Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Klaus Nygård

# Single page architecture as basis for web applications

Master's Thesis
Espoo, June 6, 2015

Supervisor: Professor Petri Vuorimaa
Advisor: D.Sc. (Tech.) Jari Kleimola

# Abbreviations and Acronyms

| | |
|---|---|
| AMD | Asynchronous Module Definition |
| AJAX | Asynchronous JavaScript and XML |
| API | Application Programming Interface |
| CDN | Content Delivery Network |
| CORS | Cross-Origin Resource Sharing |
| CPU | Central Processing Unit |
| CSS | Cascading Style Sheet |
| CSS3 | Third revision of CSS |
| DOM | Document Object Model |
| GPS | Global Positioning System |
| HTML | Hypertext Markup Language |
| HTML5 | Fifth revision of HTML |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol Secure |
| IDE | Integrated Development Environment |
| IP | Internet Protocol |
| JS | JavaScript |
| JSON | JavaScript Object Notation |
| MathML | Mathematical Markup Language |
| MVC | Model-View-Controller |
| MVP | Model-View-Presentation |
| MVVM | Model-View-ViewModel |
| OpenGL ES | Open Graphics Library for Embedded Systems |
| REST | Representational State Transfer |
| UDP | User Datagram Protocol |
| UI | User Interface |
| URI | Uniform Resource Identifier |
| SDK | Software Development Kit |
| SaaS | Software as a Service |
| SOAP | Simple Object Access Protocol |

| | |
|---|---|
| SVG | Scalable Vector Graphics |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| UI | User Interface |
| W3C | World Wide Web Consortium |
| WHATWG | Web Hypertext Application Technology Working Group |
| WebGL | Web Graphics Library |
| WebRTC | Web Real-Time Communication |
| WLAN | Wireless Local Area Network |
| WWW | World Wide Web |
| XHR | XMLHttpRequest |
| XML | eXtensible Markup Language |

# Contents

**Abbreviations and Acronyms** **iv**

**1 Introduction** **1**
1.1 Objectives and research questions . . . . . . . . . . . . . . . . . 2
1.2 Structure of the Thesis . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Web as an Application Platform** **4**
2.1 Architecture of Web Applications . . . . . . . . . . . . . . . . . 4
2.1.1 Presentation tier . . . . . . . . . . . . . . . . . . . . . . . 6
2.1.2 Business logic tier . . . . . . . . . . . . . . . . . . . . . . 6
2.1.3 Data tier . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Advantages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

**3 Overview of Web Technologies** **13**
3.1 HTML5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1.1 Elements and semantics . . . . . . . . . . . . . . . . . . . 14
3.1.2 Media support . . . . . . . . . . . . . . . . . . . . . . . . 15
3.1.3 Canvas . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.1.4 Drag-and-drop . . . . . . . . . . . . . . . . . . . . . . . . 16
3.1.5 Web Messaging . . . . . . . . . . . . . . . . . . . . . . . . 16
3.1.6 Browser history management . . . . . . . . . . . . . . . . 16
3.1.7 Offline Web Applications . . . . . . . . . . . . . . . . . . 16
3.2 Cascading Style Sheets . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 JavaScript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.4 Related specifications . . . . . . . . . . . . . . . . . . . . . . . . 20
3.4.1 Web Storage . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.4.2 WebSockets . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.4.3 Web Real-Time Communication . . . . . . . . . . . . . . 22
3.4.4 Touch Events . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.4.5 Web Workers . . . . . . . . . . . . . . . . . . . . . . . . . 23

3.4.6 Service Workers . . . 23
3.4.7 Server-Sent Events . . . 24
3.4.8 Cross-Origin Resource Sharing . . . 24
3.4.9 File API . . . 24
3.4.10 Geolocation . . . 25
3.4.11 Scalable Vector Graphics . . . 25
3.4.12 Web Audio . . . 25
3.4.13 WebGL . . . 26

**4 Single Page Architecture** **27**
4.1 Key Concepts and Components . . . 27
4.1.1 AJAX . . . 27
4.1.2 REST . . . 29
4.1.3 Separation of Concerns . . . 31
4.1.4 Data binding . . . 32
4.1.5 Routing . . . 33
4.2 Overview of Popular Frameworks . . . 34
4.2.1 AngularJS . . . 36
4.2.2 Backbone . . . 37
4.2.3 React . . . 38
4.2.4 EmberJS . . . 39

**5 Implementations with AngularJS** **41**
5.1 Specifications . . . 41
5.2 System Architecture . . . 43
5.3 Development tools . . . 44
5.4 Libraries . . . 45
5.5 Technologies . . . 46
5.6 Application structure . . . 47
5.7 User Interfaces . . . 51

**6 Discussion** **58**
6.1 Development and distribution . . . 58
6.2 Performance . . . 59
6.3 User Experience . . . 60
6.4 Summary . . . 60
6.5 Future Work . . . 63

**7 Conclusions** **65**

# Chapter 1

# Introduction

Since its birth in 1990, the World Wide Web has served as an universal platform to serve content to all parts of the world. HTML (Hypertext Markup Language) was designed to be the standard document format for the web, which it still is. However, along with the breakthrough of smartphones, the web has been disrupted by the rise of new marketplaces, such as Apple App Store and Google Play. The physical Internet has taken a major role not only in serving information via web, but also working as a layer for transporting data to and between applications that are not in web.

Applications that are built for specific platforms or operating systems by specific tools, are called *native applications*. Such applications are often distributed in the Internet via dedicated marketplaces or web sites. Due to the popularity of the web, native applications are more and more rivalled by *web applications*. Previously the web applications were not considered an alternative to the native applications — mostly due to their inferior performance and dependency on an Internet connection. Nevertheless, as the Internet connection speeds are less and less a restrictive constraint for web applications, and advanced web technologies like HTML5, CSS3 (Cascading Style Sheet, version 3), AJAX (Asynchronous JavaScript and XML) and WebSockets have emerged, web applications have potential to replace a great proportion of the native applications.

The distinction of a "native" and a "web" application can be unclear. In this thesis, by *native applications* I mean applications that are not run in a web browser but as individually packaged containers, often identified as "traditional applications". By *web applications* I mean applications that are run in a web browser environment and are built with web technologies, primarily with HTML, CSS and JavaScript. What might be confusing, however, is that web applications can *also* be packaged as native applications. Such applications are often referred to as *hybrid applications* and besides

being distributed like native applications, they can utilize the native APIs (Application Programming Interface) of the operating system they are run on. Contrary to web applications such applications have the benefit of using many of the features that modern devices offer that web browsers do not, such as accelerometer, flash light or access to native features like notifications that are used in many modern mobile operating systems.

## 1.1 Objectives and research questions

The primary objective of this thesis is to evaluate the web as a platform for applications rather than as a traditional document platform. Also, being a requirement for modern web applications, the suitability and challenges of the single page architecture will be assessed.

We know that HTML was designed document-based web in mind, rendering certain problems in creating application-style web pages. In this thesis, I examine what these problems are and what practical means there are to overcome them.

1. What are the main challenges and advantages of building and distributing web applications?

2. How does single page architecture solve or relate to these challenges and advantages?

3. What practical means do we have to build single page applications?

This thesis provides a comprehensive overview of current state of web technologies, such as HTML5, CSS3 and JavaScript. I evaluate the suitability of those technologies for building applications that might replace native applications. Also, I examine concepts and techniques that are not commonly utilized in traditional web sites but solve relevant problems in applications.

One of the motivators behind single page architecture is the need to enhance the user experience in the web, thus usability of web applications is also studied. I review what performance and User Interface (UI) related problems appear in web applications and how those are solved.

## 1.2 Structure of the Thesis

In Chapter 2 I present the architecture, challenges and advantages of web applications in general. Chapter 3 provides an overview of web technologies

that are related to building complex applications. Next, in Chapter 4 I introduce the concepts and components of single page architecture and an overview of popular frameworks.

A practical examination is conducted in Chapter 5, where I present three single page applications that I built. I discuss the results of the applications and share my thoughts about future work in Chapter 6. Finally, I conclude the thesis in Chapter 7.

# Chapter 2

# Web as an Application Platform

The evolution of the World Wide Web during its lifetime of 25 years is remarkable. Initially the web pages were simple text-based documents that were connected via hyperlinks. The web evolved gradually towards a rich-content platform lead by the use of plug-in components such as Flash, QuickTime, RealPlayer and Shockwave. They allowed web pages to display content that was not possible otherwise. The web pages became increasingly interactive and reminded the user more of multimedia presentations than of text documents. [61, 62]

Today, the web is undergoing another evolutionary change: the web is rivalling desktop software as an application platform. Taivalsaari and Mikkonen discuss on their publications of the evolution of the web. They predict that the web will ultimately win the battle of the main platform for end user software, and conventional binary programs will be confined to system software. The future of the software industry and software engineering research will be determined by this battle between web applications and native applications. [41, 61, 62]. Web applications have also been considered one of the greatest interest in the web of the last decade. [51]

In this chapter, I discuss the background of web applications. First, I examine a common structure and related technologies required by the applications. The technologies will be overviewed in more detail in the next chapter. Second, I examine the challenges of the web as an application platform and finally take a look at the advantages provided.

## 2.1 Architecture of Web Applications

First, it is important to make a distinction between a web page and a web application. By *web pages* I mean classic pages that are rendered on the

server and typically reload the page on most or all requests – even if they behave or the functionality is application-like. By *web applications* I mean "pages" that are loaded only once and utilize the server primarily just for exchanging data. In this thesis, I focus solely on web applications.

Despite the standardized components of the web, a web application can be built in many different ways. Contrary to many native application platforms that provide dedicated libraries, SDKs, APIs and development environments, the web as a development environment is very diverse. [51] It is up to the the developer to select a development stack as they like, the good being in that there are almost no restrictions on how to design the architecture.

One way to identify application architectures is to divide the application to tiers by its fundamental areas: data logic, business logic and presentation logic, as shown by Figure 2.1 below. Such tiers can be recognized, e.g.,

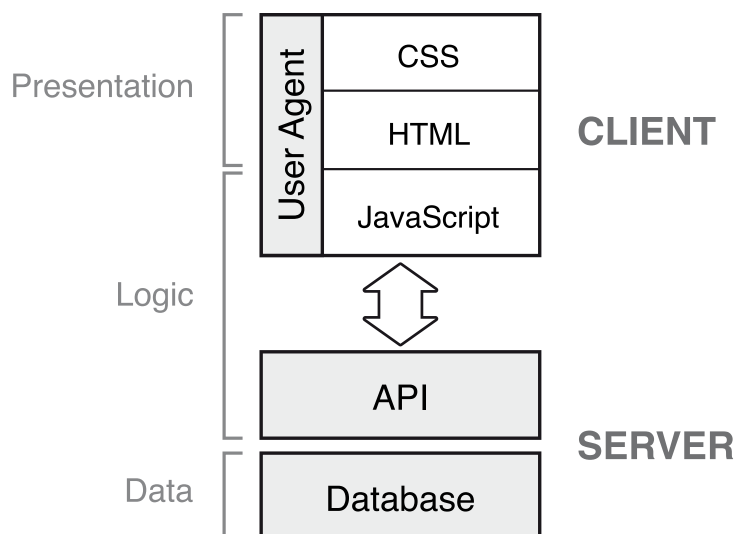


Figure 2.1: 3-tier web application architecture, adapted from [48].

as *1-tier*, *2-tier* and *3-tier* architectures. In 1-tier architecture there is no clear separation in code when it comes to the fundamental responsibilities, whereas in 2-tier architecture presentation logic and data logic are separated but both augmented with the business logic. 3-tier architecture separates all the areas and is considered superior to the former two considering many aspects of building software, including better performance, interoperability and maintainability. [48]

Web applications built with single page architecture follow the 3-tier

model. The application running in the browser is purely the presentation layer, whereas business logic layer can not be distinctly divided to either, but is partly implemented in both. Servers and external services are used for containing and exchanging the data. Thus, page architecture is an architecture for the presentation tier implementation and not comparable to the 3-tier architecture.

### 2.1.1 Presentation tier

The user interface is built with HTML, CSS and JavaScript. New page loads are rendered by the internal browser mechanism, but as is case with single page architecture, only the start page is rendered and further updates are triggered by JavaScript.

HTML is used to define the layout and sections of the web page and CSS defines how they are displayed. Based on the scripts and HTML, browser builds a Document Object Model (DOM), which represents the current structure and content of the web document (page). [20]

### 2.1.2 Business logic tier

As can be seen from the Figure 2.1, the application logic is divided to both: to the client and to the server. In this thesis, I do not examine business logic on server-side, but only on the client-side. Client-side logic is implemented in JavaScript, which is a dynamic language used in all web browsers. JavaScript is examined in more detail in a later chapter.

After the introduction of AJAX (Asynchronous JavaScript And XML) [17] in 2005, the web has gained momentum in separating the business logic from the presentation layer as opposed to classical web. [38] The essential concept in this 3-tier architectural division is the ability to loosely couple the client with the server, allowing the logic to be changed with as little effort as possible. [48]

In single page architecture, the separation of presentation and data layers implies the separation of the DOM and the data. With the help of AJAX, it is possible to build web pages where the data is loaded asynchronously with multiple requests and the application state is retained between those requests. [39] With this model, the DOM is built by fetching the data from external resources, whereas in classical web sites the DOM is built only based on the initial data, effectively HTML, fetched from the server.

### 2.1.3 Data tier

Web applications typically rely on external databases, most likely SQL or NoSQL databases. The data is consumed via an interface like REST or SOAP that serializes and exchanges the data between the database and the client application.

More and more web applications rely their storage on the client-side, namely the browser's, storage capabilities. Technologies like Web Storage [23] for storing structured data and Indexed Database API [37] for key-value data are in standardization process by W3C[1]. [50]

The focus on this thesis is only on the client-side data, which in practice means the fetched data from the server via HTTP requests.

## 2.2 Challenges

Origins of the web in document-oriented information sharing yield a number of issues that hinder the development of applications for a web-browser environment. The over 20-year history of the Web and the rapid development of the Internet has resulted in new technologies being adapted faster by developers than they can be standardized – causing lots of diversification in technology adoption. Some of the practical challenges are identified in more detail below:

**Application complexity**
The Internet web sites range from small, non-interactive personal web sites to large and complex software products that perform advanced business functions. Nowadays, complex web application development needs a diverse team having considerably many fields of expertise. [29, 51]

**Browser semantics**
Mainly due to historical reasons, many of the browser features do not fit to the context of applications. For instance, the buttons used to navigate between pages, namely 'back' and 'forward' buttons, have no semantic meaning in many applications where the state is persitent. Also, browser's generic context menus rarely provide any useful functionality. [62]

**Same-origin policy**
Web browsers implement a security-related policy that is used to pre-

[1] http://www.w3.org

vent some of the Cross-site Request Forgery (CSRF) attacks. For example, a web application loaded from a web-site can access resources from the same domain, but not arbitrary resources from other web sites. Thus, it prevents reading data from external resources also on purpose, which is often necessary when fetching data from a REST API for instance. To overcome this, the web application developer must make special arrangements on the server, like set up a proxy to access the web site. A better mechanism, called Cross-Origin Resource Sharing (CORS), has been recommended by W3C. It provides a way for web servers to support cross-site access, but is not controllable by the application developer. [55, 62, 63]

**Software engineering principles**

Again, in the early web the primary conceptual components of web sites were documents, pages and forms. Thus, it was not software engineering in the same sense as it is today. This has resulted in the lack of software engineering principles like *modularity and reusability*, *separation of concerns*, *well-defined (manifest) interfaces* and *information hiding*. [40]

Compared to conventional applications, the web is very dynamic by its nature, which causes fundamental changes in the development style. Other languages like C++, C# and Java have static type checking and generally well-defined interfaces, whereas JavaScript lacks these completely. The web applications need a stepwise approach[2] and an exploratory programming style, which mainstream software developers are often unaware of. [62]

**Performance**

Previously, the performance problems of web browsers were primarily related to online connectivity and network latency issues. However, today, as the web serves more and more application-style websites, the performance issues of the browser also as a runtime environment has become apparent. [62] The issues have been tackled by the development of JavaScript engines, one of which has become so fast that it is used even in server-side applications on a platform called Node.js[3]). However, a single fast JavaScript engine does not solve the performance problem, as many times the application must be runnable even

[2] In stepwise approach to programming, the program is developed gradually in steps. Each step is decomposed from one or several instructions and is broken up into a number of subtasks. The program description and the implementation can be refined in parallel.

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

on the slowest JavaScript engines. Additionally, the perceived performance is not only determined by raw JavaScript execution speed, but by browser's properties including the operation speed of DOM interaction, rendering speed of graphics like SVG, multi-threading and utilization of GPU. [43] Also, the memory management capabilities of JavaScript virtual machines have been denoted to have poorly suitable to large, long-running applications. [62]

**Fragmentation**

In an ecosystem of native applications, fragmentation typically means the phenomenon when the developer has to maintain multiple codebases or have custom builds for different devices and platforms. This adds unnecessary complexity and the ideal situation is always to have a single codebase with a straightforward build and distribution process. In practice, the fragmentation is apparent especially in mobile applications, where the developer has to customize the application to work on Android 2.x and Android 4.x operating systems.

Web applications are not subject to similar fragmentation as conventional applications. The browser environment is considered universal and the same code is supposed to work in every browser. Instead, the fragmentation challenges are related to browser diversity, rapid development of technologies and the fact that the web browser has not initially been designed to be a runtime environment for applications. The amount of different frameworks and libraries built and being built for web is significant. The fact that there is a plethora of options regarding the technologies may cause new application to be built with unfamiliar technologies and be difficult for developers to adapt to.

**HTML semantics**

HTML5, of which the final version was published in 2014, added new semantics and APIs for complex web applications. It included important features like `<audio>` and `<video>` elements. [24] Also, it included semantic replacements for generic block elements, including `<nav>` and `<footer>` elements. However, it has been criticized that HTML is not suitable for constructing *application* user interfaces due to the fact the elements are designed to represent document-oriented structure rather than a UI layout. [5] It can be argued, though, that HTML is well suitable for the web due to its focus on content rather than presentation leading to better adaptation to different screen sizes. [16]

There are attempts to solve the semantics issue in a standardized way.

Web Components[4] is an umbrella term for technologies being standardized to allow defining custom elements. Polymer[5] is a technology being developed by Google built on top of Web Components. It adds further functionality to custom elements, such as templating and two-way data binding. However, the technologies are still relatively new and are not recommended for production use.

**Monetization**

A lot has changed in the evolution of distribution and monetization model of software since the early days of the Web. Still in the 1990s, the primary way of monetizing applications was to offer them to be purchased by CDs on the shelves of physical stores. Some of the applications needed a unique serial number to unlock the application. In 2000s, the applications were also sold and downloaded online from web sites. 2010s was the revolution of dedicated application stores like Apple's App Store and Google's Play Store.

Compared to conventional applications, web applications face a challenge of proper monetization in order to be commercially viable. The web applications are available for everyone on the web – theoretically for free. Nevertheless, the current trend is towards Software as a Service (SaaS) monetization model, which requires users to register to the application as a service and pay a periodic fee. Also, the web applications could be packaged as native applications and distributed in the application stores, or they could use advertising as the monetization model. The monetization model of web applications is not essentially a problem, but merely a challenge.

**Distribution and promotion**

Non-trivially, there are certain challenges related to distributing and promoting web applications. Consider application stores, where the store provider actively validates the uploaded applications, promotes the top applications and the users write reviews that help other users to evaluate the applications. As of today, there is no such centralized organization conducting that for web applications. Finding and validating web applications is a challenge for the end users.

**Network dependency**

Most of the web applications today require an Internet connection in order to work. Nonetheless, certain applications, like utilities and tools,

[4] `http://webcomponents.org`
[5] `https://www.polymer-project.org`

should also work offline in order to compete with native applications that are not network dependent.

## 2.3 Advantages

The largest advantage of the web as an application platform is the immense popularity of the web and the web browser. All major operating systems ranging from mobile to desktop devices include a web browser by default. The applications deployed to web can be considered to be instantly accessible all over the world. Once the application is deployed or upgraded, it is immediately available. Thus, there is no need to manually download, install or upgrade web applications, which is almost an unfair advantage compared to conventional applications.

Traditional applications – before the time of application stores – were often developed for long before releasing. It was not uncommon to have multiple months, or even years long of release cycle. This lead to problems when critical bugs appeared in the software. The bugs could be patched, but distributing the patch, informing and making the users install it would be challenging. Application stores takes a leap forward in this sense, as the stores notify users of updates and streamlines the install process. Yet, the developers have to often wait for the approval of the update to be available at the application store, slowing the process. In the web, there is no middlemen in distributing and the bug fixes can be virtually instant. It is up to the developer to decide when and how often to update the application. For example, a Finnish newspaper "Kauppalehti"[6] deploys their single page web application four times a day by average. [46]

Web applications are suitable for agile development style. The process style of building web applications tends to be spiral-like, a continuous iterative process between implementation, testing and maintenance. The dynamic language qualities of JavaScript fit to such rapid and light development process. [51]

The web is cross-platform. Conventional applications have the challenge of supporting different types of operating systems, devices or CPU architectures [62]. Theoretically, web applications can be run on any device and on any browser. Web applications are not without challenges, however. In order to work on a plethora on devices and browser, they need to be compatible with different browser versions and even different versions of web technologies like HTML. They need to adjust to different screen sizes in order to be

[6] `http://www.kauppalehti.fi`

usable. Despite these challenges, the fact that there is no need to write the same application with multiple languages and development kits, is a great advantage.

# Chapter 3

# Overview of Web Technologies

In this chapter, I present an overview of the web technologies that are related to building applications for the web. These technologies include those that are often referred to as "Web 2.0 technologies", intended for the creation of collaborative and interactive, rich desktop-style applications, as identified by Mikkonen and Taivalsaari. [62]

Although not being the focus of the thesis, it is crucial to understand the technologies in order to make sense of the concepts that differentiate web applications from traditional web sites and provide functionality familiar from conventional applications. I evaluate the role and importance of these technologies regarding building modern web applications.

I have divided the technologies into four categories. In the first three sections I present the base components for every website and web application. In the fourth section, I present additional APIs that have been standardized or are in common usage across the web.

## 3.1 HTML5

HTML is the markup language used on all web pages to define and describe the documents displayed by the web browser. HTML5 not only implies the markup language, but it is often used to refer to the set of features and APIs released along with HTML5 specification. The relevance of HTML5 related to web applications can be assessed by the fact that due to historical reasons: it was originally named as "Web Applications 1.0" by WHATWG (Web Hypertext Application Technology Working Group)[1] until renamed to "HTML5" by World Wide Web Consortium (W3C)[2]. [49]

[1] http://whatwg.org

[2] http://www.w3.org

HTML5 is the successor of HTML 4.01, which was the previous standard released in 1999. The first draft of HTML5 was published in 2008. The final recommendation of the specification by W3C was not released until October 2014, but HTML5 was in general use long before that. The focus of developing HTML5 was specifically towards web applications, which was thought to be most lacking. In the end, HTML5 addresses lots of practical problems and supports building dynamic and social sites that require various features. [34]

In this chapter, I take a look at the HTML5 and related technologies and evaluate them from the application point of view. It is hard to define which technologies – or which only – are clearly related to HTML5, but this overview focuses on those that are important for web applications.

### 3.1.1 Elements and semantics

Elements are the most central component of HTML. They are used to describe the document and its content. The elements are associated with certain *semantics*, which means that the element has certain meanings. For example, the `h1` element represents the highest level heading in the document. [24] The intended use of elements is to use them only for the correct semantic purpose, but web applications often violate this principle. As I discussed in the Challenges section, web application developers want to describe UIs rather than documents. It is an open question whether this kind of violation is actually harmful. The semantic correctness is useful when the content is indexed by a search engine, for example. However, the nature of web applications is often not to present content in a traditional way and the content of such applications is not usually linkable.

The HTML5 standard divides elements into zero or more categories grouped with similar characteristics of the elements. [24] The categories and example elements are presented below.

- **Metadata**: Elements located typically in the `head` sections. Sets behaviour and presentation of the document and relations to other documents.

  *Examples:* `link`, `script`, `style`, `title`

- **Flow content**: Most elements used in the body of documents and applications.

  *Examples:* `a`, `button`, `div`, `form`, `span`, `table`

- **Sectioning**: Elements that define sections, i.e., scopes of other elements like headings and footers.

*Examples:* `article`, `asude`, `nav`, `section`

- **Heading**: Elements that define headers for sections in the section's scope.

  *Examples:* `h1`, `h2`, `h6`

- **Phrasing**: Text elements and other elements that phrase content.

  *Examples:* `cite`, `code`, `small`, `sub`

- **Embedded**: Elements that import another resources into the document.

  *Examples:* `audio`, `video`, `canvas`, `iframe`

- **Interactive**: Elements that are specifically intended for user interaction.

  *Examples:* `audio` and `video` with `controls`, `button`, `input`

### 3.1.2 Media support

HTML5 added two new media elements: `<video>` an `<audio>`. Both of them support basic operations like play, pause and mute/unmute. Also, the elements can be controlled programmatically in JavaScript. [34]

Previously web pages had to rely on plug-in components, such as Flash and QuickTime, to play media content. The plug-ins had to be separately installed and the support from different browsers was inconsistent. HTML5 playback is being adopted by many media-focused companies including YouTube and Netflix, both of which have transitioned to HTML5 videos as the standard playback option. Moreover, at the time of launching iPhone, one of the revolutionary devices regarding the Web, Apple famously announced to stop supporting Flash on its devices – HTML5 most likely being the motivator. The standardized media playback is without question an important feature that enhances the user experience of the web browser since no additional plug-ins installs are needed.

### 3.1.3 Canvas

HTML5 defines the `<canvas>` element, which is essentially a way to draw advanced two-dimensional graphics programmatically. Its intended use is in generating charts, composition images and animations. In practice `<canvas>` is a rectangular bitmap area on the page among the other elements. The drawing API provides a way to draw simple shapes, text, paths, images, gradients and ways to modify their properties. [34]

### 3.1.4 Drag-and-drop

Drag-and-drop is an interaction paradigm where the user can move an element freely within the user interface, by dragging and dropping with the mouse. With this API, it is easy to create advanced user interfaces without third party JavaScript libraries.

Drag-and-drop is often considered a HTML5 feature, as it appeared in the draft versions of HTML5 specification. Nonetheless, the final specification was postponed to HTML5.1 specification that is yet to be released. Drag-and-drop support is, however, implemented in all major desktop browsers – even IE 8 [1]. Thus, it is included here. Also, it is worth noting that drag-and-drop is not an user interaction paradigm on mobile devices.

### 3.1.5 Web Messaging

Web Messaging or cross-document messaging specification allows documents to documents to communicate with each other. For example a document containing an `iframe` element pointing to an external web site, could send message to the document loaded into the `iframe` The communication is not limited by a same-origin policy. Thus, use of the API needs extra care to protect users from abuse. [21]

### 3.1.6 Browser history management

HTML5 brings a significantly important feature from the single page architecture point of view: Browser History API. The API allows to manipulate the navigation history programmatically without the need of interacting with browser navigation controls [24]. This allows to single page applications to simulate the transition in history, although technically, they never move back-and-forth between pages.

### 3.1.7 Offline Web Applications

Classical web sites are designed to work online and are seldom concerned by support. Web applications, however, should in many cases work also offline. The user might move around drop the connection occasionally or the network connection could be unreliable for an unknown reason. Such situations should be tolerated by the application.

In HTML5 applications, this is addressed by application cache and browser state detection. The application cache is created by defining *cache manifest*, which is a simple text file that lists the resources that need to be accessed

offline. HTML5 also defines functionality to detect current network status and the events fired on the status change. [64]

## 3.2 Cascading Style Sheets

Cascading Style Sheets (CSS) is the styling language for the Web. It is used to format the layout and elements on the web page written in HTML. The latest version of CSS is CSS3 (CSS Level 3). The previous standardized version is CSS2.1 and since then CSS3 has brought lots of new and crucial features to build web applications. CSS3 supports features like gradients, transitions, animations, grid layouts and custom fonts.

CSS can be even more powerful used in conjunction with a *CSS preprocessor*. The current trend is not writing raw CSS, but to compile to it by using more advanced languages that introduce features like modules, variables, maps, functions, mixins and loops. [31, 57] The most popular preprocessors are arguably SASS (Syntactically Awesome Style Sheets)[3] and Less[4]. Preprocessors are very useful in styling large websites and applications due to the possibility for more declarative code with the help of modules and variables.

Cross-browser support is one of the major challenges in CSS. Given a combination of HTML, CSS and browser parameters like viewport size, every browser is expected to render it equally. In practice it, however, is not the case. No browser implements all CSS features or they implement special non-standardized keywords. This adds to the developer's overhead and is one of the problems solved by preprocessors.

Considering building web applications, CSS3 has really some important features. Web applications often have to work on multiple devices, resulting in a plethora of screen sizes and resolutions. Media queries feature can detect the view-port size and specify style rules accordingly. [32] Fonts specification[5] allow developers to easily integrate custom fonts, including traditional typefaces and icons. Animation support[6] allows the applications to take advantage of the browser's hardware acceleration and be used to create UIs that compete with desktop software.

[3] `http://sass-lang.com/`
[4] `http://lesscss.org/`
[5] `http://www.w3.org/TR/css3-fonts/`
[6] `http://www.w3.org/TR/css3-animations/`

## 3.3 JavaScript

JavaScript is used everywhere in the web and works on all modern web browsers – on desktop computers, tablets, smart phones, televisions, game consoles. A majority of all websites utilize JavaScript and the software industry is shifting towards web applications built with JavaScript instead of classical languages like C# and Java. Moreover, some operating systems have adopted the web standards as the presentation layer for native applications, including Windows 8, Firefox OS and Chrome OS. Therefore, JavaScript is considered one of the most important and most popular languages in the world. [9, 12, 15]

JavaSript was originally developed at Netscape and the name "JavaScript" was trademark licensed from Sun Microsystems. ECMA (European Computer Manufacturer's Association) standardized the language, and due to trademark issues, the standardized version of the language was finally called "ECMAScript". However, because the language is so widely known as "JavaScript" and that is the term practically everyone uses, we mostly call the language also in this thesis "JavaScript" instead of "ECMAScript". It is also worth noting that, despite the name, JavaScript has nothing to do with Java. [15]

The current widely adopted version of JavaScript is ECMAScript 5 and it is defined in the Standard ECMA-262[7]. Version 4 was never released and ECMAScript 3 was the previous major release.

ECMAScript is a high-level interpreted language that suits well to object-oriented and functional programming styles. [15] It has been argued that the language have some badly designed parts but also well designed ideas including loose typing, dynamic objects and expressive object literal notation. [9]. JavaScript supports features like encapsulation, polymorphism, multiple inheritance and composition. [12] JavaScript was once considered a toy-like language, but it has outgrown the roots of a mere scripting-language and is today considered an efficient general-purpose language. [15]

JavaScript does not only have lots of expression power, but despite its just-in-time compilation in the web browser, it is also very performant. The JavaScript programs are event driven and non-blocking, which compensates for performance overhead derived from garbage collection and dynamic binding. In practice, JavaScript is also the only language in the world where the same code can be run in both the web browser and the server. This has been enabled by the JavaScript environment called Node.js[8], that is built on

[7] `http://www.ecma-international.org/publications/standards/Ecma-262.htm`

[8] `https://nodejs.org`

Google's V8 JavaScript engine. [12]

During the time of writing this thesis, there has been a buzz about the next version of ECMAScript. The new version is called "ECMAScript 6" and has also been informally referred to as "JavaScript 2015". The final version of the draft was released on April 14th 2015 and will be voted for approval on June 2015. The feature-set of the new ECMAScript is frozen, but minor editorial and bug fixes may still be made. [27]

The reason why ECMAScript 6 brings so much excitement among the developer community can be understood when looking at the specification: it brings many new useful features. Some of the most important new features are listed below [11, 25]:

- **Arrow functions.** ECMAScript 6 introduces a shorthand `=>` to define arrow functions, which are lexically scoped functions in either expression or statement bodies.
- **Classes.** ECMAScript 6 defines classes that are familiar to developers from more classical languages like C++, C# and Java. However, the prototypal inheritance is not changed – classes are merely a syntactic sugar.
- **Modules.** ECMAScript 6 brings a language-level support for component definitions. It allows developers to encapsulate their code and define dependencies (imports). Until now, the only way to load other modules inside code was to rely on 3rd party implementations of specifications like CommonJS[9] and AMD (Asynchronous Module Loading)[10].
- **Data structures.** The new standard defines new data structures for common algorithms: **Map**, **Set**, **WeakMap**, **WeakSet**. They provide some useful features, such as arbitrary values for keys in Map, and can be used to improve memory management.
- **Block scoping.** ECMAScript 6 introduces a new keyword `let` to define a block scoped variable. As opposed to using `var`, `let` helps to overcome confusion often associated with JavaScript's feature called *hoisting*.
- **Proxies** enable defining custom behavior for fundamental operations into objects, like interception, virtualization and logging.

[9] `http://www.commonjs.org`
[10] `http://github.com/amdjs/amdjs-api/wiki/AMD`

- **Promises** is a new library natively provided by ECMAScript. Promises are important for the asynchronous programming pattern present especially in web applications. Promises have been implemented by many existing JavaScript libraries.

- **Unicode.** ECMAScript 6 brings non-breaking additions to support full Unicode, which is a great help in building global applications in JavaScript.

The new features will address some of the major challenges that we already discussed in the challenges section. Consider complexity, software engineering principles and performance challenges for example: the improved modularity, classes and data structures in ECMAScript 6 will most likely affect the way web applications are built in a major way. The chances are that the new ECMAScript definition will streamline and unify web application development in general, very good.

## 3.4 Related specifications

### 3.4.1 Web Storage

Traditionally the data stored on the browser by the accessed web site has been limited to cookies. Web Storage specification addresses the issues that arise in web applications when needed to overcome the limitations of cookies, such as the size limitation. The maximum size of a cookie is generally only 4KB. They also have another problem: cookies are transmitted back and forth on every request. It is a potential security risk when the connection is not encrypted. [34]

Web Storage API provides a simple way to store and retrieve JavaScript objects. The developer can choose whether to use `sessionStorage` or `localStorage`, which either persists the data between sessions or does not, respectively. Moreover, Web Storage supports values as high as a few megabytes, which makes it more suitable for storing document and file data contrary to cookies. [34]

### 3.4.2 WebSockets

Conventionally the client-server communication on web applications has been limited to HTTP (Hypertext Transfer Protocol). HTTP is a stateless and synchronous protocol that operates on top the TCP/IP protocol stack. Nowadays, there is a need for two-communication between the clients and the

servers. For example, the server might want to send push notifications to the client or the client would want wait for an asynchronous answer from the server to a request. However, HTTP was not originally designed to maintain long-living bi-directional data streams. It has been worked around by XHR-polling (XMLHttpRequest Long Polling), which is essentially a way to periodically ask the server if there is a message. The problem with XHR-polling is the overhead caused by multiple redundant HTTP requests. Each HTTP request includes effectively the same header information and requires a new TCP connection to be opened every time. [2, 59]

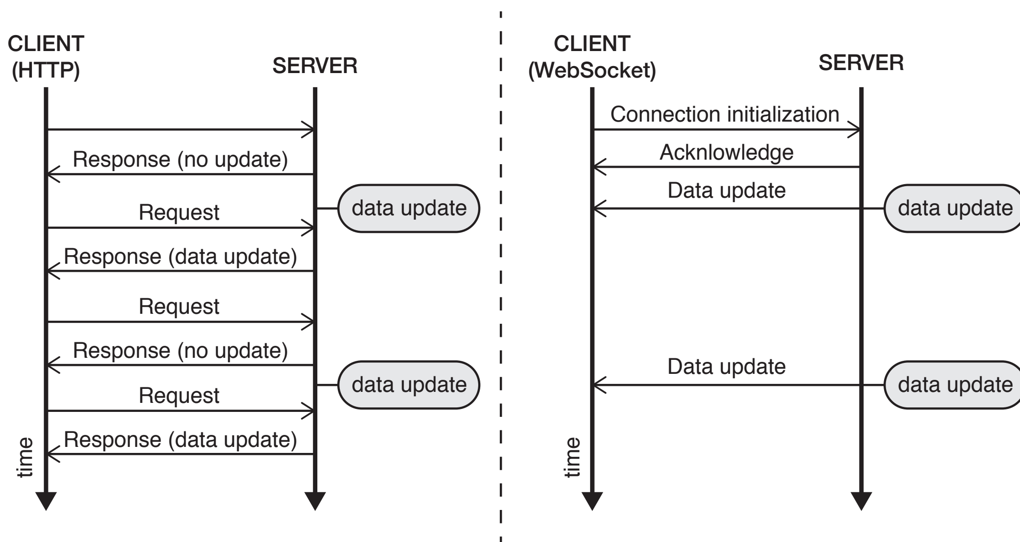


Figure 3.1: HTTP and WebSocket communication patterns compared, adapted from [2] and [59]. HTTP requests on the left, bi-directional WebSocket messages on the right.

Websockets were designed to overcome many of these limitations. Websockets are initiated through an HTTP request, which is then upgraded to a websocket connection by a special upgrade-message. HTTP and WebSocket communication patterns are compared in figure 3.1. The websocket connection is run on top of a long-lived TCP socket and allows bi-directional communication. [2] Compared to HTTP packets, the overhead of packets sent over a websocket connection is considerably smaller. Websockets can provide a 500:1 to 1000:1 reduction in header traffic and 3:1 reduction in latency. [35] The WebSocket API[11] have been standardized by W3C and the

[11] http://www.w3.org/TR/websockets/

websocket protocol is defined by IETF RCWEB group in IETF RFC 6455[12]

Due to the low overhead in traffic and low latency, websocket are very capable of handling realtime traffic. Thus, it is a suitable communication protocol for client-server-client pattern messaging focused applications, such as chats, games, logging and monitoring. From the web as an application platform perspective, websockets can be considered a major technology in bringing web applications closer to desktop applications which have been able to utilize raw TCP/IP connection for high-speed and realtime communication for decades.

### 3.4.3 Web Real-Time Communication

W3C is working on standardizing Web Real-Time Communication (WebRTC) API, which introduces peer-to-peer (P2P) connections to web browsers. WebRTC enables data and media streaming in a stateless fashion (consider UDP in contrast to TCP), which is suitable for streaming video and audio, for instance. The RTC architecture is on direct client-to-client communication and server as the mediator, as shown on figure 3.2. [4, 33].

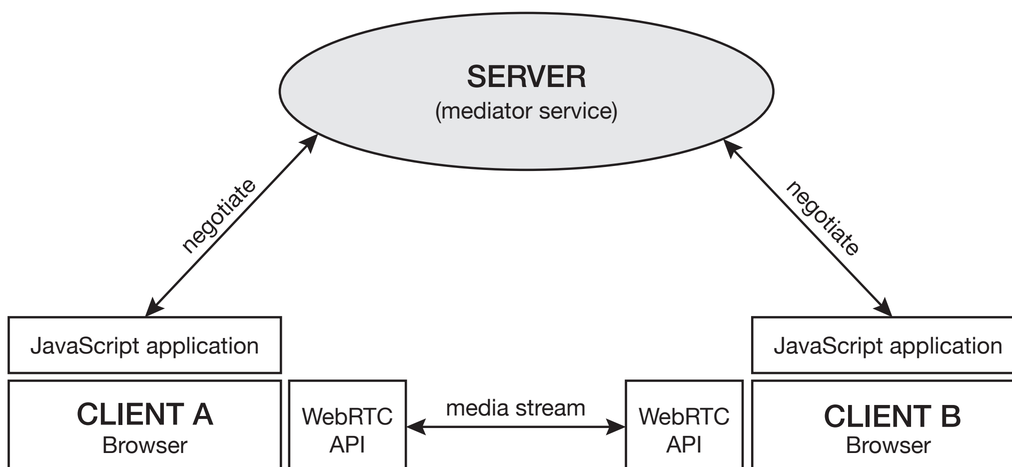


Figure 3.2: A typical WebRTC architecture between clients and a mediator service.

The WebRTC standard allows web browsers to interact with devices like microphones, webcameras and speakers. It allows to record, display and show media to or from local devices, which makes the web browser powerful regarding rich internet applications. The challenges are in security and

[12] http://tools.ietf.org/html/rfc6455s

privacy – the user has to have the control when a local device is recorded. Yet, there is also the possibility use this technology to *improving* security and accessibility: consider face and voice recognition technologies.

### 3.4.4 Touch Events

Conventionally the web was browsed with a pointer device and a keyboard, but today the web is used more and more with touch-based devices like tablets and smartphones. In order for the web to serve as an application platform, it has to adapt to the underlying operating system from the user experience point of view. This has caused the need to support new interaction functionality, such as swiping and pinching, also in the web browser.

The Touch Events specification introduces an API to capture events for multi-touch interaction. It defines simple events such as `touchstart`, `touchmove` and `touchend` that allow developers take advantage of the multi-touch gestures. [58]

### 3.4.5 Web Workers

Web Workers is an API to enable multi-threading in JavaScript applications and take advantage of multi-core CPUs (Central Processing Units). Desktop application have always been able to leverage multiple CPU threads to run, for instance, user interface and a computation process in different threads. This would be beneficial also in heavy web applications to keep resource-heavy operations from freezing the UI. Following the trend of applications moving to web, Web Workers is a crucial API for running advanced applications in the browser.

Web Workers can be utilized by detaching a fragment of JavaScript to run in a separate thread, i.e. "worker". The workers normally include long-lasting background tasks or code that is too expensive to be executed in the main thread. There are also limitations to Web Workers: they cannot access the DOM and they are tricky to debug. [34, 66]

### 3.4.6 Service Workers

Service Workers is a new API underway, which deals with the assumption that network is reachable. It enables to register event-driven scripts independent from the web pages or applications. Service Workers can work as interceptors between page requests and the network, and thus override default network behavior e.g. by generating content when the network is not reachable. The API is also intended to replace the deprecated AppCache

API and support caching for offline use. Service Worker scripts are run asynchronously on separate threads and they can be utilized for features like push messaging, background scheduling and synchronization. [56]

### 3.4.7 Server-Sent Events

Server Sent Events (SSE) is a browser API standardized by W3C. It provides a simple way for the browser to listen to event streams on the server. It partially overcomes the limitation of HTTP: receiving data without polling. However, it does not enable real bi-directional communication, since the standard only allows server-to-client messaging. Contrary to regular HTTP connections, the SSE connection is persistent. [22] It could be utilized, for instance, to send push notifications or real-time data updates.

### 3.4.8 Cross-Origin Resource Sharing

Cross-origin HTTP requests include an `Origin` header, which announces the server the request was sent from. Due to security reasons, it cannot be changed. Browsers implement same-origin policy to prevent CSRF attacks, as we discussed in the previous chapter. This often leads to issues when the web application needs to get access to external resources, such as APIs or a CDN (Content Delivery Network).

Cross-Origin Resource Sharing (CORS) addresses the issue by allowing the server to set allowed origins in the HTTP header. It also specifies a way to negotiate the access control restriction with preflight requests. [63]

It is worth noting that same-origin policiy is merely an agreed policy and is not guaranteed to be implemented in all browsers. Developers should not rely on it being implemented when considering the security of their web applications. A user might use a hacked browser or a simple plug-in to overcome limits set by same-origin policy. Nevertheless, exploiting the policy would only harm the user himself.

### 3.4.9 File API

Accessing and reading local files is often required by applications. In the web browser this is addressed by the File API, which allows asking the user to select files and then read the contents. User can select the files by either using a modal initiated by HTML `input` element or by dragging and dropping. The standard provides also a way to follow the progress of loading event by

listening to a ProgressEvent[13]. [44, 53]

### 3.4.10 Geolocation

Geolocation API provides a way to locate the user device based on the underlying device API. The device implementation could use GPS (Global Positioning System) signals or a database based on Wi-Fi signals to locate the user, for instance. The geolocation API on the browser is, however, agnostic to the underlying implementation. [52]

### 3.4.11 Scalable Vector Graphics

Scalable Vector Graphics (SVG) standard is an XML-based image format for two-dimensional graphics. It supports interaction and animation with JavaScript. SVG can be used to draw many types of graphics, including basic shapes, paths, text, fills, strokes, gradients, patterns and filters. Additionally, it supports masking and embedding bitmap images. [10]

SVG reduces the need of embedding bitmap images to web pages or applications. Especially, during the era of mobile phones and high-resolution displays the ability to include and display images in a vector format is exceedingly important. Web developers often want to build UIs that look and behave the same regardless the screen resolution. Without a scalable vector format that is achievable merely by including multiple versions of a single image, that are sized accordingly to the screen resolutions. SVG eliminates the need, as it can be scaled to any screen resolution in a pixel-perfect manner. [7]

### 3.4.12 Web Audio

The history of playing audio in the web browser has ranged from browser-specific, non-standardized implementations to third party plug-ins like Flash. HTML5 introduced the `<audio>` element but it has many limitations considering advanced audio in web applications: it has no timing controls, limited number of sounds can be played at once, no pre-buffering and no ability to apply real-time effects. [60]

Web Audio is a low-level JavaScript API to overcome the limitations of the `<audio>` element. It allows audio to be loaded and played with JavaScript, without an element in the DOM. It defines advanced functionalities, like adjusting timing, latency, volume and pitch, and also combining

[13] Progress Events. W3C Recommendation. `http://www.w3.org/TR/2014/REC-progress-events-20140211`

multiple different audio files. Moreover, it allows sound synthesization and processing directly in the browser. [60]

### 3.4.13 WebGL

WebGL (Web Graphics Library) is a low-level API for rendering 3D graphics in the web browser. It enables developers to include 3D context in the HTML with pure JavaScript, that was previously impossible without third party browser plug-ins. WebGL works on majority of the desktop browsers and in a growing number of mobile browsers. [47]

The WebGL library is developed and maintainer by the Khronos Group[14]. Major browser vendors including Apple, Google, Mozilla and Opera belong to the WebGL Working Group. [28] The WebGL 1.0 specification[15] was released in 2011 and it is based on OpenGL ES 2.0 (Open Graphics Library for Embedded Systems). The WebGL 2.0 specification[16] is currently in the making and it will be based on OpenGL ES 3.0.

WebGL uses the `<canvas>` element introduced in HTML5. Thus, being rendered on a regular HTML element, 3D rendering can be combined with other web content. The underlying OpenGL is abstracted in a JavaScript API that is used in conjuction with the typed arrays. Typed arrays specification[17] was created to allow better memory management and performance for resource-heavy WebGL applications. [47]

Web browser make a very powerful 3D platform. JavaScript and WebGL are cross-platform technologies, meaning the 3D can be presented on a vast amount of different devices, ranging from mobile phones to televisions. WebGL being designed for web, it is easy to use and hassle-free for the end users. It is expectable to be adopted in advanced user interfaces and virtual reality applications in the near future. Improving the visual interfaces have been proven to be an effective way to add value to a web service, especially in the domains of entertainment, learning and commerce [6].

[14] http://khronos.org
[15] http://www.khronos.org/registry/webgl/specs/1.0/
[16] http://www.khronos.org/registry/webgl/specs/latest/2.0/
[17] http://www.khronos.org/registry/typedarray/specs/latest/

# Chapter 4

# Single Page Architecture

Despite the popularity of the web, web applications often suffer from inferior interactivity and responsiveness compared to native applications. Prior to single page architecture (SPA), classic web applications needed the entire interface to be refreshed due to the multi-page design pattern. [39] SPA aims to enhance the user experience of web applications by improving the UI responsiveness and interaction.

In this chapter, I present key concepts and components of the single page architecture. Since there is no single right way to implement a SPA, I also present some of the most popular framework implementations.

## 4.1 Key Concepts and Components

This section presents the key concepts and components required by single page applications. The first two, AJAX and REST, are enabling technologies that underlie the frameworks and the rest are common SPA-specific concepts. It is essential to understand these technologies in order to effectively work with single page applications.

### 4.1.1 AJAX

Today, AJAX is the main technology on the web used to make rich client applications. It allows an application to update new data to the DOM without refreshing the whole page. Traditional web applications force users to wait until a response from the server when transitioning to a new view. Allowing users to interact with the page during the process, showing progress and status, vastly enhances the user experience in single page architecture. [65]

AJAX is based on the following core components: HTML, CSS, DOM,

XMLHttpRequest object and JavaScript. The components could be described as follows: HTML and CSS are used for standard rendering process, DOM for dynamic displaying and interaction, XMLHttpRequest for data exchange and JavaScript as the glue and logic. The components are well supported in all major browsers. [65]

AJAX enables developers to create web applications that are based merely on state changes in the client application, contrary to keeping session on the server and rendering static pages accordingly. This is substantially different from the classic synchronous request-wait-response-continue model, as visualized in figure 4.1. Moreover, the amount of data exchanged between browser and the server is reduced, thus resulting in improved responsiveness. [39, 65]

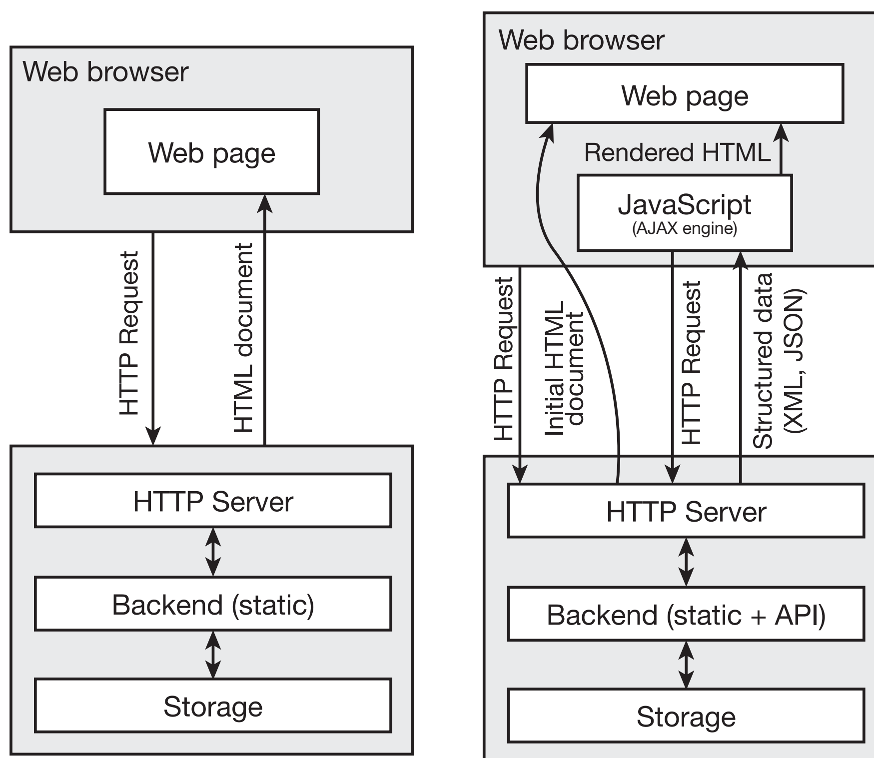


Figure 4.1: Comparison of regular web page requests (left) to asynchronous AJAX requests (right).

AJAX processes are asynchronous and thus they allow multiple operations to be done at once. This helps to increase the interactivity on the client side. [39] For example, user could submit a form, see a progress bar and

continue navigating the page. When there is response from the server, the user would get an acknowledgement of the form submission. This technique could be scaled and used to build immensely complex native-like applications, such as Gmail. Google was one of the first companies to understand the full potential of AJAX. [14]

### 4.1.2 REST

Representational State Transfer (REST) describes an architectural style often used in conjunction with AJAX-powered applications. The idea behind REST is to make the application data be accessible in components, i.e., resources. REST can be described in five constraints as listed below: [19]

1. **Resource Identification.** All resources must use Uniform Resource Identifiers (URI).
2. **Uniform Interface.** The resources are accessible through a uniform interface, which is typically HTTP.
3. **Self-Describing Messages.** A REST interface uses a known and agreed-upon resource format, such as XML or JSON for data exchange. Thus, no individual negotiations for servers and clients are needed.
4. **Hypermedia Driving Application State.** Clients consuming a REST service must follow the links found in the resources. Thus, it is possible to explore the service without dedicated discovery formats.
5. **Stateless Interactions.** The requests must be self-contained, meaning that they need to contain all the information is needed to be included in a single request. This results in statelessness, i.e., there is no state information in the HTTP requests. A state, however, could persist on the client or the server.

Web applications communicate with REST services via standard HTTP requests. HTTP defines action-like semantics for the operations that are called verbs. Each verb has two properties: safety, which implies whether the method may change the resource's state or data, and idempotency, which implies whether it can be assumed that equal multiple requests does not cause side effects different from sending a single request. The HTTP verbs are compared and explained in more detail in the table 4.1: [14]

| **Verb** | **Description** | **Safe** | **Idempotent** |
|---|---|---|---|
| `GET` | Retrieve a resource | Yes | Yes |
| `HEAD` | Retrieve resource information. Effectively the same as `GET` request's HTTP headers. | Yes | Yes |
| `PUT` | Update an existing resource. | No | Yes |
| `DELETE` | Delete an existing resource. | No | Yes |
| `POST` | Add a new resource or request an action on an existing request | No | No |
| `OPTIONS` | Query for the supported HTTP verbs for a resource. | Yes | Yes |
| `PATCH` | Update a part of an existing resource. *Note: this method is not part of the original HTTP 1.1 but was added later.* | No | Yes |

Table 4.1: HTTP Verbs

REST services are beneficial to single page applications, because they allow loose coupling of the back-end (server) and the front-end (client). Consider a company that wants to serve as many customers as possible. They will likely to build a REST service, which can not be consumed only by a web application, but also by native mobile and desktop applications. Classical web applications are tightly coupled with the business logic on the backend and limit the possibilities of extending the service.

An alternative technology to REST is SOAP (Simple Object Access Protocol). SOAP is an enterprise-level technology often coupled with WSDL (Web Service Description Language) and UDDI (Universal Description Discovery and Integration). SOAP-based services have proven to be unpopular among web developers and the use is typically limited to legacy systems. Compared to SOAP, which is a complex and requires orchestrating services, REST is lightweight and supports adaptation in changing environments, i.e., in dynamic web. Lanthaler and Gütl argue in their paper that compared to SOAP, REST-based services can be more scalable, reliable and visible and are the preferred choice for Internet-scale applications. [36] Thus, I focus solely in REST in this thesis.

### 4.1.3 Separation of Concerns

Separation of Concerns (SoC) is a software engineering principle implying modularity: a pattern in architectural design that decomposes software behaviour in encapsulated units. It can be achieved by grouping together logically related elements, which ultimately results in low coupling. [30] Breaking a large application into small units reduces code complexity. In single page architecture, SoC is essentially a Model-Controller-View (MVC) pattern, which is found also in classical web application frameworks. [12]

MVC (see figure 4.2) is a common software architecture pattern that is used to separate the visual data representation from the underlying model. Traditional web applications utilize MVC on the server where the application is written in C#, Java, Python or Ruby, for example. However, modern JavaScript applications transfer the MVC logic completely from the server to the client. [42]

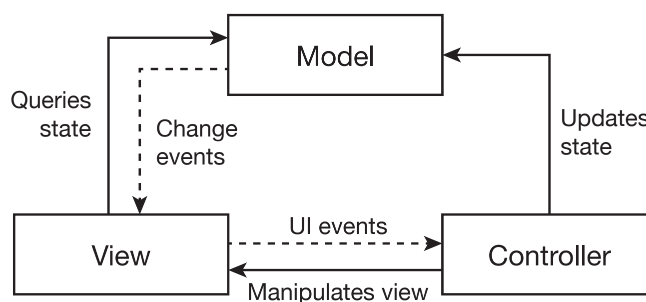


Figure 4.2: Model-View-Controller architecture. [42]

In fact, the development of single page architectures has spawned other patterns as well. MVP (Model-View-Presenter) and MVVM (Model-View-ViewModel) are architectural patterns that are derivatives of MVC. In MVP presentation model is delegate of the data and directs both: view and model. MVVM is essentially MVC used in conjunction with two-way data binding, which I will introduce in the next section. [42] Therefore, I use "MV*" as a broad term for addressing MVC, MVP and MVVM in single page applications.

Implementing MV* pattern in a framework ultimately leads to components including templates, data models and controllers. Some frameworks introduce new features, like "directives" by AngularJS, that further modularize the architecture in smaller MVC components (consider MVC inside MVC). Templates are typically plain HTML, which is written either in separate files or directly into a JavaScript component. Data models are typically

written either pure JavaScript object prototypes or by extending skeleton objects provided by the framework. Controller pattern is usually strictly defined by the framework, which then connects it to related processes like routing and dependency injection.

### 4.1.4 Data binding

Data binding is the process of establishing a connection between the application UI and the business logic. It is one of the core features of any web application that separates the logic in MVC or MVC-like pattern. There are two kind of data binding processes: one-way and two-way data binding. Conventional applications bind data only in one direction, i.e. implements one-way binding. In one-way binding, the view (template) is updated to reflect the model (data). The conceptual difference between these two techniques is visualized in figure 4.3. [18]

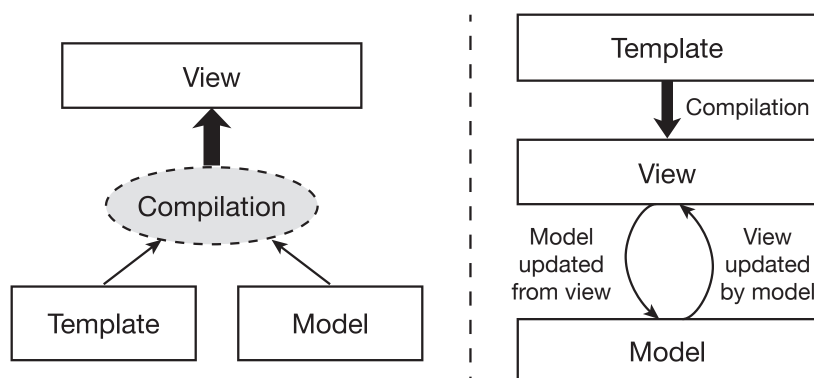


Figure 4.3: One-way data binding (left) vs. two-way data binding (right). Adapted from [18].

Two-way data binding is a technique introduced by modern SPA frameworks like AngularJS and Ember. It allows data to flow to both directions: changes in view update the model and changes in model update the view. In practice this could happen when there are multiple independent components updating the view: if the view is updated by one component, the changes would propagate also to the other components. This is a complex, but powerful technique and allows very rapid development without setting event-handlers to update the view.

### 4.1.5 Routing

Router is a module that delegates requests from predefined URIs to the proper functions. In MV* architectural pattern this is typically a map-structure that defines URI-patterns, assigns them a controller and passes parameters captured from the URI. Listing 4.1 shows an excerpt of JavaScript code as a trivial example to configure routing in AngularJS. It configures two routes: route1 with "template-1.html" and "FirstController"; route2 with "template-2.html" and "SecondController". Route2 additionally parses a parameter "param" from the URI and passes it to the controller.

```
1  $routeProvider.
2      when('/route1', {
3          templateUrl: 'template-1.html',
4          controller: 'FirstController'
5      }).
6      when('/route2/:param', {
7          templateUrl: 'template-2.html',
8          controller: 'SecondController'
9      }).
10     otherwise({
11         redirectTo: '/'
12     });
```

Listing 4.1: Route configuration example in AngularJS 1.3.

Finding resources on the web is based on URIs, thus routing is an essential feature in every web application. This also differentiates web applications from native applications, since in native applications there is no way to save the application state in an URI or an URI-like object. Thus, URIs allow the application state to be saved as bookmarks or to be sent as links to other users. Not all single page applications, however, support saving the application state in the URI. Also, serializing complex states to an URI is difficult and often impossible.

Routing is non-trivial for web applications, since application UIs often implement a "go-back/go-forward" or "undo/redo" functionalities. This means that the application has to retain history, i.e., save changes of state to memory. Many SPA frameworks exploit the History API[1] that was introduced in HTML5. However, the API is not supported by all major web browsers – especially the older ones like IE 8. Some frameworks implement a compatibility mode to support old browsers known as "hashbang mode". It prefixes the URIs with hash and exclamation symbols to allow changing the URI programmatically.

[1] http://www.w3.org/TR/2011/WD-html5-20110113/history.html

## 4.2 Overview of Popular Frameworks

In this section I overview some of the most popular JavaScript frameworks that are designed for single page applications. I look at what has been emphasized in the design of the frameworks and what are their main concepts and principles.

I chose to overview the following four open-source frameworks: AngularJS[2], Backbone[3], React[4] and Ember[5]. The selection was made by considering the following properties: Google trends[6], amount of questions tagged in Stackoverflow[7] and GitHub[8] stars. Google trends evaluate the popularity based on keywords used in searches, Stackoverflow is a very popular question & answer platform for developers and GitHub is a Git-repository provider primarily for open-source projects. Stars in GitHub resemble the amount of people marked the project as their favourite.

| **Keyword** | **Stackoverflow questions (tags)** | **GitHub stars** |
|---|---|---|
| **AngularJS** | **91015** | **37973** |
| **BackboneJS** | **17821** | **21569** |
| CanJS | 170 | 1135 |
| Durandal | 1551 | 1179 |
| **EmberJS** | **14886** | **13538** |
| KnockoutJS | 6318 | 13504 |
| Meteor | 11330 | 24509 |
| **ReactJS** | **2483** | **21142** |
| SpineJS | 180 | 2954 |

Table 4.2: JavaScript SPA framework popularity comparison (measured 28 April 2015).

It is easy to see from the figure 4.4 and the table 4.2 that AngularJS is

[2]`http://angularjs.org`
[3]`http://backbonejs.org`
[4]`http://facebook.github.io/react/`
[5]`http://emberjs.com`
[6]`http://www.google.fi/trends`
[7]`http://stackoverflow.com`
[8]`http://github.com`

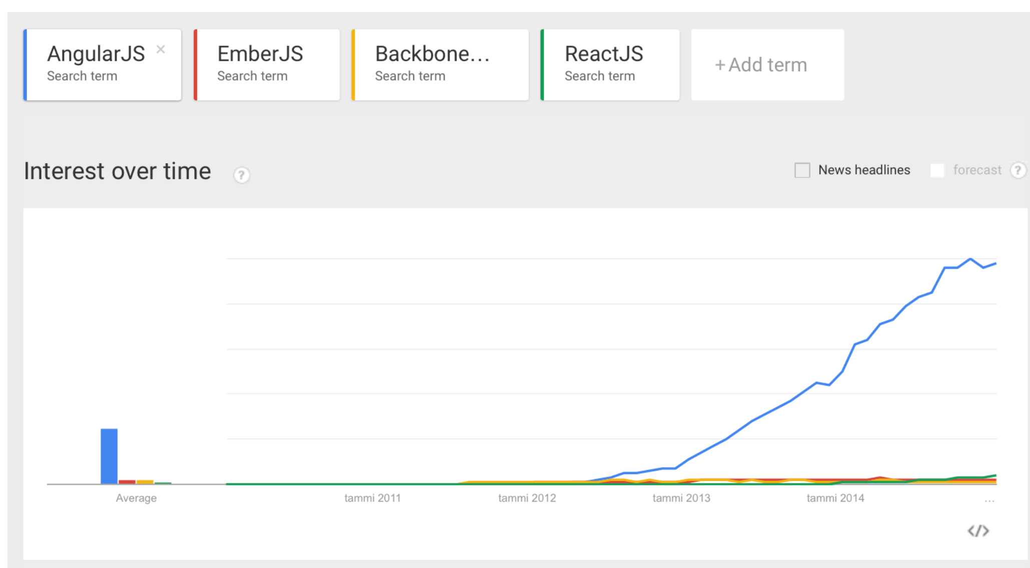


Figure 4.4: Google trends for the selected frameworks (2010–2015). The y-axis represents the percentage of the highest interest rate of given data set.

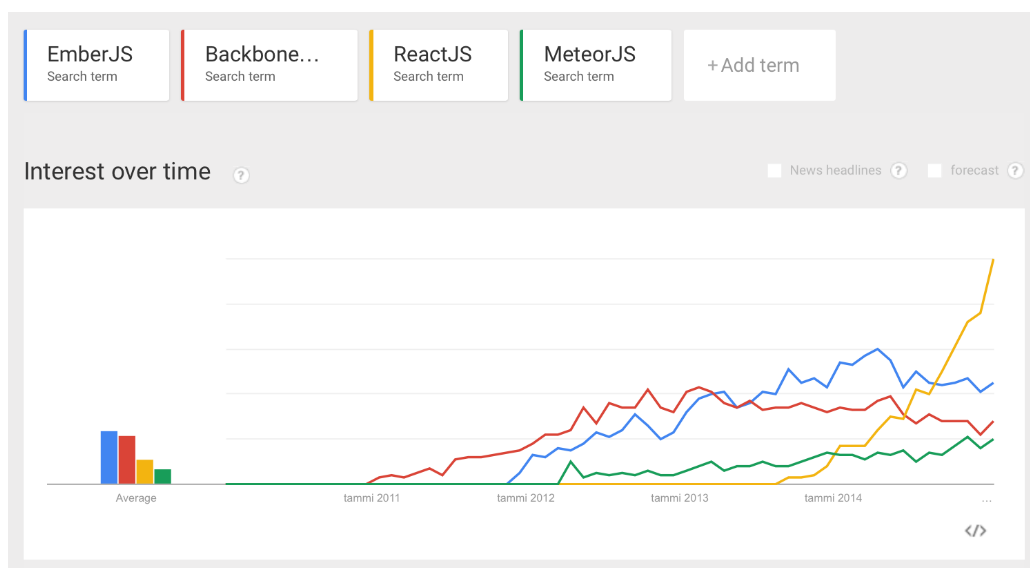


Figure 4.5: Google trends for the selected frameworks (2010–2015) including Meteor and excluding AngularJS. The y-axis represents the percentage of the highest interest rate of given data set.

currently the most popular framework by a large margin. It is also evident, that React has gained much of popularity in a short period of time. Despite the low numbers for Stackoverflow questions, it is on par on GitHub stars with Backbone, which has the longest history.

To clarify the differences between the next popular frameworks, figure 4.4 displays the trend without AngularJS. Meteor[9] is also seemingly popular framework, but is based on very different concepts than the other frameworks. It is more of a platform rather than a framework, that involves also server-side technologies. Thus, it was excluded from the overview.

### 4.2.1 AngularJS

AngularJS is a SPA framework that was first released in 2009. Version 1.0 was released in 2012 and the framework is considered to have become popular in 2013. As of writing the thesis, the newest stable version of AngularJS is 1.3. Version 1.4 is a release candidate version and a completely new version 2.0 is in the works. This thesis focuses solely on version 1.3.

The framework is being maintained and developed by Google. According to the developers, AngularJS is built around the belief that declarative code is superior to imperative when it comes to building UIs, and imperative code is better at expressing business logic. The principles behind AngularJS are listed as follows: [18]

- Business logic should be decoupled from the DOM manipulation logic, which results in better testability.
- Testing should be valued as high as writing the actual code, since structuring the code affects dramatically testing.
- The client app should be loosely coupled from the server side. This makes parallel development of both, client and server sides, possible. Also, it allows reuse of both sides.
- Framework should guide the developer through the whole application building process, including UI design, writing business logic and testing.
- Common tasks should be trivial to accomplish but difficult tasks should not be restricted.

AngularJS introduced many new techniques and concepts that help building single page applications efficiently. New concepts include scopes (for

[9] `https://www.meteor.com`

controller-view data binding), services, providers, dependency injection and directives. AngularJS makes heavy use of two-way data binding, but supports also one-way binding. I review the concepts in more detail in the implementation chapter.

AngularJS is an opinionated framework, meaning it defines rather strict patterns which have to be followed in order to build applications the "AngularJS-way". While it may take a while to learn the AngularJS style, the pre-defined patterns can be beneficial for beginners with little experience. Nevertheless, an agreed-upon structure in an application makes it easy to collaborate on a project if everyone knows the framework. If the framework is not opinionated, many people will have different ways of accomplishing similar tasks, thus there will be collisions in the preferred methods and patterns.

### 4.2.2 Backbone

Backbone takes more of a minimalistic approach compared to a full-featured framework, like AngularJS. It provides many helpful functions to build a single page application, but does not provide an opinionated structure for the application. Thus, it is considered more a library than a framework. It is intended to be used as a foundation for SPA framework or for small applications as is.

The library provides patterns to define object-oriented structures, namely "models" and "collections". Also, it introduces event handling functionality, view models and router with history functionality. [45] To manipulate the DOM, Backbone is almost always paired with a library like jQuery[10] or Zepto.js[11], that make it easy traverse and manipulate the HTML document.

Backbone is often used with an extension called "Marionette", which aims to make it easy to build large-scale applications. Together they form an entirety that is more of a framework in a sense that it provides a streamlined architecture for SPA projects. It adds scalability via modularization, reduces boilerplate code, eases view management and extends some features like routing and event handling. [45]

Behind the scenes, Backbone makes use of a popular library called "Underscore.js" and uses it as basis for many of its features. In fact, Underscore.js is written by the same author ass Backbone. Underscore.js and similar libraries, including Lodash[12] and Lazy.js[13], are often used individually to provide functional features to any JavaScript application.

[10] http://jquery.com
[11] http://zeptojs.com
[12] http://lodash.com
[13] http://danieltao.com/lazy.js/

Evidently,as its name suggests, Backbone is merely a “backbone”. Experienced developers that want flexibility, compose the application from components made by different vendors or want to convert an existing web application to a single page application, will probably prefer Backbone-like framework. However, adapting a low-opinionated framework, will likely require previous understanding of building a single page applications to effectively solve problems that large applications might cause. Thus, beginners might not adapt to Backbone very quickly. On the other hand, Backbone does not introduce such new concepts, like AngularJS does, that might be completely new to a developer and cause a steep learning curve. Nevertheless, a certain benefit Backbone has, is the light weight and small footprint. It is easy to start converting existing applications to Backbone, since it is possible to compose (only) from the features the developer likes and use any existing libraries accordingly.

### 4.2.3 React

React is a very new library maintained and developed by Facebook. The first version of React became available to the public around 2013 when Facebook decided to publish the framework as open-source. Since then it has been adopted rapidly by the developer community. It has caused a lot of buzz, since it is fundamentally different from most of the popular frameworks.

React is not a framework alone, but it has a separate section in this work because of the major way it affects a single page application architecture. It is often considered just the “V” in MVC, and together with a library like Flux[14], it becomes more framework-like. However, even React and Flux together does not form a framework, but more like a set of agreed-upon patterns. Noteworthy to say, React works well along with any other light weight framework, such as Backbone. However, Flux is often preferred, since it is also made by Facebook and designed the React-style architecture in mind.

The fundamentals of React consist of the following three ideas:

- Create simple components that do not mutate the original data.
- Render only what has been changed to DOM. This improves performance,
- Use one-way reactive data flow to make it easy to reason the component relationships.

[14]`http://facebook.github.io/flux/`

A typical application built with React and Flux would consist of components called dispatcher, stores and views (see figure 4.6). Dispatcher is a component that gathers action events and delegates them to views (React components). Views may save only minor UI-related state, but never the actual application data. The data is saved in stores from which the views read. According to the developers of Flux, this kind of style can be called “functional reactive programming”. [26]

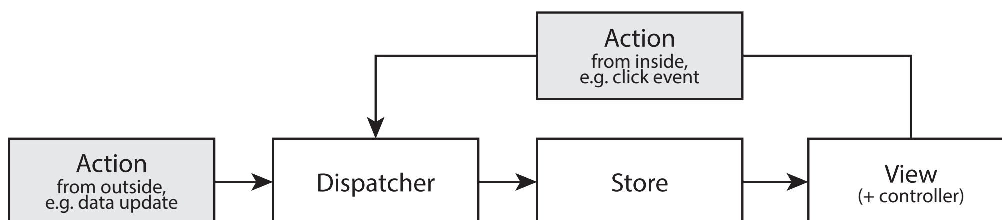


Figure 4.6: Data flow model in Flux+React. [26]

React is very well suited for building reusable components. Its rendering speed is said to be very fast, and thus it would suit also to creating hybrid mobile applications and DOM-heavy components, such as lists with hundreds of items. It can be paired theoretically with any MVC framework and replace the “V”. It is developer friendly due to the very loosely coupled components and a simple data flow mechanism.

### 4.2.4 EmberJS

Ember is a full-featured SPA framework. It is considered very opinionated and it reduces lots of boilerplate code by providing naming conventions. The framework is based on the following core concepts [13]:

- **Templates.** Ember utilizes a templating library called “Handlebars”[15], which supports expressions, outlets (template helpers) and components (custom HTML elements).
- **Router.** The router is used to translate URL into a series of nested templates. Ember automatizes the URL serialization based on models.
- **Components.** In Ember, a component is a custom HTML tag described in Handlebars templates and its behaviour is implemented in JavaScript.

[15] `http://handlebarsjs.com`

- **Models.** Models are objects that store persistent state. Ember implements various helpers for models, including records (pending objects to be saved to the server), adapters (translates requests to a certain server), serializers (turns models intro raw JSON objects), and automatic caching.

The design of Ember is much like in AngularJS. They consist of similar concepts and principles, such as two-way data binding. Compared, Ember is more opinionated and lacks similar module-level dependency injection. Ember has more boilerplate functionality and its concepts are higher level, i.e. more abstract, than in AngularJS. This makes it easy for beginners, but experienced developers will likely want to understand what is under the hood of Ember.

# Chapter 5

# Implementations with AngularJS

In this chapter, I present three single page applications that I implemented using the AngularJS framework, HTML5 and related web technologies. First, I describe the specification and requirements of the applications, and then overview the used tools, techniques and practices. Finally, I present the design and the results.

AngularJS was selected as the framework due to the popularity of the framework, as discussed in the previous chapter. Also, as a full-featured framework, it is well suitable for building applications from the scratch. JavaScript was selected over native technologies to support multiple platforms as easily as possible.

I built the applications for my startup Fresgo, which aims to create new business in the field of food industry. The company focuses in providing a way to buy food by a mobile phone. The product is a web platform, which provides a tablet/desktop application for restaurants to receive orders, and a mobile application for consumers to place the orders.

## 5.1 Specifications

In this section, I present short specifications to give a concrete scope for the implementation. For the sake of simplicity, I name the applications according to the target users as follows: "Restaurant application", "Consumer application" and "Monitoring application".

### Restaurant application

The restaurant application will run on a tablet device, namely on an iPad. We chose iPad as the single supported device, since the company determined the

used devices. Thus, there was no need to support multiple different platforms and devices. Moreover, iPad's hardware and web browser performance have a good reputation among developers. Also, other than development-related properties were considered, such as operating system stability and usability, appearance at a restaurant environment, availability of stands and resale value.

The application should work in an extremely challenging environment: a busy, crowded and loud cafeteria with either a Wi-Fi or a 3G Internet connection. The application should be constantly powered, online and be able to recover from connection drops. The status of the connectivity and orders should be updated to the server in real time. It should notify the restaurant personnel of new orders and reminders with different types of audio notifications. The application should emphasize good user experience, implement a touch interface and prevent erroneous misclicks by design.

Being deployed in multiple locations, it should be effortless to update the clients remotely. This should not require interaction from the restaurant personnel. Thus, the iPad should be always on and ready to receive orders.

### Consumer application

The consumer application should work on iOS and Android operating systems. It should be available in the native application stores, thus it will be implemented as a hybrid application, i.e., packaged as a native application that utilizes web technologies. The application should be usable in various mobile phones and in various screen sizes.

Consumers need to be able to register, sign in and set a payment card to the application. Also, the most relevant part of the application is the purchasing feature: consumers should be able to select a restaurant, view its menus, choose products and add them to a virtual shopping basket. Finally, the application should provide a way to order the content of the basket.

The application should work on slow Internet connections. The interface needs to be optimized for touch use and it should be clear and simple enough to be used on the go.

### Monitoring application

The monitoring application should allow the company (the service provider) to monitor the restaurant application clients. It should display in real time the connection and order statuses in the restaurants. It should show recent log messages in order to reveal errors. The application has no strict requirements for the platform or performance, since it is used merely as a tool by

the developers.

## 5.2 System Architecture

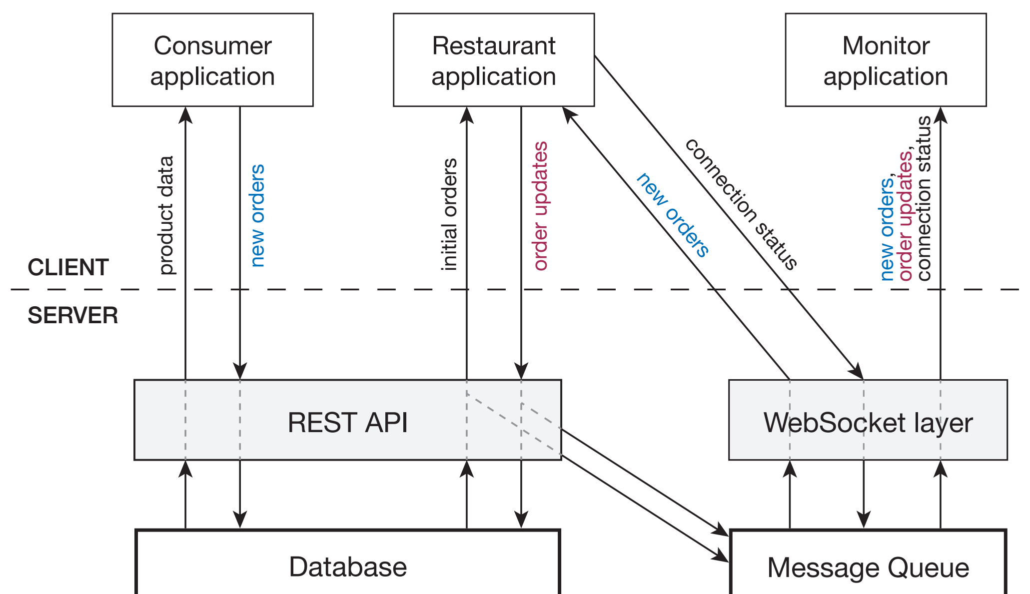


Figure 5.1: System architecture, data flow and storage.

In addition to the three client applications, the system consists of the following four main components: REST API, WebSocket layer, database and a message queue. The relationships of the components are illustrated in figure 5.1 and the purposes described below:

**REST API**

The main application logic is implemented on the client side, and a REST API is used as an interface to send and retrieve data between the client and the server. REST API provides the following functionalities: authentication, authorization, validation and manipulation of data. Consumer application retrieves the restaurant and product data and sends new orders through the API. Restaurant application retrieves initial orders through the API and posts updates to the orders (e.g. "in-progress" or "done" updates).

**WebSocket layer**

The WebSocket layer is used for real time and bi-directional communication. It notifies the restaurant application of new orders and receives its connection status. Also, it logs events to the message queue, which can be viewed by the monitor application.

**Database**

The database is used as a persistent data store. The REST API works as an interface to communicate with the database.

**Message queue**

The message queue works as a buffer for the real time messages and as a non-persistent storage for log messages.

## 5.3 Development tools

Efficient development of single page applications require tools that are different from the ones designed for traditional web applications. Such tools automatize and streamline the development process and are also crucial to follow good software engineering principles. In these applications, I used tools for testing, building (compilation) and serving automatization.

For build automatization I used tools called Grunt[1] and Gulp[2]. They minimise the files distributed to the user by concatenating the HTML, JavaScript and CSS files. Moreover, they remove unnecessary whitespace and optimize images. This greatly reduces the amount and size of files to download and results in faster load time for the end user.

Grunt and Gulp were also configured to serve files locally. Since single page applications use URLs to link to sub pages, which are technically just a single page, a special arrangement for the HTTP server is typically needed. All HTTP requests were configured to be redirected to the root document, namely `index.html`. Using a tool like Grunt or Gulp made it easy to use the same configuration for serving among all developers.

A tool called Bower[3] was used to automatize the installation of required packages. It allows to define the packages in a single text file, download and install them by a single command. This greatly lessens the efforts needed to install new or update out of date packages – especially in a team when the configuration file can be shared through the version control system.

[1] `http://gruntjs.com`
[2] `http://gulpjs.com`
[3] `http://bower.io`

Testing is an important part of software development process. For unit testing I used Jasmine[4] test suite and a test runner called Karma[5], which is designed especially for AngularJS. Karma runs on Node.js and it brings a tremendous advantage compared to conventional web application test runners: paired with a UI-less web engine (e.g., PhantomJS[6]) it can run constantly on the background and provide almost instant feedback of the test results. A modern computer runs tens and hundreds of tests in a matter of seconds. This supports well test driven development style.

## 5.4 Libraries

In addition to AngularJS, I used many UI and utility libraries. Consumer and restaurant applications depended heavily on a UI framework called Ionic[7], which is built on top of AngularJS, and thus it cannot be used in conjunction with any other SPA framework. Ionic implements a plethora of UI components that mimic the looks and behaviour of many mobile native platform UI components. For instance, such components include buttons, content areas, form elements, popups, modals and slide menus.

Monitor application utilized another UI library called Bootstrap[8], which similarly implements UI styles for buttons, forms, typography, grids, alerts, progress bars etc. However, Bootstrap is not primarily designed for mobile and touch use, but rather it is designed in a "mobile first" manner. Essentially, this means it will work on smaller screen sizes, but does not mimic behaviour known from mobile operating systems.

Lo-dash[9] is a utility library that is useful for any JavaScript application, and was used in all three applications. It implements patterns that are commonly used in functional programming style. It encourages using immutable data structures, and thus improves code readability and quality.

Apache Cordova[10] is a library and a tool to package web applications as native applications (hybrid applications). It does not affect the way to build web applications other than allowing the use of native APIs. I packaged the consumer and restaurant applications with Cordova and distributed it to Apple App Store and Google Play Store. Only splash screens and status bars

[4] `http://jasmine.github.io`
[5] `http://karma-runner.github.io`
[6] `http://phantomjs.org`
[7] `http://ionicframework.com`
[8] `http://getbootstrap.com`
[9] `http://lodash.com`
[10] `http://cordova.apache.org`

were customized for the applications with Cordova APIs. We configured the web view on Android to run Google Chrome with the help of library called Crosswalk[11].

## 5.5 Technologies

In this section, I present the used web technologies that were introduced in chapter 3. The technologies and their usage descriptions are listed below.

**Touch Events**
Touch events were used extensively in the touch-based consumer and restaurant applications. The UI components included slidable, draggable and scrollable elements.

**Media elements**
HTML5 `<audio>` element enabled us to play audio notifications on the restaurant application. However, we replaced the functionality later by Web Audio API.

**Web Messaging**
A special arrangement in the restaurant application was made with iframes. The iframe used web messaging to communicate with the containing page.

**Browser history management**
Browser history management was used by AngularUI Router to enable back and forward navigation functionalities in the consumer application.

**Cascading Style Sheets**
I used CSS comprehensively to style the applications. I used SASS and LESS preprocessors to take advantage of the advanced functionalities and to better organize the files. Ionic utilized many advanced CSS properties including transitions and the “flexbox” display mode for elements.

**Web Storage**
I used Web Storage for caching data locally in the consumer application. It prevented unnecessary requests and the user could start using the application immediately after launch, as the most important data was saved in the cache.

[11] `http://crosswalk-project.org`

**WebSockets**
I used WebSockets in the restaurant and monitoring applications to send and receive orders, order updates and log messages.

**Cross-Origin Resource Sharing**
CORS configuration made it possible to use the same REST API for the consumer and restaurant applications without special arrangements.

**Scalable Vector Graphics**
I implemented all icons in SVG format. Ionic provides an extensive icon kit where all the icons are mapped to easy-to-use CSS classes.

**Web Audio**
Advanced audio functionalities were needed, such as programmatically stopping and playing the audio and concurrent audio tracks. This caused problems with the `<audio>` element. Thus, migration to the Web Audio API was done.

## 5.6 Application structure

### Routing

The routing was implemented using an AngularJS library called AngularUI Router[12]. Compared to the router included in AngularJS, it provides much more functionality to build complex applications. It implements nested and parallel views and is organised around *states* rather than *routes*. States are like routes where the associated URL is optional.

The routing mechanism in AngularJS allows attaching animations to the pages. Transitions between views are common especially in mobile applications, where the smaller view cannot contain simultaneously as much information as in desktop applications. Transitions help user to conceive the navigation structure and give an impression of responsiveness. For example, in the consumer application many of the subsequent pages have a sliding transition in between. The transition changes direction based on whether user is navigating backwards or forwards. The animations are implemented by CSS3 transitions and are triggered by JavaScript based on the state change.

An example excerpt of routing in the consumer application is shown on listing 5.1. AngularUI Router allows to define also abstract states, which are states that cannot be rendered as is. The abstract states must be inherited by regular states that may or may not override the abstract state's data or

[12] `http://github.com/angular-ui/ui-router`

add sub-views. Thus, the excerpt only defines two actual renderable states, named `master.menu.items` and `master.menu.item`. Moreover, the URL is inherited from the parent states and results in these two URLs to access the states: `/menu/items` and `/menu/items/:categoryId/:itemId` where `:categoryId` and `:itemId` are variables. The variables can be captured in the controller as shown in listing 5.2.

```
...
$stateProvider
  .state('master', {
    url: '',
    abstract: true,
    controller: 'MainController',
    templateUrl: 'templates/master.html'
  })
  .state('master.menu', {
    abstract: true,
    url: '/menu',
    templateUrl: 'templates/menu.html'
  })
  .state('master.menu.items', {
    url: '/items',
    controller: 'MenuItemsController',
    templateUrl: 'templates/menu-items.html'
  })
  .state('master.menu.item', {
    url: '/:categoryId/:itemId',
    templateUrl: 'templates/menu-item.html',
    controller: 'ItemController'
  })
...
```

Listing 5.1: Excerpt of state configuration in the consumer application.

```
...
.controller('ItemController', function($scope, $stateParams)
{
  ...
  var categoryId = $stateParams.categoryId;
  var itemId = $stateParams.itemId;
  ...
});
...
```

Listing 5.2: Excerpt of state configuration in the consumer application.

### Modularization

Being able to divide application into logical modules is essential in every programming language and environment. This encourages reusability and testability, that are fundamentally important aspects of programming. AngularJS approaches modularization via technique known as dependency injection.

Dependency injection is an internal implementation in AngularJS and is not a standard way of modularization in single page applications. The technique works by defining modules as functions that are given a name as a string, which is then parsed by AngularJS. An example of module definition is shown in listing 5.3. It first defines an AngularJS module named `fresgo.controllers.MenuController` and specifies it to use another module named `ionic`. In the latter part it defines the content of the module, which in this case is a controller named `MenuController`. The function parameters of the controller are injected components from the modules, `$scope` being an internal component in AngularJS and `$ionicModal` being injected from the `ionic` module.

```
1 angular.module('fresgo.controllers.MenuController', [
2   'ionic'
3 ])
4 .controller('MenuController', function($scope, $ionicModal) {
5   ...
6 });
```

Listing 5.3: Excerpt of state configuration in the consumer application.

This technique makes it effortless to write new modules in a way that developers know which other modules they require. However, the drawback is that the modules are compatible only for AngularJS applications if no other arrangements are made. Furthermore, depending on modules that depend on other modules can be tricky, since there have to be a way to download the whole hierarchy of the dependencies. This has been solved by the Bower package manager, but it adds additional complexity.

### Data flow

AngularJS implements two-way data binding as I discussed previously in section 4.1.4. The binding of data between the controller and the view in the applications is made with AngularJS `$scope` service. Templates that are paired with a controller are given a scope, which is accessible directly from both; templates and controllers. Thus, attaching a click event to a template as shown in listing 5.4, can trigger an action defined in the controller. The `ng-click` attribute is a way in AngularJS to bind to a click event.

```
1 <button ng-click="goBack()"></button>
```

Listing 5.4: Example of binding to a click event in an HTML template

The scopes can be inherited also by binding data only in one direction. This would create a one-way bound scope from the parent scope where the changes would not propagate to the parent, but changes from the parent would propagate to the child scope. Furthermore, the scopes can be created also as isolated, in which scope properties are copied from the parent but the data is not bound in either direction. Defining the bindings strictly helps to create reusable components and improve the performance.

Data flows in the applications hierarchically from the top to the bottom. Controller-view pairs create child scopes to parent (or master) scopes, as also happens in listing 5.1 state configuration example. Mostly, all the hard-coded data is written in the templates directly. All the essential content is downloaded via the REST API using AngularJS `$http` service component, and propagated to the templates from the controllers.

### UI Components

UI components in AngularJS are called “directives”. They are essentially controller-view couplings that can be given a custom HTML element or attribute name. Moreover, the scope bindings and template compilation can be customized. Directives are an important technique to achieve modularity and testability. They respond directly to the “separation of concerns” software engineering principle.

Listings 5.5 and 5.6 show a simple example of a custom directive definition and usage, which are modified from the actual application implementation for demonstration purpose. The listing 5.5 defines both the business logic and the template. The scope property binds to an outer scope to get access to the interpreted value typed between the curly brackets in the listing 5.6.

```
1  angular.module('fresgo.directives.countdownTimer', [])
2  .directive('countdownTimer', function() {
3    return {
4      restrict: 'AE',
5      replace: true,
6      transclude: false,
7      scope: {
8        time: '@countdownTimer'
9      },
10     template: '<span>{{ timeLeft }}</span>',
11     controller: function($scope, $timeout) {
12         // logic implementation
13     }
```

```
14   }
15 });
```

Listing 5.5: Excerpt example of a countdownTimer directive definition

```
1 <countdown-timer="{{order.pickupTime * 60}}"></countdown-
    timer>
```

Listing 5.6: Example of using countdownTimer directive in a HTML template.

The applications were implemented to make heavy use of directives. Almost all UI components that were used more than once, were directives. In fact, even the router uses directives to render the views and most of the Ionic components are directives.

## 5.7 User Interfaces

Figure 5.2 shows screenshots captured on an iPhone 5 (iOS 8.3) and Samsung Galaxy S3 (Android 4.4). It can be seen that the UI scales nicely to both screen sizes. It is almost impossible to test the UI on all different screen sizes, but it can be assumed to work properly at least on the ones between the tested sizes. As can be seen from figures 5.4, 5.6, 5.5 and 5.3, the UI follows patterns present today in many native applications: headers, footers, side menu, navigation buttons and list elements.

Figure 5.3 shows a registration modal that is opened on top of the rest of application. The modal is animated on opening and closing, which cannot be seen from a static screenshot. A similar approach can be seen in figure 5.4 that shows a loading icon with a darkened background. Altough techically it is not a modal, i.e., a separate page, it is also placed on top of the rest of the application. This kind of behavior is nearly impossible – and cumbersome at best – to accomplish without single page architecture.

The three dots on the figure 5.3 indicate a slidable container. In other words, it is a content container that includes three individual content areas in the DOM, which are displayed separately. This is made possible by Ionic's special `ion-slide-box` directive that uses CSS positioning techniques and JavaScript to achieve the effect.

A side-menu functionality is displayed on figure 5.5. It is the application's main navigation menu and can be opened either by clicking the menu icon on the top left or by a swipe gesture from the left. A side-menu could be implemented without single page architecture, but fresh page loads would not allow animations between the views. In this application, when the user clicks a button on the side menu, the side menu is automatically hidden by a

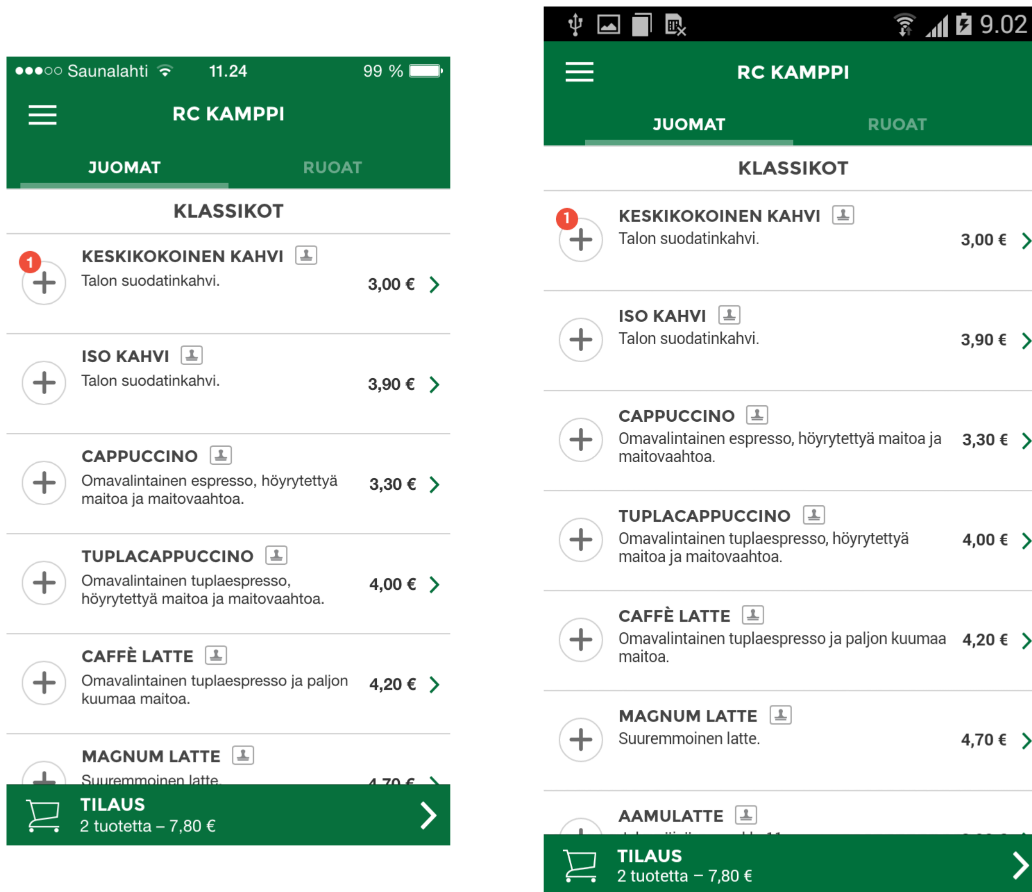


Figure 5.2: Screenshot of the consumer application (left: iPhone 5, right: Samsung Galaxy S3).

sliding animation to the left and a new view is loaded to the main view area simultaneously.

Drag and release functionality, an interaction paradigm that is better known from native applications, is presented in figure 5.6. The list of restaurants can be updated by dragging the list down and then releasing. This is tricky, because browsers behave differently when the user scrolls to the edges of the page. Some browsers implement a “rubber band” functionality, which expresses an elastic bouncing effect when user scrolls past the top or bottom of the page. Ionic does not rely on the browser’s implementation but disables the standard behaviour with CSS rules and implements a custom rubber band effect by JavaScript. Thus, the dragging is completely controllable programmatically and allows drag and release hooks to be implemented.

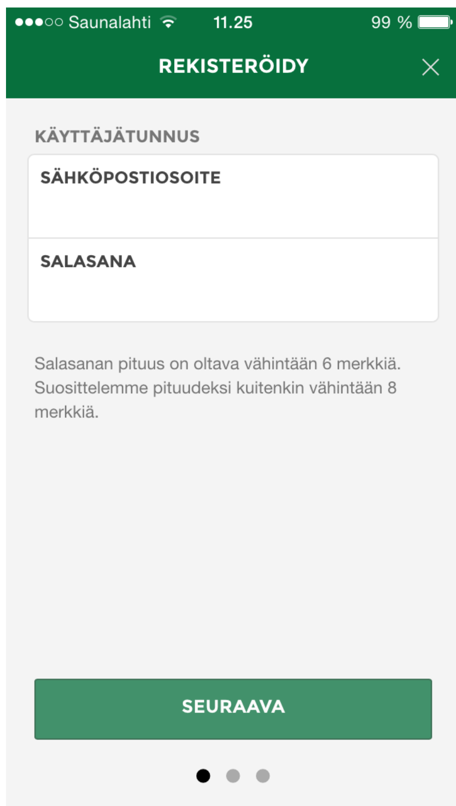


Figure 5.3: Intermittent loading screen on the consumer application.

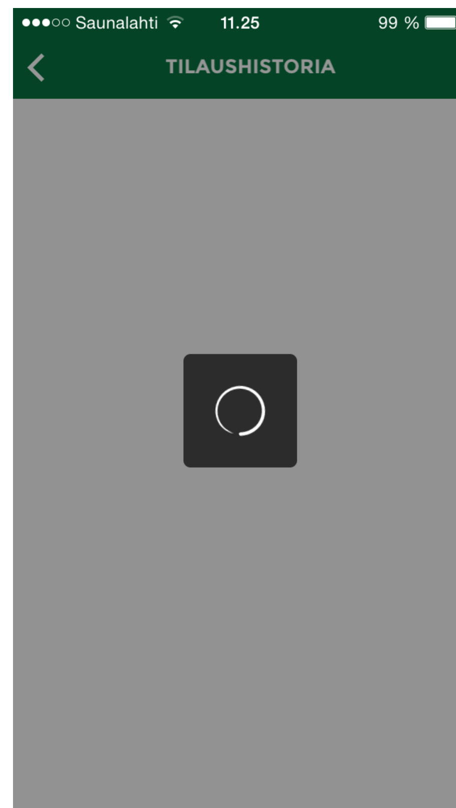


Figure 5.4: Registration screen on the consumer application.

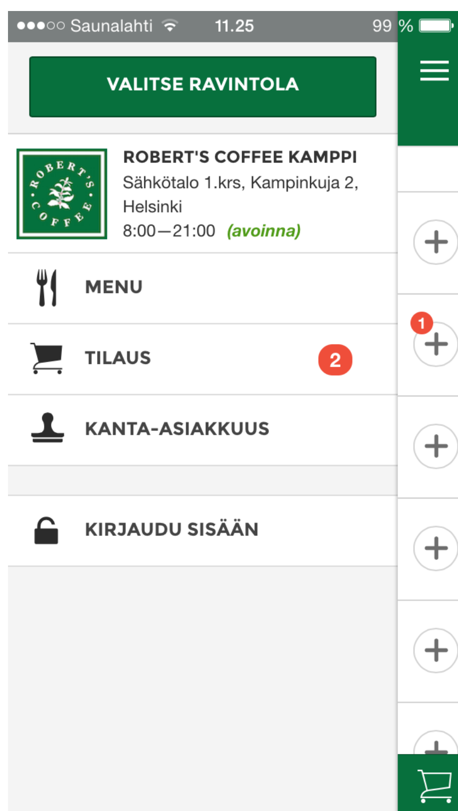


Figure 5.5: Side menu on the consumer application.

Figure 5.6: Drag and release functionality on the consumer application while dragging towards the bottom of the screen.

The restaurant application is presented in figure 5.7. The user interface follows tab-based navigation paradigm. The bottom bar at the screen is a persistent element that does not change when the view. Icons on the bar work as tab selectors that change the main view and inside the main view there can be sub-links and sub-views. Also, the navigation bar is fixed to the bottom so that it is not affected by scrolling the view.

Figure 5.8 shows a screenshot of the monitoring application. The UI the simplest of all three and does not include hierarchical navigation. The application displays the log messages for restaurant application clients in real time and allows filtering. It could be rather easily implemented without single page architecture by simply using AJAX requests and updating the DOM partially. However, there would be no benefits compared to SPA and the benefits of MV* architecture would be lost.

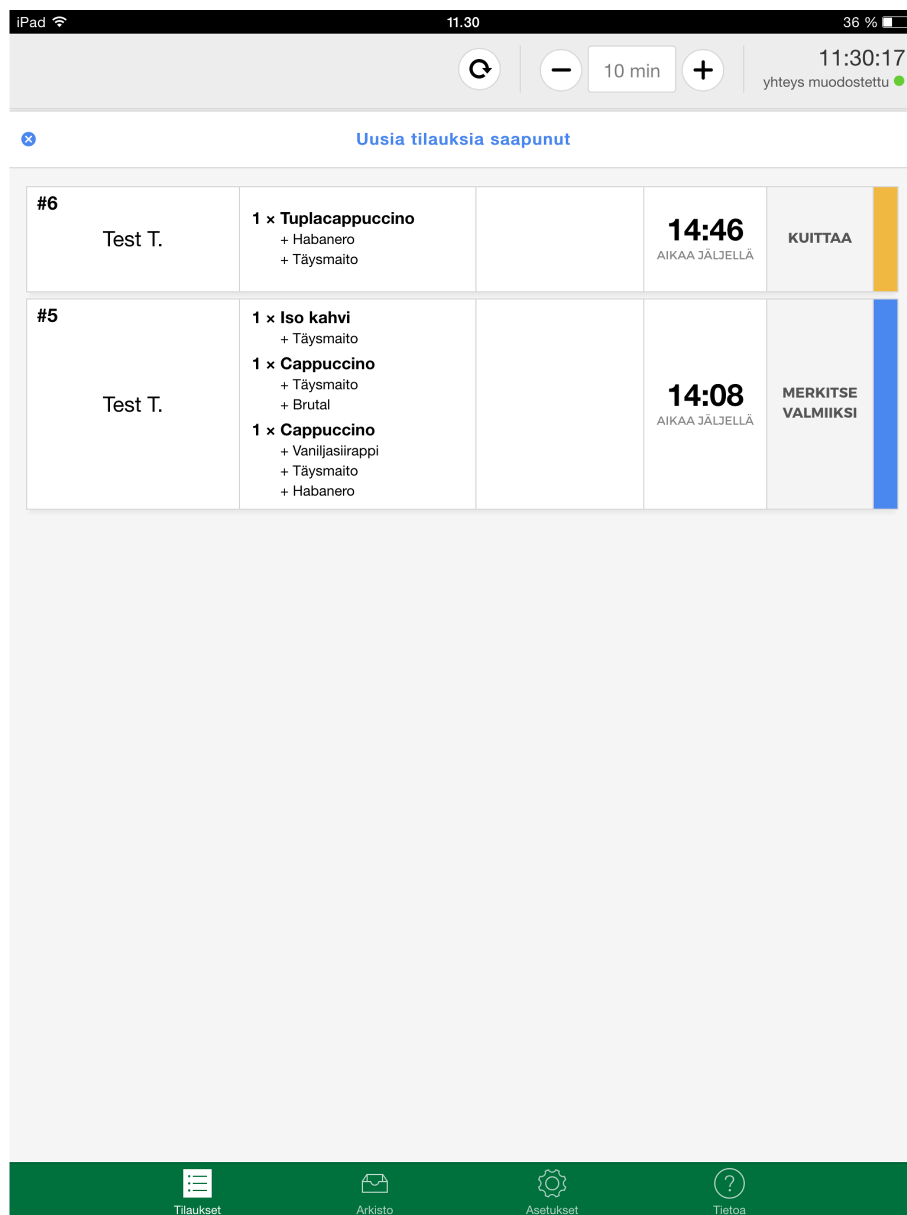


Figure 5.7: Screenshot of the main view in the restaurant application captured on an iPad.

Fresgo Client Monitor

https://[illegible]/#/

Fresgo Client Monitor connected

Clients

| Conn. | Id | Restaurant | Username | Status | Latest log | IP | Active |
|---|---|---|---|---|---|---|---|
| 9ee0 | 46 | Robert's Coffee Corner | | online | updated order #2 status to "finished" (08.05.2015 09:14:00) | | |
| 50d3 | 40 | Robert's Coffee Wanha kauppahalli | | online | went online (08.05.2015 06:00:04) | | |
| 6ac1 | 38 | Robert's Coffee Hietalahden kauppahalli | | online | went online (08.05.2015 06:00:04) | | |
| 44cf | 37 | Robert's Coffee Paahtimokauppa | | online | went online (06.05.2015 12:00:31) | | |
| d14d | 16 | Robert's Coffee Citykäytävä | | online | went online (08.05.2015 13:01:13) | | |
| def9 | 17 | Robert's Coffee Stockmann Netcup | | online | updated order #59 status to "finished" (07.05.2015 08:55:54) | | |
| 3c09 | 19 | Robert's Coffee Kamppi | | online | received order #62 (08.05.2015 13:16:31) | | |
| d53e | 36 | Robert's Coffee Kluuvi | | online | went online (08.05.2015 13:21:35) | | |

all connections

Log

```
08.05.2015 12:26:15 – (0cd5) [16] [Robert's Coffee Citykäytävä] : went offline
08.05.2015 12:26:51 – (d7f5) [16] [Robert's Coffee Citykäytävä] : went online
08.05.2015 12:29:10 – (d7f5) [16] [Robert's Coffee Citykäytävä] : went offline
08.05.2015 12:55:25 – (bed5) [16] [Robert's Coffee Citykäytävä] : went online
08.05.2015 13:01:11 – (bed5) [16] [Robert's Coffee Citykäytävä] : manual system reload performed
08.05.2015 13:01:12 – (bed5) [16] [Robert's Coffee Citykäytävä] : went offline
08.05.2015 13:01:13 – (d14d) [16] [Robert's Coffee Citykäytävä] : went online
08.05.2015 13:16:30 – (3c09) [19] [Robert's Coffee Kamppi] : pending acknowledgedment for order #62
08.05.2015 13:16:31 – (3c09) [19] [Robert's Coffee Kamppi] : received order #62
08.05.2015 13:21:05 – (d252) [36] [Robert's Coffee Kluuvi] : went online
08.05.2015 13:21:33 – (d252) [36] [Robert's Coffee Kluuvi] : manual system reload performed
08.05.2015 13:21:33 – (d252) [36] [Robert's Coffee Kluuvi] : went offline
08.05.2015 13:21:35 – (d53e) [36] [Robert's Coffee Kluuvi] : went online
```

Figure 5.8: Screenshot of the monitoring application on a desktop web browser (Chrome 42).

# Chapter 6

# Discussion

This chapter sums up the implementation and discusses the results. I present advantages and disadvantages of single page architecture and assess how it worked as basis for web applications. Also, I share my thoughts of my potential future work as well as the future of web applications.

## 6.1 Development and distribution

I learned how to build and distribute web applications that take advantage of multiple web technologies and AngularJS. I utilized many tools related to development and distribution workflow. Moreover, the community of JavaScript and web application developers became familiar.

The tools for the development appeared to be very diverse. It took considerable amount of time to build workflow configurations with Grunt and Gulp, but in the end they saved lots of time. There were plug-ins for all the required tasks and I did not face any major compatibility issues. However, the more inexperienced developers in front-end web application development found all the required tools to add rather a lot of cognitive overhead. As the projects grew, the tools to streamline and share configurations became invaluable.

As discussed in section 2.2 of the challenges of JavaScript development, I used stepwise approach and exploratory programming style. This was made possible by these tremendous tools for testing: Jasmine, Karma and PhantomJS. AngularJS has been designed testability in mind, and the dependency injection technique I presented in section 5.6 proved to make testing very efficient. Writing and running of tests was simple and powerful, since I could run the test suite constantly on the background providing instant feedback as I wrote code. I have not encountered such testability on other platforms

and consider it a major advantage.

One of our goals was to be able to update the restaurant applications as effortlessly as possible. This was achieved by packaging the application with Cordova and pointing it to download the HTML, CSS and JavaScript files from the server. Thus, the application can be distributed from the application store, but I chose to install it on the iPads with the development tools. The solution allows us to update the source files on the server and then send a reload command. This way the applications can be updated almost instantly and no interaction from the restaurant employees are needed. The ideal solution took some effort to find, but proved to work well in the production.

Without most of the technologies listed in section 5.5 creating a single page architecture would had been almost impossible or very hard. AngularJS and Ionic profoundly utilize touch events, browser history API, CSS, Web Storage and SVG. In addition, Web Audio and WebSockets were crucial technologies to build rich applications. The SPA enabled rapid development to develop the applications as the skeleton models were already implemented.

## 6.2 Performance

In the previous section, I mentioned to have deployed the restaurant applications as hybrid applications. In fact, the applications were first deployed as a regular web applications and were accessed as "pinned to home screen" applications on the iPads. IOS allows the user to pin arbitrary web pages as applications on the home screen, and depending on the page's support it hides the web browser's UI. I used this technique to achieve a clean UI and avoided uploading the applications separately to the iPads. Nonetheless, this caused prominent performance problems resulting in non-tolerable delays in registration of touch events and stuttering transition effects. The performance was seemingly better on the web browser than on the application pinned to home screen and I could not trace the root of the issue. Our best assessment is that the performance of the pinned applications on iOS 8 are restrained on purpose to actuate the use of the native application store.

The applications were developed mainly on desktop browsers and some problems did not occur until testing on the mobile devices. The bi-directional data binding in AngularJS proved to cause JavaScript performance problems when running on the mobile devices. This was clearly caused by the scope digestion mechanism, which creates and evaluates the bindings. These problems were solved by simplifying the bindings and by reducing some unnecessary two-way bindings to one-way bindings. In the end, the JavaScript performance was excellent.

## 6.3 User Experience

User experience is largely determined by the fluency and usability of the user interface. Single page architecture improves it by allowing to easily show intermediate transitions and animations between the views. Moreover, as shown in screenshots in the section 5.7, SPA enables very native-like UI elements including hidden slide menus, slidable content boxes and modals. Still, SPA is not necessarily required to create such UI elements, but rather makes it effortless and maintainable.

The UI I built for the consumer application was the most complex of the all three. Ionic's components worked exceedingly well and they were easy to customize. I consider the result of the mobile interface to be very fluent and native-like without any major drawbacks.

I first built the restaurant application using Bootstrap as the UI component library. However, I was not satisfied with the UI on touch use. Thus, I later migrated it to Ionic. Due to the single page architecture and modularization of AngularJS, the migration was easy and I achieved better functionality for touch use rather effortlessly. Again, Ionic proved to work really well from the user experience point of view.

## 6.4 Summary

In this section I summarize the results by analysing the challenges of the web as an application platform presented in section (2.2).

**Application complexity**

Although I did not implement considerably large applications, there was prominent complexity. One of the causes was that there is no *generally accepted/known*, i.e., "standardized way" of building client-side web applications. This lead to a steep learning curve for most of our developers having background in server-side and native mobile technologies. Besides understanding the concepts of AngularJS, such as directives, services, routing and scope digestion, the developers had to master SCSS and LESS preprocessors, and be able to configure Gulp and Grunt to make adaptations to the work flow. Moreover, the use of many libraries lead to scattered documentation and complexity via leaky abstraction[1]. There is variation also in the tools being used, which adds to the equation.

[1]Leaky abstraction in software development happens when an abstraction unpurposely limits the actual implementation and it ultimately leads to unnecessary complexity.

**Browser semantics**

Looking at many native applications, such as text processing, spreadsheet editing, image manipulation, calendar and messaging applications, reveals interaction paradigms very different from a web browser. Moreover, none of our applications followed a traditional page-based navigation model that the web browsers are originally designed for. I realized that the browser semantics (i.e., browser UI) are rather unsuitable for applications. Back and forward button, reload button and address bar are seldomly needed for applications. Many views do not use back-and-forth history states, and if they do, the application usually has a custom UI to address the navigation. Luckily, on iOS it is possible to get rid of the browser UI by pinning the application on the home screen or by packaging it with Cordova.

**Same-origin policy**

CORS was used as the solution to access the REST API from all the applications. However, this was not affected by the application architecture.

**Software engineering principles**

I was able to follow good software engineering principles for the most parts. Modularity and reusability were achived via good design in the AngularJS application architecture. Most UI components were declared separately from the business logic emphasizing reusability. I found the absence of static typing to hinder the development at some parts and overcame it with stepwise approach, i.e., test-driven development style.

**Performance**

As I discussed in the previous sections, I found the performance to be generally satisfactory and excellent at best. This was achieved with the use of CSS for hardware accelerated animations, Crosswalk plugin for the Anrdoid and single page architecture. Nonetheless, I found that it is easy to neglect the best practices and thus compromise JavaScript performance, especially due to the design of AngularJS.

**Fragmentation**

Fragmentation appeared to be a true issue when building web applications in long-term. I encountered the problem of libraries deprecating fast and publication of new versions of frameworks only in a matter of weeks. It caused some migration problems and took extra effort to

update the software. However, I was satisfied at the speed of bug fixes in the libraries, thus not getting stuck at framework-level limitations.

Contrary to a common case native application development, we did not have to maintain multiple codebases to support both iOS and Android applications. In fact, the consumer application was developed with a desktop web browser and worked on the iOS web view without major problems. Later, an Android conversion was made and it caused minor incompatibilities. The problems were caused by the differences in the native web browsers, but many of them were solved by utilizing Crosswalk that replaced Android's native web view with Chrome.

**HTML semantics**

I certainly reognize that the standard HTML elements are not useful for describing application-style user interfaces. However, it was easy to create custom elements by utilizing AngularJS directives. Custom tags are not valid HTML5 markup, but are supported in modern web browsers.

**Monetization**

Considering monetization in web applications, a limitation is that they cannot be sold in a similar way as native applications. There are no stores for web applications and the client-side source is, albeit only in theory, freely downloadable and copyable. However, I do not consider this to be a serious limitation. Web applications usually require users to register and monetize the application by either a one-time registration fee or a periodical fee. The major drawbacks in this are that there is no single central place to look for the applications and that the user has to have multiple registrations/accounts to different applications.

In our case there is no registration or periodical fee for the application. Thus, the use of the application for the consumer is free and the restaurants pay for the use of the platform. Hence, there are many ways to monetize an application, which I discussed already in section 2.2. The consumer application was distributed to a native application store, but even if it was not, the web as a platform would not had been a restriction considering monetization in our case.

**Distribution and promotion**

Distribution is closely related to the monetization. The company was able to took advantage of the native application stores by packaging the consumer application with Cordova. This was an efficient way to distribute the application to consumers. However, there is no such

distribution channel for applications running in the web browser. The advantage of web applications is that they do not have to be separately downloaded and installed, and a disadvantage is that they are harder to find.

**Network dependency**
I found that it is possible to support offline use by caching. Single application architecture supports this really well, since the application is usually downloaded at once and the business logic is implemented on the client. Single page applications need network connection only to communicate with APIs, which is the same for native applications. Conventional web applications cannot be used offline, because the business logic is implemented on the server. Thus, only an initial page load can be cached.

## 6.5 Future Work

Some of the stuttering and performance problems might had been avoided with the use of Web Workers. For example, externalising the computation of scope digestion in AngularJS to workers could helped, but the feature would have to be supported by the core of AngularJS. Also, moving the WebSockets event handling to a separate thread might be beneficial especially on heavy use of events.

I chose not to use AMD or CommonJS to modularize the JavaScript files. Instead, I simply included all files with the standard `<script>` tags. This was possible due to the small sizes of the projects, but any larger projects would benefit from using modularizing the JavaScript files. For example, a project including multiple applications with cross-references in source files would benefit from including only the referred (imported) files. I also felt that using AMD or similar adds unnecessary complexity and might not be worth the additional development and distribution workflow configurations. Moreover, I am looking for a native implementation for the modules, which will be implemented in ECMAScript 6.

I discovered that it is easy to overcome the limitations of HTML semantics by AngularJS directives, and many of the UI components I implemented were directives. Standard elements in HTML5, such as `<div>`, `<span>` or `<article>` are not meaningful to user interfaces. Consider elements named `<map>` or `<sidebar>`, for instance; they describe the UI element in a meaningful way. Seemingly, there is a trend towards this kind of component-based development. React framework have been recently adopted by developers

and it is based on the idea of constructing self-containing, independent and reusable custom elements. Moreover, Web Components and Polymer similar technologies being standardized to allow these kind of custom elements.

Today, JavaScript can be run also on the server, which is enabled by Node.js. This is also transforming the way web applications are built. In a new, *isomorphic*, design pattern the web pages or applications can be fully rendered also on the server, improving initial loading speed and optimization for search engines. This sets new requirements also on single page application frameworks: the application state must be transferable between the client and the server. Of the frameworks presented in section 4.2, React has been used to create isomorphic applications. Moreover, Meteor, which was briefly touched earlier, is designed as an isomporphic framework from the ground up. [8, 54] However, using this technique would had only minor performance effects on our applications.

In the near future, ECMAScript 6 (ES6) will likely have a profound effect on how to build web applications. New versions of AngularJS and Ionic are in the works being completely based on ES6 and taking advantage of its new features. Yet, the lack of tools and guides, smaller community and immaturity of ES6 were reasons why I developed the applications with ECMAScript 5. I am looking forward to utilize especially the module system in ES6, which will probably resolve the issues regarding modularization that I discussed earlier.

During the writing of this thesis, a new version of HTTP protocol was in the works. HTTP/2 (Hypertext Transfer Protocol Version 2) was released in May 2015, and it provides major new features compared to the earlier version, HTTP/1.1. It supports i.a. multiplexing of requests, streams, prioritization and header compression. In practice this means less established connections, reduced perception of latency and bi-directional communication. [3] As it becomes mainstream, HTTP/2 will also likely affect web applications in a major way. It will reduce the need of using WebSockets and Server-Sent Events. Moreover, concatenating source files will no longer have as large impact on performance as earlier. Thus, it will likely simplify application development and distribution.

# Chapter 7

# Conclusions

The web is one of the most rapidly evolving technologies in the world. This has been proven by the emergence of recent technologies, such as HTML5, WebSockets, ECMAScript 6 and HTTP/2. The community of the web, which is largely based on open source, adopts new technologies quickly and almost imminently spawns a plethora of derived technologies, libraries and frameworks. Single page architecture is a combination of some of the recent derived technologies, including browser history manipulation, advanced DOM manipulation, asynchronous loading and rendering of data. I evaluated its suitability for building applications for the web and assessed how the Web serves as an application platform.

I found that single page architecture is currently in the epicentre of web application development. In its essence, it is rather a simple concept and does not introduce any unforeseen techniques. However, the power of SPA lies in the way how it fundamentally changes the style of web application development. It transfers the handling of business logic from the server to the client. This brings web applications a major step closer to native applications in terms of the architecture. Thus, being radically different from conventional web applications, single page applications derive new needs for new purposes, such as Web Workers for dividing computation to threads and Web Sockets for real time communication.

Based on results of the applications I built with AngularJS, we found the performance of single page applications to be very satisfactory. I also found that using SPA as basis for the applications makes it very effortless to build an user interface that utilizes application-like UI components and beautiful transitioning animations. Nonetheless, major challenges are the semantics of HTML and the web browser, which I found are not suitable for applications. Moreover, SPA supports building applications for offline usage by making it possible to load the whole application at once. Overall, I can confidently

state that SPA enhances the user experience of web applications.

An unquestionable advantage of web applications is the possibility to run them on almost all platforms and devices. Web applications, however, have been criticized of poor performance and user experience especially on mobile devices. I found that a framework that focuses on touch-based use, like Ionic I used, can truly rival native applications in terms of user experience. Again, fundamentally this has been enabled by the single page architecture. To get a more comprehensive comparison between web and native applications, one would have to create similar applications with both technologies. I predict that the results of such a comparison regarding user experience, however, would still slightly favour native applications. The advantage of web application would be the simplicity of development and support for multiple platforms with one codebase.

Fragmentation and complexity are some of the challenges in application development. I found many new concepts introduced AngularJS to increase cognitive overload before they became evident. Also, to develop the applications I had to learn many new tools that support the workflow and distribution. Complexity was increased because of those new concepts and tools. However, I felt that AngularJS and SPA also reduced complexity via modularization and component-based design. I found AngularJS directives to be a powerful technique to create reusable components, and discussed another popular framework, React, which emphasises similar component design. I predict that in the near future the applications will increasingly emphasize composition of components design pattern. With the advent of Web Components technology they will likely not be specific to frameworks, but generally reusable components.

The web seems to be undergoing a change where the web applications are less and less coupled with the server-side implementations. This has spawned an interesting era in the Web, where new technologies and libraries are spawning and old are deprecating faster than ever. The web as an application platform is seeking for its form, and I predict it will continue to do so for plenty of years. In the near future, the advent of new fundamental technologies including ECMAScript 6 and HTTP/2 will have a profound effect on how the web will evolve. Single page architecture have proved to be a prime mover for rich web applications and I predict it to slowly become the standard way of building applications for the Web.

# Bibliography

[1] Can I Use: drag-and-drop. Available at: http://caniuse.com/#feat=dragndrop. Accessed 17-April-2015.

[2] Agarwal, S. Real-time Web Application Roadblock: Performance Penalty of HTML Sockets. In *2012 IEEE International Conference on Communications (ICC)* (Ottawa, June 2012), IEEE, pp. 1225–1229.

[3] Belshe, M., BitGo, R. Peon, G., Thomson, M., and Mozilla. Hypertext Transfer Protocol Version 2 (HTTP/2) standard definition (RTC 7540). Available at: http://www.rfc-editor.org/rfc/rfc7540.txt. Accessed 16-May-2015.

[4] Bergkvist, A., Burnett, D. C., Jennings, C., and Narayanan, A. WebRTC 1.0: Real-time Communication Between Browsers (working draft). Available at: http://www.w3.org/TR/2015/WD-webrtc-20150210/. Accessed 18-April-2015.

[5] Bleigh, M. Let's Kill Semantic HTML. Available at: http://divshot.com/blog/opinion/lets-kill-semantic-html/ (2014). Accessed 14-April-2015.

[6] Bochicchio, M. A., Longo, A., and Vaira, L. Extending Web applications with 3D features. In *2011 13th IEEE International Symposium on Web Systems Evolution (WSE)* (Williamsburg, September 2013), pp. 93–96.

[7] Bushell, D. Resolution independence with SVG. Available at: http://www.smashingmagazine.com/2012/01/16/resolution-independence-with-svg/. Accessed 19-April-2015.

[8] Creamer, J. React To The Future With Isomorphic Apps. Available at: http://www.smashingmagazine.com/2015/04/21/react-to-the-future-with-isomorphic-apps/. Accessed 16-May-2015.

[9] Crockford, D. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[10] Dahlström, E., Dengler, P., Grasso, A., Lilley, C., McCormack, C., Schepers, D., Watt, J., Ferraiolo, J., Jun, F., and Jackson, D. Scalable Vector Graphics (SVG) 1.1 (Second Edition). W3C Recommendation, W3C, Aug 2014. Available at: `http://www.w3.org/TR/SVG/`. Accessed 19-April-2015.

[11] Ecma International. *Final draft Standard ECMA-262 6th Edition (Rev 38)*, April 2015. Available at: `http://wiki.ecmascript.org/lib/exe/fetch.php?id=harmony%3Aspecification_drafts&cache=cache&media=harmony:ecma-262_6th_edition_final_draft_-04-14-15.pdf`.

[12] Elliott, E. *Programming JavaScript Applications*. O'Reilly Media, 2014.

[13] EmberJS. EmberJS v1.10.0 Guides. Available at: `http://guides.emberjs.com/v1.10.0/`. Accessed 28-April-2015.

[14] Fink, G., and Flatow, I. *Pro Single Page Application Development*. Apress, 2014.

[15] Flanagan, D. *JavaScript: The Definitive Guide 6th Edition*. O'Reilly Media, 2011.

[16] Foster, J. The Semantics of HTML and XAML. Available at: `http://www.codefoster.com/semantics/` (2014). Accessed 14-April-2015.

[17] Garrett, J. J. Ajax: A New Approach to Web Applications. 5. Available at: `https://courses.cs.washington.edu/courses/cse490h/07sp/readings/ajax_adaptive_path.pdf`. Accessed 11-April-2015.

[18] Google Inc. AngularJS (v.1.13.14) Developer Guide. Available at: `http://code.angularjs.org/1.3.14/docs/guide`. Accessed 23-April-2015.

[19] Guinard, D., Trifa, V., and Wilde, E. A Resource Oriented Architecture for the Web of Things. In *Internet of Things (IOT)* (Tokyo, November 2010), IEEE, pp. 1–8.

[20] Hégaret, P. L., Whitmer, R., and Wood, L. Document Object Model. Available at: `http://www.w3.org/DOM/`. Accessed 11-April-2015.

[21] Hickson, I. HTML5 Web Messaging API proposed recommendation by W3C. Available at: http://www.w3.org/TR/webmessaging/. Accessed 17-April-2015.

[22] Hickson, I. Server-Sent Events. W3C Recommendation, W3C, Feb 2015. Available at: http://www.w3.org/TR/2015/REC-eventsource-20150203/. Accessed 19-April-2015.

[23] Hickson, I. Web Storage recommendation by W3C. Available at: http://www.w3.org/TR/2013/REC-webstorage-20130730/. Accessed 13-April-2015.

[24] Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E. D., O'Connor, E., and Pfeiffer, S. HTML5 specification by W3C. Available at: http://www.w3.org/TR/2014/REC-html5-20141028/. Accessed 15-April-2015.

[25] Hoban, L. ECMAScript 6 Features. Available at: http://github.com/lukehoban/es6features/. Accessed 17-April-2015.

[26] Inc., F. Flux documentation. Available at: http://facebook.github.io/flux/docs/overview.html. Accessed 26-April-2015.

[27] International, E. Final draft Standard Ecma-262 6th Edition / April 2015. Available at: http://wiki.ecmascript.org/lib/exe/fetch.php?id=harmony%3Aspecification_drafts&cache=cache&media=harmony:ecma-262_6th_edition_final_draft_-04-14-15.pdf. Accessed 17-April-2015.

[28] Khronos Group. OpenGL ES 2.0 for the Web. Available at: http://www.khronos.org/webgl/. Accessed 19-April-2015.

[29] Laine, M., Shestakov, D., Litvinova, E., and Vuorimaa, P. Toward Unified Web Application Development. *IT Professional 13*, 5 (September 2011), 30–36.

[30] Laplante, P. A. *What Every Engineer Should Know About Software Engineering.* CRC Press, Taylor & Francis Group, 2007.

[31] Less.js. Less language features. Available at: http://lesscss.org/features/. Accessed 20-April-2015.

[32] Lie, H. W., Çelik, T., Glazman, D., and van Kesteren, A. Media Queries. W3C Recommendation, W3C, Jun 2015. Available

at: http://www.w3.org/TR/2012/REC-css3-mediaqueries-20120619/. Accessed 20-April-2015.

[33] LORETO, S., AND ROMANO, S. P. Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts. *Internet Computing, IEEE 16*, 5 (September 2012), 68–73.

[34] LUBBERS, P., ALBERS, B., AND SALIM, F. *Pro HTML5 Programming*. Apress, 2010.

[35] LUBBERS, P., AND GRECO, F. HTML5 Web Sockets: A quantum leap in scalability for the Web. Available at: http://www.websocket.org/quantum.html. Accessed 18-April-2015.

[36] MARKUS LANTHALER AND CHRISTIAN GÜTL. Towards a RESTful Service Ecosystem. In *4th IEEE International Conference on Digital Ecosystems and Technologies (DEST)* (Dubai, April 2010), IEEE, pp. 209–214.

[37] MEHTA, N., SICKING, J., GRAFF, E., POPESCU, A., ORLOW, J., AND BELL, J. Indexed Database API. Available at: http://www.w3.org/TR/IndexedDB/. Accessed 13-April-2015.

[38] MESBAH, A., AND VAN DEURSEN, A. An Architectural Style for Ajax. In *The Working IEEE/IFIP Conference on Software Architecture* (Mumbai, January 2007), IEEE, p. 9.

[39] MESBAH, A., AND VAN DEURSEN, A. Migrating Multi-page Web Applications to Single-page AJAX Interfaces. In *11th European Conference on Software Maintenance and Reengineering, 2007. CSMR '07* (Amsterdam, March 2007), IEEE, pp. 181–190.

[40] MIKKONEN, T., AND TAIVALSAARI, A. Web Applications — Spaghetti Code for the 21st Century. In *Sixth International Conference on Software Engineering Research, Management and Applications* (August 2008), IEEE, pp. 319–328.

[41] MIKKONEN, T., AND TAIVALSAARI, A. Apps vs. Open Web: The Battle of the Decade. In *Proceedings of the 2nd Workshop on Software Engineering for Mobile Application Development* (Santa Monica, California, USA, MSE'2011), pp. 22–26.

[42] MONTEIRO, F. *Learning Single-page Web Application Development*. Packt Publishing, 2014.

[43] Mullany, M. 5 Myths About Mobile Web Performance. Available at: http://www.sencha.com/blog/5-myths-about-mobile-web-performance/ (2013). Accessed 14-April-2015.

[44] Network, M. D. Using files from web applications. Available at: http://developer.mozilla.org/en-US/docs/Using_files_from_web_applications. Accessed 19-April-2015.

[45] Osmani, A. *Developing Backbone.js Applications*. O'Reilly Media, 2013.

[46] Palo, A. HelsinkiJS & DevOpsFinland january 2015 meeting. Available at: http://developers.almamedia.fi/helsinkijs-devopsfinland-january-2015/. Accessed 16-April-2015.

[47] Parisi, T. *WebGL: Up and Running*. O'Reilly Media, 2012.

[48] Peacock, R. Distributed architecture technologies. *IT Professional 2*, 3 (2000), 58–60.

[49] Pilgrim, M. *HTML5 Up and Running*. O'Reilly Media / Google Press, 2010.

[50] Pohja, M. *Web Application User Interface Technologies*. PhD thesis, Aalto University, May 2011.

[51] Pomonis, T., Christodoulou, S. P., and Gizas, A. B. Towards Web 3.0: A Unified Development Process for Web Applications Combining Semantic Web and Web 2.0 Technologies. *Engineering Management Reviews (EMR) 2*, 2 (June 2013), 45–53.

[52] Popescu, A. Geolocation API Specification. W3C Recommendation, W3C, Oct 2013. Available at: http://www.w3.org/TR/2013/REC-geolocation-API-20131024/. Accessed 18-April-2015.

[53] Ranganathan, A., and Sicking, J. File API. W3C Working Draft, W3C, Sept 2013. Available at: http://www.w3.org/TR/2013/WD-FileAPI-20130912/. Accessed 19-April-2015.

[54] Rosa, A. D. Isomorphic JavaScript Applications – the Future of Web? Available at: http://www.sitepoint.com/isomorphic-javascript-applications/. Accessed 16-May-2015.

[55] Ruderman, J. Same-origin policy. Available at: `http://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy/`. Accessed 14-April-2015.

[56] Russell, A., Song, J., and Archibald, J. Service Workers. W3C Working Draf, W3C, Feb 2015. Available at: `http://www.w3.org/TR/2015/WD-service-workers-20150205/`. Accessed 02-June-2015.

[57] Sass. SASS (Syntactically Awesome StyleSheets). Available at: `http://sass-lang.com/documentation/file.SASS_REFERENCE.html`. Accessed 20-April-2015.

[58] Schepers, D., Moon, S., Brubeck, M., and Barstow, A. Touch Events recommendation by W3C. Available at: `http://www.w3.org/TR/2013/REC-touch-events-20131010/`. Accessed 18-April-2015.

[59] Skvorc, D., Horvat, M., and Srbljic, S. Performance Evaluation of Websocket Protocol for Implementation of Full-Duplex Web Streams. In *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th* (Opatija, Croatia, May 2014), IEEE, pp. 1003–1008.

[60] Smus, B. *Web Audio API: Advanced Sound for Games and Interactive Apps*. O'Reilly Media, 2013.

[61] Taivalsaari, A., and Mikkonen, T. The Web as an Application Platform: The Saga Continues. In *Software Engineering and Advanced Applications (SEAA)* (2011), 37th EUROMICRO Conference, IEEE, pp. 170–174.

[62] Taivalsaari, A., Mikkonen, T., Ingalls, D., and Palacz, K. Web Browser as an Application Platform. In *Software Engineering and Advanced Applications* (September 2008), 34th Euromicro Conference, IEEE, pp. 293–302.

[63] van Kesteren, A. Cross-Origin Resource Sharing. W3C Recommendation, W3C, Jan 2014. Available at: `http://www.w3.org/TR/2014/REC-cors-20140116/`. Accessed 18-April-2015.

[64] WHATWG. HTML Living Standard by WHATWG. Available at: `http://html.spec.whatwg.org/multipage/`. Accessed 17-April-2015.

[65] Yuping, J. Research and Application of Ajax Technology in Web Development. In *IEEE Workshop on Electronics, Computer and Applications* (May 2014), IEEE, pp. 256–260.

[66] Zbierski, M., and Makosiej, P. Bring the Cloud to Your Mobile: Transparent Offloading of HTML5 Web Workers. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science* (Singapore, December 2014), IEEE, pp. 198–203.