JavaScript in the browser

Why JavaScript in the browser?

Now that we can structure the document and its content with HTML, and that we can style it in many ways with CSS, we may want to go beyond those capabilities (dynamically change the HTML document, perform some operations based on some current circumstances, …). JavaScript was created with exactly that goal in mind, and is still being used as such; its recent popular renewal as a language for scripts and web back-ends using Node.js is quite recent. All browsers expect HTML, CSS and JavaScript; Whatever you want to write for the web, even the most complex applications (even GMail, Google Maps, …) will have no choice but to be composed of those three technologies if they want to be understood by browsers.

A quick history to know where we come from

Note: the JavaScript language has **nothing** to do with the Java language. Please make sure not to use the wrong word!

Step 1: Browser War and standardization

Before the Browser War, the first “mainstream” browser was called Mosaic. It existed between 1993 and 1997 and had very limited features. Netscape released Netscape Navigator in 1995 in an attempt to shake things up and bring more features to the web experience. Quickly, Netscape Navigator had ~80% market share; but then quickly came Microsoft Internet Explorer 1.

The Browser War was a time when web browser market share was shared between Netscape Navigator and Microsoft Internet Explorer, and each browser was trying to one-up the other with fancy features for developers. It is a time that saw the birth of frames, CSS, and… JavaScript! Brendan Eich, back then a developer at Netscape, wrote the first version of JavaScript in 10 days (and it kinda still shows in the quirks of the language!), in order to ship it before Internet Explorer, with almost no documentation. People making websites were quick to use it to do nice things with Netscape Navigator, so Microsoft shipped a reverse-engineered version of it a few days later.

Of course, with this kind of approach, behavior in browsers were inconsistent, so Netscape ended up submitting the language for standardization with the ECMA, where it was given its actual name: ECMAscript (but everybody keeps saying JavaScript, so you should keep calling it that too).

Step 2: shy steps towards consistency

Netscape Navigator 5 was taking far too long to build because of heavy legacy issues, so Netscape decided to give it up and to prepare for Netscape Navigator 6, rebuilt entirely from the ground up. Rebuilding something this large entirely from the ground up is notoriously always a very bad decision in product management, and this occurrence didn't fail the rule: Netscape disappeared for years while its browser took much longer than expected to be built, making a boulevard for Microsoft Internet Explorer to take virtually all of the market share.

The result for JavaScript is that for years, Microsoft didn't follow through on their promise to make their version of JavaScript compatible with the ECMA standard. They had good reasons not to: since almost the entire planet was using their product, their incompatibility with other browsers was kind of a competitive advantage. Since developers who could only focus their efforts on one version of JavaScript were focusing on Microsoft Internet Explorer, their websites were effectively broken in other browsers.

This gave birth to Javascript libraries whose goals were to harmonize a single language API that worked in all browsers, such as JQuery (which comes with other features too).

Step 3: towards a more mature language

After years of JavaScript developers having to either produce one code per browser, or use libraries like JQuery, Microsoft's strategy eventually caught up to them for two main reasons:

- Rich front-end applications requiring heavy JavaScript became more and more mainstream, and Microsoft's willful years of tardiness compared to other browsers led Internet Explorer (circa version 6, 7 and 8) to be much slower than the rest of the competition. People started to install other browsers in order to use comfortably their Gmail, Google Maps, etc.
- Microsoft's behavior was so obviously anti-constructive that they got a ton of lawsuits from competitors and from governments, most of which they lost. One of the most painful outcomes for them was the "ballot screen": they had to display a screen on installation presenting their competing browsers to all of their European customers. This led European users to even more migrate to other browsers. This was a wake-up call for Microsoft, which reconsidered entirely their strategy, and came up with Internet Explorer versions (circa version 9 or 10) that were finally reasonably compatible with all relevant specs, and with JavaScript engine speeds comparable to their competitors. More recently, to make Internet Explorer and its terrible reputation a thing of the past, they have rebranded it as "Microsoft Edge", and adopted a much more open strategy.

Microsoft 180-degre shift was immensely beneficial to JavaScript for browsers, since the same JavaScript code can now be ran transparently on all browsers with little risk; also, now that the whole industry is caught up, JavaScript's pace of standardization for new features has dramatically increased within the past few years, and JavaScript is slowly becoming less and less like the quirky language that was created in 10 days as a marketing stunt in the mid-90s, and more and more like a modern, full-featured language.

JQuery & co

While JQuery's syntax is very appreciated by developers, one of its obvious tradeoffs is one of performance: all of your users are loading and running a library bringing a ton of features, while you might only need a handful.

It used to make obvious sense up until recently, but JQuery's two main upsides are now eroding:

- It was immensely useful in writing code to safely work on all browsers; but now that Microsoft has caught up and is "fighting" for Microsoft Explorer < 9 to disappear from their clients' computers, this has become less and less a problem as those browsers disappear.
- Even today, it makes quite a few annoying and/or verbose Javascript syntaxes (such as making an Ajax call, or "selecting" an HTML element) much more developer-friendly. But for the newest versions of ECMAscript, many JQuery syntaxes have been used as inspiration and are directly offered in native "vanilla" JavaScript.

Even though JQuery is still heavily used in a lot of production websites, developers are much more prompt to reconsider using it in their websites if they only need a handful of features (see the website called "You might not need JQuery", offering easy solutions in native "vanilla" JS to reproduce the most used features in JQuery).

A more lightweight and recent library that has been popular is Underscore.js, which was later forked as another project called lodash (got the pun?). It only attempted to make JavaScript's syntax less verbose (and didn't attempt to solve the browsers' inconsistencies). Since they do not intend to tackle the differences between browsers, they are as relevant when using JavaScript in the browser as with Node.js. Maybe someday, ECMAscript will catch up to the syntactic sugar offered by Underscore and lodash, but waiting for this, they are quite popular today, and should remain so in the foreseeable future.

Some feats of JavaScript in the browser

Now that you're amazingly comfortable with JavaScript in Node.js (!!!), you might wonder what the differences are between the JavaScript you've experienced there and the JavaScript you're going to experience in the browser. Here are some differences:

The JavaScript version In Node.js, if a new version of JavaScript is stabilized, then you can update Node.js on your computer/server, and suddenly, you can use the latest fancy features of JavaScript. In the browser, you know nothing about the engine running your code, since it's the user's browser. They might be running a very old browser, that doesn't understand the latest features. Therefore, most recent upgrades to JavaScript are not usable by front-end developers for months or years. Use the "Can I use?" website to find out what features are reasonably implemented by browsers.

The "special" objects The browser is giving you access to its features through a number of objects, like the `window` object, which represents the browser tab this code is running in, the `navigator` object, giving information about the browser used in general, etc. Most of those APIs are not standardized, but they are typically the same across browsers.

The DOM Standing for “Document Object Model”, the DOM is the HTML code as the browser “understands” it, i.e. as a tree. If you’re uncomfortable with why your browser thinks of your HTML code as a tree, you should read this: What is the Document Object Model? (/rltoken/Tf6AsSG-UUxeneuRSstfrw). The browser comes with the DOM API, which allows you to travel through the tree as needed: find elements in it, add or remove nodes, travel up or down the tree, … Those APIs are made available through the document object, representing the HTML document in which the code runs.

The event-orientation Since JavaScript in the browser is meant to perform actions when certain things happen, callbacks are not being run after I/O operations are performed like in Node.js, but when some events happen. Those events can be triggered by the user (a click on a button, the browser window being resized, …) or sometimes not (an image isn’t loading properly, the HTML webpage has finished loading, …)

Tools

You have many features in your browser’s dev tools meant to help you: a JavaScript console where you can run code that will directly apply to the webpage that is opened, a debugger allowing you to add breakpoints, etc. Take some time to get familiar with them…