The Web as a Software Platform: Ten Years Later

Antero Taivalsaari[1,2] and Tommi Mikkonen[2,3]
[1]*Nokia Technologies, Hatanpään valtatie 30, Tampere, Finland*
[2]*Department of Pervasive Computing, Tampere University of Technology, Korkeakoulunkatu 1, Tampere, Finland*
[3]*Department of Computer Science, University of Helsinki, Gustaf Hällströmin katu 2b, Helsinki, Finland*

Keywords: Web Programming, Web Applications, Live Object Systems, JavaScript, HTML5, Lively Kernel.

Abstract: In the past ten years, the Web has become a dominant deployment environment for new software systems and applications. In view of its current popularity, it is easy to forget that only 10-15 years ago hardly any developer would write serious software applications for the Web. Today, the use of the web browser as a software platform is commonplace, and JavaScript has become one of the most popular programming languages in the world. In this paper we revisit some predictions that were made over ten years ago when the Lively Kernel project was started back in 2006. Ten years later, most of the elements of the original vision have been fulfilled, although not entirely in the fashion we originally envisioned. We look back at the Lively Kernel vision, reflecting our original goals to the state of the art in web programming today.

1 INTRODUCTION

The widespread adoption of the World Wide Web has fundamentally changed the landscape of software development. In the past years, the Web has become the *de facto* deployment environment for new software systems and applications. Office productivity applications and corporate tools such as invoicing, purchasing and expense reporting systems have migrated to the Web. Banking, insurance and retail industries – to name a few – have been transformed profoundly by the emergence of web-based applications and internet services. Academic papers such as this one are now commonly written using collaborative, browser-based environments instead of traditional, installed office suites. Even software development is nowadays often performed using interactive, web-based tools.

Only ten years ago, the world looked very different still. Back in 2006, very few developers would write software for the Web, let alone consider using JavaScript – or any other web technology, for that matter – for writing any serious software applications (Casteleyn et al., 2014). Today, the Software as a Service (SaaS) model (Turner et al., 2003) is prevalent, and interactive, dynamic software development for the Web has become commonplace. In fact, traditional installed applications now maintain a stronghold only in the mobile realm, where the number of mobile apps (especially for iOS and Android devices) has exploded in recent years (Petsas et al., 2013). In contrast, the number of applications that people install on their personal computers has been in steady decline over the past years. The majority of activities on personal computers are now performed using a web browser, leveraging the Software as a Service (SaaS) model (VisionMobile, 2016).

In late 2005, when we started talking about creating an interactive, browser-based programming environment entirely in JavaScript, the idea was met with a lot of contempt at Sun Microsystems where we were working at the time. JavaScript was viewed as a toy language that was suitable for writing scripts no more than few lines long. Furthermore, the idea of using the web browser as a software platform was found highly questionable by many of our colleagues.

The Lively Kernel project (Taivalsaari et al., 2008b; Ingalls et al., 2008) – started at Sun Microsystems Labs back in 2006 – created one of the first fully interactive, self-sustaining, web-based software development environment that was built on the assumption that the web browser would become a credible, full-fledged software platform (http://lively-kernel.org/). While the Lively system is not very widely known or used today, it did pave the way – for its part – for today's Software as a Service based software development systems and live web programming more broadly. A recently published ten-year anniversary paper summarizes the roots, highlights and

Taivalsaari, A. and Mikkonen, T.
The Web as a Software Platform: Ten Years Later.
DOI: 10.5220/0006234800410050
In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 41-50
ISBN: 978-989-758-246-2

evolution of the Lively Kernel from the technical perspective over the past ten years (Ingalls et al., 2016).

While the ten-year anniversary paper looked at Lively mainly from the viewpoint of live object programming, in this paper we take a different angle and look back at the vision presented ten years ago, reflecting our original goals to the state of the art in web programming today. We will also take a look at the evolution of the Web as a software platform and its impact on the software industry. We will then look onwards to the future, highlighting relevant technical areas for future work. This paper is a follow-up to a series of earlier papers in which we have tracked the evolution of the Web as a software platform (Mikkonen and Taivalsaari, 2007; Taivalsaari et al., 2008a; Taivalsaari et al., 2011; Anttonen et al., 2011; Taivalsaari and Mikkonen, 2011).

The structure of this paper is as follows. In Section 2, we will first provide a retrospective on the origins of the Lively Kernel project, including the broader vision behind it. In Section 3, we dive deeper into the original goals of the Lively web platform, followed by some discussion on meeting those goals in Section 4. In Sections 5 and 6, we take a look at the state of the art in web programming today, reflecting the current state to the goals defined ten years ago. In Section 7, we make some predictions and proposals for future work. Finally, Section 8 concludes the paper with some final remarks.

2 RETROSPECTIVE

The roots of the Lively Kernel project can be traced back to early conversations and discussions in 2005 that soon converged on themes related to web programming. It was becoming obvious to us that while the web browser was not designed to be software platform, the browser was nevertheless going to become an important platform, since the browser offered a channel to distribute software globally in an entirely novel, "frictionless" fashion – without the conventional hassles and distribution costs associated with shrink-wrapped or manually installed applications.

While some of our team members had been closely involved in the development of the Java programming language ecosystem for years, it also became clear to us that JavaScript (Flanagan, 2011) – for better or worse – would become the next major programming language. After all, JavaScript was the only programming language that was supported by all the major web browsers. While looking at JavaScript, a language once considered as the neglected little sister of the Java language, we realized that JavaScript actually offered many attractive qualities such as support for first class functions, reflection (albeit in a limited form only) and fully interactive program execution much in the same fashion as Smalltalk systems three decades earlier (Goldberg and Robson, 1983).

In July 2006, we gave a presentation to our management to launch a new initiative on web programming and JavaScript. The key arguments in that presentation were the following:

1. The World Wide Web will be the next major target platform.
2. The Web Browser will effectively be the new operating system.
3. JavaScript is the *de facto* programming language of the Web.

While there is nothing controversial about these statements in 2017, back in 2006 these were still highly questionable claims. Those days, the web browser was regarded only as a tool for viewing web pages, i.e., documents with little interactive content apart from some animated GIF images and Flash advertisements. Similarly, JavaScript and CSS (Cascading Style Sheets) were used primarily as tools to enliven static web content – not as tools for implementing any serious applications. Furthermore, the concepts of *Software as a Service* (SaaS) and *Platform as a Service* (PaaS) were yet to be popularized – Salesforce.com would launch their Force.com SaaS platform in 2007.

Given Sun's stewardship and continued commitment to the Java platform, it was not surprising that our proposal received only lukewarm interest. Nevertheless, in August 2006, we received a permission and some funding to start a project to build the initial prototype and demos. The initial project was quite small, with only four people. The early history of the project – described in the ten-year anniversary paper (Ingalls et al., 2016) – resulted in the first public release of the Lively Kernel on October 1, 2007.

To cut a long story short, the timing of our project turned out to be about six months too late. In early 2007, just as we had built the reasonably feature complete, internal version of the Lively Kernel, it had already been decided that JavaFXScript (https://en.wikipedia.org/wiki/JavaFXScript) was to become the official scripting language for the Java platform. Any projects related to JavaScript were viewed as a distraction to that strategy. Thus, our project was limited to being a research project only; a lot of publications were written and a number of successful external demonstrations were held over the next years. In hindsight, had we been able to show compelling demos about 3-6 months earlier before

some critical decisions were made, the future might have looked different.

3 REVISITING THE ORIGINAL VISION AND GOALS

The vision behind the Lively Kernel project has been presented in a number of papers years ago (Taivalsaari et al., 2008b; Ingalls et al., 2008; Taivalsaari et al., 2008a). Ever since the beginning of the Lively Kernel project (originally known as project Flair and later simply as Lively), we argued that web programming should be fully interactive, very much in the same fashion as Smalltalk systems had already been in the 1970s and 1980s (Goldberg and Robson, 1983), allowing programmers to interact on all the objects directly and fully interactively. We also argued that the web browser could be used as a fully self-contained, self-sustaining programming environment; in other words, the user could accomplish all the programming, debugging and application execution tasks without ever leaving the web browser. In modern web parlance, the entire system is just a big Single-Page Application (SPA) (Mesbah and Van Deursen, 2007; Mikowski and Powell, 2013).

In designing the Lively Kernel, we had the following technical goals (Taivalsaari et al., 2008b):

1. *Self-sufficiency*. The entire system is just a web page. All the development, debugging and application execution tasks can be performed without ever leaving the browser. Applications live in the environment much in the same fashion as applications in Smalltalk systems do (Goldberg and Robson, 1983). (Among other things, this meant that the source code of the system was maintained inside the system itself; apart from a small kernel, there was no conventional source code representation of the system available outside the system.)
2. *Liveness and malleability*. The entire system is designed to be live, interactive and dynamically editable in the same fashion as the Smalltalk systems are. Almost anything in the system can be changed on the fly. All the parts in the backend as well as in the frontend can be programmed interactively without the traditional compile, link, run, crash, debug, and begin-all-over cycle.
3. *Uniform and consistent development and user experience*. A central goal in the design of the Lively Kernel was uniformity. We wanted to build a platform using a minimum number of underlying concepts. Everything in the Lively Kernel is an object, and all the visual objects in the system are *morphs* that can be manipulated in a consistent fashion.
4. *Support for direct manipulation and desktop-style user experience*. When the Lively Kernel effort was started, web browsers still offered a rather clunky user experience that seemed like a throwback to an earlier era predating desktop apps. Our aim was to make the web browser a fully interactive environment supporting direct manipulation much in the same fashion as Smalltalk systems and desktop operating systems were long before the advent of the web browser.

More concisely, the objective was to support highly dynamic, desktop-style applications and application development with rich graphics and direct manipulation capabilities, but without the installation or upgrade hassles that conventional applications had.

In realizing the vision, some key ingredients and principles were borrowed from Smalltalk systems, including the focus (some might even say infatuation) on *uniformity*. Our goal was to build a platform using a minimum number of underlying technologies and concepts. This was in striking contrast with dominant web technologies that utilize a diverse array of technologies such as HTML, CSS, DOM, JavaScript, PHP, XML, and so on. In the Lively Kernel the goal was to do as much as possible using a single technology: JavaScript. We chose JavaScript because of its ubiquitous availability in the web browsers and because of its syntactic resemblance to highly popular languages such as C, C++ and Java (in contrast, Smalltalk had never truly become a mainstream programming language because of its unusual syntax).

In realizing the vision, we leveraged the dynamic aspects of JavaScript, especially the ability to modify applications at runtime. Such capabilities are an essential ingredient in building a malleable web programming environment that allows applications to be developed interactively and collaboratively.

At the implementation level, the Lively Kernel also leveraged Ajax-style asynchronous HTTP network communication (the *XMLHttpRequest* mechanism) that had just become available in major browsers some months earlier. In the absence of asynchronous networking, it would not have been possible to build a truly interactive user experience without blocking the user interface when network operation requests to backend services were being made.

Back in 2006, it still seemed feasible to try to replace the Document Object Model, HTML and CSS with interfaces that were more uniform and amenable to programmatic, desktop-style application development. We were heavily influenced and inspired by the *Morphic* graphics architecture that had been created

over a decade earlier by Randy Smith, John Maloney, Bay-Wei Chang and other talented engineers as part of the Self system (Maloney and Smith, 1995). Morphic was a portable, scene-graph based 2D rendering and composition architecture with built-in affine transformation and matrix functionality that allowed any piece of graphics to be resized, rescaled and rotated programmatically with a rich set of imperative APIs. Our goal was to replace the DOM with something equally powerful, utilizing the (SVG) DOM as the underlying implementation architecture. By replacing the DOM with a JavaScript-based reimplementation of Morphic, the hope was that we could eventually turn the entire World Wide Web into a dramatically more interactive, visual, live construction environment. Such an environment would allow people not just to share documents but also create applications and components collaboratively in a seamless and fully interactive fashion.

The overall Lively Kernel vision is captured well in a videotaped lecture given by Dan Ingalls at JSConf in Scottsdale, Arizona in late 2012[1]. The video summarizes the key Lively features that were familiar from Smalltalk systems of yore. However, to most web developers – and to most mainstream software developers even today – these were and still are rather unusual, unfamiliar features, offering much more flexibility and malleability than average developers are accustomed to.

4 DISCUSSION

To the best of our knowledge, the Lively Kernel was the first system to implement a purely browser-based ”zero-installation” web programming environment with rich, interactive graphics and built-in development and debugging tools. It was rather unique at the time, while drawing a lot of inspiration from the Smalltalk-80 system (Goldberg and Robson, 1983) as well as from the Morphic rendering architecture developed originally for the Self programming language (Maloney and Smith, 1995; Ungar and Smith, 1987). The Lively Kernel preserved the central qualities of the Smalltalk programming environment, and reintroduced them in the context of the Web so that the user would not need any other execution environment than a reasonably compatible web browser.

Perhaps the most unique (and fateful) decision in the original Lively Kernel development was the decision to abandon the web browser's Document Object Model (DOM) as the primary developer-facing rendering architecture. The DOM was – and still is – a rather complex global data structure that holds the runtime document tree representing a web page inside the web browser. Web developers manipulate the contents of their web pages by poking and tweaking the DOM tree from their HTML, CSS and JavaScript code. While the DOM is widely used, it effectively violates several established software engineering principles, exposing the implementation details of the user interface as a large global data structure and as a set of global variables that can potentially conflict with each other if the web application downloads components from multiple sources.

In hindsight, it should have been pretty obvious that by the late 2000s the adoption of the DOM – as well as the ”holy trinity” of HTML, CSS and JavaScript – was already so prevalent and deeply ingrained in web development that attempting to replace those technologies with something different should have been seen as an impossible mission. Furthermore, the immaturity of SVG implementations in web browsers back then – such as the lack of proper font support for text rendering – caused us considerable implementation headaches. In later versions of the Lively system, the rendering architecture was generalized to support other underlying rendering technologies such as HTML and the HTML5 Canvas API. These later steps in the evolution of the system have been summarized in the ten-year anniversary paper (Ingalls et al., 2016).

5 STATE OF THE ART IN WEB PROGRAMMING: TEN YEARS LATER

Let us next take a look at the state of the art in web programming today, reflecting our original objectives and ideas to the present situation in the industry.

The Web and the Software as a Service (SaaS) Model have Redefined Personal Computing. Today, the use of the Web as a software platform and the benefits of the Software as a Service model are widely understood (Turner et al., 2003; Bouzid and Rennyson, 2015). For better or worse, the web browser has become the most commonly used desktop application; often the users no longer open any other applications than just the browser. Effectively, for many average computer users today, the browser *is* the computer.

A recent VisionMobile developer survey report strongly confirmed this observation, proposing the following key trends in year 2016 (VisionMobile,

[1] https://www.youtube.com/watch?v=QTJRwKOFddc

2016):

- The browser has become the default interface for desktop applications.
- If the browser isn't used to run the desktop app, it is being used to distribute it.
- ChromeOS is gaining a foothold in Southern Asia.

Based on the points above, it is fair to say that the Web and the Software as a Service model have redefined the notion of personal computing in the past ten years. Although conventional desktop applications do still exist and are still widely used, desktop applications and their deployment model are now primarily web-based. Perhaps the most representative example of this ongoing paradigm shift is Microsoft's web-based Office 365 productivity suite (https://www.office.com/) that replaces Microsoft's earlier (native) Office suite – the most iconic and prevalent software product of the earlier PC era. This trend has also sparked the introduction of totally new computing device categories, such as Google's purely browser-based Chromebook personal computers (https://www.google.com/chromebook/) running the ChromeOS operating system.

JavaScript has Become a Popular Programming Language. Due to the central role of the web browser, JavaScript has become one of the most popular programming languages in the world, just as we anticipated ten years ago. While JavaScript language standardization work was stalled for many years, there is now major progress on the standards front. The ECMAScript 6 Specification was finally published in June 2015 (ECMAInternational, 2015), followed by ECMAScript 7 a year later (ECMAInternational, 2016). Although the suitability of the JavaScript language for large masses of software developers can be debated, ECMAScript 6 (also known as ECMAScript 2015) is actually a pretty decent and expressive programming language, providing support for features such as modules, class declarations, lexical block scoping, iterators and generators, promises for asynchronous programming, and proper tail calls.

Interactive, Visual Development on the Web has Become Commonplace. From the viewpoint of the original Lively vision, it is interesting to note that interactive, visual development for the Web has become commonplace. There are numerous interactive HTML5 programming environments such as *Cloud9* (https://c9.io/), *Codepen.io* (http://codepen.io/), *Dabblet* (http://dabblet.com/), *JSBin* (https://jsbin.com/), *JSFiddle* (https://jsfiddle.net/), *LiveWeave* (http://liveweave.com/) and *Plunker* (https://plnkr.co/) that capture many of the original qualities of the Lively vision – such as the ability to perform software development entirely within the confines of the web browser.

In addition, there are *web curation systems* (see (Lupfer et al., 2016)) and JavaScript visualization libraries such as *Chart.js* (http://www.chartjs.org/), *Cola.js* (http://marvl.infotech.monash.edu/webcola/), *D3* (https://d3js.org/) and *Vis.js* (http://visjs.org/) that provide rich, interactive, animated 2D and 3D visualizations for the Web, very much in the same fashion as we envisioned when we started the work on Lively back in 2006. A central difference, though, is that these new libraries are intended primarily for data visualization rather than for general-purpose application development.

Web Browser Performance and JavaScript Performance have Improved Dramatically. While the original versions of the Lively Kernel ran slowly, advances in web browsers and high-performance JavaScript engines soon changed the situation dramatically. The emergence of Google's Chrome web browser and the V8 JavaScript engine – created by some of our former colleagues from Sun – kick-started web browser performance wars. Raw JavaScript execution speed increased by three orders of magnitude between years 2006 and 2013, effectively repeating the same dramatic performance advances that had occurred with Java virtual machines ten years earlier when those VMs evolved from simple interpreter-based systems to using advanced adaptive just-in-time compilation techniques. Although improvements in the UI rendering area have been less dramatic, from the end user's perspective today's web browsers are easily 10-20 times faster than ten years ago (Wagner, 2016). This has made it possible to run serious applications in the web browser. (Sadly, this has also enabled much richer use of interactive advertisements on web sites.)

HTML, CSS and the DOM Turned Out to be Much More Persistent than we thought. The browser and JavaScript performance improvements – while definitely impressive – were not really unforeseen to us. We were convinced that the performance problems of the browser and JavaScript would ultimately get resolved. However, what was unforeseen to us how "sticky" the original core technologies in web development – HTML, CSS and JavaScript – as well as the use of the DOM would be. Our assumption was that software developers would prefer having a more uniform, conventional set of imperative graphics APIs – supporting direct, programmatic object manipulation much in the same fashion as in conventional desktop operating systems – instead of using features that were originally designed for document layout rather than for programming.

Furthermore, when we gave presentations in web developers conferences, reminding web developers of traditional software engineering principles such as modularity, separation of concerns and the general importance of keeping specifications and public interfaces separate from implementation details (Parnas, 1972), web developers shrugged and noted that the use of HTML, CSS and JavaScript already gave them the necessary separation. Likewise, the ability to manipulate graphics by poking the global DOM tree from anywhere in the application was seen as a normal way of doing things rather than as something that would raise any concerns.

In recent years, things have gone in a better direction given the earlier mentioned modularity mechanisms that have been added to the ECMAScript language, as well as upcoming support for *Web Components* (https://www.w3.org/TR/#tr_Web_Components).

Web Components bring component-based software engineering principles to the World Wide Web, including the interoperability of higher-level HTML elements, encapsulation, information hiding and the general ability to create reusable, higher-level UI components that can be added flexibly to web applications.

The Worlds of JavaScript and Web Programming are Highly Fragmented. The number of JavaScript libraries and frameworks has grown almost exponentially in the past years. According to the recent estimates, there are now over 1,300 publicly released JavaScript libraries available (https://www.javascripting.com/). Interestingly, there is still very little convergence yet, except for some annually changing trends, with some libraries and frameworks gaining momentum one year, only to lose their momentum to some newer frameworks some time later. For instance, the once dominant *Prototype.js* and *jQuery* libraries are now being forgotten. While *Angular.js* seemed to capture the most developer mindshare only two years ago, it is currently the *React.js* ecosystem that seems to capture the majority of developer attention. Furthermore, the use of JavaScript on the server side, most notably *Node.js* (https://nodejs.org/en/) and its associated NPM ecosystem (consisting of tens of thousands of publicly available components and modules) creates further fragmentation.

JavaScript has Become a Very Popular Language also on the Server Side. As a follow-up to the previous point, JavaScript has become an extremely popular language also on the server side. There is a lot of ongoing innovation in this area, including the current trend towards *isomorphic* applications in which the frontend and backend functionality are both written in JavaScript, and may even share the same code (http://isomorphic.net/). Server-side JavaScript development is a very broad and fascinating topic; however, it falls outside our original scope and thus we will not dive deeper into it in this paper.

Mobile Computing is Still Dominated by Apps – for now. During the original development of the Lively Kernel, we were aiming at making the system run well also on mobile devices. Although the feasibility of running the system on mobile devices was demonstrated, in practice mobile devices and browsers were still so slow those days that no serious mobile Lively applications could be built. Furthermore, the considerably smaller screen sizes and different input modalities made it difficult to run desktop applications on mobile devices.

The technical reasons for the desktop and mobile app divergence are well understood nowadays (Charland and Leroux, 2011; Joorabchi et al., 2013). One approach for tackling the shortcomings of the Web as a mobile platform is to use cross-platform or hybrid app designs (Dalmasso et al., 2013; Casteleyn et al., 2014). In the late 2000s, so called *Rich Internet Application* (RIA) platforms such as Adobe AIR, Apache Cordova (Wargo, 2015) (formerly PhoneGap) and Microsoft Silverlight (Moroney, 2010) were very popular. RIA systems were an attempt to bring alternative programming languages and libraries to the Web in the form of browser plug-in components that each provided a complete, more efficient platform runtime (see (Casteleyn et al., 2014)). However, just as it was predicted in (Taivalsaari and Mikkonen, 2011), the RIA phenomenon turned out to be rather short-lived.

More broadly, it is interesting to note that in the past ten years desktop computing and mobile computing have evolved in entirely different directions. While personal computers are now driven mostly by the Software as a Service model, mobile devices are still dominated by native or hybrid apps. This divergence is unlikely to continue indefinitely. There are already indications that desktop and mobile operating systems will ultimately converge. For instance, Microsoft's latest Windows 10 Mobile operating system represents an attempt to unify Windows application platform across multiple device classes.

The convergence between mobile and desktop platforms will be driven by several factors, including the increasingly powerful CPUs in mobile devices, blurring lines between different types of computing devices (phones, "phablets", tablets, tablets with detachable keyboards, ultraportable laptops, and so on), the growing number of devices that the average com-

puter users have in their daily lives, and the subsequent need for computing environments in which applications and data stay automatically in sync between all the devices (Levin, 2014). In such multi-device environments, the users will expect a more seamless, *liquid* software experience that allows the users to pick the most applicable device and then effortlessly move to another device (e.g., with bigger screen or larger keyboard) to continue the current activities. A recently published a *Liquid Software Manifesto* to summarize predictions and expectations in this area (Taivalsaari et al., 2014). Microsoft's *Continuum* (Microsoft Corporation,) and Apple's *Handoff/Continuity* functionality (Gruman, 2014) already represent an early manifestation of such functionality.

Instant Worldwide Deployment and Dramatically Faster Release Cycles have Become Commonplace. When the Lively Kernel project was started, the majority of software deployments at Sun were still done in a conventional fashion by distributing physical CDs/DVDs or by making new binaries available on the Web. New software releases occurred relatively infrequently, perhaps a few times per year for major software products such as the Java SDK. In contrast, web-based systems allow changes to be published pretty much instantly worldwide.

Since the Lively Kernel was one of the first systems to boldly enter such an instant deployment model, we had no support from tools and techniques that have later been introduced in the context of continuous deployment (Leppänen et al., 2015). Instead, all such complications had to be handled as a part of the manual development process.

In hindsight, it is amazing how quickly the traditional deployment model was replaced by instant worldwide deployment enabled by the Software as a Service model. This has resulted in dramatically faster release cycles as well as to the rise of entirely new continuous development and deployment practices methodologies across the industry, including DevOps (Debois, 2011). These topics are now so widely studied and documented that we do not need to dive more deeply into them in this paper.

6 STATE OF THE ART IN WEB PROGRAMMING: REMAINING TECHNICAL CHALLENGES

The technical challenges associated with web programming are still largely the same as ten years ago. Below we list some of the key challenges.

Limited Access to Local Resources or Host Platform Capabilities. Web documents and scripts are run in a sandbox that places restrictions on the resources and host platform capabilities that the web browser can access. For instance, access to local files on the machine in which the web browser is being run is not allowed, apart from reading and writing cookies and using localStorage. While these security restrictions prevent malicious access, they make it difficult to build web applications that utilize local resources or host platform capabilities. Consequently, the functionality that can be offered by web applications is inevitably more limited than that of native applications.

Completeness of Applications is Difficult to Determine. Web applications are generally so dynamic that it is impossible to know statically – ahead of application execution – if all the structures that the program depends on will be available at runtime. While web browsers are designed to be error-tolerant and will ignore incomplete or missing elements, in some cases the absence of elements can lead to fatal runtime problems that are impossible to detect before execution. Furthermore, with scripting languages such as JavaScript applications can even modify themselves on the fly, and there is no way to statically detect the possible errors resulting from such modifications. Consequently, web applications require significantly more testing to make sure that all the possible application behaviors and paths of execution are covered.

Fine-grained Security Model is Missing. A key point in all the limitations related to networking and security is the need for a more fine-grained security model for web applications. On the Web, applications are still second-class citizens that are at the mercy of the classic, one size fits all sandbox security model of the web browser. This means that decisions about security are determined primarily by the site (origin) from which the application is loaded, and not by the specific needs of the application itself.

Incompatible Browser Implementations; Lack and Disregard of Standards; Cornucopia of Overlapping Features and Standards. Just like ten years ago, a central problem in web application development is browser incompatibility. This is partly due to the rapid pace in specifying new browser features and the gradual emergence of such features in actual browser releases. However, there are also legitimate business reasons for many of the incompatibilities, arising from intellectual property rights issues, e.g., in the media codec area. While there has been tremendous progress in improving basic web browser compatibility, overall the situation is still much different from, e.g., Java development where comprehensive compatibility toolkits were in place early on to ensure compatibility and standards compliance.

At the same time, there are too many competing standards that partially overlap each other. For instance, in the area of graphics rendering, a standards-compatible web browser offers at least five built-in development and rendering models. These standards include the dominant Document Object Model (DOM) rendering architecture. They also include the *Canvas 2D Context API* (also known as the Canvas API, https://www.w3.org/TR/2dcontext/) as well as *WebGL* (http://www.khronos.org/webgl/). Additionally, there are important technologies such as *Scalable Vector Graphics* (SVG) and *Web Components* that complement the basic DOM architecture. On top of these basic rendering technologies, an extremely rich library and tool ecosystem has emerged, providing a cornucopia of choices for the developer.

More broadly, in striking contrast with the situation ten years ago, there is now an incredible amount of innovation in the web development area. New libraries and tools have become available almost on a weekly (if not daily) basis (see, e.g., http://www.javascripting.com/). The rapid pace of innovation and rather uncontrolled, organic evolution of the Web have resulted in a situation in which there are numerous ways to build applications on the Web – many more than most people realize, and also arguably more than are really needed. This has put the developers in a complex position in which it is difficult to choose technologies that would be guaranteed to still be around and supported ten years from now.

These things said, there are also many positive developments in the compatibility area. As already mentioned, there has finally been tremendous progress in ECMAScript (JavaScript) language standardization (ECMAInternational, 2015; ECMAInternational, 2016). Furthermore, newer browsers – in particular Microsoft's Edge browser (https://www.microsoft.com/en-us/windows/microsoft-edge) that has replaced Internet Explorer – are significantly more compatible with each other than dominant browsers ten years ago. We are confident that similar compatibility improvements will find eventually their way also to mobile web browsers that still have more significant feature deviations today (see http://mobilehtml5.org/ for an overview).

In the same vein, aforementioned Web Components offer hope that well-known (but hitherto missing) software engineering principles and practices will eventually find their way into the web browser, including modularity and the ability to create higher-level, general-purpose UI components that can be flexibly added to web applications. Web components are still the "dark horse" in web development – they are little known to most developers, and it is difficult to place betting odds on their eventual success.

7 LOOKING FORWARD

The field of web programming today still bears the imprint of the document-oriented – as opposed to application-oriented – roots of the Web. The programming capabilities of the Web have largely been an afterthought – designed originally by non-programmers for relatively simple scripting tasks.

While the adoption of Single-Page Application (SPA) development style (Mesbah and Van Deursen, 2007; Mikowski and Powell, 2013) and its support in popular frameworks such as Angular (Jadhav et al., 2015) have improved the overall user experience especially for those web sites that want to behave more like classic desktop applications, we think that the characteristics of typical web applications today still do not reflect the true potential of the Web as an application platform. Furthermore, while social media features have made the use of the Web generally more interactive and collaborative, the general user experience is still strongly reminiscent of the strictly page-based *back-forward-reload* metaphor introduced by the NCSA Mosaic browser back in the early 1990s (Darken, 1998).

In many ways, the original Lively system already was much more interactive, dynamic and collaborative than an average web application today. However, at this point we are clearly past the point where the foundations of the Web could be altered significantly. Thus, instead of attempting to "fix" the Web, we have shifted our attention to new areas in which dynamic programming capabilities will likely play a central role in the future.

Looking forward, we predict that the current transition towards the Internet of Things (IoT) and the Web of Things (WoT) will drive the industry towards systems that have much better support for interactive development and programming. We are moving to the *Programmable World Era* in which literally all everyday objects will be connected to the Internet and will have enough computing, storage and networking capabilities to host a dynamic programming environment, thus turning everyday objects remotely programmable (Wasik, 2013; Taivalsaari and Mikkonen, 2017).

For better or worse, everyday objects around us will have more computing power, storage capacity and network bandwidth than computers that were used for running Smalltalk systems in the 1970s and 1980s. The availability and presence of such capa-

bilities will open up tremendous possibilities for entirely new types of applications and services. Many of the platforms under development for the IoT domain leverage Node.js, which effectively means that JavaScript may well become the *de facto* programming language for IoT applications as well.

The Internet of Things area offers a natural playground for dynamic programming capabilities provided by systems such as the Lively Kernel. To this end, we plan to harness and leverage the Lively environment as a web-based graphical end-user programming environment for IoT devices, with the goal to realize the broader Programmable World vision by implementing the same kind of direct manipulation capabilities that demonstrated earlier. The key difference is that rather than just making the World Wide Web more lively, we now aim at making the entire world around us programmable in an effortless and lively fashion (”Lively Things”).

8 CONCLUSIONS

The Lively Kernel project created one of the first fully interactive, self-sustaining web-based software development environments that was built on the assumption that the web browser would one day become a credible, full-fledged software platform. Today – over ten years after the inception of the Lively Kernel project – most of the elements of the original Lively vision have been fulfilled, although not entirely in a fashion we originally envisioned.

In this paper, we have revisited the Lively vision, reflecting the original goals to the state of the art in web programming today. The emergence of the web browser as an application platform has inspired numerous systems to take advantage of the features and capabilities that we embraced when starting the project. While the Lively Kernel itself did not become very widely known or used, it did pave the way – for its part – for today’s Software as a Service based software development systems and dynamic web programming more broadly.

We believe that the interactive, web-based development capabilities will become even more important in the future as the industry moves towards the *Programmable World Era* in which everyday objects around us will become connected and programmable.

ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland (project 295913).

REFERENCES

Anttonen, M., Salminen, A., Mikkonen, T., and Taivalsaari, A. (2011). Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC’2011, TaiChung, Taiwan, March 21-25, 2011), ACM Press, proceedings vol 1*, pages 800–807.

Bouzid, A. and Rennyson, D. (2015). *The Art of SaaS: A Primer on the Fundamentals of Building and Running a Successful SaaS Business*. Xlibris.

Casteleyn, S., Garrigós, I., and Mazón, J.-N. (2014). Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. *ACM Trans. Web*, 8(3):18:1–18:46.

Charland, A. and Leroux, B. (2011). Mobile Application Development: Web vs. Native. *Communications of the ACM*, 54(5):49–53.

Dalmasso, I., Datta, S. K., Bonnet, C., and Nikaein, N. (2013). Survey, Comparison and Evaluation of Cross Platform Mobile Application Development Tools. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 323–328. IEEE.

Darken, R. (1998). Breaking the Mosaic Mold. *IEEE Internet Computing*, 2(3):97.

Debois, P. (2011). Devops: A Software Revolution in the Making. *Journal of Information Technology Management*, 24(8):3–39.

ECMAInternational (2015). ECMAScript 2015 Language Specification, Standard ECMA-262, 6th Edition, June 2015. http://www.ecma-international.org/ecma-262/6.0/. [Online; accessed 22-Feb-2017].

ECMAInternational (2016). ECMAScript 2016 Language Specification, Standard ECMA-262, 7th Edition, June 2016. http://www.ecma-international.org/ecma-262/7.0/. [Online; accessed 22-Feb-2017].

Flanagan, D. (2011). *JavaScript: The Definitive Guide, 6th edition*. O’Reilly Media.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

Gruman, G. (Oct. 7, 2014). Apple’s Handoff: What Works, and What Doesn’t. *InfoWorld*.

Ingalls, D., Felgentreff, T., Hirschfeld, R., Krahn, R., Lincke, J., Röder, M., Taivalsaari, A., and Mikkonen, T. (2016). A World of Active Objects for Work and Play: The First Ten Years of Lively. In *Proceedings of SPLASH’2016 Onward! Track (Amsterdam, the Netherlands, October 30 - November 4, 2016)*, pages 238–249.

Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., and Mikkonen, T. (2008). The Lively Kernel – a Self-Supporting System on a Web Page. In *Self-Sustaining Systems (S3’2008, Potsdam, Germany, May 15-16, 2008), Lecture Notes in Computer Science LNCS5146*, pages 31–50. Springer-Verlag.

Jadhav, M. A., Sawant, B. R., and Deshmukh, A. (2015). Single Page Application using AngularJS. *International Journal of Computer Science and Information Technologies*, 6(3).

Joorabchi, M. E., Mesbah, A., and Kruchten, P. (2013). Real Challenges in Mobile App Development. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24. IEEE.

Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V., and Männistö, T. (2015). The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32(2):64–72.

Levin, M. (2014). *Designing Multi-Device Experiences: An Ecosystem Approach to User Experiences Across Devices*. O'Reilly Media, Inc.

Lupfer, N., Kerne, A., Webb, A. M., and Linder, R. (2016). Patterns of Free-form Curation: Visual Thinking with Web Content. In *Proceedings of the 2016 ACM on Multimedia Conference (MM'16, Amsterdam, The Netherlands, October 15-19, 2016)*, pages 12–21.

Maloney, J. and Smith, R. (1995). Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of UIST'95*, pages 21–28.

Mesbah, A. and Van Deursen, A. (2007). Migrating Multi-Page Web Applications to Single-Page Ajax Interfaces. In *11th European Conference on Software Maintenance and Reengineering CSMR'07*, pages 181–190. IEEE.

Microsoft Corporation. Continuum. http://www.windowscentral.com/continuum.

Mikkonen, T. and Taivalsaari, A. (2007). Web Applications: Spaghetti Code for the 21st Century. *Technical Report TR-2007-166, Sun Microsystems Labs, June 2007*.

Mikowski, M. S. and Powell, J. C. (2013). *Single Page Web Applications: JavaScript End-to-End*. Manning.

Moroney, L. (2010). *Microsoft Silverlight 4 Step by Step*. Microsoft Press.

Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058.

Petsas, T., Papadogiannakis, A., Polychronakis, M., Markatos, E. P., and Karagiannis, T. (2013). Rise of the Planet of the Apps: A Systematic Study of the Mobile App Ecosystem. In *Proceedings of the 2013 Internet Measurement Conference*, pages 277–290. ACM.

Taivalsaari, A. and Mikkonen, T. (2011). The Web as an Application Platform: The Saga Continues. In *37th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'2011, Oulu, Finland, August 30 - September 2, 2011)*, pages 170–174. IEEE Computer Society.

Taivalsaari, A. and Mikkonen, T. (2017). Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software, Jan/Feb 2017*, 34(1):72–80.

Taivalsaari, A., Mikkonen, T., Anttonen, M., and Salminen, A. (2011). The Death of Binary Software: End User Software Moves to the Web. In *Proceedings of the 9th International Conference on Creating, Connecting and Collaborating through Computing (C5'2011, Kyoto, Japan, January 18-20, 2011)*, pages 17–23. IEEE Computer Society.

Taivalsaari, A., Mikkonen, T., Ingalls, D., and Palacz, K. (2008a). Web Browser as an Application Platform. In *34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'2008, Parma, Italy, September 3-5, 2008)*, pages 293–302. IEEE Computer Society.

Taivalsaari, A., Mikkonen, T., Ingalls, D., and Palacz, K. (2008b). Web Browser as an Application Platform: The Lively Kernel Experience. Technical report, Technical Report TR-2008-175, Sun Microsystems Laboratories.

Taivalsaari, A., Mikkonen, T., and Systä, K. (2014). Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture. In *Proceedings of COMPSAC'2014*.

Turner, M., Budgen, D., and Brereton, P. (2003). Turning Software into a Service. *Computer*, 36(10):38–44.

Ungar, D. and Smith, R. (1987). Self: The Power of Simplicity. In *Proceedings of OOPSLA'87*, pages 227–241.

VisionMobile (2016). Cloud and Desktop Developer Landscape. http://www.visionmobile.com/product/cloud-and-desktop-developer-landscape/. [Online; accessed 5-March-2016].

Wagner, J. L. (2016). *Web Performance in Action: Building Fast Web Pages*. Manning.

Wargo, J. M. (2015). *Apache Cordova 4 Programming*. Pearson Education.

Wasik, B. (2013). In the Programmable World, All Our Objects Will Act as One. *Wired (May 2013)*, page 462.