Documentation

What is Modernizr?

Modernizr is a small JavaScript library that *detects* the availability of native implementations for next-generation web technologies, i.e. features that stem from the HTML5 and CSS3 specifications. Many of these features are already implemented in at least one major browser (most of them in two or more), and what Modernizr does is, very simply, tell you whether the current browser has this feature natively implemented or not.

Unlike with the traditional—but highly unreliable—method of doing "UA sniffing," which is detecting a browser by its (user-configurable) `navigator.userAgent` property, Modernizr does actual feature detection to reliably discern what the various browsers can and cannot do. After all, the same rendering engine may not necessarily support the same things, and some users change their userAgent string to get around poorly developed websites that don't let them through otherwise.

Modernizr aims to bring an end to the UA sniffing practice. Using feature detection is a *more reliable* mechanic to establish what you can and cannot do in the current browser, and Modernizr makes it convenient for you in a variety of ways:

1. It tests for over 40 next-generation features, all in a matter of milliseconds
2. It creates a JavaScript object (named `Modernizr`) that contains the results of these tests as boolean properties
3. It adds classes to the `html` element that explain precisely what features are and are **not** natively supported
4. It provides a script loader so you can pull in polyfills to backfill functionality in old browsers

With this knowledge that Modernizr gives you, you can take advantage of these new features in the browsers that can render or utilize them, and still have easy *and reliable* means of controlling the situation for the browsers that cannot.

Installing Modernizr

Head over to the Download page, and select which features you intend to use in your project. This way we can provide the slimmest bits of code that you'll need. Hit Generate and you've got your own custom build of Modernizr. If you don't know yet what features you'll be using, get the Development version which contains them all, but is uncompressed.

Drop the script tags in the `<head>` of your HTML. For best performance, you should have them follow after your stylesheet references. The reason we recommend placing Modernizr in the head is two-fold: the HTML5 Shiv (that enables HTML5 elements in IE) must execute before the `<body>`, and if you're using any of the CSS classes that Modernizr adds, you'll want to prevent a FOUC.

If you don't support IE8 and don't need to worry about FOUC, feel free to include modernizr.js whereever.

Polyfills and Modernizr

The name "Modernizr" might throw you for a second, we'll admit. The library does allow you to use the new HTML5 sectioning elements in IE, but aside from that, it doesn't *modernize* any other features. The name Modernizr actually stems from the goal of modernizing our development practices (and ourselves). However! Modernizr still pairs extremely well with scripts that do provide support when native browser support is lacking. In general, these scripts are called polyfills.

polyfill *(n)*: a JavaScript shim that replicates the standard API for older browsers

So a websocket polyfill would create a `window.WebSocket` global with the same properties and methods on it as a native implementation. That means you can develop for the future, with the real API, and only load your compatibility polyfills on browsers that do not support that API or feature.

And good news for you, **there is a polyfill for nearly every HTML5 feature that Modernizr detects**. Yup. So in most cases, you can use a HTML5 or CSS3 feature and be able to replicate it in non-supporting browsers. Yes, not only can you use HTML5 today, but you can use it in the past, too!

But that leads to a big, fat warning: **just because you can use a polyfill doesn't mean you should**. You want to deliver the best user experience possible, which means it should be quick! Loading five compatibility scripts for IE7 so it looks and works the exact same as Chrome and Opera isn't a wise choice. There are no hard and fast rules, but keep in mind how adding more scripts to the page can impact the user experience. And remember, none of your users view your site in more than one browser; It's okay if it looks and acts differently.

Now for more on how to effectively use and serve polyfills for all your different users, read on…

Modernizr.load() tutorial

`Modernizr.load` is a resource loader (CSS and JavaScript) that was made to specifically to work side-by-side with Modernizr. It's optional in your build, but if you are loading polyfills, There's a good chance it can save you some bandwidth and boost performance a bit.

`Modernizr.load` syntax is generally very easy to understand, so we'll start with a basic example:

```
Modernizr.load({
  test: Modernizr.geolocation,
  yep : 'geo.js',
  nope: 'geo-polyfill.js'
});
```

In this example, we decided that we should load a different script depending on whether geolocation is supported in the host browser or not. By doing this, you save users from having to download code that their browser does not need. This increases page performance, and offers a clear place to build a healthy amount of abstraction to your polyfills (both 'geo.js' and 'geo-polyfill.js' would seem the same to the rest of the app, even if they're implemented differently).

There's a good chance that you're not terribly unhappy with your script downloading speeds now, but you'll be happy to know that `Modernizr.load` does not slow anything down, and can sometimes offer a small boost in performance by loading scripts asynchronously and in parallel. There's a lot of variables to weigh in this area, so we suggest that you try a few things to make sure you're getting maximum performance for your specific situation.

`Modernizr.load` is small and simple, but it can do quite a bit of heavy-lifting for you. Here is a slightly more complicated example of using `Modernizr.load` when your scripts rely on more than one Modernizr feature-test. A good technique is to wrap up multiple polyfill scripts into a single 'oldbrowser' type script, that way you don't end up loading too many scripts at once. Here's how that might work:

```
// Give Modernizr.load a string, an object, or an array of strings and
objects
Modernizr.load([
  // Presentational polyfills
  {
      // Logical list of things we would normally need
      test : Modernizr.fontface && Modernizr.canvas && Modernizr.cssgradients,
```

```
      // Modernizr.load loads css and javascript by default
      nope : ['presentational-polyfill.js', 'presentational.css']
  },
  // Functional polyfills
  {
      // This just has to be truthy
      test : Modernizr.websockets && window.JSON,
      // socket-io.js and json2.js
      nope : 'functional-polyfills.js',
      // You can also give arrays of resources to load.
      both : [ 'app.js', 'extra.js' ],
      complete : function () {
      // Run this after everything in this group has downloaded
      // and executed, as well everything in all previous groups
      myApp.init();
      }
  },
  // Run your analytics after you've already kicked off all the rest
  // of your app.
  'post-analytics.js'
]);
```

There's a lot that you can do with `Modernizr.load`. It was released standalone as yepnope.js but it was made specifically with Modernizr and feature-testing in mind. You can check out the full docs for `Modernizr.load` at yepnopejs.com.

One cool feature of `Modernizr.load` is it's complete decoupling of load and execution of scripts. That might not mean much to you, but users of the HTML5 Boilerplate might be familiar with the Google CDN copy of jQuery fallback. It looks something like this:

```
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.js"></
script>
    <script>window.jQuery || document.write('<script src="js/libs/jquery-
1.7.1.min.js">\x3C/script>')</script>
```

It works by trying to load in the script, and then immediately after, testing to see if the jQuery object is available. If It's not, then they load a local copy of jQuery as a fallback. This is generally not that easy in script-loaders, but `Modernizr.load` has got you covered. You can just use the same logic, and `Modernizr.load` will handle the execution order for you:

```
Modernizr.load([
  {
      load: '//ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.js',
      complete: function () {
      if ( !window.jQuery ) {
            Modernizr.load('js/libs/jquery-1.7.1.min.js');
      }
      }
  },
  {
      // This will wait for the fallback to load and
      // execute if it needs to.
      load: 'needs-jQuery.js'
  }
]);
```

A quick note, though: only use this technique as a method for fallbacks, because it can hurt performance by forcing scripts to load in serial instead of in parallel.

`Modernizr.load` is extensible as well. You can load your own prefixes and filters for your scripts. For more advanced information on that, you can check out the docs on the yepnopejs.com homepage. You can even go as far as to replace entire steps of the loading process and replace them with custom logic. So start using it and saving those precious, precious bytes.

How Modernizr works

For a lot of the tests, Modernizr does its "magic" by creating an element, setting a specific style instruction on that element and then immediately retrieving it. Browsers that understand the instruction will return something sensible; browsers that don't understand it will return nothing or "undefined".

Many tests in the documentation come with a small code sample to illustrate how you could use that specific test in your web development workflow. Actual use cases come in many more varieties, though. The possible uses of Modernizr are limited only by your imagination.

If you're really interested in the details of how Modernizr works, look at the source code of modernizr.js and the various feature detects, and dig into the project on github.

HTML 5 elements in IE

Modernizr runs through a little loop in JavaScript to enable the various elements from HTML5 (as well as `abbr`) for styling in Internet Explorer. Note that this does not mean it suddenly makes IE support the Audio or Video element, it just means that you can use `section` instead of `div` and style them in CSS. you'll also probably want to set many of these elements to `display:block`; see the HTML5 Boilerplate CSS for an example. As of Modernizr 1.5, this script is identical to what is used in the popular html5shim/html5shiv library. Both also enable printability of HTML5 elements in IE6-8, though you might want to try out the performance hit if you have over 100kb of css.

Supported browsers

We support IE6+, Firefox 3.5+, Opera 9.6+, Safari 2+, Chrome. On mobile, we support iOS's mobile Safari, Android's WebKit browser, Opera Mobile, Firefox Mobile and whilst we're still doing more testing we believe we support Blackberry 6+.

Features detected by Modernizr

CSS features

Feature	`Modernizr` JS object property / CSS classname
@font-face	fontface
background-size	backgroundsize
border-image	borderimage
border-radius	borderradius
box-shadow	boxshadow
Flexible Box Model	flexbox
The flexible box model (aka flexbox) offers a different way for positioning elements, that addresses some of the shortcomings of float-based layouts.	

/* Simulated box shadow using borders: */
.box {
  border-bottom: 1px solid #666;
  border-right: 1px solid #777;
}
.boxshadow div.box {
  border: none;
  -webkit-box-shadow: #666 1px 1px 1px;
  -moz-box-shadow: #666 1px 1px 1px;
  box-shadow: #666 1px 1px 1px;
}	

hsla()	hsla

In most cases you don't need to use the rgba and hsla classes, as browsers can do the fallback naturally:
.my_container { /* no .hsla use necessary! */
  background-color: #ccc; /* light grey fallback */
  background-color: hsla(0, 0%, 100%, .5); /* white with alpha */
}	

Multiple backgrounds	multiplebgs
opacity	opacity
rgba()	rgba
text-shadow	textshadow

Firefox 3.0 false-positives this test, but there is no known fix to that. All IEs, including IE9, do not support text-shadow and thus fail to deliver delight.
.glowy { /* ghosted letters */
  color: transparent;
  text-shadow: 0 0 10px black;
}
.no-textshadow {
  color: black;
}	

CSS Animations	cssanimations
Android 2.1-3 will pass this test, but can only animate a single property. Android 4+ behaves as expected.	
CSS Columns	csscolumns
Generated Content (:before/:after)	generatedcontent
CSS Gradients	cssgradients
As there is a WebKit bug which causes needless downloads of	

background-image assets, there is a best way of combo'ing a background-image with CSS gradients, so everyone gets the right experience with no extra HTTP overhead:
`.no-js .glossy,`
`.no-cssgradients .glossy {`
`background: url("images/glossybutton.png");`
`}`
 	
`.cssgradients .glossy {`
`background-image: linear-gradient(top, #555, #333);}`	

CSS Reflections	cssreflections
CSS 2D Transforms	csstransforms
CSS 3D Transforms	csstransforms3d
CSS Transitions	csstransitions
Transitions can typically be used without using Modernizr's specific CSS class or JavaScript property, but for those occasions you want parts of your site to look and/or behave differently they are available. A good example use case is to build Modernizr into an animation engine, which uses native CSS Transitions in the browsers that have it, and relies on JavaScript for the animation in browsers that don't. Isotope uses a Modernizr feature detect to use transitions along with transforms to achieve full hardware acceleration when possible, falling back to jQuery-based animation otherwise.	

HTML5 features

Feature	`Modernizr` JS object property / CSS classname
applicationCache	applicationcache
Canvas	canvas
If `Modernizr.canvas` tests false, you may want to load FlashCanvas or excanvas.	
Canvas Text	canvastext
Drag and Drop	draganddrop
hashchange Event	hashchange
History Management	history
HTML5 Audio	audio

If audio support is detected, Modernizr assesses which formats the current browser will play. Currently, Modernizr tests `ogg`, `mp3`, `wav` and `m4a`.
Important: The values of these properties are not true booleans. Instead, Modernizr matches the HTML5 spec in returning a string representing the browser's level of confidence that it can handle that codec. These return values are an empty string (negative response), "maybe" and "probably". The empty string is falsy, in other words: `Modernizr.audio.ogg == ''` and `'' == false`
`var audio = new Audio();`
`audio.src = Modernizr.audio.ogg ?`
`'background.ogg' :`
`Modernizr.audio.mp3 ?`
`'background.mp3' :`
`'background.m4a';`
`audio.play();`	

HTML5 Video	video
See Video for Everybody by Kroc Camen for a JavaScript-less way to use HTML5 `video` with graceful fallbacks for all browsers. With Modernizr's `video` detection, you can use CSS and JavaScript styling to further enhance the look and/or interactivity when the browser does support `video`. If video support is detected, Modernizr assesses which formats the current browser will play. Currently, Modernizr tests `ogg`, `webm` and `h264`. Also, read the important note about audio formats above, as the same return values apply to video.	
IndexedDB	indexeddb
Input Attributes	
HTML5 introduces many new attributes for input elements. Modernizr tests for these: `autocomplete`, `autofocus`, `list`, `placeholder`, `max`, `min`, `multiple`, `pattern`, `required`, `step`. These new attributes can do things such as: enable native autocomplete, mimic elem.focus() at page load, do form field hinting, and do client-side validation. View the HTML5 input attribute support page from Mike Taylor to see them in action.	

Input Types	
HTML5 introduces thirteen new values for the `<input>`'s type attribute. They are as follows: `search, tel, url, email, datetime, date, month, week, time, datetime-local, number, range, color.` These types can enable native datepickers, colorpickers, URL validation, and so on. If a browser doesn't support a given type, it will be rendered as a text field. Modernizr cannot detect that date inputs create a datepicker, the color input create a colorpicker, and so on—it will detect that the input values are sanitized based on the spec. In the case of WebKit, we have received confirmation that sanitization will not be added without the UI widgets being in place.	
localStorage	localstorage
Cross-window Messaging	postmessage
sessionStorage	sessionstorage
Web Sockets	websockets
Web SQL Database	websqldatabase
Web Workers	webworkers

Miscellaneous

Feature	`Modernizr` JS object property / CSS classname
Geolocation API	geolocation
Inline SVG	inlinesvg
SMIL	smil
SVG	svg
SVG Clip paths	svgclippaths
This test is only for clip paths in SVG, not necessarily applying SVG clippaths to HTML content. View this demo of SVG clippaths to see the feature in action. If you're curious about clippaths on HTML, read the comments here.	
Touch Events	touch
The Modernizr.touch test only indicates if the browser supports touch events, which does not necessarily reflect a touchscreen device. For example, Palm Pre / WebOS (touch) phones do not support touch events and thus	

fail this test. Additionally, Chrome (desktop) used to lie about its support on this, but that has since been rectified. Modernizr also tests for Multitouch Support via a media query, which is how Firefox 4 exposes that for Windows 7 tablets. For more info, see the Modernizr touch tests.
`if (Modernizr.touch){`
`  // bind to touchstart, touchmove, etc and watch `event.streamId``
`} else {`
`  // bind to normal click, mousemove, etc`
`}`	

WebGL	~~webgl~~

`if (Modernizr.webgl){`
`  loadAllWebGLScripts(); // webgl assets can easily be > 300k`
`} else {`
`  var msg = 'With a different browser you'll get to see the WebGL experience here: \`
`  get.webgl.org.';`
`  document.getElementById( '#notice' ).innerHTML = msg;`
`}`	

Additional existing tests via plugins

In addition to the tests in Modernizr "core", many other common tests are available as Modernizr plugins in the `/feature-detects/` folder of the github repo. If you did not find a test you're looking for on this page here, try there or the issue tracker.

Modernizr Methods

There is a pair of methods available, giving you some added functionality you'll probably enjoy if you do a lot of CSS3 and responsive design.

Modernizr.prefixed()

Detecting CSS feature prefixes

`Modernizr.prefixed(str)`

Modernizr.prefixed() returns the prefixed or nonprefixed property name variant of your input

```
Modernizr.prefixed('boxSizing') // 'MozBoxSizing'
```

Properties must be passed as DOM-style camelCase, rather than `box-sizing` hyphenated style. Return values will also be the camelCase variant, if you need to translate that to hyphenated style use:

```
str.replace(/([A-Z])/g, function(str,m1){ return '-' + m1.toLowerCase(); }
).replace(/^ms-/,'-ms-');
```

If you're trying to ascertain which transition end event to bind to, you might do something like...

```
var transEndEventNames = {
    'WebkitTransition' : 'webkitTransitionEnd',
    'MozTransition'    : 'transitionend',
    'OTransition'      : 'oTransitionEnd',
    'msTransition'     : 'MSTransitionEnd',
    'transition'       : 'transitionend'
},
```

```
transEndEventName = transEndEventNames[ Modernizr.prefixed('transition') ];

// ... bind to the transEndEventName event...
```

(For a more in-depth tutorial on how to use Modernizr.prefixed, see this tutorial by Andi Smith)

Detecting DOM feature prefixes

`Modernizr.prefixed(str, obj[, scope])`

You can use `Modernizr.prefixed` to find prefixed DOM properties and methods easily. For example:

```
Modernizr.prefixed('requestAnimationFrame', window) // fn
```

By passing a second argument, you declare you are looking for a prefixed method or property on that object. If it finds something, but isn't a function, it'll return the object, boolean, number or value that it finds. For example, the quota management API:

```
Modernizr.prefixed('StorageInfo', window) // object.
```

If it finds a function, it'll return a bound function. By default the function will be bound to the second argument. But you can change that by declaring a third argument that it should be bound to.

```
// basic use
Modernizr.prefixed('requestAnimationFrame', window) // fn

// creating a rAF polyfill
window.rAF = Modernizr.prefixed('requestAnimationFrame', window) ||
function(){ ...

// passing a scope
var ms = Modernizr.prefixed("matchesSelector", HTMLElement.prototype,
document.body)
ms('body') // true
```

Lastly, if you want to search for a prefixed DOM feature and only want the name of it back, pass `false` as the 3rd argument:

```
Modernizr.prefixed('requestAnimationFrame', window, false) //
 'webkitRequestAnimationFrame'
```

If you want more, read our github ticket for this feature. Also, we included a `Function.prototype.bind` polyfill in Modernizr 2.5 by default. It supports this feature, but you're welcome to use it in your app code, as well.

(For a more in-depth tutorial on how to use Modernizr.prefixed, see this tutorial by Andi Smith)

mq() media query testing

JavaScript method:

`Modernizr.mq(str)`

Modernizr.mq tests a given media query, live against the current state of the window

A few important notes:

- If a browser does not support media queries at all (eg. oldIE) the mq() will always return false
- A max-width or orientation query will be evaluated against the current state, which may change later.
- You should specify the media type, though commonly the `all` type is good enough:
  `Modernizr.mq('only all and (max-width: 400px)')`
- You must specify values. Eg. If you are testing support for the min-width media query use:
  `Modernizr.mq('(min-width: 0px)')`

Sample usage:

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

Extensibility

The tests in Modernizr core may not cover all your cases, but there is a full set of APIs available for you to plug your own feature tests into Modernizr. Additionally, many of them have been collected already in the `/feature-detects/` folder and in the wiki.

addTest() Plugin API

JavaScript method:

```
Modernizr.addTest(str, fn)
Modernizr.addTest(str, bool)
Modernizr.addTest({str: fn, str2: fn2})
Modernizr.addTest({str: bool, str2: fn})
```

You may want to test additional features that Modernizr currently does not support. For that, you can use the addTest function. For example, some users have requested tests for IE's float double margin bug, and support for `position:fixed`. All the current tests that we've codified are available in the Github. Using addTest, you can add these yourself and get the exact same API as the fully supported tests.

New signatures for this method of accepting a boolean or an object were added for Modernizr 2

Sample Usage:

```
// Test for <track> element  support
Modernizr.addTest('track', function(){
  var video = document.createElement('video');
  return typeof video.addTextTrack === 'function'
});
```

Assuming the above test passes, there will now be a `.track` class on the HTML element and `Modernizr.track` will be true. IE6, of course, will now have a `.no-positionfixed` class.

testStyles()

JavaScript method:

```
Modernizr.testStyles(str,fn[, nodes, testnames])
```

Modernizr.testStyles() allows you to add custom styles to the document and test an element afterwards. An element with the id of "modernizr" is injected into the page.

```
  Modernizr.testStyles('#modernizr { width: 9px; color: papayawhip; }',
function(elem, rule){
      Modernizr.addTest('width', elem.offsetWidth == 9);
  });
```

If your test requires more than a single element to be injected you can specify the `nodes` argument which accepts a number indicating how many extra child div elements you want injected inside the main element. These extra elements will default to and id of "modernizr[n]" where n is a number. If you wish to specify a specific id the `tesnames` argument allows you to pass an array of id's to apply to the child elements.

```
  Modernizr.testStyles('#modernizr { width: 9px; color: papayawhip; }',
function(elem, rule){
      Modernizr.addTest('width', elem.offsetWidth == 9);
  }, 2, ["video", "image"]);
```

testProp()

JavaScript method:

```
Modernizr.testProp(str)
```

`Modernizr.testProp()` investigates whether a given style property is recognized. Note that the property names must be provided in the camelCase variant.

```
Modernizr.testProp('pointerEvents')  // true
```

testAllProps()

JavaScript method:

`Modernizr.`**`testAllProps(str)`**

`Modernizr.testAllProps()` investigates whether a given style property, or any of its vendor-prefixed variants, is recognized. Note that the property names must be provided in the camelCase variant.

```
Modernizr.testAllProps('boxSizing')  // true
```

hasEvent()

JavaScript method:

`Modernizr.`**`hasEvent(str [,elem])`**

`Modernizr.hasEvent()` detects support for a given event, with an optional element to test on

```
Modernizr.hasEvent('gesturestart', elem)  // true
```

_prefixes & _domPrefixes

JavaScript properties:

`Modernizr.`**`_prefixes`** `Modernizr.`**`_domPrefixes`**

The vendor prefixes you'll have to test against. Look in the modernizr source or below to see how to effectively `join()` these arrays in order to test your style properties.

```
Modernizr._prefixes.join('prop' + ':value; ');
/* prop:value; -webkit-prop:value; -moz-prop:value;
   -o-prop:value; -ms-prop:value; -khtml-prop:value; */

'prop:' +  Modernizr._prefixes.join('value; prop:') + 'blah';
/* prop:value; prop:-webkit-value; prop:-moz-value;
   prop:-o-value; prop:-ms-value; prop:-khtml-value; prop:blah; */

'prop:value; ' + Modernizr._domPrefixes.join('Prop' + ':value; ') + ':value';
/* prop:value; WebkitProp:value; MozProp:value;
   OProp:value; msProp:value; Khtml:value */
```