Implementing feature detection

Feature detection involves working out whether a browser supports a certain block of code, and running different code dependent on whether it does (or doesn't), so that the browser can always provide a working experience rather crashing/erroring in some browsers. This article details how to write your own simple feature detection, how to use a library to speed up implementation, and native features for feature detection such as `@supports`.

Prerequisites:	Familiarity with the core HTML, CSS, and JavaScript languages; an idea of the high level principles of cross browser testing.
Objective:	To understand what the concept of feature detection is, and be able to implement suitable solutions in CSS and JavaScript.

The concept of feature detection

The idea behind feature detection is that you can run a test to determine whether a feature is supported in the current browser, and then conditionally run code to provide an acceptable experience both in browsers that *do* support the feature, and browser that *don't*. If you don't do this, browsers that don't support the features you are using in your code won't display your sites properly and will just fail, creating a bad user experience.

Let's recap and look at the example we touched on in our Handling common JavaScript problems — the Geolocation API (which exposes available location data for the device the web browser is running on) has a main entry point for its use, a `geolocation` property available on the global Navigator object. Therefore, you can detect whether the browser supports geolocation or not by using something like the following:

```
if("geolocation" in navigator) {
  navigator.geolocation.getCurrentPosition(function(position) {
    // show the location on a map, perhaps using the Google Maps API
  });
} else {
  // Give the user a choice of static maps instead perhaps
}
```

It is probably better to use an established feature detection library however, rather than writing your own all the time. Modernizr is the industry standard for feature detection tests, and we'll look at that later on.

Before we move on, we'd like to say one thing up front — don't confuse feature detection with **browser sniffing** (detecting what specific browser is accessing the site) — this is a terrible practice that should be discouraged at all costs. See Using bad browser sniffing code for more details.

Writing your own feature detection tests

In this section we'll look at implementing your own feature detection tests, in both CSS and JavaScript.

CSS

You can write tests for CSS features by testing for the existance of *element.style.property* (e.g. `paragraph.style.transform`) in JavaScript.

A classic example might be to test for Flexbox support in a browser; for browsers that supports the newest Flexbox spec, we could use a flexible and robust flex layout. For browsers that don't, we could use a floated layout that works OK, although it is slightly more brittle and hacky, and not as cool-looking.

Let's implement something that demonstrates this, although we'll keep it simple for now.

1. Start by making local copies of our `css-feature-detect.html`, `flex-layout.css`, `float-layout-css`, and `basic-styling.css` files. Save them in a new directory.
2. We will add the HTML5 Shiv to our example too, so that the HTML5 semantic elements will style properly in older versions of IE. Download the latest version (see Manual installation), unzip the ZIP file, copy the `html5shiv-printshiv.min.js` and `html5shiv.min.js` files into your example directory, and link to one of the files by putting the following under your `<title>` element:

```
<script src="html5shiv.min.js"></script>
```

3. Have a look at your example CSS files — you'll see that `basic-styling.css` handles all the styling that we want to give to every browser, whereas the other two CSS files contain the CSS we want to selectively apply to browser depending on their support levels. You can look at the different effects these two files they have by manually changing the CSS file referred to by the second `<link>` element, but let's instead implement some JavaScript to automatically swap them as needed.
4. First, remove the contents of the second `<link>` element's `href` attribute. We will fill this in dynamically later on.
5. Next, add a `<script></script>` element at the bottom of your body (just before the closing `</body>` tag).
6. Give it the following contents:

```
var conditional = document.querySelector('.conditional');
var testElem = document.createElement('div');
if(testElem.style.flex !== undefined && testElem.style.flexFlow
  conditional.setAttribute('href', 'flex-layout.css');
} else {
  conditional.setAttribute('href', 'float-layout.css');
}
```

Here we are grabbing a reference to the second `<link>` element, and creating a `<div>` element as part of our test. In our conditional statement, we test that the `flex` and `flex-flow` properties exist in the browser. Note how the JavaScript representations of those properties that are stored inside the `HTMLElement.style` object use lower camel case, not hyphens, to separate the words.

> **Note**: If you have trouble getting this to work, you can compare it to our css-feature-detect-finished.html code (see also the live version).

When you save everything and try out your example, you should see the flexbox layout applied to the page if the browser supports modern flexbox, and the float layout if not.

> **Note**: Often such an approach is overkill for a minor feature detection problem — you can often get away with using multiple vendor prefixes and fallback properties, as described in CSS fallback behavior and Handling CSS prefixes.

@supports

In recent times, CSS has had its own native feature detection mechanism introduced — the `@supports` at-rule. This works in a similar manner to media queries (see also Responsive design problems) — except that instead of selectively applying CSS depending on a media feature like a resolution, screen width or aspect ratio, it selectively applies CSS depending on whether a CSS feature is supported.

For example, we could rewrite our previous example to use `@supports` — see `supports-feature-detect.html` and `supports-styling.css`. If you look at the latter, you'll see a couple of `@supports` blocks, for example:

```css
@supports (flex-flow: row) and (flex: 1) {

  main {
    display: flex;
  }

  main div {
    padding-right: 4%;
    flex: 1;
  }

  main div:last-child {
    padding-right: 0;
  }

}
```

This at-rule block applies the CSS rule within only if the current browser supports both the `flex-flow: row` and `flex: 1` declarations. For each condition to work, you need to include a complete declaration (not just a property name) and NOT include the semi-colon on the end.

`@supports` also has `OR` and `NOT` logic available — the other block applies the float layout if the flexbox properties are not available:

```
@supports not (flex-flow: row) and (flex: 1) {

  /* rules in here */

}
```

This may look a lot more convenient than the previous example — we can do all of our feature detection in CSS, no JavaScript required, and we can handle all the logic in a single CSS file, cutting down on HTTP requests. the problem here is browser support — `@supports` is not supported at all in IE, and only supported in very recent versions of Safari/iOS WebKit (9+/9.2+), whereas the JavaScript version should work in much older browsers (probably back to IE8 or 9, although older versions of IE will have additional problems, such as not supporting `Document.querySelector`, and having a messed up box model).

JavaScript

We already saw an example of a JavaScript feature detection test earlier on. Generally, such tests are done via one of the following common patterns:

Summary of JavaScript feature detection techniques

Feature detection type	Explanation	Example

Feature detection type	Explanation	Example
if member in object	Check whether a certain method or property (typically an entry point into using the API or other feature you are detecting for) exists in its parent Object.	`if("geolocation" in navigator) { ... }`

Property on element	Create an element in memory using `Document.createElement()` and then check if a property exists on it. The example shown is a way of detecting HTML5 Canvas support.	`function supports_canvas() {`
		`return !!document.createElement('canvas').ge`
		`}`
		
		`if(supports_canvas()) { ... }`

Method on element return value	Create an element in memory using `Document.createElement()` and then check if a method exists on it. If it does, check what value it returns.	See Dive Into HTML5 Video Formats detection test
Property on element retains value	Create an element in memory using `Document.createElement()`, set a property to a certain value, then check to see if the value is retained.	See Dive into HTML5 `<input>` types detection test

Note: The double `NOT` in the above example (`!!`) is a way to force a return value to become a "proper" boolean value, rather than a Truthy/Falsy value that may skew the results.

The Dive into HTML5 Detecting HTML5 Features page has a lot more useful feature detection tests besides the ones listed above, and you can generally find a feature detection test for most things by searching for "detect support for YOUR-FEATURE-HERE" in your favourite search engine. Bear in mind though that some features are known to be undetectable — see Modernizr's list of Undetectables.

matchMedia

We also wanted to mention the `Window.matchMedia` JavaScript feature at this point too. This is a property that allows you to run media query tests inside JavaScript. It looks like this:

```
if (window.matchMedia("(max-width: 480px)").matches) {
  // run JavaScript in here.
}
```

As an example, our Snapshot demo makes use of it to selectively apply the Brick JavaScript library and use it to handle the UI layout, but only for the small screen layout (480px wide or less). We first use the `media` attribute to only apply the Brick CSS to the page if the page width is 480px or less:

```
<link href="dist/brick.css" type="text/css" rel="stylesheet" media="all an
```

We then use `matchMedia()` in the JavaScript several times, to only run Brick navigation functions if we are on the small screen layout (in wider screen layouts, everything can be seen at once, so we don't need to navigate between different views).

```
if (window.matchMedia("(max-width: 480px)").matches) {
  deck.shuffleTo(1);
}
```

Using Modernizr to implement feature detection

It is possible to implement your own feature detection tests using techniques like the ones detailed above. You might as well use a dedicated feature detection library however, as it makes things much easier. The mother of all feature detect libraries is Modernizr, and it can detect just about everything you'll ever need. Let's look at how to use it now.

When you are experimenting with Modernizr you might as well use the development build, which includes every possible feature detection test. Download this now by:

1. Clicking on the Development build link.
2. Clicking the big pink *Build* button in the page that comes up.
3. Clicking the top *Download* link in the dialog box that appears.

Save it somewhere sensible, like the directory you've been creating your other examples for in this article.

When you are using Modernizr in production, you can go to the Download page you've already visited and click the plus buttons for only the features you need feature detects for. Then when you click the *Build* button, you'll download a custom build containing only those feature detects, making for a much smaller file size.

CSS

Let's have a look at how Modernizr works in terms of selectively applying CSS.

1. First, make a copy of `supports-feature-detect.html` and `supports-styling.css`. Save them as `modernizr-css.html` and `modernizr-css.css`.
2. Update your `<link>` element in your HTML so it points to the correct CSS file (you should also update you `<title>` element to something more suitable!):
   ```
   <link href="modernizr-css.css" rel="stylesheet">
   ```
3. Above this `<link>` element, add a `<script>` element to apply the Modernizr library to the page, as shown below. This needs to be applied to the page before any CSS (or JavaScript) that might make use of it.
   ```
   <script src="modernizr-custom.js"></script>
   ```
4. Now edit your opening `<html>` tag, so that it looks like this:
   ```
   <html class="no-js">
   ```

At this point, try loading your page, and you'll get an idea of how Modernizr works for CSS features. If you look at the DOM inspector of your browser's developer tools, you'll see that Modernizr has updated your `<html>` `class` value like so:

```
<html class="js no-htmlimports sizes flash transferables applicationcache
blob-constructor cookies cors ...AND LOADS MORE VALUES!>
```

It now contains a large number of classes that indicate the support status of different technology features. As an example, if the browser didn't support flexbox at all, `<html>` would be given a class name of `no-flexbox`. If it did support modern flexbox, it would get a class name of `flexbox`. If you search through the class list, you'll also see others relating to flexbox, like:

- `flexboxlegacy` for the old flexbox spec (2009).
- `flexboxtweener` for the 2011 inbetween syntax supported by IE10
- `flexwrap` for the `flex-wrap` property, which isn't present in some implementations.

Note: You can find a list of what all the class names mean — see Features detected by Modernizr.

Moving on, let's update our CSS to use Modernizr rather than `@supports`. Go into `modernizr-css.css`, and replace the two `@supports` blocks with the following:

```
/* Properties for browsers with modern flexbox */

.flexbox main {
  display: flex;
}

.flexbox main div {
  padding-right: 4%;
  flex: 1;
}

.flexbox main div:last-child {
  padding-right: 0;
}

/* Fallbacks for browsers that don't support modern flexbox */

.no-flexbox main div {
  width: 22%;
  float: left;
  padding-right: 4%;
}


.no-flexbox main div:last-child {
```

```
.no-flexbox main div:last-child {
  padding-right: 0;
}

.no-flexbox footer {
  clear: left;
}
```

So how does this work? Because all those class names have been put on the `<html>` element, you can target browsers that do or don't support a feature using specific descendant selectors. So here we're applying the top set of rules only to browsers that do support flexbox, and the bottom set of rules only to browsers that don't (`no-flexbox`).

Note: Bear in mind that all of Modernizr's HTML and JavaScript feature tests are also reported in these class names, so you can quite happily apply CSS selectively based on whether the browser supports HTML or JavaScript features, if needed.

Note: If you have trouble getting this to work, check your code against our `modernizr-css.html` and `modernizr-css.css` files (see this runing live also).

JavaScript

Modernizr is also equally well-prepared for implementing JavaScript feature detects too. It does this by making the global `Modernizr` object available to the page it is applied to, which contains results of the feature detects as `true`/`false` properties.

For example, load up our `modernizr-css.html` example in your browser, then try going to your JavaScript console and typing in `Modernizr.` followed by some of those class names (they are the same here too). For example:

```
Modernizr.flexbox
Modernizr.websqldatabase
Modernizr.xhr2
Modernizr.fetch
```

The console will return `true`/`false` values to indicate whether your browser supports those features or not.

Let's look at an example to show how you'd use those properties.

1. First of all, make a local copy of the `modernizr-js.html` example file.
2. Attach the Modernizr library to the HTML using a `<script>` element, as we have done in previous demos. Put it above the existing `<script>` element, which is attaching the Google Maps API to the page.
3. Next, fill in the `YOUR-API-KEY` placeholder text in the second `<script>` element (as it is now) with a valid Google Maps API key. To get a key, sign in to a Google account, go to the Get a Key/Authentication page, then click the blue *Get a Key* button and follow the instructions.
4. Finally, add another `<script>` element at the bottom of the HTML body (just before the `</body>` tag), and put the following script inside the tags:

```
if(Modernizr.geolocation) {

  navigator.geolocation.getCurrentPosition(function(position) {

    var latlng = new google.maps.LatLng(position.coords.latitude
    var myOptions = {
      zoom: 8,
      center: latlng,
      mapTypeId: google.maps.MapTypeId.TERRAIN,
      disableDefaultUI: true
    }
    var map = new google.maps.Map(document.getElementById("map_ca
  });

} else {
  var para = document.createElement('p');
  para.textContent = 'Argh, no geolocation!';
  document.body.appendChild(para);
}
```

Try your example out! Here we use the `Modernizr.geolocation` test to check whether geolocation is supported by the current browser. If it is, we run some code that gets your device's current location, and plots it on a Google Map.

Summary

This article covered feature detection in a reasonable amount of detail, going through the main concepts and showing you how to both implement your own feature detection tests and use the Modernizr library to implement tests more easily.

Next up, we'll start looking at automated testing.

Previous | Overview: Cross browser testing | Next

In this module