On the Web Platform Cornucopia

Tommi Mikkonen[1], Cesare Pautasso[2], Kari Systä[3] and Antero Taivalsaari[4]

[1]University of Helsinki, Helsinki, Finland
[2]USI, Lugano, Swizerland
[3]Tampere University, Tampere, Finland
[4]Nokia Bell Labs, Tampere, Finland
tommi.mikkonen@helsinki.fi, cesare.pautasso@usi.ch,
kari.systa@tuni.fi, antero.taivalsaari@nokia-bell-labs.com

Abstract. The evolution of the Web browser has been organic, with new features introduced on a pragmatic basis rather than following a clear rational design. This evolution has resulted in a cornucopia of overlapping features and redundant choices for developing Web applications. These choices include multiple architecture and rendering models, different communication primitives and protocols, and a variety of local storage mechanisms. In this position paper we examine the underlying reasons for this historic evolution. We argue that without a sound engineering approach and some fundamental rethinking there will be a growing risk that the Web may no longer be a viable, open software platform in the long run.

Keywords: Web Platform, Technology Design Space, Software Engineering Principles, Web Engineering, Progressive Web Applications, HTML5

1 Introduction

The Web was originally designed for global document publishing. The scripting capabilities that were added later evolved into a myriad of overlapping programming capabilities. Today, after more than two decades of organic evolution, there are (too) many ways to build software for the Web platform, with developers continuously chasing after the latest and greatest frameworks and development paradigms[1]. This evolution has been driven by competition against other platforms (e.g., native mobile applications or traditional desktop applications) and by competition within the Web platform (i.e., among Web browser vendors and framework developers). The result is a cornucopia of choices providing a rich, complex and ever-growing set of features that need to be mastered by developers and maintained by browser vendors.

In this position paper we look at the design of the Web platform from a Web Engineering standpoint to present a deeper understanding of the driving forces behind its growth and evolution; such continuous organic evolution has made us

[1] https://stateofjs.com/

concerned of the long term sustainability of the Web as a software platform. In addition, we compare the Web to more traditional software platforms, especially in light of how they fulfill MacLennan's classic software engineering principles [1]. We focus on two key principles, *simplicity* and *consistency*: there should be a minimum number of non-overlapping concepts with simple rules for their combination; things that are similar should also look similar and different things should look different. Using the words of Brooks: "It is better to have a system reflect one set of design ideas than to have one that contains many good but independent and uncoordinated ideas" [2].

In continuation to our previous papers (e.g., [3,4,5]), we claim that the evolution of the Web browser has been driven by new features introduced on a pragmatic basis (often purely for commercial needs) instead of being based on a justified long-term design rationale[2]. This has resulted in redundant features, including multiple architecture and rendering models, communication primitives and protocols, local storage mechanisms and programming language constructs. In this paper, we study this cornucopia of overlapping features inside the browser, their origins, relations, and intended use cases. Sharing the concerns of many[3], we argue that without a sound engineering approach and fundamental rethinking there will be a growing risk that the Web may no longer be seen as the open, universal, stable and viable software platform it has attempted to become in the past decades [7].

2 Evolution of the Web as a Software Platform

In the early life of the Web, Web pages were truly *pages*, i.e., semi-structured textual documents that contained some primitive forms of multimedia content (e.g., images) without animation support (except for the controversial blinking tag that was eliminated later) or any interactivity beyond the basic ability to navigate back and forth between pages. Navigation was based on simple *hyperlinks*, and a new Web page was loaded from the Web server each time the user followed a link. For reading user input some pages were presented as *forms* with simple text fields, buttons and selection widgets.

The Web became increasingly interactive when Web pages started containing animated graphics and audio tracks, some of which were rendered by browser plug-in components (such as Java Applets, Flash, RealPlayer, Quicktime and Shockwave), while others were scripted using the JavaScript language [8], introduced in Netscape Navigator version 2.0B almost as an afterthought in December 1995. From technological standpoint, this phase was driven by efforts and investment into various competing browser plugins, which would provide a proprietary browser-independent layer and a common runtime environment for applications that could deliver an interactive user experience beyond the limits of the underlying Web browser.

[2] See for example the controversial decisions to provide standard support for Digital Rights Management in clear conflict with the Open Web Principle [6].

[3] `https://extensiblewebmanifesto.org/`

The introduction of dynamic *DHTML*, Cascading Style Sheets (CSS), and the JavaScript language, with programmatic access to the Document Object Model (DOM) and the underlying HTTP client (with the XMLHttpRequest object), enabled highly interactive Web pages with built-in support for modifying the content of Web pages without reloading them. This effectively decoupled the page navigation lifecycle from the communication exchanged with the server, while still using the original HTTP protocol. This technology mix became known as Ajax (Asynchronous JavaScript and XML) [9]. In this phase, the Web started moving in directions that were unforeseen by its original designer, with Web sites behaving more like multimedia presentations and traditional low-latency rich client applications rather than simple static pages.

In the early 2000s, the concept of Software as a Service (SaaS) emerged. At that point, people realized that the ability to offer software applications seamlessly over the Web and then perform instant worldwide updates would require a truly universal runtime execution environment, which become familiar under the HTML5 brand [10]. Observing the benefits of this model, developers started to build Web sites that behave much like desktop applications [11], for example, by allowing Web pages to be updated partially, rather than requiring the entire page to be refreshed. This also increased the demand for a full-fledged programming language that could be used directly from inside the Web browser instead of relying on external plug-in components. Since JavaScript was readily available inside every browser [12], it became a target of significant investment, especially to improve its performance [13] and add language features to make the language more amenable to developers.

This brings us to the current Web platform, featuring a continuously evolving, evergreen Web browser, with support – at the time of writing – for two hundred and one HTML5-related specifications published by the W3C and the Web Hypertext Application Technology Working Group (WHATWG). Out of these 59 have been abandoned or are already superseded[4]. These specifications define a myriad of APIs, formats and browser features for building progressive Web applications [14] that can run in a standards-compatible Web browser.

3 Redundancy Within Web Platform APIs

The evolution of the Web differs from the evolution of many other software platforms. While many popular open source platforms have grown under the leadership of a benevolent dictator, the governance of the Web has shifted from a design-by-committee approach to multiple committees with many players pulling the evolution in different directions.

Just like classic native software platforms, the modern Web browser offers a number of APIs that provide abstractions to applications. However, taking graphics rendering as the first example, the browser includes multiple alternatives – DOM/DHTML, 2D Canvas, SVG, and 3D WebGL – each of which

[4] `http://html5-overview.net/`

introduces a very different development paradigm, e.g., regarding whether rendering is performed declaratively or programmatically, or whether the graphics is expected to be managed by the browser or explicitly by the developers themselves [3]. In contrast, many graphical user interface toolkits used in traditional software platforms contain several layers of rendering capabilities for developing graphical user interfaces, with carefully designed abstractions and implementation layers. In such designs, one can identify distinct layers that implement intermediate abstractions, which at some point in history were intended for the programmers, but which over time were complemented with simpler and more powerful higher-level interfaces. Unlike in the Web browser, these lower-level APIs form a layered architecture wherein each layer offers a coherent set of abstractions that are open for 3rd party developers, and not only to the internal implementation of the framework itself.

Such layered designs are common in software platforms. For instance, layering is used in the Unix/Linux file system abstractions, providing uniform access to several types of storage and communication devices. Instead, Web application developers need to choose between overlapping APIs for storing data inside the browser, such as Cookies, Local/Session Storage, or IndexedDB (without counting the now deprecated WebSQL). Another example is the area of communication mechanisms. On many traditional software platforms, there is a stack of protocols, such as the TCP/IP stack, wherein each layer provides an abstraction of a particular type of service. In the browser there are different protocols (HTTP version 1, 1.1, 2.0, 3.0, WebSockets, WebRTC) exposed through several APIs and programming techniques, such as Programmatic HTTP (originally known as XMLHttpRequest; now being replaced by Fetch), Server-Sent Events, or Service Workers – again partially redundant and at the API level mostly unrelated to each other.

In the area of programming languages, JavaScript is the *lingua franca* of the Web. After making the jump to also server side with Node.js, JavaScript has become one of the most widely used programming languages in the world. This popularity has put a lot of pressure on JavaScript language evolution, e.g., to improve its performance, clean up some of its original idiosyncrasies, and generally make the language more approachable to developers familiar with other languages (such as Java). Out of many possible examples, we mention (1) variable declaration constructs (the attempt to replace the original `var` with function scoping with `let` and `const` with block scoping), and (2) three contrasting approaches to inheritance: prototypical, parasitic (or closure-based), and the recently added class-based (with mixins). (3) For asynchronous event-based programming, developers can choose from callbacks, futures/promises and the recently added `async/await` constructs. Each addition is intended to be an improvement over the previous one(s), but the language keeps growing after improvement added on top of previously added constructs.

Things mentioned above are just few examples of the ongoing emergence of redundant options available to Web developers within what we call the Web

Platform Cornucopia. Next, we place the focus on the sources of this cornucopia, and study some of the patterns in browser API formation.

4 Patterns for Browser API Formation: A Technical View

Vendor-specific browser APIs and features. Looking back to the history of browser evolution, almost all the facilities we take for granted were once specific to one browser vendor. Today's core features, such as JavaScript, Cascading Style Sheets (CSS) and various HTML tags – apart from a small core of HTML1 tags originating from Tim Berners-Lee – were vendor-specific at some point during the so-called browser wars[5]. While conventions such as prefixing features with vendor-specific names helps ensure that developers are more aware of vendor-specific extensions (with the expectation that they eventually become part of standards), in some cases vendor-specific extensions spread into other browsers even before they are standardized. For instance, some `-webkit` prefixed features are supported by non-WebKit browsers. In the light of recent developments, the era of browser-specific features may not be over yet, since each browser vendor follows a different roadmap in embracing and implementing new standards.

Plugin components. Historically, Web browser plugin components played an important role in the development of browser features. Probably the best example are video codecs inside the browser. For a long time, the Flash player[6] was the dominant technology in that role, whereas now almost all browsers include HTML5 video support, and the role of Flash is diminishing rapidly.

Versioned recommendations versus living standards. Today, there are two sets of guidelines that the browser vendors should follow. W3C[7] aims at providing versioned recommendations, whereas the WHATWG[8] community introduces living, continuously evolving standards. Both forums advance at a different pace, and their operations are uncoordinated[9]. While at present the differences are small, we expect that more and more diverging features will be proposed in the long run.

New hardware. The introduction of new hardware capabilities can spark the need for new software platform APIs. Examples pertaining to the Web platform include the Geolocation API and the WebGL API[10], which is almost identical to OpenGL[11]. The total the number of direct-access low-level APIs in the context of the Web browser is still low. Key reasons for this are the concerns regarding the security of the Web browser sandbox, and privacy concerns due to increased user fingerprinting exposure.

[5] Wikipedia `https://en.wikipedia.org/wiki/Browser_wars` in fact lists three separate browser wars (1995-2001, 2004-2018, 2018-present).
[6] `https://www.adobe.com/fi/products/flashplayer.html`
[7] `https://www.w3.org/`
[8] `https://whatwg.org/`
[9] `https://dzone.com/articles/w3c-vs-whatwg-html5-specs`
[10] `http://www.khronos.org/webgl/`
[11] `https://www.opengl.org/`

Web frameworks. Yet another source of new browser features is the evolution of Web frameworks – especially those frameworks that reach dominant status. Dominant frameworks are used so extensively in application development that their abstractions start gradually "leaking into" standards as well. There are many examples, e.g., in the area of model-view-controller programming patterns, data binding, and reactive programming. A very concrete example are the jQuery[12] library's $ selectors, which became part of the standard DOM as *document.querySelector*.

Language pre-compilers. JavaScript has effectively taken the role of the "assembly language of the Web" – literally in the case of the WebAssembly subset [15]. There are a growing number of languages such as CoffeeScript, TypeScript, Elm, Emscripten, RubyJS, Pyjamas, Processing.js, Scala.js, ClojureScript and PharoJS that can be compiled or translated into JavaScript. These languages take advantage of the JavaScript engine in the Web browser or in the cloud backend (Node.js) as a universal execution target platform.

A similar pattern can be observed with CSS – the declarative language for styling Web content. Preprocessors such as Less, Sass, or SCSS offer additional stylesheet features (e.g., variables, macros, mixins, nesting of selectors within formatting rules, and the parent selector) that can be compiled into plain CSS. Features of these higher-level languages have also started trickling down into the core platform. Good examples are, e.g., CSS variables and computed expressions, and JavaScript arrow functions.

5 Web Frameworks to the Rescue?

Ideally, in a software development environment there should be only one, clearly the best and most obvious way to accomplish each task. However, in Web development – even in a generic Web browser without add-on components or libraries – there are several overlapping ways to accomplish even the most basic tasks. A popular approach to bring back simplicity and consistency into the development process is to leverage Web frameworks. Applications built on top of higher-level frameworks typically use the lower-level browser APIs only indirectly, through the frameworks. For instance, the XMLHttpRequest API mentioned earlier was available in many browsers well before it became a key building block for Single-Page Web Applications and AJAX [9]. Similarly, one can view Mercure[13] as a variation of the well-established Server-Sent Events (SSE) technology, augmented with additional library support. Likewise, the controversial CSSinJS proposal[14] (using the JavaScript syntax to encode CSS rules) would remove the need to use one of the three core languages of the Web platform (HTML, CSS and JavaScript).

This role is where Web frameworks excel. They offer a more coordinated and coherent set of development facilities, development patterns and experiences,

[12] `https://jquery.com/`
[13] `https://github.com/dunglas/mercure`
[14] `https://cssinjs.org/`

but in the end rely on standard browser APIs. Frameworks can be designed to follow established software engineering principles, bypassing the incoherent and rather organically evolved features underneath. Furthermore, it is possible to take specifics of the application genre and habits of the developer community into account. Furthermore, the frameworks can propose higher-level APIs that hide the diversity of the underlying APIs. Movements such as Progressive Web Apps set their goals even further by considering mobile devices, too. New rendering and visualization techniques such as WebVR[15] and WebXR Device API[16] take the Web towards virtual and augmented reality rendering with new APIs, building upon the well-established WebGL API discussed above. Obviously, facilities provided by the most successful libraries and frameworks can eventually gravitate down to be adopted as standard browser features, following the pattern presented above.

Unfortunately, instead of forming a set of compatible components that build upon each others' strengths, many of the most commonly used frameworks partly overlap, thus extending the Web cornucopia to the framework area, making it difficult for developers to pick the right one. Furthermore, probably even more so than with the core browser, the decision regarding which framework to use reflects the current trends and fashion instead of careful consideration of application needs. The oversupply of Web frameworks, as well as the risk of them being abandoned as they get replaced by yet another frameworks, unfortunately diminishes their role as drivers for Web browser API evolution.

6 Conclusion

To conclude, we are surprised how poorly the Web platform meets the consistency and simplicity advocated by software engineering principles. Given the current popularity of the Web browser as a software platform, we are even more surprised how little discussion these issues have generated in the Software Engineering and Web Engineering communities. This discussion raises many questions: Is the Web Platform Cornucopia viable for the Web ecosystem? What is the effort required to maintain a reasonably coherent Web platform in the long term? Are Web developers becoming more or less productive over the years? Is Web Engineering dominated by the frameworks or the browser features (or vendors) underneath them? Will there be a coherent, long-lasting set of key frameworks that are deliberately set to drive browser API evolution? How much time and effort is spent by the developers in rewriting their code to follow the rapid evolution of frameworks, or to port Web applications between frameworks? Will browser vendors give up, leading to a "monoculture" in which only one browser engine remains? While such a monoculture would have a better chance to keep the feature cornucopia under control, would it really be the desirable end state for the long term sustainability of the Web? All these questions ultimately boil down to where innovation is happening: within browsers, within frameworks, in

[15] `https://webvr.info/`
[16] `https://immersive-web.github.io/webxr/`

the applications above (further growing the stack), or outside or beside the Web (making it irrelevant), e.g., in areas such as mobile or pervasive computing. In the Web of Things area HTTP-like protocols have been adopted, but otherwise the adoption of the Web as a platform for running embedded software is still rare. Also, as the Web platform keeps changing, will the users be forced to upgrade their hardware (upgrading browsers may require to upgrade the OS which may make the hardware obsolete) just like it is effectively required in the mobile apps area today)? Has the Web lost its way in the area of backwards and forward compatibility? While the first Web site can still be opened in a browser 30 years later, what is the likelihood that today's Web applications can still run unmodified in future browsers in the late 2040s?

References

1. MacLennan, B.J.: Principles of Programming Languages Design, Evaluation, and Implementation (3rd edition). Oxford University Press (1999)
2. Brooks Jr, F.P.: The Design of Design: Essays from a Computer Scientist. Pearson Education (2010)
3. Taivalsaari, A., Mikkonen, T., Pautasso, C., Systa, K.: Comparing the Built-In Application Architecture Models in the Web Browser. In: 2017 IEEE International Conference on Software Architecture (ICSA), IEEE (2017) 51–54
4. Gallidabino, A., Pautasso, C.: Maturity Model for Liquid Web Architectures. In: International Conference on Web Engineering, Springer (2017) 206–224
5. Taivalsaari, A., Mikkonen, T., Systä, K., Pautasso, C.: Web User Interface Implementation Technologies: An Underview. In: Proceedings of the 14th International Conference on Web Information Systems and Technologies, WEBIST 2018, Seville, Spain, September 18-20, 2018. (2018) 127–136
6. Daubs, M.S.: HTML5, Digital Rights Management (DRM), and the Rhetoric of Openness. Journal of Media Critiques [JMC] **3**(9) (2017)
7. Mikkonen, T., Taivalsaari, A.: Reports of the Web's Death are Greatly Exaggerated. Computer **44**(5) (2011) 30–36
8. Flanagan, D.: JavaScript: The Definitive Guide, 6th edition. O'Reilly Media (2011)
9. Paulson, L.D.: Building Rich Web Applications with Ajax. Computer **38**(10) (2005) 14–17
10. Anthes, G.: HTML5 Leads a Web Revolution. Communications of the ACM **55**(7) (2012) 16–17
11. Fraternali, P., Rossi, G., Sánchez-Figueroa, F.: Rich Internet Applications. IEEE Internet Computing **14**(3) (2010) 9–12
12. Severance, C.: JavaScript: Designing a Language in 10 Days. Computer **45**(2) (2012)
13. Richards, G., Gal, A., Eich, B., Vitek, J.: Automated Construction of JavaScript Benchmarks. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '11, ACM (2011) 677–694
14. Ater, T.: Building Progressive Web Apps: Bringing the Power of Native to the Browser. O'Reilly (2017)
15. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the Web up to Speed with WebAssembly. In: ACM SIGPLAN Notices. Volume 52., ACM (2017) 185–200