Cutting the mustard: then and now

Cutting the mustard is a term first defined by the BBC to explain how to test if a browser supported the features needed to run a web application. In the example below we need to make sure that the browser supports query selectors, local storage and event listeners before we can build the app with these features. The traditional code works like this:

```
if('querySelector' in document
     && 'localStorage' in 'localStorage' 'addEventListener' in window) {
     'addEventListener'// bootstrap the javascript application
}
```

I've also seen it converted to a function for ease of writing. We first define the items that make the browser cut the mustard and make them the return value of our check function

```
function cutsMustard() {
  return 'querySelector' in document
     && 'localStorage' in window
     && 'addEventListener' in window;
}

if (cutsMustard()) {
  console.log('We cut the mustard!!1');
}
```

Cutting the mustard today

in Cutting the mustard - 2018 edition Stefan Baumgartner suggests different strategies for "cutting the mustard" with current browsers.

Using feature detection is still important but, perhaps the best way to test features according to Baumgartner, is to write modules instead of scripts and then

use the `type="module"` attribute of the script tag, like this.

```
<script type="module">
  import { init } from './client.js';

  init();
</script>
```

The browsers that don't support modules will not undertand the script tag above and skip it, and the code inside, completely. Problem solved, right?

Not so fast, cowboy.

We don't need to serve identical content to all devices but we need to ensure that our core experiences are inclusive of as many people as possible with as good an experience as we can. From my perspective, cutting the mustard should not include ES2017 features until they are more widely supported both in desktop and mobile.

Interestingly, when I asked Stephan whether his strategy includes mobile the answer sounded very much like "who cares".

> What's mobile but a set of browsers with a set of features :-) If your mobile browser supports modules, it cuts the mustard.
>
> — Stefan Baumgartner (@ddprrt) November 29, 2018

Because a lot of mobile browsers do not support modules they "don't cut the mustard" if we apply the strict definition of his mustard test.

This works great for people who have the latest and greatest browsers and operaating systems but what happens if they don't? I like Bruce Lawson's idea of the ***World Wide Web*** not the ***Wealthy Western Web*** so a lot of the following ideas go towards wide-spread access rather than developer comfort and ergonomics.

Do we need to transpile?

Inside the module we can be more confident about support for modern features like spread operators, async/await and other powerful features of the web

platform. But it's an all-or-nothing deal. What happens if we have to support older browsers like those on the following list, some of which may support some of the capabilities we need for our applications but not others?

- Opera Mobile
- Samsung Internet
- QQ Browser
- Baidu
- Android Webview
- Older versions of Chrome for Android

Sure, the browsers are evergreen and should update automatically, unless there is a corporate policy in place that prevents updated or the users in mobile devices are wary of bandwidth financial cost or storage to upgrade the browser to one that supports modules.

So I'll start with the assumption that we need to transpile our code and that "latest and greatest" features are a progressive enhancement for those who can afford it.

One way to do it is to provide the transpiled bundle only to those browsers that don't support modules.

As discussed earlier, browsers that don't support modules will skip scripts with `type="module"` and ignore all their contents.

We can use the `nomodule` attribute to do the reverse. Browsers that understand modules will ignore script elements with `nomodule` attribute.

The code below will load scripts based on whether the browser supports modules or not.

```
<script type="module"
        src="./ecmascript-modules-main.js"></script>

<script nomodule src="./bundled-main.js"></script></script>
```

This example will do the same thing with inline code.

```
<script type="module">
  alert('The browser DOES support ES modules');
</script>
<script nomodule></script>
  alert('The browser DOES NOT support ES modules');
</script>
```

We can also just work with feature detection for those features that we actually need rather than assuming that because we have modules we have everything else we need.

In this example we need to be sure the browser supports async functions and await keywords, service worker, local storage and indexedDB.

I'm using a proposal by Jouni for doing feature tests for async functions.

The local storage detection code is from How I detect and use localStorage: a simple JavaScript pattern by Mathias Bynens.

The idea remains the same: we test for each feature and, if needed, we configure the test to account for vendor prefixes or different syntax for the same API (shouldn't happen in evergreen browsers but it wouldn't be uncommon for a browser to support prefixed and unprefixed versions of the same API)

```
// Feature detect local storage
var storage;
var fail;
var uid;
try {
  uid = new Date;
  (storage = window.localStorage).setItem(uid, uid);
  fail = storage.getItem(uid) != uid;
  storage.removeItem(uid);
  fail && (storage = false);
} catch (err) {
  // empty by design
}
```

```
// Feature detect for async function
var __modernScript__;
try {
  eval("typeof Object.getPrototypeOf(async function() {}).constructor ===
  __modernScript__ = document.createElement('script');
  __modernScript__.type = 'text/javascript';
  __modernScript__.src = '##URL##';
  document.getElementsByTagName('head')[0].appendChild(__modernScript__);
} catch (err) {
  "typeof Object.getPrototypeOf(async function() {}).constructor === 'func
}

function cutsMustard() {
  return 'serviceWorker' in navigator
    && window.indexedDB
    && storage
    && __modernScript__
    && 'addEventListener' in window;
}

if (cutsMustard()) {
  console.log('We cut the mustard!!1');
}
```

Yes, feature detection means more work both in the scripts themselves and in tooling but it ensures that we provide our core experience to as wide an audience as possible.

To me this is all a matter of willingness to put the effort into creating engaging experiences that are progressively enhanced for the latest and greatest. If it means adding tooling or extra code to make things work, then so be it.

And cutting the mustard is not a static process. It may be that our next project can leverage wider support for those features we'd like to use to cut the mustard today.