# Client-Side Cornucopia : Comparing the Built-In Application Architecture Models in the Web Browser

Taivalsaari, Antero

Springer Nature Switzerland
2019

Taivalsaari , A , Mikkonen , T , Pautasso , C & Systä , K 2019 , Client-Side Cornucopia : Comparing the Built-In Application Architecture Models in the Web Browser . in M J Escalona , F D Mayo , T A Majchrzak & V Monfort (eds) , Web Information Systems and Technologies : 14th International Conference, WEBIST 2018, Seville, Spain, September þÿ1 8 2 0 , 2 0 1 8 , R e v i s e d S e l e c t e d P a p e r s . L e c t u r e N o t e s i n B u s i n e s s I n f o vol. 372 , Springer Nature Switzerland , Cham , pp. 1-24 , Web Information Systems and Technologies , Sevilla , Spain , 18/09/2018 . https://doi.org/10.1007/978-3-030-35330-8_1

http://hdl.handle.net/10138/324750
https://doi.org/10.1007/978-3-030-35330-8_1

acceptedVersion

# Client-Side Cornucopia: Comparing the Built-In Application Architecture Models in the Web Browser

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

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

**Abstract.** The programming capabilities of the Web can be viewed as an afterthought, designed originally by non-programmers for relatively simple scripting tasks. This has resulted in cornucopia of partially overlapping options for building applications. Depending on one's viewpoint, a generic standards-compatible web browser supports three, four or five built-in application rendering and programming models. In this paper, we give an overview and comparison of these built-in client-side web application architectures in light of the established software engineering principles. We also reflect on our earlier work in this area, and provide an expanded discussion of the current situation. In conclusion, while the dominance of the base HTML/CSS/JS technologies cannot be ignored, we expect Web Components and WebGL to gain more popularity as the world moves towards increasingly complex web applications, including systems supporting virtual and augmented reality.
***Keywords***—Web programming, single page web applications, web components, web application architectures, rendering engines, web rendering, web browser

## 1 Introduction

The World Wide Web has become such an integral part of our lives that it is often forgotten that the Web has existed only about thirty years. The original design sketches related to the World Wide Web date back to the late 1980s. The first web browser prototype for the NeXT computer was completed by Tim Berners-Lee in December 1990. The first version of the Mosaic web browser was made available publicly in February 1993, and the first commercially successful browser – Netscape Navigator – was released in late 1994. Widespread commercial use of the Web took off in the late 1990s.

In the end of the 1990s and in the early 2000s, the web browser became the most commonly used computer program, sparking a revolution that has transformed not only commerce but communication, social life and politics as

well. In desktop computers, nearly all the important tasks are now performed using the web browser. Even mobile applications today can be viewed merely as "mirrors into the cloud". While native mobile apps may still offer UI frameworks and widgets that are (at least for now) better suited to the limited screen size and input modalities of the devices, valuable content has moved gradually away from mobile devices to cloud-based services, thus reducing the original role of the mobile apps considerably.

Interestingly, the programming capabilities of the Web have largely been an afterthought – designed originally by non-programmers for relatively simple scripting tasks. Due to different needs and motivations, there are many ways to make things on the Web – many more than people generally realize. Furthermore, over the years these features have evolved in a rather haphazard fashion. Consequently, there are various ways to build applications on the Web – even without considering any extensions or thousands of add-on libraries. Depending on one's viewpoint, the web browser natively supports three, four or five different built-in application rendering and development models. Thousands of libraries and frameworks have then been implemented on top of these built-in models. Furthermore, in addition to application architectures that partition applications more coarsely into server and client side components, it is increasingly possible to fine-tune the application logic by moving code flexibly between the client and the server, as originally noted in [15].

Even though a lot of the application logic in web applications may run on the server, the rendering capabilities of the web browser are crucial in creating the presentation layer of the applications. In this paper, we provide a comparison of the *built-in client-side web application architectures*, i.e., the programming capabilities that the web browsers provide out-of-the-box before any additional libraries are loaded. This is a topic that has received surprisingly little attention in the literature. While there are countless articles on specific web development technologies, and thousands of libraries have been developed *on top of the browser*, there are few if any papers comparing the built-in user interface development models offered by the browser itself. The choice between these alternative development models has a significant impact on the overall architecture and structure of the resulting web applications. The choices are made more difficult by the fact that the web browser offers a number of overlapping features to accomplish even basic tasks, reflecting the historical, organic evolution of the web browser as an application platform.

This paper is motivated by the *recent trend toward simpler, more basic approaches in web development*. According to a study carried out a few years ago, the vast majority (up to 86%) of web developers felt that the Web and JavaScript ecosystems have become far too complex (`http://stateofjs.com/2016`). There is a movement to go back to the roots of web application development by building directly upon what the web browser can provide without the added layers introduced by various libraries and frameworks. The "*zero framework manifesto*" crystallizes this desire for simplicity [3]. However, as will be shown in this paper, even the "vanilla" browser offers a cornucopia of choices when it comes to

application development. The paper is based on our earlier articles [29,27], and it has been extended from those to provide a more comprehensive view to web development in general as well as deeper technical discussion on the implications.

The structure of this paper is as follows. In Section 2, we provide an overview on the evolution of the web browser as an application platform. In Section 3, we dive into the built-in user interface development and rendering models offered by modern web browsers: (1) DOM, (2) Canvas, (3) WebGL, (4) SVG, and (5) Web Components. In Section 4, we provide a comparison and presumed use cases of the presented technologies, and in Section 5, we list some broader considerations and observations. In Section 6, we revisit our earlier predictions made in [29,27], followed by avenues for future work in Section 7. Finally, Section 8 concludes the paper with some final remarks.

## 2 Evolution of the Web Browser as an Application Platform

The history of computing and the software industry is characterized by disruptive periods and paradigm shifts that have typically occurred every 10-15 years. Back in the 1960s and 1970s, mainframe computers gave way to minicomputers. In the early 1980s personal computers sparked a revolution, making computers affordable to ordinary people, and ultimately killing a number of very successful minicomputer manufacturers such as Digital Equipment Corporation as a side effect. In the 1990s, the emergence of the World Wide Web transformed personal computers from standalone computing "islands" to network-connected web terminals. In the early 2000s, mobile phones were opened up for third party application development as well. Today, the dominant computing environment clearly is the Web, with native apps complementing it in various ways especially in the mobile domain [17,18].

Over time, the World Wide Web has evolved from its humble origins as a *document sharing system* to a massively popular hypermedia application and content distribution environment – in short, the most powerful information dissemination environment in the history of humankind. This evolution has not taken place in a fortnight; it has not followed a carefully designed master plan either. Although the World Wide Web Consortium (W3C) has seemingly been in charge of the evolution of the Web, in practice the evolution has been driven largely by dominant web browser vendors: Mozilla, Microsoft, Apple, Google and (to a lesser degree) Opera. Over the years, these companies have had divergent, often misaligned business interests. While browser compatibility has improved dramatically in recent years, the browser landscape is still truly a mosaic or cornucopia of features, reflecting organic evolution – or a tug of war if you will – between different commercial vendors over time.

Before delving into more technical topics, let us briefly revisit the evolution of the web browser as a software platform [26,24,1].

**Classic Web**. In the early life of the Web, web pages were truly *pages*, i.e., page-structured documents that contained primarily text with interspersed

images, without animation or any interactive content. Navigation between pages was based on simple *hyperlinks*, and a new web page was loaded from the web server each time the user clicked on a link. There was no need for asynchronous network communication between the browser and the web server. For reading used input some pages were presented as *forms*, with simple textual fields and the possibility to use basic widgets such as buttons and combo (selection) boxes. These types of "classic web" pages were characteristic of the early life of the Web in the early 1990s.

**Hybrid Web**. In the second phase, web pages became increasingly interactive. Web pages started containing animated graphics and plug-in components that allowed richer, more interactive content to be displayed. This phase coincided with the commercial takeoff of the Web during the dot-com boom of the late 1990s when companies realized that they could create commercially valuable web sites by displaying advertisements or by selling merchandise and services over the Web. Plug-in components such as Flash, RealPlayer, Quicktime and Shockwave were introduced to make it possible to construct web pages with visually enticing, interactive multimedia, allowing advanced animations, movie clips and audio tracks to be inserted in web pages.

With the introduction of *DHTML* – the combination of HTML, Cascading Style Sheets (CSS), the JavaScript language [6], and the Document Object Model (DOM) – it became possible to create interactive web pages with built-in support for more advanced graphics and animation. The JavaScript language, introduced in Netscape Navigator version 2.0B almost as an afterthought in December 1995, made it possible to build animated interactive content by scripting directly the web browser.

In this phase, the Web started moving in directions that were unforeseen by its original designer, with web sites behaving more like multimedia presentations rather than static pages. Content mashups and web site cross-linking became popular and communication protocols between the browser and the server became increasingly advanced. Navigation was no longer based solely on hyperlinks. For instance, Flash apps supported drag-and-drop and direct clicking/events on various types of objects, whereas originally no support for such features existed in browsers.

**The Web as an Application Platform**. In the early 2000s, the concept of Software as a Service (SaaS) emerged. Salesforce.com pioneered the use of the Web as a CRM application platform in the early 2000s, demonstrating and validating the use of the Web and the web browser as a viable target platform for business applications. At that point, people realized that the ability to offer software applications seamlessly over the Web and then perform instant worldwide software updates could offer unsurpassed business benefits.

As a result of these observed benefits, people started to build web sites that behave much like desktop applications, for example, by allowing web pages to be updated partially, rather than requiring the entire page to be refreshed. Such systems often eschewed link-based navigation and utilized direct manipulation techniques (e.g., drag and drop features) borrowed from desktop-style applica-

tions instead. Interest in the use of the browser as an application platform was reinforced by the introduction of Ajax (Asynchronous JavaScript and XML) [8]. The key idea in Ajax was to use *asynchronous network communication* between the client and the server to decouple user interface updates from network requests. This made it possible to build web sites that do not necessarily block when interacting with the server and thus behave much like desktop applications, for example, by allowing web pages to be updated asynchronously one user interface element at a time, rather than requiring the entire page to be updated each and every time something changed. Although Ajax was primarily a specific technique rather than a complete development model or platform, it fueled further interest in building "Web 2.0" applications that could run in a standard web browser. 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 any external plug-in components.

After the introduction of Ajax and the concept of *Single Page Applications* (SPAs) [12], the number of web development frameworks on top of the web browser has exploded. Today, there are over 1,400 officially listed JavaScript libraries (see `http://www.javascripting.com/`).

**Server-Side JavaScript**. The use of client-side web development technologies has spread also to other domains. For instance, after the introduction of the V8 high-performance JavaScript engine (`https://developers.google.com/v8/`), the use of the JavaScript language has quickly spread into server-side development as well. As a result, Node.js (`https://nodejs.org/`) has become a vast ecosystem of its own; according to a popular saying, there is an "NPM module for nearly everything". In fact, the NPM (Node Package Manager) ecosystem has been growing even faster in recent years than the client-side JavaScript ecosystem. According to `npmjs.com`, there are more than 800,000 NPM packages at the time of this writing.

As already mentioned earlier, in this paper we shall focus only on client-side technologies and only on those technologies that have been included natively in standards-compatible web browsers. We feel that this is an area that is surprisingly poorly covered by existing research.

**Non-Standard Development Models and Architectures**. For the sake of completeness, it should be mentioned that over the years web browsers have supported various additional client-side rendering and development models. For instance, Java applets were an early attempt to include Java language and Java virtual machine (JVM) support directly in a web browser. However, because of the immaturity of the technology (e.g., inadequate performance of early JVMs) and Microsoft's vigilant resistance, applets never became an officially supported browser feature. For years, Microsoft had their alternative technologies, such as ActiveX, available (only) in Microsoft Internet Explorer. For a while, the use of various browser plug-in components offering application execution capabilities – such as Adobe Flash or Shockwave – was extremely popular.

In the late 2000s, so called *Rich Internet Application* (RIA) platforms such as Adobe AIR or Microsoft Silverlight were very much in vogue. RIA systems

were an attempt to reintroduce alternative programming languages and libraries in the context of the Web in the form of browser plug-in components that each provided a complete platform runtime. For a comprehensive overview of RIA systems, refer to Castelyn's survey [4]. However, just as it was predicted in [24], the RIA phenomenon turned out to be rather short-lived. The same seems to be true also of various attempts to support native code execution directly from within the web browser. For instance, Google's Native Client offers a sandbox for running compiled C and C++ code in the browser, but it has not become very popular. Mozilla's classic NPAPI (Netscape Plugin Application Programming Interface) – introduced originally by Netscape in 1995 – has effectively been removed from all the major browsers; for instance, Google Chrome stopped supporting it already in 2015. Although there are some interesting ongoing efforts in this area – such as the W3C WebAssembly effort (`http://webassembly.org/`), it is now increasingly difficult to extend the programming capabilities of the web browser without modifying the source code of the browser itself (and thus creating non-standard, custom browsers).

## 3 Client-Side Web Rendering Architectures – An Underview

As summarized above, the history of the Web has undergone a number of evolutionary phases, reflecting the *document-oriented* – as opposed to *application-oriented* – origins of the Web. Nearly all the application development capabilities of the Web have been an afterthought, and have emerged as a result of divergent technical needs and business interests instead of careful planning and coordination.

As a result of the browser evolution that has occurred in the past two decades, today's web browsers support a mishmash of complementary, partially overlapping rendering and development models. These include the dominant "holy trinity" of HTML, CSS and JavaScript, and its underlying Document Object Model (DOM) rendering architecture. They also include the *Canvas 2D Context API* as well as *WebGL*. Additionally, there are important technologies such as *Scalable Vector Graphics* (SVG) and *Web Components* that complement the basic DOM architecture.

The choice between the rendering architectures can have significant implications on the structure of client-side web applications. Effectively, all of the technologies mentioned above introduce their own distinct programming models and approaches that the developers are expected to use. Furthermore, all of them have varying levels of framework, library and tool support available to simplify the actual application development work on top of the underlying development model. The DOM-based approach is by far the most popular and most deeply ingrained, but the other technologies deserve a fair glimpse as well.

Below we will dive more deeply into each technology. We will start with the DOM, Canvas and WebGL models, because these three technologies can be regarded more distinctly as three separate technologies. We will then dive into

SVG and Web Components, which introduce their own programming models but which are closely coupled with the underlying DOM architecture at the implementation level.

### 3.1 DOM / DHTML

In web parlance, the *Document Object Model* (DOM) is a platform-neutral API that allows programs and scripts to dynamically access and update the content, structure and style of web documents. Document Object Model in the foundation for *Dynamic HTML* – the combination of HTML, Cascading Style Sheets (CSS) and JavaScript – that allows web documents to be created and manipulated using a combination of declarative and imperative development styles. Logically, the DOM can be viewed as an *attribute tree* that represents the contents of the web page that is currently displayed by the web browser. Programmatic interfaces are provided for manipulating the contents of the DOM tree from HTML, CSS and JavaScript.

In the web browser, the DOM serves as the foundation for a *retained (automatically managed) graphics architecture*. In such a system, the application developer has no direct, immediate control over rendering. Rather, all the drawing is performed indirectly by manipulating the DOM tree by adding, removing and modifying its nodes; the browser will then decide how to optimally lay out and render the display after each change.

Over the years, the capabilities of the DOM have evolved significantly. The evolution of the DOM has been described in a number of sources, including Flanagan's JavaScript "bible" [6]. In this paper we will not go into details, but it is useful to provide a summary since this evolution partially explains why the browser offers such a cornucopia of overlapping functionality.

- *DOM Level 1* specification – published in 1998 – defines the core HTML (and XML) document models. It specifies the basic functionality for document navigation.
- *DOM Level 2* specification – published in 2000 – defines the stylesheet object model, and provides methods for manipulating the style information attached to a document. It also enables traversals on the document and provides support for XML namespaces. Furthermore, it defines the *event model* for web documents, including the event listener and event flow, capturing, bubbling, and cancellation functionality.
- *DOM Level 3* specification – released as a number of separate documents in 2001-2004 – defines document loading and saving capabilities, as well as provides document validation support. In addition, it also addresses document views and formatting, and specifies the keyboard events and event groups, and how to handle them.
- *DOM Level 4* specification refers to a "living document" that is kept up to date with the latest decisions of the WHATWG/DOM working group[1].

[1] https://dom.spec.whatwg.org/

DOM attributes can be manipulated from HTML, CSS, JavaScript, and to some extent also XML code. As a result, a number of entirely different development styles are possible, ranging from purely imperative usage to a combination of declarative styles using HTML and CSS. For instance, it is possible to create impressive 2D/3D animations using the CSS animation capabilities without writing a single line of imperative JavaScript code.

Below is a "classic" DHTML example that defines a text paragraph and an input button in HTML. The input button definition includes an `onclick` event handler function that – when clicked – hides the text paragraph by changing its visibility style attribute to '`hidden`'.

```
<!DOCTYPE html>
<html><body>
<p id="text">This is a piece of text.</p>

<input type="button" value="Hide text"
onclick="document.getElementById('text').style.visibility='
    hidden'">

</body></html>
```

In practice, very few developers use the raw, low-level DOM interfaces directly nowadays. The DOM and DHTML serve as the foundation for an extremely rich library and tool ecosystem that has emerged on top of the base technologies. The manipulation of DOM attributes is usually performed using higher-level convenience functions provided by popular JavaScript / CSS libraries and frameworks.

### 3.2 Canvas

The Canvas (officially known as the *Canvas 2D Context API*) is an HTML5 feature that enables dynamic, scriptable rendering of two-dimensional (2D) shapes and bitmap images (`https://www.w3.org/TR/2dcontext/`). It is a low level, imperative API that does not provide any built-in scene graph or advanced event handling capabilities. It that regard, Canvas offers much lower level graphics support than the DOM or SVG APIs that will automatically manage and (re)render complex graphics elements.

Canvas objects are drawn in *immediate mode*. This means that once a shape such as a rectangle is drawn using Canvas API calls, the rectangle is immediately forgotten by the system. If the position of the rectangle needs to be changed, the entire scene needs to be repainted, including any objects that might have been invalidated (covered) by the rectangle. In the equivalent DOM or SVG case, one could simply change the position attributes of the rectangle, and the browser would then automatically determine how to optimally re-render all the affected objects.

The code snippet below provides a minimal example of Canvas API usage. In this example, we first instantiate a 2D canvas graphics context of size 100x100

after declaring the corresponding HTML element. We then imperatively draw a full circle with a 40 pixel radius in the middle of the canvas using the Canvas 2D Context JavaScript API.

```
<!DOCTYPE html>
<html><body>

<canvas id="myCanvas" width="100" height="100">
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.arc(50,50,40,0,2*Math.PI);
ctx.stroke();
</script>

</body></html>
```

Note that in these simple examples we are mixing HTML and JavaScript code. In real-world examples, it would be a good practice to keep declarative HTML code and imperative JavaScript code in separate files. We will discuss programming style implications later in Section 4.

The event handling capabilities of the Canvas API are minimal. A limited form of event handling is supported by the Canvas API with *hit regions* (`https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Hit_regions_and_accessibility`).

Conceptually, Canvas is a low level API upon which a higher-level rendering engine might be built. Although canvas elements are created in the browser as subelements in the DOM, it is entirely possible to create just one large canvas element, and then perform all the application rendering and event handling inside that element. There are JavaScript libraries that add event handling and scene graph capabilities to the canvas element. For instance, with *Paper.js* (`http://paperjs.org/`) or *Fabric.js* (`http://fabricjs.com/`) libraries, it is possible to paint a canvas in layers, and then recreate specific layers, instead of having to repaint the entire scene manually each time. Thus, the Canvas API can be used as a full-fledged application rendering model of its own.

The Canvas element was was initially introduced by Apple in 2004 for use inside their own Mac OS X WebKit component in order to support applications such as Dashboard widgets in the Safari browser. In 2005, the Canvas element was adopted in version 1.8 of Gecko browsers and Opera in 2006. The Canvas API was later standardized by the Web Hypertext Application Technology Working Group (WHATWG).

The adoption of the Canvas API was originally hindered by Apple's intellectual property claims over this API. From technical viewpoint, adoption was also slowed down by the fact that the Canvas API expressiveness is significantly more limited than the well-established, mature immediate-mode graphics APIs that were available in mainstream operating systems already a decade or two earlier.

Microsoft's DirectX API – originally introduced in Windows 95 – is a good example of a substantially more comprehensive API. Nowadays the Canvas API is supported by all the main web browsers; in spite of its technical limitations, the Canvas API has a thriving library ecosystem as well.

### 3.3 WebGL

*WebGL* (`http://www.khronos.org/webgl/`) is a cross-platform web standard for hardware accelerated 3D graphics API developed by Khronos Group, Mozilla, and a consortium of other companies including Apple, Google and Opera. The main feature that WebGL brings to the Web is the ability to display 3D graphics natively in the web browser without any plug-in components. WebGL is based on OpenGL ES 2.0 (`http://www.khronos.org/opengles`), and it leverages the OpenGL shading language GLSL. A comprehensive JavaScript API is provided to open up OpenGL programming capabilities to JavaScript programmers.

In a nutshell, WebGL provides a JavaScript API for rendering interactive, immediate-mode 3D (and 2D) graphics within any compatible web browser without the use of plug-in components. WebGL is integrated into major web browsers, enabling Graphics Processing Unit (GPU) accelerated usage of physics and image processing and effects in web applications. WebGL applications consist of control code written in JavaScript and shader code that is typically executed on a GPU.

WebGL is widely supported in modern desktop browsers. Today, even all the major mobile browsers (excluding Opera Mini) support WebGL by default. However, actual usability of WebGL functions is dependent on various factors such as the GPU supporting it. Even in many desktop computers WebGL applications may run poorly unless the computer has a graphics card that provides sufficient capabilities to process OpenGL functions efficiently.

Just like the Canvas API discussed above, the WebGL API is a rather low-level API that does not automatically manage rendering or support high-level events. From the application developer's viewpoint, the WebGL API is in fact too cumbersome to use directly without utility libraries. For instance, setting up typical view transformation shaders (e.g., for view frustum), loading scene graphs and 3D objects in the popular industry formats can be very tedious and requires writing a lot of source code.

Given the verbosity of shader definitions, we do not provide any code samples here. However, there are excellent WebGL examples on the Web. For instance, the following link contains a great example of an animated, rotating, textured cube with lighting effects: `http://www.sw-engineering-candies.com/snippets/webgl/hello-world/`.

Because of the complexity and the low level nature of the raw WebGL APIs, many JavaScript convenience libraries have been built or ported onto WebGL in order to facilitate development. Examples of such libraries include *A-Frame, BabylonJS, three.js, O3D, OSG.JS, CopperLicht* and *GLGE*.

### 3.4 SVG

Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics with support for interactivity, affine transformations and animation. The *SVG Specification* [30] is an open standard published by the World Wide Web Consortium (W3C) originally in 2001. Although bitmap images were supported since the early days of the Web (the `<IMG>` tag was introduced in the Mosaic browser in 1992), vector graphics support came much later via SVG.

The code snippet below provides a simple example of an SVG object definition that renders an automatically scaling W3C logo to the screen[2].

```
<div id="w3clogo">
<svg xmlns='http://www.w3.org/2000/svg' viewBox="0 0 131 76">
  <path d="M36,5l12,41l12-41h33v4l-13,21c30,10,2,69-21,28l7-2
    c15,27,33,-22,3,-19v-4l12-20h-15l-17,59h-11-13-42l-12,42h
    -11-20-67h9l12,41l8-28l-4-13h9" fill='#005A9C'/>
  <path d="M94,53c15,32,30,14,35,7l-1-7c-16,26-32,3-34,0M122
    ,16c-10-21-34,0-21,30c-5-30 16,-38 23,-21l5-10l-2-9"/>
</svg>
</div>
```

While SVG was originally just a vector image format, SVG support has been integrated closely with the web browser to provide comprehensive means for creating interactive, resolution-independent content for the Web. Just like with the HTML DOM, SVG images can be manipulated using DOM APIs via HTML, CSS and JavaScript code. This makes it possible to create shapes such as lines, Bezier/elliptical curves, polygons, paths and text and images that be resized, rescaled and rotated programmatically using a set of built-in affine transformation and matrix functions.

The code sample below serves as an example of interactive SVG that defines a circle object that is capable of changing its size in response to mouse input.

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="6cm" height="5cm" viewBox="0 0 600 500" xmlns="
    http://www.w3.org/2000/svg" version="1.1">

  <!-- Change the radius with each click -->
  <script type="application/ecmascript">
    function circle_click(evt) {
      var circle = evt.target;
      var currentRadius = circle.getAttribute("r");
      if (currentRadius == 100) {
        circle.setAttribute("r", currentRadius*2);
      } else {
        circle.setAttribute("r", currentRadius*0.5);
```

[2] `https://dev.w3.org/SVG/tools/svgweb/samples/svg-files/w3c.svg`

```
15        }
16      }
17    </script>
18
19    <!-- Define circle with onclick event handler -->
20    <circle onclick="circle_click(evt)" cx="300" cy="225" r="100
        " fill="blue"/>
21  </svg>
```

As illustrated in the example, the SVG scene graph enables event handlers to be associated with objects, so a circle object may respond to an `onClick` event or other events. To get the same functionality with Canvas, one would have to implement the code to manually match the coordinates of the mouse click with the coordinates of the drawn circle in order to determine whether it was clicked.

Just like with the HTML DOM, SVG support in the web browser is based on a *retained (managed) graphics architecture*. Inside the browser, each SVG shape is represented as an object in a *scene graph* that is rendered to the display automatically by the web browser. When the attributes of an SVG object are changed, the browser will calculate the most optimal way to re-render the scene, including the other objects that may have been impacted by the change.

In the earlier days of the Web, SVG was the only mechanism to implement a scalable, "morphic" graphics system, which is why the SVG DOM API was used as the foundation for graphics implementation, e.g., in the original Lively Kernel web programming system that provided a self-supporting development environment inside the browser [26,11]. The following link provides a reference to a more comprehensive, "Lively-like" example of an SVG-based application that includes interactive capabilities (image rescaling and rotation based on mouse events) as well: `https://dev.w3.org/SVG/tools/svgweb/samples/svg-files/photos.svg/`.

In general, it is important to summarize that in the context of the Web, SVG is *much more than just an image format*. Together with event handling capabilities, affine transformations, gradient support, clipping, masking and composition features, SVG can be used as the basis for a full-fledged, standalone graphical application architecture or windowing system.

### 3.5 Web Components

*Web Components* (`https://www.w3.org/TR/#tr_Web_Components`) are a set of features added to the HTML and DOM specifications to enable the creation of reusable widgets or components in web documents and applications. The intention behind web components is to 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.

An important motivation for web components is the *fundamentally brittle nature of the Document Object Model*. The brittleness comes from the global

nature of elements in the DOM created by HTML, CSS and JavaScript code. For example, when you use a new HTML `id` or `class` in your web application or page, there is no easy way to find out if it will conflict with an existing name used by the page already earlier. Subtle bugs creep up, style selectors can suddenly go out of control, and performance can suffer, especially when attempting to combine code written by multiple authors [16]. Over the years various tools and libraries have been invented circumvent the issues, but the fundamental brittleness issues remain. The other important motivation is the *fixed nature of the standard set of HTML elements*. Web components make it possible to extend the basic set of components and support dynamically downloadable components across different web pages or applications.

Web components are built on top of a concept known as the *Shadow DOM*. In technical terms, the Shadow DOM introduces the concept of parallel "shadow" subtrees in the Document Object Model. These subtrees can be viewed conceptually as "icebergs" that expose only their tip while the implementation details remain invisible (and inaccessible) under the surface. Unlike regular branches in the DOM tree, shadow trees provide support for *scoped styles* and *DOM encapsulation*, thus obeying the well-known separation of concerns and modularity principles that encourage strong decoupling between public interfaces and implementation details [19]. Utilizing the Shadow DOM, the programmer can bundle CSS with HTML markup, hide implementation details, and create self-contained reusable components in vanilla JavaScript without exposing the implementation details or having to follow awkward naming conventions to ensure unique naming.

At the technical level, a shadow DOM tree is just normal DOM tree with two differences: 1) how it is created and used, and 2) how it behaves in relation to the rest of the web page. Normally, the programmer creates DOM nodes and appends those nodes as children of another element. With shadow DOM, the programmer creates a *scoped DOM tree* that is attached to the element but that is separate from its actual children. The element it is attached to is its *shadow host*. Anything that the programmer adds to the shadow tree becomes local to the hosting element, including `<style>`. This is how shadow DOM achieves CSS style scoping.

The following listing presents a minimal web component example that creates a text editor that automatically resizes itself as text is entered in the text area:

```
<!DOCTYPE html>
<html><head>
<link rel="import" href="basic-autosize-textarea.html" >
</head><body>
<p>Automatically resizing text input component:</p>
<basic-autosize-textarea>Edit me!
</basic-autosize-textarea>
</body></html>
```

Note that up until recently, many browsers did not support web components yet. Therefore, they had to be emulated in the form of *polyfill libraries* that im-

plement the missing functionality (`http://webcomponents.org/polyfills/`). As of this writing, native support for the Shadow DOM is available all major web browsers except Microsoft Explorer and Microsoft Edge. For latest status, refer to `http://caniuse.com/#feat=shadowdom/`.

## 4 Comparison and Primary Use Cases

The technologies described in the previous section are rather different, with divergent design goals and varying and partially overlapping functionality. As a result, it is not easy to perform an objective comparison, or provide measurements on, e.g., development efficiency or ease of use. In general, ease of development or use in the context of the Web is highly subjective and dependent on one's background, e.g., whether the developer is a classically trained software engineer or a web developer who has never written software for target platforms other than the Web.

In this section we first provide a comparison that begins with an overview table that gives a summary of the basic differences between the presented technologies. Second, we discuss the primary use cases for the different technologies. Broader technical and architectural implications will be discussed separately in Section 5.

### 4.1 Technology Comparison: An Overview

An overview and a summary of the different approaches is presented in Table 1. The table covers topics such as the overall development paradigm (imperative vs. declarative), rendering architecture (retained/managed vs. immediate), information hiding support, primary intended usage domain and current popularity. We also provide impressions on more subjective factors such as technology maturity, abstraction level and ease of code reuse. Finally, the table summarizes whether each technology provides support for defining animations in a declarative fashion (as opposed to having to write lengthy JavaScript timer scripts to drive animations), as well as whether the technology is supported by mobile browsers.

### 4.2 Primary Use Cases

While the presented five technologies are all fully functional and Turing complete in the sense that they can be used for writing any imaginable application within the context of the sandbox offered by the web browser, these technologies are originally intended for different purposes and use cases. To begin with, each of the technologies introduces their own distinct programming style(s). This is especially true of the Canvas and WebGL technologies that are much lower level, imperative APIs that require significantly more manual labor, e.g., in the placement of graphics and in driving the rendering process. In contrast, DOM/DHTML, SVG and Web Component programming is performed at a

**Table 1.** Comparison of Built-In Client-Side Rendering Technologies [29]

| | **DOM / DHTML** | **Canvas** | **WebGL** | **SVG** | **Web Components** |
|---|---|---|---|---|---|
| Development Paradigm | Declarative and imperative | Imperative | Imperative | Declarative and imperative | Declarative and imperative |
| Rendering Architecture | Retained | Immediate (explicit repainting required) | Immediate (explicit repainting required) | Retained | Retained |
| Information Hiding | No | Not applicable (no namespace support) | Not applicable | No (except when creating multiple SVG images) | Yes (Shadow DOM encapsulation and scoped styles) |
| Primary Usage Domain | Documents and forms | 2D graphics (e.g., in games) | 3D/2D graphics especially in games and VR/AR | 2D image rendering | Web applications and graphical user interfaces |
| Popularity | Ubiquitous | Popular in specific use cases | Limited | Popular in specific use cases | Growing |
| Technology Maturity | Mature | Mature | Mature | Mature | Emerging (standardization underway) |
| Abstraction Level | Medium | Very low | Low | Medium | High |
| Ease of Code Reuse | Low to medium | Low | Medium (shaders) | Low to high (high as an image format) | High |
| Declarative Animation Support | Yes | No | No | Yes | Yes |
| Mobile Browser Support | Yes | Yes | Yes | Yes | Not in Microsoft browsers |

higher level and require much less imperative control and attention over rendering. That said, DOM/DHTML and SVG programming can be performed in a number of very different ways depending on whether the developer prefers a declarative development style (relying only on HTML and CSS) or imperative development style (developing primarily in JavaScript).

The following bullets provide a basic characterization on the primary baseline use cases for each technology.

- *DOM/DHTML.* HTML was originally developed as a declarative markup language for creating static documents and forms. Over the years, the use of DOM/DHTML has expanded to almost every imaginable use case. Today, DOM-based development approach dominates the web development landscape. This approach is declarative in nature, so the browser largely decides about rendering; this simplifies the development of web sites that look like documents, but can complicate the creation of sites that should behave like desktop applications or require control of the display at pixel level.
- *Canvas.* The Canvas API was introduced at a time when there was no other way to render lines, circles, rectangles or other low-level graphics imperatively inside the browser. For a number of reasons that were highlighted earlier, the Canvas API is significantly less capable that it ideally should be. Currently, the Canvas API is utilized primarily by game developers. It is also used occasionally inside regular web pages to include custom graphical content, although the majority of such use cases can often be completed more conveniently in SVG.
- *WebGL.* From technical viewpoint, WebGL is basically a thin JavaScript wrapper over native OpenGL interfaces for providing a programmatic API inside the web browser to achieve hardware-accelerated (GPU) rendering. As a result, the use cases of WebGL are a direct derivative of the OpenGL use cases, including (especially) game development, computer-aided design (CAD), scientific visualization, flight simulation, virtual reality, or any other case in which advanced 3D (or 2D) graphics rendering capabilities are needed. WebGL is an imperative, low-level API that places a lot of requirements on developer skills. Until recent years, the use of WebGL was still marginal, but it has steadily gained importance as the need to render VR/AR content in the web browser increases.
- *SVG.* In the context of the web browser, SVG has a dual role. First and foremost, SVG is a vector image format for rendering scalable graphics content on web pages. However, SVG can also be used as a rich, generic graphics context to drive scene graph based applications with support for complex event handling, affine transformations (rotation, zooming, scaling, shearing), gradients, clipping, masking and object composition. Given that the basic DOM has evolved over the years to support these capabilities, in practice SVG is used mainly as an image format. Thus, the importance of the broader application development use cases for SVG is nowadays small.
- *Web Components.* Web components are the "dark horse" in web development – they are still little known to most developers, and it is difficult to

place betting odds on their eventual success. Web components reintroduce well-known (but hitherto missing) software engineering principles and practices 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 cater to nearly any imaginable use case but they are especially well-suited to the development of full-fledged web applications that require an extensible set of GUI widgets.

## 5 Broader Considerations

According to MacLennan's classic software engineering principles [13], some of the most fundamental principles in software development are *simplicity* and *consistency*: There should be a minimum number of concepts with simple rules for their combination; things that are similar should also look similar, and different things should look different. Unfortunately, the web browser violates these and several other key principles in a number of ways, as evidenced by the above observations.

**Overlapping capabilities**. 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 rendering tasks. It is not easy to provide recommendations on specific technologies to use, except for those tasks in which immediate-mode graphics is required (in which case either the Canvas or WebGL API will have to be utilized). In most cases, developers will end up using the basic DOM/DHTML approach, complemented with various libraries.

**Mismatching development styles**. When composing web applications even using the basic DOM/DHTML approach, the developers commonly face a mixture of declarative and imperative programming styles. They may also have to use a combination of retained and immediate-mode graphics especially when aiming at applications that are usable across different screen sizes – following responsive web design [14]. In general, imperative versus declarative and unmanaged versus managed graphics rendering provide different facilities and require different considerations, and the implementation mechanisms can be completely different. In fact, such adaptation could have been yet another dimension to compare.

**Incompatible and incoherent abstractions**. The abstractions and programming patterns supported by Canvas and WebGL APIs are very different from DOM/DHTML and SVG programming. Web components introduce yet another abstraction layer that has been patched on top of the DOM/DHTML. In general, the features supported by the browser reflect organic evolution of features over the years rather than any carefully master-planned architectural design. For instance, patterns and styles required for Canvas and WebGL programming are very different from DOM or SVG; Web Component (Shadow DOM) programming requires yet another programming style. When these programming

patterns are combined – as often happens when using code from other parties – confusing situations may emerge.

Given the organic evolution of the web ecosystem, it is nevertheless fairly safe to predict that we will not go back to a less diverse web ecosystem or have a chance to radically simplify the feature set of the web browser. It is impossible to put the genie back to the bottle. For example, recent versions of the JavaScript language – from ECMAScript6 to ECMAScript9 – have introduced a lot of new language functionality (promises, generators and decorators, to list a few), thus ensuring that library rewriting and evolution will be swift in the coming years, creating further diversity and potential confusion for application developers.

**Fashion-driven development**. Over the past years there has been a notable trend in the library area towards fashion-driven development. By this we refer to the developers' tendency to surf on the wave of newest and most dominant "alpha" development frameworks. For instance, the once hugely popular *Prototype.js* and *JQuery.js* libraries were largely replaced by *Knockout.js* and *Backbone.js* in 2012. Back in 2014, *Angular.js* was by far the most dominant alpha framework, while in 2016-2017 it was the *React.js + Redux.js* ecosystem that seemed to be capturing the majority of developer attention, with *Vue.js* then foreseen as the most likely next dominant framework. As witnessed by the somewhat unfortunate evolution of the Angular ecosystem over the years, the alpha frameworks have a tendency to evolve very quickly once they get developers' attention, leading into compatibility issues. To make the matters worse, once the next fashionable alpha framework emerges and hordes of developers start jumping ship onto the new one, it becomes questionable to what extent one can build long-lasting business-critical applications and services, e.g., for the medical industry in which products must commonly have a minimum lifetime of twenty years. With the present pace of upgrades, the browser and the web server as the runtime environment would be almost completely replaced by patches, upgrades, and updates; similarly, most of the libraries would be replaced several times by newer, more fashionable ones.

**Opportunistic design and "cargo cult" programming**. In web development there has historically been a strong tradition of *mashup-based development*: searching, selecting, pickling, mashing up and glueing together disparate libraries and pieces of software [9]. Often such development has the characteristics of *cargo cult programming*: ritually including code and program structures that serve no real purpose or that the programmer has chosen to include because hundreds of other developers have done so – without really understanding why. The popularity of opportunistic design has exploded because of the success of *Node.js* (`https://nodejs.org/`) and its *Node Package Manager (NPM)* ecosystem (`https://www.npmjs.com/`) – nowadays, there are over 800,000 reusable NPM modules available for nearly all imaginable tasks. While this approach can save a lot of work and open up interesting opportunities for large-scale code reuse [21], this approach does not foster development of reliable, long-lasting applications, because even the smallest changes in the constituent components

– each of which evolves separately and independently – can break applications [22].

**Violation of established software engineering principles**. Although many web developers may not realize this, the web browser violates many established software engineering principles, including the lack of *information hiding*, lack of *manifest interfaces*, lack of *orthogonality*, and lack of (aforementioned) simplicity and consistency [13]. These observations were reported already over ten years ago [16], but little has happened to fix the issues, apart from libraries that aim at introducing their own way of engineering web applications. The absence of solid engineering principles is easy to understand given that the web browser was originally designed to be a document distribution environment rather than a "real" application execution environment. However, the current popularity of the Web as the software platform makes it very unfortunate that these important principles have been ignored. Currently the web components are the best – and perhaps also the only – chance to reintroduce some of these important principles to the heart of the Web.

In the broader picture, the deficiencies of the web browser as a software platform are being tackled with an abundance of libraries. As of this writing, there are more than 1,400 officially listed JavaScript libraries in `javascripting.com`, with new ones being introduced on a weekly if not daily basis. Although many of the libraries are domain-specific, a lot of them are aimed squarely at solving the architectural limitations of the web browser, e.g., to provide a consistent set of manifest interfaces to perform various programming tasks. Over the years, JavaScript libraries have evolved from mere convenience function libraries to full-fledged Model-View-Controller (MVC) frameworks providing extensive UI component sets, application state management, network communication and database interfaces, and so on. In general, these will not necessarily help in tackling the above characteristics but may rather add a new layer of complexity on top of them.

## 6 Revisiting Our Earlier Predictions and Considerations

As mentioned in the beginning, this paper is an expanded, revisited version of papers that were published earlier [29,27]. In this section, we will revisit our earlier predictions and considerations in the light of more recent technologies and approaches to web applications.

**The emergence of Virtual DOM technologies**. Out of the technologies discussed in [29,27], DOM/DHTML has maintained its dominant role as the baseline technology as we predicted. The majority of libraries and applications that have been developed over the years are built on top of the standard DOM/DHTML approach. What we did not foresee, however, was the introduction of techniques that effectively replicate and virtualize the behavior of the Document Object Model in order to gain additional programmatic control over rendering. These new approaches can be viewed as a derivative of the Shadow DOM model introduced by Web Components, except that in these

approaches the DOM is externalized and replicated outside the built-in Document Object Model, thus allowing libraries and applications to work around some of the limitations and built-in assumptions that the web browser imposes on application development. Simply put, Virtual DOM trees (see `https://bitsofco.de/understanding-the-virtual-dom/`) are copies of the original DOM; these copies can be manipulated and updated independently of the browser-level DOM APIs, thus bypassing any immediate impact on the browser's rendering process. Once all the updates have been made in the virtual DOM, the changes need to pushed back (copied) to the original DOM in an optimized way. The Virtual DOM approach can considerably improve rendering performance as well as enhance the overall smoothness of web user interfaces in comparison to traditional DOM manipulation in which applications have very limited control over rendering.

**Emerging support for Virtual and Augmented Reality**. In our previous work, we foresaw increased popularity of WebGL that enables browser-based, installation-free, high-performance applications for viewing VR/AR content. These features will inevitably gain more popularity, as the world moves towards richer media experiences, and the standard DOM/DHTML model is unable to support the necessary features. To this end, further rendering and visualization techniques that build on WebGL have been proposed. These include *WebVR* (https://webvr.info/) and *WebXR Device API* (https://immersive-web.github.io/webxr/), which take the Web towards virtual and augmented reality rendering with new APIs. In addition, we predicted that WebGL would also be increasingly important for game developers; however, the elimination of the "last safe bastion" of traditional binary applications (as indicated in our earlier paper) – allowing the creation of portable high-performance applications in the context of the web browser – has not yet taken place. Similarly, we pointed out that the Web would benefit from a high-performance, low-level 2D graphics API that would provide a more comprehensive feature set and direct drawing capabilities without any historical development baggage of the Canvas API. However, at the time of this writing, there is no such standardization effort in sight.

**Web Components**. Regarding web components, it is still too early to declare victory or failure. Since web components offer a more disciplined approach to DOM/DHTML programming, reintroduce established software engineering principles, and generally alleviate the "spaghetti code" issues that have resurfaced with the Web [25], we would certainly like to see them succeed. In reality, the main obstacle to the wider adoption of web components are the predominant JavaScript libraries that also provide additional abstraction layers on top of the underlying DOM and basic browser features. Hence, the future of web components is fundamentally affected by the evolution of JavaScript library landscape and associated features.

**JavaScript library landscape**. Earlier in this paper, we noted that JavaScript library evolution has followed a fashion-driven approach in which a few frameworks have dominated the landscape for a few years, only to be superseded by new dominant frameworks some years later. Interestingly, this trend seems to

have waned in the past two years. While the flood of new, lesser known front-end and backend frameworks has continued as strongly as ever (as witnessed in `http://www.javascripting.com/` or in the constantly increasing number of NPM packages), the popularity of top three dominant front-end libraries has not changed much in the past two years.

When writing the first manuscripts of our earlier papers in late 2016/early 2017 [29,27], the top five frontend frameworks were Vue.js, React, Angular, Angular 1, and Inferno (`https://risingstars.js.org/2016/en/`). As of this writing (January 2019), Vue.js, React, and Angular are still the top three, followed by Hyperapp and Omi (`https://risingstars.js.org/2018/en/#section-framework`). At the same time, jQuery is still used rather extensively (`https://w3techs.com/technologies/details/js-jquery/all/all`), implying that some web frameworks can also have an extended lifespan. However, the introduction and enamoration with new, fashionable frameworks is by no means over. For instance, *Weex* (`https://weex.incubator.apache.org/`) has recently gained popularity rapidly in the mobile domain.

## 7 Future Work

In this paper we have scratched only the surface of architectural issues related to web applications, as we intentionally narrowed down our analysis into one specific area: the built-in application rendering technologies in a modern web browser. The web ecosystem provides a cornucopia of choices in many other areas. Consequently, there are several avenues for future research and directions to expand this work towards different dimensions of the Web as an application platform.

We are currently encouraging ourselves and our students to perform similar studies, e.g., on the cornucopia of communication mechanisms and methods used in web applications, including Ajax [8], Comet [5], Server-Sent Events [10], WebSockets [20], WebRTC [2]), and to some extent also Web/Shared Workers [31]. In addition, persistent storage in the context of the web browser is an interesting topic, although the design space in that context is far more limited.

Cornucopia associated with front-end Web frameworks is an even more diverse area to study than the technologies inside the browser itself. So far, we have briefly studied only the most popular mainstream frameworks, but a more in-depth look would definitely be an interesting direction for future work, in particular since the many libraries provide facilities that are similar to those of the technologies inside the browser.

Finally, server-side web development is yet another rich area for future work. As already mentioned, over the past few years, an extremely prolific ecosystem has emerged around Node.js, and there are a lot of additional open source technologies for nearly every imaginable aspect of server-side development. For instance, data acquisition and analytics solutions such as Apache Kafka, Storm and Spark have become very popular. From architectural standpoint, the recent trend towards *isomorphic JavaScript* is also extremely relevant. In web devel-

opment, an isomorphic application is one whose code can run unmodified both in the server and the client [23]. Such capabilities are relevant, e.g., in realizing *liquid software* that allows applications to seamlessly migrate across multiple devices [28,7].

## 8 Conclusions

Over the past twenty-five years or so, the World Wide Web has evolved from a document sharing system to a full-fledged programming environment. This evolution has taken place organically, and new technologies have been constantly introduced to help developers create compelling web systems.

As a consequence, web development today presents a cornucopia of choices on all fronts. Both on the client side and the server side, there exist a large number of competing, overlapping technologies, and new libraries and tools become available almost on a daily basis. The rapid pace of innovation has put the developers in a complex position 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.

In this paper, we have investigated one of the perhaps most overlooked areas in web development: the client-side web rendering architectures that have been built into the generic web browser. We compared five built-in rendering and application development models, followed by some predictions, discussion and avenues for future research.

As Alan Kay once aptly put it, "simple things should be simple, and complex things should be possible". In web development today, pretty much everything is possible, but really not at all in the simplest possible way. While the World Wide Web is one of the most important innovations for humankind, for web application developers things are still likely to get even more complicated until they get any simpler.

## References


1. M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari. Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 800–807. ACM, 2011.
2. A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, B. Aboba, T. Brandstetter, and J.-I. Bruaroey. WebRTC 1.0: Real-time Communication Between Browsers. *W3C Candidate Recommendation, 27 September 2018, available at: https://www.w3.org/TR/webrtc/*.
3. Bitworking.org. Zero Framework Manifesto: No More JS Frameworks, 2014. `https://bitworking.org/news/2014/05/zero_framework_manifesto`.
4. S. Casteleyn, I. Garrigós, and J.-N. Mazón. Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. *ACM Trans. Web*, 8(3):18:1–18:46, July 2014.
5. D. Crane and P. McCarthy. *What Are Comet and Reverse Ajax?* Springer, 2009.

6. D. Flanagan. *JavaScript: The Definitive Guide, 6th edition.* O'Reilly Media, 2011.
7. A. Gallidabino, C. Pautasso, V. Ilvonen, T. Mikkonen, K. Systä, J.-P. Voutilainen, and A. Taivalsaari. On the Architecture of Liquid Software: Technology Alternatives and Design Space. In *Proc. of WICSA*, 2016.
8. J. J. Garrett. Ajax: A New Approach to Web Applications. http://adaptivepath.org/ideas/ajax-new-approach-web-applications/, 18 February 2005.
9. B. Hartmann, S. Doorley, and S. R. Klemmer. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing*, 7(3):46–54, 2008.
10. I. Hickson. Server-Sent Events. *W3C Recommendation 03 February 2015, latest version available at http://www.w3.org/TR/eventsource/*, 2015.
11. D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel: a Self-Supporting System on a Web Page. In *Self-Sustaining Systems*, pages 31–50. Springer, 2008.
12. M. A. Jadhav, B. R. Sawant, and A. Deshmukh. Single Page Application using AngularJS. *International Journal of Computer Science and Information Technologies*, 6(3), 2015.
13. B. J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation, 3rd edition.* Oxford University Press, 1999.
14. E. Marcotte. *Responsive Web Design.* Editions Eyrolles, 2011.
15. E. Meijer. Democratizing the Cloud. In *Companion Proc. of OOPSLA'07*, pages 858–859, 2007.
16. T. Mikkonen and A. Taivalsaari. Web Applications — Spaghetti Code for the 21st Century. In *Proc. Int'l Conf. Software Engineering Research, Management and Applications (SERA'2008, Prague, Czech Republic, August 20-22, 2008)*, pages 319–328. IEEE Computer Society, 2008.
17. T. Mikkonen and A. Taivalsaari. Apps vs. Open Web: The Battle of the Decade. In *Proceedings of the 2nd Workshop on Software Engineering for Mobile Application Development*, pages 22–26. MSE Santa Monica, CA, 2011.
18. T. Mikkonen and A. Taivalsaari. Cloud Computing and its Impact on Mobile Software Development: Two Roads Diverged. *Journal of Systems and Software*, 86(9):2318–2320, 2013.
19. D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
20. V. Pimentel and B. G. Nickerson. Communicating and Displaying Real-time Data with WebSocket. *IEEE Internet Computing*, 16(4):45–53, 2012.
21. A. Salminen and T. Mikkonen. Mashups: Software Ecosystems for the Web Era. In *IWSECO@ ICSOB*, pages 18–32, 2012.
22. A. Salminen, T. Mikkonen, F. Nyrhinen, and A. Taivalsaari. Developing Client-Side Mashups: Experiences, Guidelines and the Road Ahead. In *Proc. 14th Int'l Academic MindTrek Conference: Envisioning Future Media Environments*, pages 161–168. ACM, 2010.
23. J. Strimpel and M. Najim. *Building Isomorphic JavaScript Apps: From Concept to Implementation to Real-world Solutions.* " O'Reilly Media, Inc.", 2016.
24. A. Taivalsaari and T. Mikkonen. The Web as an Application Platform: The Saga Continues. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 170–174. IEEE, 2011.
25. A. Taivalsaari and T. Mikkonen. Return of the Great Spaghetti Monster: Learnings from a Twelve-Year Adventure in Web Software Development. In *International Conference on Web Information Systems and Technologies*, pages 21–44. Springer, 2017.

26. A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web Browser as an Application Platform: the Lively Kernel Experience, Sun Labs Technical Report TR-2008-175. January 2008.
27. A. Taivalsaari, T. Mikkonen, C. Pautasso, and K. Systä. Comparing the Built-in Application Architecture Models in the Web Browser. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 51–54. IEEE, 2017.
28. A. Taivalsaari, T. Mikkonen, and K. Systä. Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture. In *38th IEEE Computer Software and Applications Conference (COMPSAC)*, pages 338–343, 2014.
29. A. Taivalsaari, T. Mikkonen, K. Systä, and C. Pautasso. 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.*, pages 127–136, 2018.
30. W3C. Scalable Vector Graphics (SVG) Specification 1.1 (Second Edition), 2011. `https://www.w3.org/TR/SVG/`.
31. W3Schools. HTML5 Web Workers. `http://www.w3schools.com/html/html5_webworkers.asp`.