real buzz kill for Modernizr). The HTML in use is as follows:

```
$(document).ready(function(){

//Add inner html to the logo  div if border radius is supported.
  if( Modernizr.borderradius ){

    var innerHtml = '<div class="logo-inner">';
```

```
          innerHtml += '<div class="circle">M</div>';
          innerHtml += '</div>';
      $('#logo').html(innerHtml);
    }

  }); //end of document ready
```

And then of course the CSS styles to follow suit:

```
  /* Logo */
  .no-borderradius  #logo{
    position: absolute;
    width: 70px;
    height: 110px;
    background: url(images/logo.png);
    top: -5px;
    left: 10%;

  }

  .borderradius #logo{
    position: absolute;
    left: 10%;
    width: 0;
    height: 70px;
    border-left: 35px solid #999;
    border-right: 35px solid #999;
    border-bottom: 35px solid transparent;

  }

  .borderradius #logo .logo-inner{

    position: absolute;
    width: 0;
    height: 70px;
    border-left: 32px solid black;
    border-right: 32px solid black;
    border-bottom: 32px solid transparent;
    left: -32px;
    top: -5px;

  }
```

```
.borderradius #logo .logo-inner .circle {

  border-radius: 50%;
  border: solid 2px #999;
  width: 50px;
  height: 50px;
  text-align: center;
  line-height: 50px;
  color: white;
  font-size: 30px;
  position: absolute;
  left: -27px;
  top: 10px;
  font-family: 'Fredoka One', Arial;

}
```

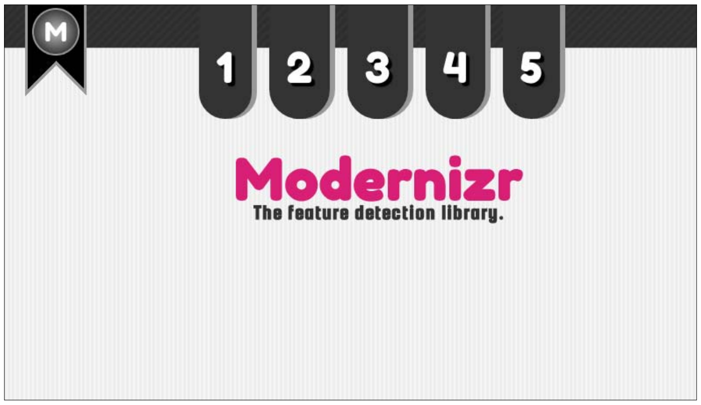
Like earlier there is a `no-` state to handle the feature not being supported and additional styles for the newly added DOM elements, `logo-inner` and `circle`.

Some key things have taken place here. The banner that holds the circular logo has been replicated with CSS using a border trick. You have seen a border consists of four edges and in this instance the `div` element itself isn't actually shown, merely its borders. The bottom border is set to transparent and the two sides are set to display, which is what creates that triangular effect and gives us a banner type look. The `logo` container div is doing the exact same thing, which is what gives it a nice grey edge. Nested inside all of this is the circle, which is achieved with `borderradius`. A circle can be created by setting the width and height of a block-level element to be identical to each other and then applying a `borderradius` property of half or in our case 50 percent.

Only one last progression is remaining and then we can move onto the graphics and other bits that we'll be building on in each additional frame panel. Let's add a little bit of polish to the logo by way of a radial gradient. Until now we've been using variations on linear gradients, so let's see what we can do with the radial version:

```
/* Add a bit of polish to the logo using a radial gradient to mimmic
reflection */
.cssgradients #logo .logo-inner .circle{

 background: -webkit-radial-gradient(circle, rgba(216, 216, 216,
0.84), black);

}
```

Now the logo has a radial gradient, which mimics a light reflection and gives it a more three-dimensional quality. Now, not only is the entire code base able to deliver the identical experience using both images and CSS, but additionally we've done some subtle enhancements by adding a gradient to the logo, as seen in the following screenshot:



Now that the core has a core and progressive version we can move on to the content within these frame panels and do even more really cool and fun things.

Clouds

The first thing I'd like to do is add some clouds behind the Modernizr headline. The HTML is as follows:

```
<hgroup>
   <h1 class"title">Modernizr</h1>
   <h2 class="subtitle">The feature detection library.</h2>
</hgroup>
  <div class="mini-cloud mini-cloud-l"></div>
  <div class="mini-cloud mini-cloud-r"></div>
```

The clouds are the same graphics we had seen earlier and as you probably must have noticed, looking at the selectors in the following code snippet there is a left and right cloud. Let's look at the CSS for them:

```
/* Mini clouds */
.mini-cloud{
  background: url(images/cloud.png) no-repeat 0 0 transparent;
  width: 162px;
  height: 104px;
  position: absolute;
}

/* Left position of mini cloud */
.mini-cloud-l{
  left: 34%;
  top: 250px;
}

/* Right position of mini cloud */
.mini-cloud-r{
  left: 55%;
  top: 278px;
}
```

I'm also going to bump the headers down a bit from the top of the page to get the alignment of all the visual elements into a better state of balance. The following code snippet will move everything into the vertical center of the frame:

```
/* Same as before but altering the position from top */
h1{
  font-family: 'Fredoka One', Arial;
  padding-top: 10%;
  font-size: 4em;
  color: #D91E76;
  top: 154px;
}

h2{
  font-family: 'Squada One', Arial,cursive;
  color: #333;
  top: 165px;
}
```

Finally let's really add the Modernizr logo above the clouds; it will sit just above the `h1` tag and rest on it just like a cake topper would on a delicious three-tiered cake.

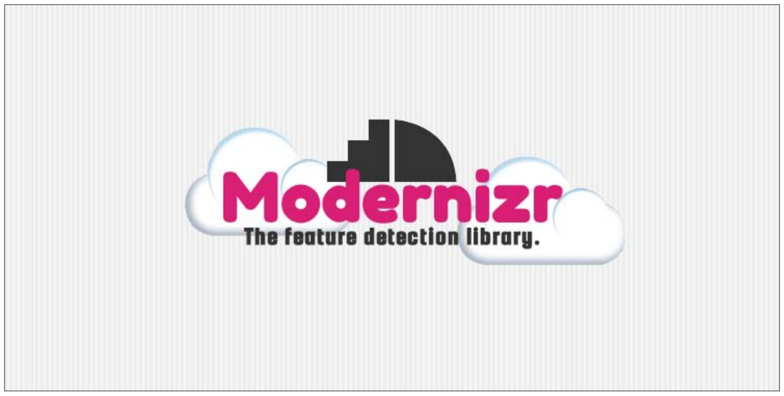
Here is the HTML. This time I'm using an inline `img` element because I'd like to cover as many bases as possible in these exercises. More commonly one might use a CSS background image, even merged with other images as a sprite to reduce HTTP requests. We'll do something a bit like that later with feature detection, but right now an inline image will do just nicely, as seen in the following code snippet:

```
<!--  The html for the logo -->
<div id="modernizr-logo">
 <img src="images/modernizr-logo.png" >
</div>
```

The CSS styles for the logo, more or less positioning it within the frame panel using the following code snippet:

```
/* Modernizr logo, more or less positioning it within the frame */
#modernizr-logo{
  position: relative;
  width: 125px;
  margin: 0 auto;
  height: 60px;
  top: 15px;
  z-index: 3;
}
```

If everything went according to plan, the logo should now look something like the following screenshot:



That's a pretty good stopping point. So far we've replaced all of the navigation and background with a pure CSS3 version using WebKit, just CSS selectors, and JavaScript partnered with CSS. After that we swapped the logo with a purely CSS version using JavaScript and spiced up the logo with a radial gradient. After this was achieved we started on the actual content of the page by adding in some clouds and the official Modernizr logo.

In the subsequent sections, frames we will be building on this current look and feel by adding in even more style elements that leverage the CSS classes, and once that's done we will start playing around with the new features we have by changing the look and experimenting with the gradients a bit more. The last two frames in particular will be focused more on having some fun with the newer features once they have been detected, although they will also bring some new things to the table.

Frame 2 – multiple backgrounds, text shadow, and RGBA color

In this section we'll be doing the following:

- Creating multiple backgrounds to change the color
- Adding a text shadow to the page for headings
- Using RGBA color for alpha

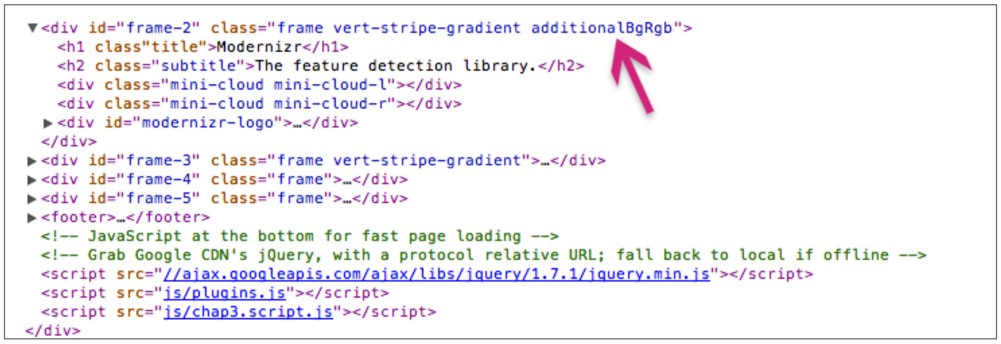
Now we're going to leave the first frame alone and head into the open waters over to frame two. In frame two, we will be detecting and then using multiple backgrounds. The original background is ok, but it would be nice to saturate it with some color. Adding color is actually quite simple with multiple background support. We'll also be taking advantage of support for RGBA, which is identical to traditional RGB color with one very key addition—alpha. Alpha for anybody familiar with things such as Photoshop is more or less the opacity of the color.

Multiple background support is as you would imagine noted in the namespace and also the DOM as `multiplebgs`. Therefore we can target the background using just the CSS as shown in the following example.

In this example, we return again to the CSS and JavaScript hybrid method. The CSS will have its own selector class and JavaScript will be in control of whether or not to place it on the `frame-2` element, as shown in the following code snippet:

```
/* Adds an additional linear gradient background. */
.additionalBgRgb{

background: -webkit-linear-gradient(0deg,rgb(7, 165, 179) 50%,
rgb(138, 214, 230) 50%), rgb(40, 194, 209);

}
//Add the additionalBgRgb class to the Frame-2 element.
$(document).ready(function(){

//Add an additional background to the frame, if supported.
if( Modernizr.cssgradients && Modernizr.multiplebgs ){
$( '#frame-2' ).addClass('additionalBgRgb');
}
}); //End of document ready.
```

If you save this all and refresh the browser, you'll now notice that the class has been added to the `frame-2` element and the background has now been changed to a very nice blue-green color, as seen in the following screenshot:



```
▼<div id="frame-2" class="frame vert-stripe-gradient additionalBgRgb">
   <h1 class"title">Modernizr</h1>
   <h2 class="subtitle">The feature detection library.</h2>
   <div class="mini-cloud mini-cloud-l"></div>
   <div class="mini-cloud mini-cloud-r"></div>
 ▶<div id="modernizr-logo">…</div>
 </div>
▶<div id="frame-3" class="frame vert-stripe-gradient">…</div>
▶<div id="frame-4" class="frame">…</div>
▶<div id="frame-5" class="frame">…</div>
▶<footer>…</footer>
 <!-- JavaScript at the bottom for fast page loading -->
 <!-- Grab Google CDN's jQuery, with a protocol relative URL; fall back to local if offline -->
 <script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
 <script src="js/plugins.js"></script>
 <script src="js/chap3.script.js"></script>
</div>
```

RGBA

Let's take this a little bit further and use RGBA and capitalize on the alpha property to tone this color down a touch. We'll do this the very same way but with RGBA color this time instead of RGB color.

In the JavaScript in the `script.js` file add the following after the previous condition:

```
//Add an additional background to the frame, if supported.
if( Modernizr.cssgradients && Modernizr.multiplebgs ){

 $( '#frame-2' ).addClass('additionalBgRgb');

}

//Simlar to RGB, using a multiple background this time using RGBA.
if( Modernizr.cssgradients
&& Modernizr.multiplebgs
&& Modernizr.rgba ){

$( '#frame-2')
.removeClass('additionalBgRgb')
.addClass('additionalBgRgba');

}
```

The CSS with RGBA will be similar, with a slight difference being the opacity, which will tone down the saturation of the color a touch. The code used to make this change is as follows:

```
.additionalBgRgba{

background: -webkit-linear-gradient(0deg,rgba(7, 165, 179, 0.6) 50%,
rgba(138, 214, 230, 0.82) 50%), rgba(40, 194, 209, 0.5);
}
```

The way we have the code set up now is that we check for multiple background support, and if found, season the `frame-2` element with a new class of `addtionalBgRgb`. Then a second check is done for multiple background and RGBA support and if found, the `addtionalBgRgb` class is removed and replaced by the `rgba` class.

This works fine, and there may be no noticeable difference in the page speed but it could be better. Let's reverse the conditionals and set the `rgb` class to be a fallback of the `rgba` class. As RGBA color is ultimately what we're after it makes sense to test for that first and if it's not present to fall back on, or settle for the `rgb` class version. The code snippet used is as follows:

```
//Simlar to RGB, using a multiple background this time using RGBA.
if( Modernizr.cssgradients
&& Modernizr.multiplebgs
&& Modernizr.rgba ){

$( '#frame-2' ).addClass('additionalBgRgba');

}

//Fall back on the rgb class.
else if( Modernizr.cssgradients && Modernizr.multiplebgs ){

$( '#frame-2' ).addClass('additionalBgRgb');
}
```

Text shadow

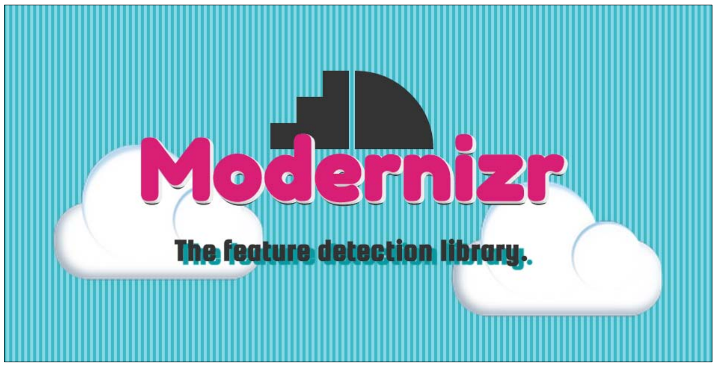
Having a custom font is great. In fact I think the world has seen about enough poor man's Helvetica for one lifetime, but that doesn't mean we have to stop there. In fact with `textshadow` support enabled we can do some really cool things with the text that we have. This time we're jumping back into the CSS; we won't need JavaScript as the fallback for this would be what we already currently see on the page.

I know that I'll be using these elements in all the frames from this point forward. So, I'll add them all now as shown in the following code snippet:

```
/* Text Shadow for current and future h1 elements */
.textshadow #frame-2 h1,
.textshadow #frame-3 h1,
.textshadow #frame-4 h1,
.textshadow #frame-5 h1{
  text-shadow: 1px 2px 6px #333;
```

```
    text-shadow: 1px 0px #eee, 0px 1px #ccc,
                 2px 1px #eee, 1px 2px #ccc,
                 3px 2px #eee, 2px 3px #333;
  }

  /* Ad text shadow to current and future h2 elements */
  .textshadow #frame-2 h2,
  .textshadow #frame-3 h2,
  .textshadow #frame-4 h2,
  .textshadow #frame-5 h2{
    text-shadow: 4px 3px 1px rgb(9, 154, 160);
  }
```

This should be fairly self-explanatory as it's been covered by other features, but let's do a quick review. All browsers that pass the `textshadow` feature test will be given a text shadow. In the case of the `h1` element's multiple shadows have been added to give it a nice raised effect.



We could do a `no-textshadow` instance as well, but the default experience doesn't warrant it as we're happy enough with how it looks should `text-shadow` be ignored by the browser not supporting it. This is a good practice in my opinion because it keeps code low, when you design, as much as possible into the base experience that you are happy with. So, minimal exceptions have to be used.

That's a pretty good place to end for this frame. We've added a nice text-shadow effect, introduced a second background so that frame 2 is not only using the original stripes but a colored RGB background as well to give it a nice blue-green color, and we reduced the saturation of the second background with the introduction of the alpha property with RGBA color.

Frame 3 – box reflect, HSLA color, and generated content

In this section, we'll be doing the following:

- Switching color to HSLA
- Adding CSS reflections
- Recreating clouds using CSS-generated content `:after` and `:before` classes
- Using CSS Transform to modify the scale of the clouds
- Swapping the miniature clouds for CSS clouds

In this section we'll introduce a new color type, HSL, which stands for Hue, Saturation, and Lightness. This is yet another way of representing color. In software programs such as Adobe Photoshop there have been multiple ways to express color for some time now.

In Photoshop, for example, documents are typically converted to CMYK (Cyan, Magenta, Yellow, and Black) for use in print, RGB for Web, and even other formats such as LAB, which isn't device dependent and was designed to approximate human vision. HSL is just another great way of expressing color in the browser as 3D. You can learn more about HSL and these other color models at `learn.colorotate.org/color-models.html`.

Even if you're perfectly happy with plain old RGB, it's good to at least be aware of these other color models as they can really open up a whole new set of color possibilities once you start playing around with them.

We'll also be adding a reflection using the `box reflect` property to the `h1` tag, and then fading it out using some gradient background trickery so that it mimics something you'd do with a layer mask gradient in an image editing program.

Once the reflection has been added and adequately "masked" to simulate a fading effect, we'll be converting the clouds to completely CSS3-based ones using the `generatedcontent` feature test. I'm going to chain selectors as a proof of concept for this example.

HSL color

The HSL color will also take full advantage of multiple backgrounds. The end result will be a background similar to the previous RGBA, but with a little bit of a different take in order to distinguish the two. The great thing about current browsers such as Google Chrome is that the developer tools have a color palette and also color conversion feature that makes experimenting with color easier than it has ever been.

Let's begin by adding the following chained selectors to the `style.css` file, which will lighten the background for `frame-3`:

```
/* Multiple Background Using hsla color */
.multiplebgs.hsla #frame-3{

background: -webkit-linear-gradient(0deg,hsla(0, 100%, 50%, 0) 50%,
hsla(190, 100%, 16%, 0.14) 50%),
hsla(185, 65%, 55%, 0.42);

}
```

Also, change the color of the text shadow for only the `h2` element of `frame-3`. This is where the chaining really takes a step up:

```
/* Example using 3 chained selectors */
.textshadow.multiplebgs.hsla #frame-3 h2.subtitle{
  text-shadow: 4px 4px 1px #fff;
}
```

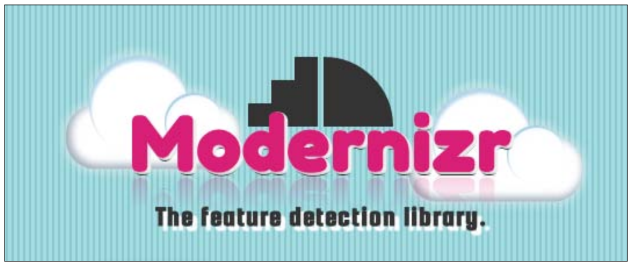
What has happened now is the background has lightened a bit and the color of the text-shadow has been changed to accommodate this as well.

Box shadow

With box shadow at our disposal we can flip an element above, left, right, or below itself. Let's flip the `h1` tag in frame 3 (and later frames) on its back and mask it with a mask that fades the element out by using a linear gradient mask. This will give it a true reflection, look, and feel. This can be done using the following code snippet:

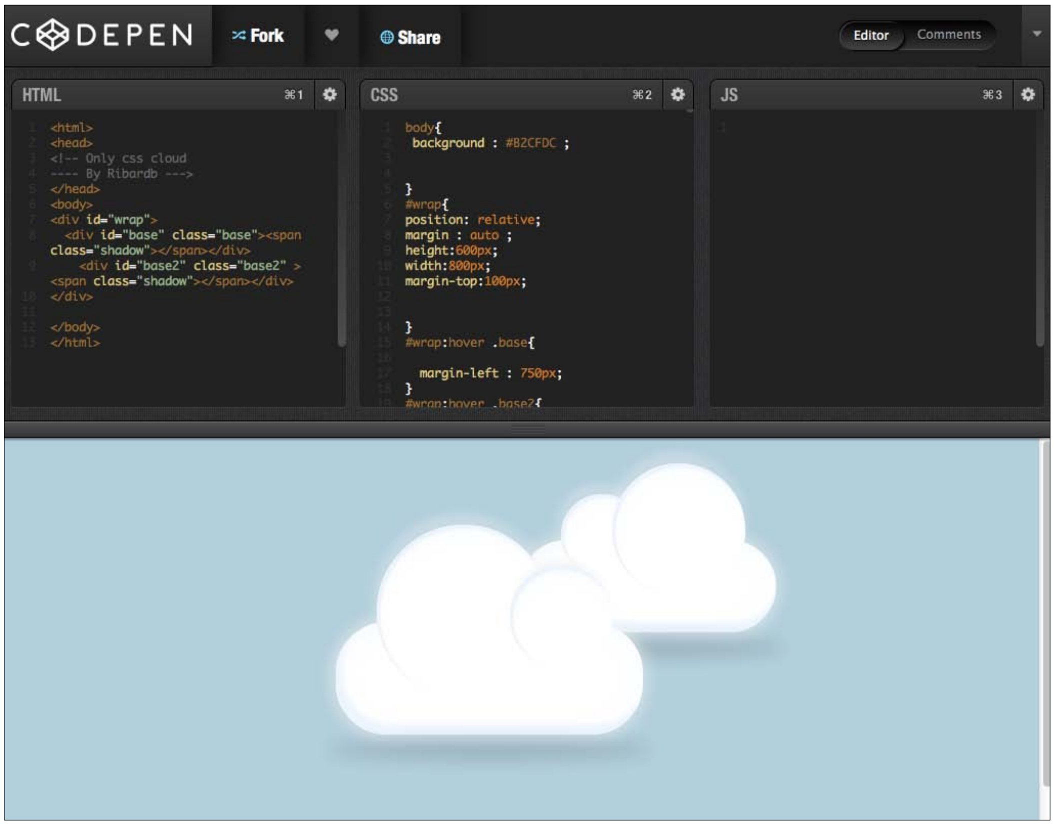
```
/* CSS Reflections */
.cssreflections.cssgradients.rgba #frame-3 h1,
.cssreflections.cssgradients.rgba #frame-4 h1,
.cssreflections.cssgradients.rgba #frame-5 h1{

  -webkit-box-reflect: below -0.35em
-webkit-gradient(linear, left top, left bottom, from(transparent),
color-stop(80%, transparent),
to(rgba(255,255,255,0.3)));

}
```

The effect of the `box shadow` property can be seen in the following screenshot:



Converting the clouds to CSS

The CSS clouds we are currently using actually began their life as purely CSS-based. I wish I could take the credit for these but they were actually created on one of my favorite websites at `codepen.io/ribardb/pen/cykFj`. The following screenshot gives you a peek at this website:



```
<html>
<head>
<!-- Only css cloud
---- By Ribardb --->
</head>
<body>
<div id="wrap">
  <div id="base" class="base"><span
class="shadow"></span></div>
    <div id="base2" class="base2" >
<span class="shadow"></span></div>
</div>

</body>
</html>
```

```
body{
 background : #B2CFDC ;


}
#wrap{
position: relative;
margin : auto ;
height:600px;
width:800px;
margin-top:100px;


}
#wrap:hover .base{

  margin-left : 750px;
}
#wrap:hover .base2{
```

We will be staying pretty true to the original way these clouds were made and only altering the color slightly using the following code snippet:

```
/* CSS Clouds class */

.cssCloud{

  background: white;
  height: 110px;
  width: 300px;
  -webkit-border-radius: 50px;
  margin: 0 auto;
  position: relative;
  z-index: 2;
  top: 3px;

    box-shadow:  inset 5px -4px 18px 4px rgba(75, 98, 120, 0.5),
            inset 7px 8px 20px 0px rgba(235, 245, 255, 0.2),
            inset 0px 0px 50px 6px rgba(235, 245, 255, 0.2),
            0px 0px 20px 6px rgba(240, 240, 240, 0.5);
}
```

This will give us the first part of the cloud, which is the base. After this we add the other two parts using CSS-generated content. Both the `:before` and `:after` classes will help to complete the clouds, as shown in the following code snippet:

```
/* Set the positioning to absolute */
.cssCloud:before, .cssCloud:after{

  content: '';
  position:absolute;

}

/* The second cloud piece */
.cssCloud:before{

  background: white;
  height: 170px;
  width : 170px;

  -webkit-border-radius: 100px;
  top:-80px;
  left:40px;
  box-shadow: inset 3px 9px 6px rgba(111, 181, 217, 0.5),
            inset 4px 15px 15px rgba(235, 245, 255, 0.2),
            inset 3px 2px 2px rgba(230, 230, 230, 0.5),
            -10px -10px 15px rgba(255, 255, 255, 0.1),
            0px -10px 15px rgba(255, 255, 255, 0.3),
            0 -10px 10px -5px rgba(0, 0, 0, 0.05);
}

/* The third and final piece. */
.cssCloud:after{

  background: white;
  height: 100px;
  width : 100px;

  -webkit-border-radius: 50px;
  top:-21px;
  left:170px;

  box-shadow: inset 2px 4px 2px rgba(111, 181, 217, 0.6),
          inset 4px 15px 15px rgba(235, 245, 255, 0.2),
          inset 3px 2px 2px rgba(230, 230, 230, 0.5),
          -10px -10px 15px rgba(255, 255, 255, 0.1),
          0 -10px 15px rgba(255, 255, 255, 0.3),
          0 -10px 10px -5px rgba(0, 0, 0, 0.05);
}
```

This is the core CSS3 cloud. However, this cloud is much larger than we need because we'll be using it again in the next section. We want it to be much smaller so that it matches the size of the image-based clouds. To do this we can make another CSS class and use that to scale the entire thing down. As this requires the additional feature CSS transforms, let's turn once again to the JavaScript in the `script.js` file to add this class to scale the cloud:

```
$(document).ready(function(){

//Frame 3 checks for CSS Gradients, Generated Content and also CSS
Transform.
if( Modernizr.cssgradients
 && Modernizr.csstransforms
 && Modernizr.generatedcontent
){

// Loop through frame panel 3 and remove the image css classes
//and then season with the CSS3 versions.
$( '#frame-3' ).find( '.mini-cloud' ).each( function(){
```

```
$(this).removeClass( 'mini-cloud' )
   .addClass( 'cssCloud smallCloud' );
          } );

  }
});
```

What's taking place in the JavaScript is that all of the necessary features are tested for as a feature set, and if all of the ducks are in a row, the original CSS classes that used images for the clouds are removed and the CSS3 ones are added in their place.

One last piece remains and that is for the clouds to be scaled down to match the original image-based versions. In the CSS file we'll be adding some classes that control scale using the following code snippet:

```
/* Classes to scale and reposition
the CSS version of the mini clouds
*/
/* Set positioning to absolute, but relative to parent and then scale
down by 50% */
.smallCloud{
  position: absolute;
  -webkit-transform: scale(0.5);
}

/* Top position the :before part */
.smallCloud:before{

  top: -85px;
}

/* Top position the :after part */
.smallCloud:after{

  top: -21px;
}

/* Re-adjust the position of the left cloud */
.mini-cloud-l.smallCloud{

  left: 25%;
}

/* Re-adjust the position of the right cloud */
```

```
.mini-cloud-r.smallCloud{

  left: 50%;
}
```

That should about do it. The clouds should now match the size and positioning of their image-based predecessors. So much so that it may be hard to tell which one is being used. So, let's add one last little bit of icing to this cake.

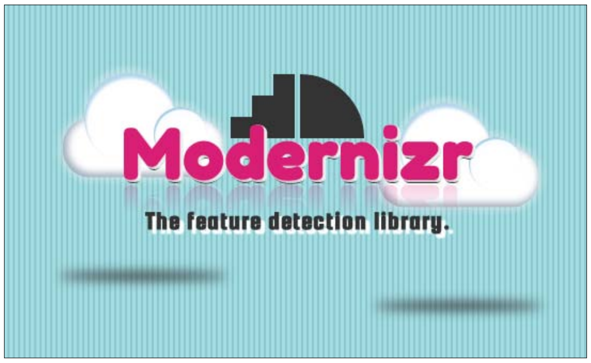
Just after each loop in the JavaScript, add one more line:

```
$(this).removeClass( 'mini-cloud' )
       .addClass( 'cssCloud smallCloud' );
//This new line will add a span element inside the cloud div.
$(this).html('<span class="shadow"></span>');
```

And in the `style.css` file, add the CSS instructions for the shadow as shown in the following code snippet:

```
/* Shadow */
.cssCloud .shadow {
  width: 300px;
  position: absolute;
  background: black;
  bottom: -185px;
  z-index: -2;
  box-shadow: 0 0 30px 8px rgba(50, 50, 50, 0.9);
  -webkit-border-radius: 50%;
}
```

Now the CSS versions will have tiny shadows underneath them, whereas the image versions will not. This should make viewing which version is in play for proof-of-concept feature detection much easier for this frame. The resulting logo is seen in the following screenshot:



Extra credit – converting the Modernizr logo to CSS

Frame 3 is now entirely CSS-based except for one last piece, the official Modernizr logo. I think we might as well go all the way here and convert that as well. The only feature that is really required is `borderradius`, which by this frame we assume is a feature that already exists since we covered it in the original `frame-1` panel.

Nonetheless I want to show one more way of using CSS only to toggle. This time I'm going to leave both of these in the DOM and use CSS to control what is to be displayed.

```
/* The CSS for the official Modernizr logo */

/* Hide by default */
#modernizr-logo .logo-wrap{
  display: none;
}

#modernizr-logo .shape-wrap{
  position: absolute;
  bottom: 0;
}

#modernizr-logo .curve-contain{

  width: 60px;
  height: 60px;
  overflow: hidden;
  position: absolute;
  right: 0;
}

#modernizr-logo .curve{

  width: 120px;
  height: 120px;
  background: #333;
  display: block;
  border-radius: 50%;
  left: -60px;
  position: absolute;
}
```

```
#modernizr-logo .block{

  display: block;
  width: 20px;
  height: 20px;
  background: #333;
  position: absolute;
  bottom: 0;
}

#modernizr-logo .block-1{

  left: 20px;
  height: 40px;
}

#modernizr-logo .block-2{

  left: 40px;
  height: 60px;
}

/* Hide image version for all future frames. */
.borderradius #frame-3 #modernizr-logo img,
.borderradius #frame-4 #modernizr-logo img,
.borderradius #frame-5 #modernizr-logo img{
  display:none;
}

/* Display CSS version for all subsequent frames. */
.borderradius #frame-3 #modernizr-logo .logo-wrap,
.borderradius #frame-4 #modernizr-logo .logo-wrap,
.borderradius #frame-5 #modernizr-logo .logo-wrap{
  display: block;
}
```

And of course the final HTML for the frame is as follows:

```
<div id="frame-3" class="frame vert-stripe-gradient">
  <h1 class="title">Modernizr</h1>
  <h2 class="subtitle">The feature detection library.</h2>
  <div class="mini-cloud mini-cloud-l"></div>
  <div class="mini-cloud mini-cloud-r"></div>
  <div id="modernizr-logo">
```

```
      <img src="images/modernizr-logo.png">

      <div class="logo-wrap">
       <div class="shape-wrap">
        <span class="block block-1"></span>
        <span class="block block-2"></span>
        <span class="block block-3"></span>
       </div>
       <div class="curve-contain">
        <span class="curve"></span>
       </div>
       </div>
       </div>
    </div>
```

What the CSS is doing in the preceding code is hiding the image element inside the `modernizr-logo` container and instead displaying the pure CSS version, if the `borderradius` feature is enabled. This is just one more way of toggling a feature on or off by way of feature detection.

That about sums up everything we needed to do for frame 3. At this point, everything has been converted to be 100 percent CSS3 by way of feature detection. We also covered as many bases as possible with the methods available to us to make this all possible. We used generated content in tandem with gradients and border radius to make the clouds. We also used box reflect to create a nice reflection for the `h1` tag. Last but not the least we used the HSL color model to alter the background of our frame.

Frame 4 – animations

In this section, we will be doing the following:

- Adding a large cloud to the frame
- Using the old striped background as a visual object instead
- Animating the Modernizr logo with rotation
- Creating a new background using CSS gradients
- Animating the miniature clouds

In this frame, we'll be adding some finishing touches and playing around a bit with what we already have. The first thing I'm going to do is add another cloud to the frame that will act as a centerpiece. It's going to reside just behind the title but in front of the smaller clouds.

I am also going to remove the previous shadows and instead have a single shadow beneath the main cloud.

The HTML for frame 4 will be as follows:

```
<div id="frame-4" class="frame">
  <h1 class="title">Modernizr</h1>
  <h2 class="subtitle">Making it rain progressive experiences.</h2>

  <div class="mini-cloud mini-cloud-l">  <span class="shadow"></span>
  </div>

  <div class="mini-cloud mini-cloud-r">
  <span class="shadow"></span>
  </div>

  <div id="modernizr-logo">
   <img src="images/modernizr-logo.png">

   <div class="shape-wrap">
   <span class="block block-1"></span>
   <span class="block block-2"></span>
   <span class="block block-3"></span>
   </div>
  <div class="curve-contain">
  <span class="curve"></span>
  </div>
  </div>

  <div class="rain-drops">
   <div class="rainbg"></div>
  </div>

 <div class="cssCloud" >
 <span class="shadow"></span>
 </div>
</div>
```

The large cloud has been added at the very end of the frame as follows:

```
<div class="cssCloud" >
 <span class="shadow"></span>
</div>
```

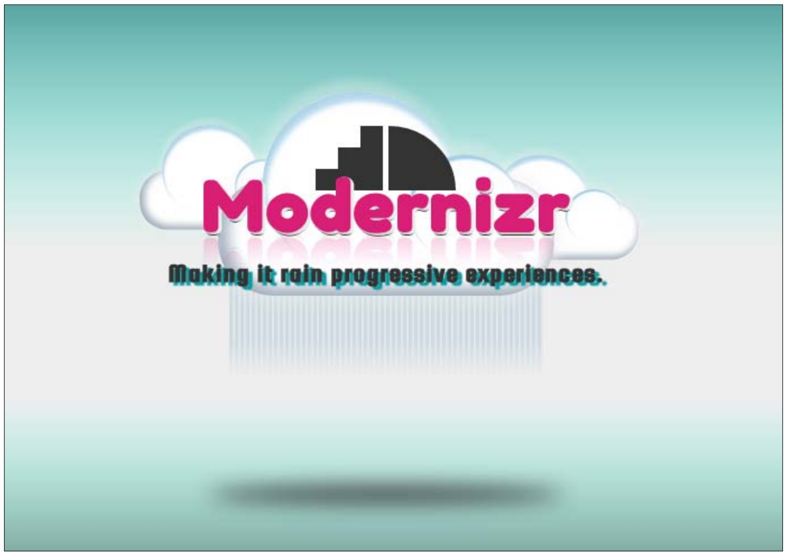
We don't actually have to do anything new here because we styled all of this in the previous frame 3 and scaled it down. Here in this frame we show it at the original scale, complete with shadow.

Also, you'll notice the addition of a `rain-drops` div with a `rainbg` div nested inside. We're going to be using this in not just this frame but the final frame as well. For this frame we will be taking the striped background used in the earlier frames and use it instead as a visual effect by also combining it with box shadow as well.

In place of the previous background we'll still be using gradients, but minus the striped bars. The CSS for both is as follows:

```
#frame-4,
#frame-5{
  background: -webkit-gradient(linear, left top, left bottom,
from(rgba(0, 51, 71, 1)), color-stop(50%, rgba(23, 255, 218, 0)),
color-stop(70%, rgba(23, 255, 218, 0)),color-stop(66%, rgba(23, 255,
255, 0)),to(rgba(51, 51, 51,0.8) ));

}

#frame-4 .rainbg,
#frame-5 .rainbg{

  height: 65px;
  width: 100%;
  position: absolute;

  background-image: -webkit-linear-gradient(0deg,rgba(135, 175, 195,
0.3) 50%, rgba(215, 225, 235, 0.6) 50%);
  background-size: 5px;

  -webkit-box-reflect: below 0 -webkit-gradient(linear, left top,
left bottom, from(transparent), color-stop(50%, transparent),
to(rgba(255,255,255,1)));
}
```

If we take a look in the browser we now have a vibrant gradient background in place of the previous striped one. Additionally, the striped background is now gone and its place is a visual element representing rain. We can see these changes in the following screenshot:



As you have probably noticed, in each frame we show a method for detecting features to be used in the frame at that particular time, but in subsequent frames we'll often statically add those features in with the assumption that we had already detected them. The `cssCloud` class in frame 4, for example. We're doing this mostly to save your eyeballs from having to stare at an overly redundant sequence of code, and to instead distill each individual idea as the best possible one. I think it's most important in these exercises to illustrate the concepts and possibilities, as opposed to following the letter of the law in every frame as we are covering a great deal of information.

In these final two frames, I am going to be bypassing illustrating feature detection and getting right down to brass tacks. I am doing this because frames 4 and 5 are representations of the more advanced, latest, and greatest browser features. Also, we have covered all of the bases pretty well as to how one would go about checking first for these features, and again it would be a bit redundant as this frame thus far uses features we have already tested for in previous exercises. There is however one final feature not yet covered, animation, which we will be exploring now.

Animating the clouds

Let's add a subtle animation to mini clouds by making them roll back and forth as though they are blowing in the wind. The first step is to define the CSS, beginning with the `keyframe` definitions as shown in the following code snippet:

```
/* The left mini cloud. */
@-webkit-keyframes cloud-l{

  from{ left: 25% }
  to{ left: 20%;}
}


/* The right mini cloud. */
@-webkit-keyframes cloud-r{

  from{ left: 50% }
  to{ left: 55%;}
}
```

The clouds are going to shift gently back and forth 5 percent from their current position. We'll of course have to add the rest of the animation parameters to the previous classes as well, using the following code snippet:

```
/* Animation params for the right cloud */
.cssanimations #frame-4 .mini-cloud-r{

  -webkit-animation-duration: 5s;
  -webkit-animation-iteration-count: 10;
  -webkit-animation-direction: alternate;
  -webkit-animation-timing-function: ease-out;
  -webkit-animation-fill-mode:both;
  -webkit-animation-delay: 5s;
  -webkit-animation-name: cloud-r;
}

/* Animation params for the left cloud */
.cssanimations #frame-4 .mini-cloud-l{

  -webkit-animation-duration: 5s;
  -webkit-animation-iteration-count: 10;
  -webkit-animation-direction: alternate;
  -webkit-animation-timing-function: ease-out;
  -webkit-animation-fill-mode:both;
```

```
      -webkit-animation-delay: 2s;
      -webkit-animation-name: cloud-1;
    }
```

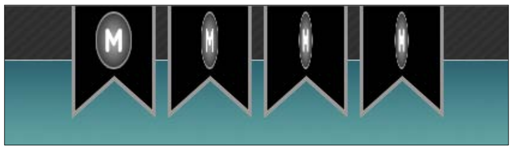
As you can see, in frame 4 if `cssanimations` is available the clouds will rock gently back and forth across the page.

How about we do one last animation example and make the logo spin, much like a coin on its edge?

We can use the following code snippet to do this:

```
    /* The animation for the logo spin. */
    @-webkit-keyframes "logo-spin" {

      from { -webkit-transform: rotateY(0deg); }
        to   { -webkit-transform: rotateY(-360deg); }
    }
    .cssanimations #logo .logo-inner .circle{

      -webkit-animation-name: logo-spin;
      -webkit-animation-duration: 4s;
      -webkit-animation-iteration-count: 10;
      -webkit-animation-direction: alternate;
      -webkit-animation-timing-function: ease-out;
      -webkit-animation-fill-mode: forwards;
      -webkit-animation-delay: 25s;
    }
```

Our screen should look something like the following:



Frame 5 – putting it all together and making it rain

In this section, we'll be doing the following:

- Creating the rain function and making it rain
- Creating the rain drops using border-radius

This last frame is all about fun. What we're going to do is use JavaScript to find each feature that passed the feature test, make that a rain drop, and make it fall out of the cloud. We're literally going to have Modernizr make it rain features.

Caveats

This final frame is built exclusively on WebKit browsers such as Google Chrome and Safari. This is because we're building things not available on many of today's browsers and with feature testing properly in place we're dealing with features that wouldn't be seen at all. These features would only be shown once the browser had "progressed" far enough to interpret them. However, for the sake of building a forward-thinking experience we'll need to be able to see what we are viewing. So, the typical feature detection guard has been dropped. We will not detect whether a feature is supported with JavaScript or CSS before we use it as we have in the past.

Building for the future means that some but not all browsers will be ready for these features. I recommend at the time of this writing viewing the last two frames in Google Chrome 20 and above.

For the HTML all we need to add is this extra part to the `rain-drops` div. It's another `div` element named `drops-wrapper` that will act as a wrapper for all of the raindrops, as seen in the following code snippet:

```
<div class="rain-drops">
   <div class="drops-wrapper"></div>
   <div class="rainbg"></div>
 </div>
```

The raindrops will be CSS and be done using `border-radius`, `transform`, and some `animation` that will make the drop fall, as shown in the following code snippet:

```
.drop {

  position: absolute;
  width: 1em;
  height: 1em;
  -webkit-border-top-left-radius: 10em;
  -webkit-border-top-right-radius: 1em;
  -webkit-border-bottom-right-radius: 10em;
  -webkit-border-bottom-left-radius: 1em;

  border-radius-topleft: 10em;
  border-radius-topright: 0;
  border-radius-bottomright: 10em;
```

```
    border-radius-bottomleft: 10em;
    border-top-left-radius: 10em;
    border-top-right-radius: 0;
    border-bottom-right-radius: 10em;
    border-bottom-left-radius: 10em;

    -webkit-transform: rotate(-45deg);
    -moz-transform: rotate(-45deg);
    -o-transform: rotate(-45deg);
    -ms-transform: rotate(-45deg);
    transform: rotate(-45deg);
    text-align: center;
    font-family: Arial;
    font-size: 1em;
    color: #fff;
    z-index: 1;
    color: #336699;
    background-color: #0567a0;
    -webkit-box-shadow: 0 0px 2px 5px rgba(11,144,213,0.5);
    box-shadow: 0 0px 2px 5px rgba(11,144,213,0.5);
    -webkit-animation-fill-mode: forwards;
    -moz-animation-fill-mode: forwards;
    -ms-animation-fill-mode: forwards;
    -o-animation-fill-mode: forwards;
    animation-fill-mode: forwards;
    -webkit-animation-name: rain;
    -moz-animation-name: rain;
    -ms-animation-name: rain;
    -o-animation-name: rain;
    animation-name: rain;
  }

  .drop p {
    -webkit-transform: rotate(45deg);
    -moz-transform: rotate(45deg);
    -o-transform: rotate(45deg);
    -ms-transform: rotate(45deg);
    transform: rotate(45deg);
  }
```

This time I've taken the liberty of adding all the vendor prefixes as well in order to give a fuller picture in our final example.

The cloud will have three drops fall at any given time. To keep things as simple as possible, there are three separate classes that position and specify the timing of the drops' descent, as shown in the following code snippet:

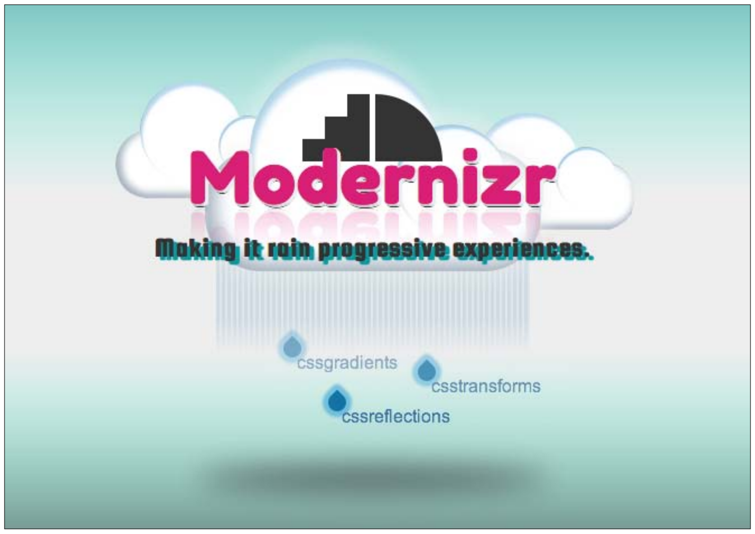
```
.drop-1 {

  left: 60px;
  -webkit-animation-duration: 3s;
}

.drop-2 {

  left: 100px;
  -webkit-animation-duration: 2s;
}

.drop-3 {

  left: 180px;
  -webkit-animation-duration: 2.5s;
}
```

Next we will add the following JavaScript:

```
/*
 * Adds a droplet for each feature that the browser has.
*/
function randomRange (min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

//The array to collect the passed tests.
var rainDrops = [];

/* Loop through the Modernizr global and collect the passed test into
an array */
(function($){

for( var prop in Modernizr ) {

    if( Modernizr.hasOwnProperty( prop ) )

        if( Modernizr[prop] === true ){
         //Push each passed test into an array.
         rainDrops.push(prop);
```

```
            }
        }

      }

    )(jQuery);

    //Loop through the array of passed tests and add raindrops to the
    cloud.
    var start = 0;
    var end = slice = 3;

    function dripFaucet(){

     var dropSet,
       drops = '';

       //Grab 3 from the results.
       dropSet = rainDrops.slice( start , end);

      for ( i=0; i< dropSet.length; i++ ){

       drops += '<div class="drop drop-'+( i + 1 )+' " style="z-index:
    '+ randomRange( 1,3 ) +' top: '+randomRange( 90,100 )+'px ;"><p>'+
    rainDrops[start + i] +'</p></div>';

      }

      //Move the pointer up in the array for the next set.
      start = end;
      end = end + slice;

      $('.rain-drops .drops-wrapper', '#frame-5').html(drops);

    }

    window.setInterval( function(){ dripFaucet(); }, 2000);
```

The JavaScript loops through the stored Modernizr tests, if a test passed, it is stored in an array. This array is then parsed and drops are added three at a time to the cloud. For WebKit browsers, the CSS drops animate by falling, as seen in the following screenshot:



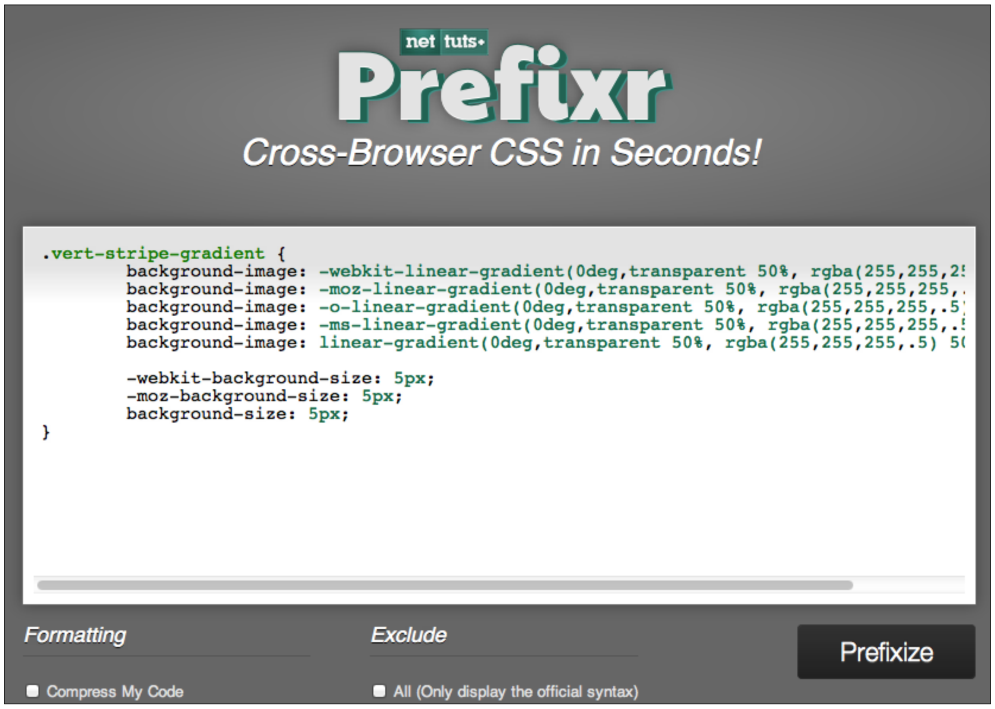
Vendor prefixing

As I mentioned earlier, this tutorial was WebKit-centric. However, the majority of these features are available in most current versions of browsers. The way that the browser in question detects these is by use of a vendor prefix.

The following is an earlier linear gradient example, fully vendor prefixed:

```
.vert-stripe-gradient {
  background-image: -webkit-linear-gradient(0deg,transparent 50%,
rgba(255,255,255,.5) 50%);
  background-image: -moz-linear-gradient(0deg,transparent 50%,
rgba(255,255,255,.5) 50%);
  background-image: -o-linear-gradient(0deg,transparent 50%,
rgba(255,255,255,.5) 50%);
  background-image: -ms-linear-gradient(0deg,transparent 50%,
rgba(255,255,255,.5) 50%);
  background-image: linear-gradient(0deg,transparent 50%,
rgba(255,255,255,.5) 50%);

  -webkit-background-size: 5px;
  -moz-background-size: 5px;
  background-size: 5px;
}
```

As you can see the CSS is iterative for each vendor and is rather cut and dry. I tend to write my code with a single vendor in mind, and then use a prefix generator such as Prefixr, which can be found at `prefixr.com`. Simply paste in your code and full vendor prefixing is done for you on the fly. Vendor prefixing plugins are also widely available for most if not all popular IDEs. The following screenshot shows us a peek how the earlier code is used on the website:



```
.vert-stripe-gradient {
        background-image: -webkit-linear-gradient(0deg,transparent 50%, rgba(255,255,25
        background-image: -moz-linear-gradient(0deg,transparent 50%, rgba(255,255,255,.
        background-image: -o-linear-gradient(0deg,transparent 50%, rgba(255,255,255,.5)
        background-image: -ms-linear-gradient(0deg,transparent 50%, rgba(255,255,255,.5
        background-image: linear-gradient(0deg,transparent 50%, rgba(255,255,255,.5) 50

        -webkit-background-size: 5px;
        -moz-background-size: 5px;
        background-size: 5px;
}
```

Prefixing with Modernizer.prefixed

Modernizr has a prefixing utility under the hood. This is helpful should you want to automatically obtain the prefix while using JavaScript. Simply pass the prefix you are looking for as an argument and the function will return the applicable vendor prefix.

For example, `Modernizr.prefixed('animation-duration')` in Google Chrome would return `WebkitAnimation-duration`.

Summary

In this chapter, we divided feature sets into five separate frames in order to represent five different levels of feature progressions. The last two frames may only work in a WebKit-enabled browser with the most edge features under the hood such as Google Chrome 20 and above, but they should really drive home the idea of building for features that aren't available across the board just yet, but most certainly will be some day. We also showed that there are limits. The last two frames use features that may pass all available tests, but that have implementations that may not work for a specific use case just yet. Also, we had a bit of fun and made it rain features.

We have now managed to illustrate how a singular experience can be enhanced by progression into the upper atmosphere of browser features. I hope that by now you are well aware of the many ways in which the features can be tested for and used in a browser, how fallbacks and progressions can be put into practice, and that your mind is swirling with all sorts of new ideas to put into practice. In the next chapter, we will cover a few more in-depth topics within the library itself, as well as some additional resources.

4
Customizing to Your Unique Needs

In this chapter we'll be covering some of the extras provided by Modernizr that go beyond just feature detection, as well as customizing the library to suit your individual needs. Up to this point we've used the library as a development build. The development build includes all of the feature tests and library extras, which will be covered in a bit more detail as well. As a performance goal however, we want to keep any overhead as light as possible. We've lightened overhead (download size) significantly by reducing HTTP requests, when we converted some images to be purely CSS. Now we can go a step further and lighten the file size of the Modernizr library by building a custom version with only the features we need. This will allow the library to download as fast as possible and in turn enhance overall page performance. Here's what this chapter will be covering:

- Custom building Modernizr for leaner overhead
- `Modernizr.load` (YepNope JS) for conditionally loading
- Loading polyfills and other conditional scripts
- Shim media queries using Respond JS
- Further reading and resources

Customizing Modernizr

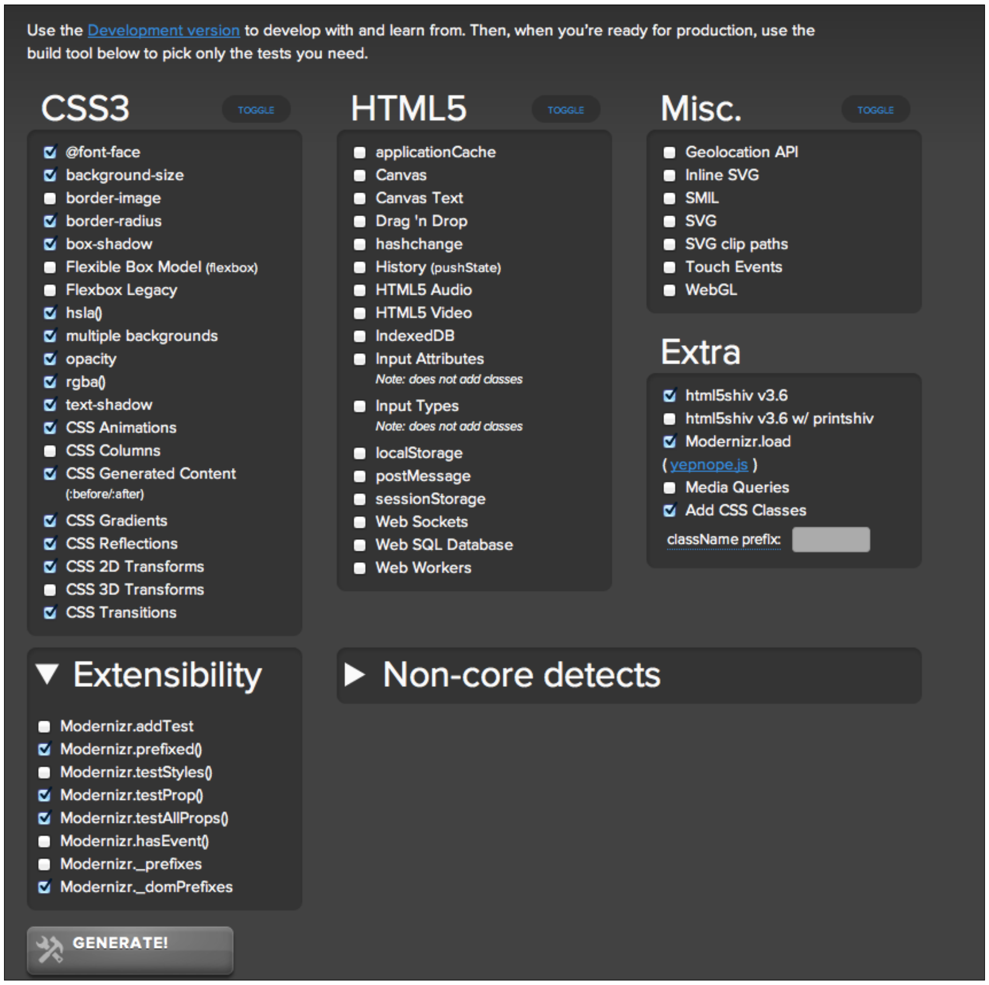
In the examples from the previous chapter, we used the full development build of the library. This however isn't recommended for production sites because the full library would be rarely needed. It is better instead to reduce the overhead file size by custom building this library with only the feature tests that are needed.

This is very easy to do by using the custom build options on the Modernizr website. Simply select **Production** as the download option and choose the boxes for the features your application needs, click on **Generate** and you're off and running. It has even been minified for you. For the previous chapter, a custom build of the library would include the following tests:

- **CSS3**
    - **@font-face**
    - **background-size**
    - **border-radius**
    - **box-shadow**
    - **hsla**
    - **multiple backgrounds**
    - **opacity**
    - **rgba**
    - **text-shadow**
    - **CSS Animations**
    - **CSS Generated Content**
    - **CSS Gradients**
    - **CSS Reflections**
    - **CSS 2D transforms**
    - **CSS Transitions**
- **Extensibility**
    - **Modernizr.prefixed**
    - **Modernizr.testProp**
    - **Modernizr.testAllProps**
    - **Modernizr._domPrefixes**
- **Extra**
    - **html5shiv**
    - **Modernizr.load**
    - **Add CSS Classes**

The extensibility part isn't actually needed if all of your vendor prefixing was done manually or by a vendor prefix generator such as Prefixr, which we touched upon in the previous chapter. It's been left in the build. So, we can cover it in a bit more detail in this chapter as well. The other extensible includes are required by `Modernizr.prefixed` and will be auto selected for you upon custom build. However, as we have them it's a good chance to highlight one of the most useful methods, the `addTest` method.

The custom build of Modernizr weighs in at about 10 KB. Compare that with the full development build we used that weighed in at around 47 KB. This means that the development version was over four times the size needed to run the application in the previous chapter. The custom build tool looks as follows on `Modernizr.com`:



Use the Development version to develop with and learn from. Then, when you're ready for production, use the build tool below to pick only the tests you need.

CSS3 TOGGLE

- ✓ @font-face
- ✓ background-size
- ☐ border-image
- ✓ border-radius
- ✓ box-shadow
- ☐ Flexible Box Model (flexbox)
- ☐ Flexbox Legacy
- ✓ hsla()
- ✓ multiple backgrounds
- ✓ opacity
- ✓ rgba()
- ✓ text-shadow
- ✓ CSS Animations
- ☐ CSS Columns
- ✓ CSS Generated Content (:before/:after)
- ✓ CSS Gradients
- ✓ CSS Reflections
- ✓ CSS 2D Transforms
- ☐ CSS 3D Transforms
- ✓ CSS Transitions

HTML5 TOGGLE

- ☐ applicationCache
- ☐ Canvas
- ☐ Canvas Text
- ☐ Drag 'n Drop
- ☐ hashchange
- ☐ History (pushState)
- ☐ HTML5 Audio
- ☐ HTML5 Video
- ☐ IndexedDB
- ☐ Input Attributes *Note: does not add classes*
- ☐ Input Types *Note: does not add classes*
- ☐ localStorage
- ☐ postMessage
- ☐ sessionStorage
- ☐ Web Sockets
- ☐ Web SQL Database
- ☐ Web Workers

Misc. TOGGLE

- ☐ Geolocation API
- ☐ Inline SVG
- ☐ SMIL
- ☐ SVG
- ☐ SVG clip paths
- ☐ Touch Events
- ☐ WebGL

Extra

- ✓ html5shiv v3.6
- ☐ html5shiv v3.6 w/ printshiv
- ✓ Modernizr.load (yepnope.js)
- ☐ Media Queries
- ✓ Add CSS Classes

className prefix:

▼ Extensibility

- ☐ Modernizr.addTest
- ✓ Modernizr.prefixed()
- ☐ Modernizr.testStyles()
- ✓ Modernizr.testProp()
- ✓ Modernizr.testAllProps()
- ☐ Modernizr.hasEvent()
- ☐ Modernizr._prefixes
- ✓ Modernizr._domPrefixes

▶ Non-core detects

GENERATE!

Why stress over kilobytes? Aren't they trivial when it comes to overhead? Now that applications are moving from the Web and expanding into almost every digital device imaginable, this makes a huge difference. By minimizing as much overhead as possible, not only does the site load as fast as it can, but also a room is made for other hungrier resources such as images. The extra 37 KB could be an entire CSS sprite.

The Modernizr.addTest plugin API

While we are on the topic of extending, Modernizr has a plugin API that you can use to extend the library and add in some of your own. For example, if you wanted to test whether Safari is running in a sort of web app mode called standalone on an iPhone, which is often the case for websites that have been bookmarked by adding to the device's home screen, you could do so by creating the following custom test:

```
//Add a test for Safari Mobile standalone mode.
Modernizr.addTest("standalone", window.navigator.standalone );
```

The second argument can either be passed as a function that returns a `true` or `false` result, or as we've done in the preceding example by passing in a property. This method is handy when not a lot of logic has to be applied and the result is a simple `true` or `false` statement. After running this function, Modernizr will now add either a `standalone` or `no-standalone` class to the page, as well as store this result along with the others.

Modernizr.load

Modernizr has a `load` method, also known as the popular YepNope JS conditional loading library. It is a hugely popular conditional loader for polyfills that is also capable of loading other resources including CSS files. Modernizr capitalizes on the capability of YepNope JS by packaging it under its hood under the method name of `load`.

This new test for `standalone` mode can now be used by `Modernizr.load` to conditionally load additional scripts. Let's say, for example, you wanted to load some special scripts when the browser is running as a web app. The new custom standalone test we created can be used to do this. In this next bit of code, our new test will be performed and a fictional `webapp.js` script will be loaded if the condition is `true`.

```
//Custom test for standalone mode.
Modernizr.addTest('standalone', window.navigator.standalone);
Modernizr.load({
  test: Modernizr.standalone,
  yep: 'webapp.js',
  complete: function () {
    console.log("Standalone was tested for and the conditional script
was loaded if needed");
  }
});
```

The test we created is added and performed using `Modernizr.addTest`. Then `Modernizr.load` runs this test and if it passes, the `yep` script—a special `webapp.js` will be loaded. The last method, named `complete`, is also run after all of this has completed. The complete method will run regardless of the test passing or failing.

`Modernizr.load`, also known as YepNope JS, is known most famously for the conditional loading of polyfills. Let's go over an example of using this helpful method to conditionally load one of our own.

Using polyfills

What exactly is a polyfill? Remy Sharp, who coined the term, defines a polyfill on his blog as follows:

> *A polyfill, or polyfiller, is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively. Flattening the API landscape if you will.*

Paul Irish defines a polyfill as follows:

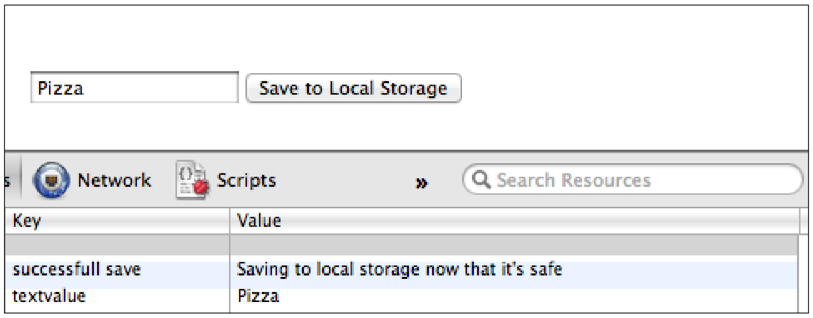
> *A shim that mimics a future API, providing fallback functionality to older browsers.*

The etymology comes from a spackling paste named Polyfilla and best summed up the ideals of what was being accomplished when creating this functionality.

In the following test, we're going to use a built-in test for local storage and load a polyfill if that test result does not pass. In this example, we'll just test against the `window` property and not using Modernizr because that particular test wasn't included in our custom build. However, that is something that can be included in the library.

```
  //Test for local storage and load a polyfill if needed.
  Modernizr.load({
    test: window.localStorage,
    nope: 'storage.js', //Local storage is not supported, load polyfill
    complete: function () {
      console.log("local storage has been polyfilled if required");
    }
  });
```

Using this code, we get something similar to the following screenshot:



Key	Value
successfull save	Saving to local storage now that it's safe
textvalue	Pizza

There are a large number of polyfills available to you on the Internet. A large list of polyfills can be found on the Modernizr Github at `https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills`.

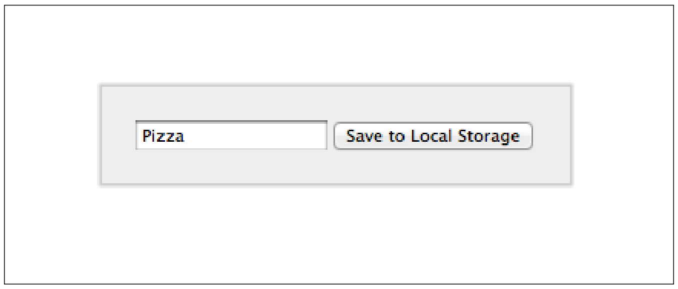
How about we try this out in a web page and put it into practice? In a new HTML page, I'm going to make a quick and simple `div` element that will wrap an input text field and a button. Upon entering a value into the text field and then clicking on the button, that value will be saved in the browser's local storage. On page refresh, the saved value will be checked for and then added to the text field if it is found. I will be using the following HTML for the page:

```
<!doctype html>
<!-- paulirish.com/2008/conditional-stylesheets-vs-css-hacks-answer-
neither/ -->
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7" lang="en">
<![endif]-->
<!--[if IE 7]>    <html class="no-js lt-ie9 lt-ie8" lang="en">
<![endif]-->
<!--[if IE 8]>    <html class="no-js lt-ie9" lang="en"> <![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js" lang="en"> <!--<![endif]-->
<head>
  <meta charset="utf-8">
<!--  Force IE to use the latest version of its rendering engine -->
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title></title>
  <meta name="description" content="A Modernizr test page.">
  <link rel="stylesheet" href="styles.css">
<script src="modernizr.custom.js"></script>
</head>
<body>
<h1>Local Storage Example</h1>
 <div id="localStorage">
 <input type="text" class="textinput"/><button>Save to Local Storage</
button>
 </div>
  <!-- JavaScript at the bottom for fast page loading -->
  <!-- Grab Google CDN's jQuery, with a protocol relative URL; fall
back to local if offline -->
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
jquery.min.js"></script>
  <script src="script.js"></script>
</body>
</html>
```

In the `styles.css` file, we'll add the basic styles for the form using the following code snippet:

```
#localStorage{
  margin: 300px auto 0;
  width: 265px;
  background: #eee;
  padding: 20px;
}
/* Add a drop shadow */
.boxshadow #localStorage {
  -webkit-box-shadow: 0 0 2px 2px #ccc;
  box-shadow: 0 0 2px 2px #ccc;
}
```

Using this code, we get something similar to the following screenshot:



As you will have already noticed it's a straightforward, bare bones text input and button wrapped by a `div` element. A stylesheet, jQuery, and JavaScript file are included similar to the naming conventions in the previous chapters. The `script.js` file is where all of the heavy lifting is going to happen. I am also going to wrap all of the JavaScript in a little bit of shorthand for the document `ready` method to ensure that the DOM will be loaded before running any of the JavaScript. I'm going to first add in the previous examples, the `localStorage` test, the `standalone` test, and the polyfill loading as seen in the following code snippet:

```
$(function(){
  //Test for localstorage and load a polyfill if needed.
  Modernizr.load({
    test: window.localStorage,
    nope: 'storage.js',
    complete: function () {
      localStorage.setItem("successfull save", "Saving to local
storage now that it's safe");
    }
  });
Modernizr.addTest( 'standalone', window.navigator.standalone );
  Modernizr.load({
    test: Modernizr.standalone,
    yep: 'specialjavascriptfunctions.js',
    complete: function () {
      console.log("Standalone was tested for, and the conditional
script was loaded if needed");
    }
  });
});
```

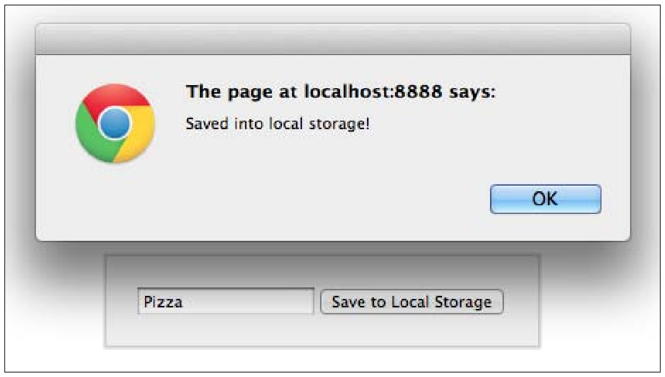
If you view this new HTML page in the browser you'll see the console has logged the `localStorage` and the `standalone` complete messages.

Now, we can add the code that will handle `localStorage` input processing:

```
$(function(){
// ...previous code from Modernizr.load, Modernizr.addTest above
   //Cache the text input, and local storage into variables.
  var textVal = $('.textinput', '#localStorage');
  var storedItem = localStorage.getItem("textvalue");
  //Check and see if anything has been stored
  //from the text input. Set the text input to have it's value.
  if( typeof storedItem === 'string' ){
    textVal.val( storedItem );
  }
```

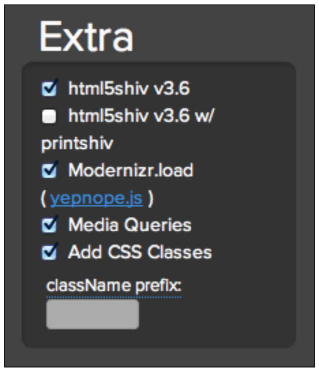
```
    //Bind a function to the click event of
    //our button that saves the value into localstorage.
    //An alert is fired after this is done.
    $('button', '#localStorage').click(function(){
      localStorage.setItem( "textvalue", textVal.val() );
      alert("Saved into local storage!");

    });
  }); //end of code
```

With the code all in place, and the polyfill conditionally loaded, we can test our form. Once the value is saved an alert will appear, as shown in the following screenshot:



Media queries with Modernizr.mq

Support for media queries must be added through the custom build screen. I've gone ahead and rebuilt a new custom download using the previous options, as well as adding media query support found under the **Extra** panel, as shown in the following screenshot:



Extra

- ☑ html5shiv v3.6
- ☐ html5shiv v3.6 w/ printshiv
- ☑ Modernizr.load (yepnope.js)
- ☑ Media Queries
- ☑ Add CSS Classes

className prefix:

With media queries added to the library, a new `mq` method is now available. One caveat that should be noted is that if a browser does not support media queries at all this method will always return `false`. This applies generally to old versions of IE.

With media queries now in our library, testing is as simple as passing an argument. Here is the sample usage from the Modernizr site:

```
Modernizr.mq('only screen and (max-width: 768px)')  // true
```

This will test the media type of "only screen" (as opposed to print, or other) for a maximum width of 768 pixels. Now, as mentioned earlier, older IE browsers with no support for media queries to speak of will always return `false`. To combat these earlier versions the Modernizr library would include a shim for this named Respond JS. However, this is no longer the case and if you want to shim in this type of support, it must be added manually due to some IE 8 crashing issues causing the need for its removal. Hopefully, future versions will resolve this bottleneck. In the next section, I'll show you how to use the `mq` method to test for support and then shim with Respond JS.

Respond

As I previously mentioned, Modernizr used to be bundled with Respond.js, however recently as of version 2.5 it was decoupled from the library itself and must now be downloaded separately and then added manually. This was due to some IE 8 crashing issues.

Because it has been removed, I won't go into a lot of detail about it beyond implementing and lightly reviewing the example from the Respond JS download website and showing how to load it as a polyfill with `Modernizr.load`.

On the Respond JS Github page, this bit of JavaScript describes itself as follows:

> *A fast & lightweight polyfill for min/max-width CSS3 Media Queries (for IE 6-8, and more).*

This means if you wanted media queries regarding this scope to be backward compatible you could leverage this tiny library to accomplish that. One such use could, for example, be if your web page was built to have a responsive layout that adapted to the width and height of the viewable area.

Here is another brief example of using a test to load in a polyfill; in this example, no test is performed and the script is explicitly loaded in for shim support:

```
Modernizr.load({
    load: Modernizr.hasmediaq,
    complete: function(){
       console.log('respond.js has been loaded');
    }
});
```

Of course you may prefer to conditionally load this polyfill; we can create and apply a new custom test using `Modernizr.mq` and instead polyfill with the script only as needed as seen in the following code snippet:

```
Modernizr.addTest( 'hasmediaq', Modernizr.mq('only all') );
  Modernizr.load({
  test: Modernizr.hasmediaq,
  nope: 'respond.js',
  complete: function(){
    console.log('respond.js has been loaded');
  }
});
```

Now that we have a media query functionality successfully polyfilled, let's put it to use. As I mentioned before, I'm going to borrow a page from the example available on the Respond JS test page but with a little bit of a twist.

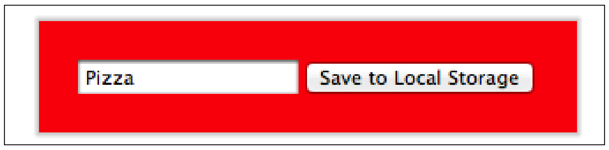
In this next step, I'll be adding the following CSS, pasted from the example on the Respond JS website, and then adapted to our example code we've already been using. In the Respond JS test case, the background color of the page changes color as different media queries are met. The example we'll use will cause the background color of just the `localStorage div` to change color as the size of the page changes. As each new condition is met, the applicable style will be applied to the element.

Add the following CSS to the `styles.css` page:

```
/*
 * Respond.js test page styles applied to our
 * custom example. Resizing the page will
 * change the color of the #localStorage div
 * Based on http://scottjehl.github.com/Respond/test/test.html
*/
/*styles for 300 and up @ 16px!*/
/* The max-width declaration below blocks this from ever working */
@media only screen and (min-width: 18.75em){
  #localStorage {
    background: yellow;
  }
}
```

```
/*styles for 480px - 620px @ 16px!*/
@media only screen and (min-width: 30em) and (max-width: 38.75em) {
  #localStorage {
    background: green;
  }
}

@media screen and (min-width: 38.75em),only
print,projector{#localStorage {background:red;}}

/*styles for 800px and up @ 16px!*/
@media screen and (min-width: 50em){
  #localStorage {
    background: blue;
  }
}

/*styles for 1100px and up @ 16px!*/
@media screen and (min-width: 68.75em){
  #localStorage {
    background: orange;
  }
}

/*styles for 1200px and up @ 16px!*/
@media screen and (min-width: 1200px){
  #localStorage {
    background: navy;
  }
}
```

Now when we resize the page you'll notice the background color changing on the `localStorage` wrapper, as seen in the following screenshot:



Putting it all together here is the final code including tests, HTML, CSS, and JavaScript loading with jQuery.

JavaScript executing on DOM ready via jQuery in `script.js` file uses the following code snippet:

```
$(function(){
  //Test for localstorage and load a polyfill if needed.
  Modernizr.load({
    test: window.localStorage,
    nope: 'storage.js', //load localstorage polyfill
    complete: function () {
      localStorage.setItem("successfull save", "Saving to local
      storage now that it's safe");
    }
  });

  //Create a test for media queries. Then load respondjs polyfill if
needed.
  Modernizr.addTest( 'hasmediaq', Modernizr.mq('only all') );
  Modernizr.load({
    test: Modernizr.hasmediaq,
    nope: 'respond.js',
    complete: function(){
      console.log('respond.js has been loaded');
    }
  });
  Modernizr.addTest( 'standalone', window.navigator.standalone );
  Modernizr.load({
    test: Modernizr.standalone,
    yep: 'specialjavascriptfunctions.js',
    complete: function () {
      console.log("Standalone was tested for and the conditional
      script was loaded if needed");
    }
  });
  //Cache the text input, and local storage into variables.
  var textVal = $('.textinput', '#localStorage');
  var storedItem = localStorage.getItem("textvalue");
  //Check and see if anything has been stored
  //from the text input. Set the text input to have its value.
  if( typeof storedItem === 'string' ){
    textVal.val( storedItem );
  }
  //Bind a function to the click event of
  //our button that saves the value into localstorage.
  //An alert is fired after this is done.
```

```
    $('button', '#localStorage').click(function(){
      localStorage.setItem( "textvalue", textVal.val() );
      alert("Saved into local storage!");
    });
  });
```

CSS styles including the color changing media queries for our `styles.css` file are used in the following code snippet:

```
/* Wrapper for the text input and button. */
#localStorage{
  margin: 300px auto 0;
  width: 265px;
  background: #eee;
  padding: 20px;
}

/* Add a drop shadow */
.boxshadow #localStorage {
  -webkit-box-shadow: 0 0 2px 2px #ccc;
  box-shadow: 0 0 2px 2px #ccc;
}

/*
 * Respond.js test page styles applied to our
 * custom example. Resizing the page will
 * change the color of the #localStorage div
 * Based on http://scottjehl.github.com/Respond/test/test.html
*/
/*styles for 300 and up @ 16px!*/
/* The max-width declaration below blocks this from ever working */
@media only screen and (min-width: 18.75em){
  #localStorage {
    background: yellow;
  }
}

/*styles for 480px - 620px @ 16px!*/
@media only screen and (min-width: 30em) and (max-width: 38.75em) {
  #localStorage {
    background: green;
  }
}
```

```
@media screen and (min-width: 38.75em),only
print,projector{#localStorage {background:red;}}

/*styles for 800px and up @ 16px!*/
@media screen and (min-width: 50em){
  #localStorage {
    background: blue;
  }
}

/*styles for 1100px and up @ 16px!*/
@media screen and (min-width: 68.75em){
  #localStorage {
    background: orange;
  }
}

/*styles for 1200px and up @ 16px!*/
@media screen and (min-width: 1200px){
  #localStorage {
    background: navy;
  }
}
```

And last but not least, the simple HTML page for inputting and saving a text string into `localStorage` is shown in the following code snippet:

```
<!doctype html>
<!-- paulirish.com/2008/conditional-stylesheets-vs-css-hacks-answer-
neither/ -->
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7" lang="en">
<![endif]-->
<!--[if IE 7]>    <html class="no-js lt-ie9 lt-ie8" lang="en">
<![endif]-->
<!--[if IE 8]>    <html class="no-js lt-ie9" lang="en"> <![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js" lang="en"> <!--<![endif]-->
<head>
  <meta charset="utf-8">
<!--  Force IE to use the latest version of its rendering engine -->
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title></title>
  <meta name="description" content="A Modernizr test page.">
  <link rel="stylesheet" href="styles.css">
  <!--  Modernizr will be included in the head of the page. We'll need
it do do some light lifting before the DOM tree renders load feature
detection and shimming  -->
```

```
<script src="modernizr.custom.js"></script>
</head>
<body>
<!-- Creates a very simple form for storing a text string into
localstorage -->
 <div id="localStorage">
 <input type="text" class="textinput"/><button>Save to Local Storage</
button>
 </div>
  <!-- JavaScript at the bottom for fast page loading -->
  <!-- Grab Google CDN's jQuery, with a protocol relative URL; fall
back to local if offline -->
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
jquery.min.js"></script>
  <script src="script.js"></script>
</body>
</html>
```

Just like that we've conditionally shimmed for and applied a series of media queries with CSS and Respond JS. We've polyfilled and saved something into `localStorage` with `Modernizr.load`. Finally, we trimmed the fat on our library by way of a custom build to get the most amount of kick with the smallest overhead.

Further reading and resources

I highly encourage you to explore the Modernizr docs in addition to using the examples in these chapters as seeds to grow ideas from. My hope is that by broadly covering all of the principles and practices in as many ways as possible the gears in your head will begin turning and you'll be filled with fresher ways of implementing previous projects as well as new approaches for what lies ahead.

Now that we've about wrapped up our feature detecting, polyfilling fun, I'd like to cover a few resources that are quite helpful and worth checking out. These are just the tip of the iceberg but should get you well on your way to building more solid apps.

jQuery's Best Friends

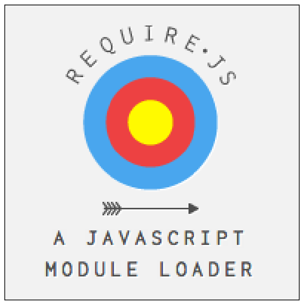
The website `jQuerysBestFriends.com` by Alex Sexton is delivered in slide format and covers just about everything important to building in JavaScript today with a nice side of humor. It is highly recommended to take the time to read through each slide, as well as get a laugh or two as Alex covers an array of practices and principles. You'll walk away more educated and full of great resources in development today.

Require JS

This is one of the best ways, if not the best, to load JavaScript into your pages and applications. Require JS allows a simple and sanity saving way of loading dependencies as needed in a module centric way. Compare this with the more traditional way of dumping every bit of JavaScript and using only a portion of it as needed. This way you can load only what you need in a manageable way.

While YepNope/`Modernizr.load` is useful for a more micro level using tests to conditionally load scripts, this is better suited for the broader picture loading of JavaScript. With an additional plugin, it is also possible to load templates as well.

In addition to the benefit of modularity and loading tools, Require JS comes packaged with a great optimizer for production code builds, including CSS. Find it at `RequireJS.org`.



Backbone JS

Backbone JS is one of my favorite MVC JavaScript frameworks and pairs quite nicely with Require JS and Modernizr. Using Backbone allows you to break up your application into Models, Views, and Collections. Find out more at `BackboneJS.org`.

Underscore

Self described as a "utility-belt library for JavaScript", and the "tie to go along with jQuery's tux", this library provides 60 or so functions, many of which are probably more highly optimized versions of those you are already using. This library even has a built-in template utility that picks up where jQuery templates left off. Find out more at `UnderscoreJS.org`.

HTML5 Rocks

Mobile, gaming, or meat and potatoes web development, this site covers it all with practical exercises, advice, and articles on virtually every topic related to the new HTML5 specification. Find out more at `Html5rocks.com`.

Summary

That's all folks! Our time has come to an end. We've introduced the idea of detecting features as an alternative to UA sniffing. We progressively built up an experience that while only attractive in WebKit browsers today, will be picked up without any extra work on your part—assuming you've vendor prefixed accordingly—once the other manufacturers have adopted those features into their builds. We even took the test results performed by the Modernizr library and literally made it rain all of those wonderful features your browser has, or will at least one day have in the not too distant future. Thanks for reading this book. Happy coding!

Index