Expertise
and insight
for the future

Ville Valkonen

Lessons learned developing a large-scale progressive web application

Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Bachelor's Thesis

10 May 2019

Author	Ville Valkonen
Title	Lessons learned developing a large-scale web application
Number of Pages	39 pages
Date	10 May 2019
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Professional Major	Digital Media
Instructor	Kari Aaltonen, Principal Lecturer

The goal of this thesis was to learn modern open-source web technologies and implement a large-scale progressive web application project using them.

The methodologies used for describing the application's features were user-centered and mainly leveraged user stories. The technologies selected for the project were based on the project's requirements and the best practices established in the industry.

In user-centered software development, the aim is to define software features from the user's point of view by telling short user stories which describe who should be able to do what and why. In the development stage, the best practices established in the industry include the usage of open-source technologies, modularity, bundling, and containerization technologies to achieve similar development and production environments. In the production stage demand based automatic horizontal scaling of the production environment is required and can be achieved by leveraging containerization technologies and distributed systems management tools.

The explanations of the numerous established practices and open-source technologies used in the project and in modern web application development, in general, can be used as reference for building large-scale progressive web applications.

The project's scale turned out to be an overwhelming hindrance for it to be implemented in a timely manner. A great deal of progress was made towards the project's architectural and detailed design, but the project remains incomplete. However, the skills and understanding required for its implementation were acquired and the project's design and development continue in the future.

Keywords	progressive, web application, development

Tekijä Otsikko Sivumäärä Aika	Ville Valkonen Opittua suuren verkkosovelluksen kehityksessä 39 sivua 10.5.2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	mediatekniikka
Ammatillinen pääaine	digitaalinen media
Ohjaaja	yliopettaja Kari Aaltonen

Opinnäytetyön tarkoituksena oli oppia käyttämään moderneja avoimen lähdekoodin verkkoteknologioita ja toteuttaa suuren mittaluokan progressiivinen verkkosovellus niitä hyväksi käyttäen.

Projektin ominaisuuksien kuvaukseen käytetyt menetelmät olivat käyttäjäkeskeisiä ja hyödynsivät lähinnä käyttäjätarinoita kuvaamaan sovelluksen ominaisuuksia. Projektissa käytetyt teknologiat perustuvat projektin vaatimuksiin ja alan vakiintuneisiin käytäntöihin.

Käyttäjäkeskeisessä ohjelmistokehityksessä ominaisuuksia pyritään määrittelemään käyttäjän näkökulmasta kertomalla lyhyitä käyttäjätarinoita, jotka kertovat, kenen täytyy pystyä tekemään mitä ja miksi. Alan vakiintuneisiin käytäntöihin kuuluu rakennusvaiheessa avoimen lähdekoodin teknologioiden hyödyntäminen, modulaarisuus, ns. bundlaus sekä säiliöintiteknologioiden käyttö kehitys- ja tuotantoympäristöjen samankaltaisuutta tavoiteltaessa. Tuotantovaiheessa tarvitaan kysynnän määrään perustuvaa automaattista tuotantoympäristön horisontaalista skaalausta, mikä voidaan tavoittaa hyödyntämällä säiliöintiteknologioita sekä hajautettujen järjestelmien hallintatyökaluja.

Esiteltyjä vakiintuneita käytäntöjä sekä projektissa käytettyjen avoimen lähdekoodin teknologioita voidaan käyttää referenssinä progressiivisten verkkosovellusten suunnittelussa ja rakennuksessa.

Ylitsepääsemättömäksi haasteeksi osoittautui projektin suuri mittaluokka, minkä takia projektin toteutusta ei saatu tehtyä ajallaan. Vaikka projektin arkkitehtuuri ja yksityiskohtainen suunnitelma edistyi huomattavasti, sitä ei saatu valmiiksi. Projektin toteuttamiseen tarvittavat taidot ja ymmärrys tulivat hankituksi, ja sen kehitys jatkuu tulevaisuudessa.

Avainsanat	progressiivinen, verkkosovellus, kehitys

Contents

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BASE	Basic Availability, Soft-state, Eventual consistency
BEM	Block Element Modifier naming convention
CAP	Consistency Availability Partitioning
CI / CD	Continuous integration, delivery and deployment
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
HTML	Hypertext Markup Language
HTTPS	Hypertext Transport Protocol Secure
IDE	integrated development environment
JS	JavaScript
JSX	XML-like syntax extension to ECMAScript
MITM	man-in-the-middle attack
MVP	minimum viable product
npm	Node package manager

OWASP	Open Web Application Security Project
PACELC	Partitioning Availability Consistency Else Latency Consistency
PWA	Progressive Web Application
PHP	PHP Hypertext Processor
SASS	Syntactically Awesome Style Sheets
SQL	Structured Query Language
TC39	Technical Committee 39
TLS	Transport Layer Security
U2F	Universal Second Factor
UI	user interface
W3C	The World Wide Web Consortium
XSS	cross-site scripting

1 Introduction

In 2014 I had an idea for a web platform. This small idea grew to be a project so large that over the last five years I have spent countless hours on its refinement and development while learning about modern practices, technologies and progressive web applications. In this thesis I take an overview of the process the project has gone through, the technologies it is based on and what I have learned while designing and developing it.

First, I will talk about what constitutes software development, the processes and stages. Then, I will focus on the web, its history and look at web application development tools and practices. I will explain some of the historical context to help understand the reasons why web technologies and practices have evolved towards the current state. Finally, I will go through a list of modern web development tools and practices and explain how they work.

I will then shift focus to the project, how it got started, and go over the iterations the project has gone through. As I go over the iterations, I will discuss the tools and technologies selected for the project and which factors led to their selection. I will also discuss some development practices and the methodologies I have used during development and why I found them the most effective. Finally, I will talk about the project's current state and the future.

2 Modern web application development

2.1 The development process

Software development, and particularly, web application development consists of many different components which should be considered from the beginning of the development process. There are many approaches to software development, each suited for different types of projects and teams. Determining which approach should be chosen depends on the nature of the project, the requirements and the team working on it.

Generally, all projects go through the following stages:

- Conceptual development
- Requirements analysis
- Architectural design
- Detailed design
- Implementation, unit testing
- Systems testing
- Deployment, maintenance and improvement

Software development practices have evolved throughout the decades and can be classified into two categories: plan-driven models and agile development models. The waterfall model is plan-driven and describes software development as a linear process where each stage leads to the next. However, realistically the process is never as linear as the model suggests because the entire domain of the project must be known from the beginning, which is rarely the case. What naturally follows is the waterfall model with feedback, which has the option to return to previous stages as necessary. This approach works for large new projects with a clear vision of the product, but scheduling may become a problem if new information about requirements emerges, or if there are flaws in the design. Agile development models are iterative and are well-suited for projects with a goal of incremental improvement. They recognize the flaws of their predecessor models and improve upon them. [1]

2.2 The stages of development

Regardless of the process, the application development process begins with defining the minimum requirements of the application based on the concept. The requirements include the target audiences and use cases i.e. who the application is for and what purpose it serves. Defining the requirements limits the scope of the project and helps to focus on the essential. The resulting set of definitions is the minimum viable product, MVP.

The target audiences and use cases are the defining factors for the application's architectural design which include the following: the software stack, target platforms and form factors, so the choices made at the early stages are crucial. The software stack is the set of technologies and tools software is built with and it includes languages, frameworks, libraries and databases. In the detailed design stage, the application's logic and data model are designed, and in the implementation stage they are implemented as code. Defining unit tests at the beginning of a project is a good way to measure the completion

rate of a project: when all the tests pass, the project can move on to the next stage. In the systems testing stage, a test production environment is set up and tested before deploying the code into the actual production environment. Lastly, the code is deployed into the production environment and maintained with the help of monitoring and logging tools while improvements are developed. Continuous integration, delivery and deployment practices greatly alleviate the process and are discussed later.

2.3 The road to progressive web applications

JavaScript is the backbone of all modern web applications. JavaScript standardization and web browsers have a complicated history of competing companies and factions driving their interests [2]. ECMA Technical Committee 39 or TC39 is the group tasked with the standardization of the ECMAScript language [3]. The language is called ECMAScript due to trademark reasons; therefore, JavaScript is a commercial name for ECMAScript [2]. In 2015, after years of conflict within the committee, TC39 finally released ECMAScript 6, later named ECMAScript 2015 [4], and changed their standardization process to one where proposals must go through four stages to be accepted as new standard ECMAScript features [5]. Because of the nature of this process, browser implementations will always lag behind the ECMAScript standard, creating the JavaScript feature gap shown in figure 1.

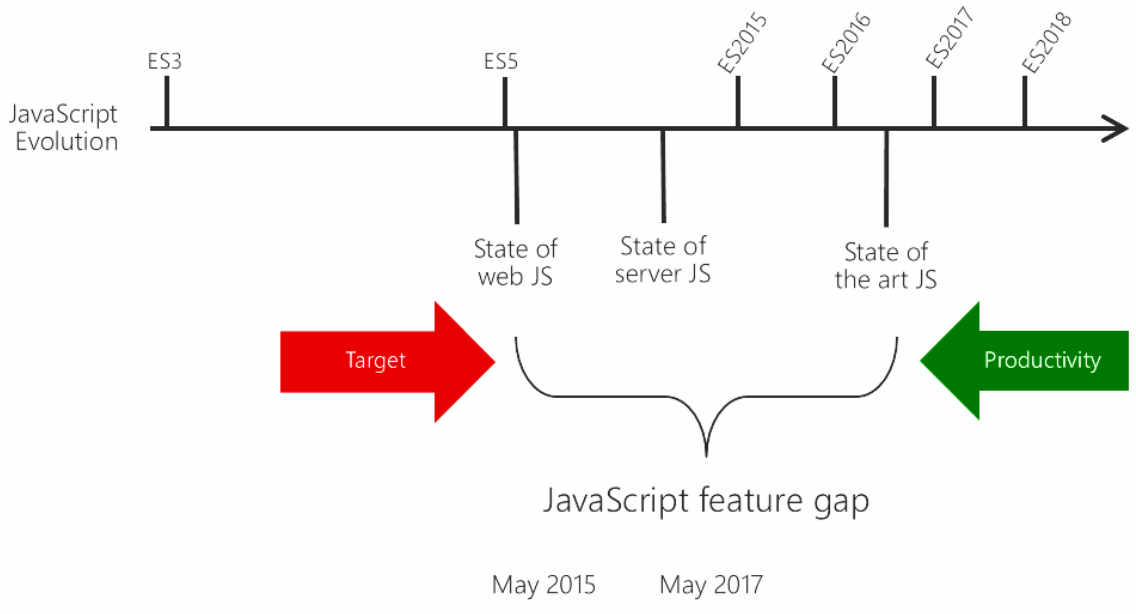

Figure 1. The JavaScript feature gap [6]

The staggering number of different browser versions has spawned services like Can I use [7] to track which ECMAScript features are supported by each browser version. Compilers like Babel [8] were developed to bridge the gap between browser implementations and the newest versions of JavaScript. Babel enables developers to use the newest JavaScript syntax while developing but provide a compatible experience for older browsers.

Near the turn of the last decade web technologies were maturing, the HTML5 specification was being developed and the ECMAScript 5 specification was recently released. Websites were mostly traditional with enhancement via JavaScript and the primary language on the server was PHP. Desktop applications were the norm and smartphones and native mobile applications were on the rise as iOS and Android were competing for the mobile market share. The idea of the web as an application platform started making its way into developers' minds to consolidate the native mobile application development process and provide a good experience for all platforms without maintaining multiple versions for different platforms and operating systems. In 2012, Facebook's Mark Zuckerberg described their heavy early investment into HTML5 as the biggest strategic mistake because the web platform was simply not ready yet [9].

Since 2008, Google had started making drastic improvements in the performance of their V8 JavaScript engine as shown in figure 2.

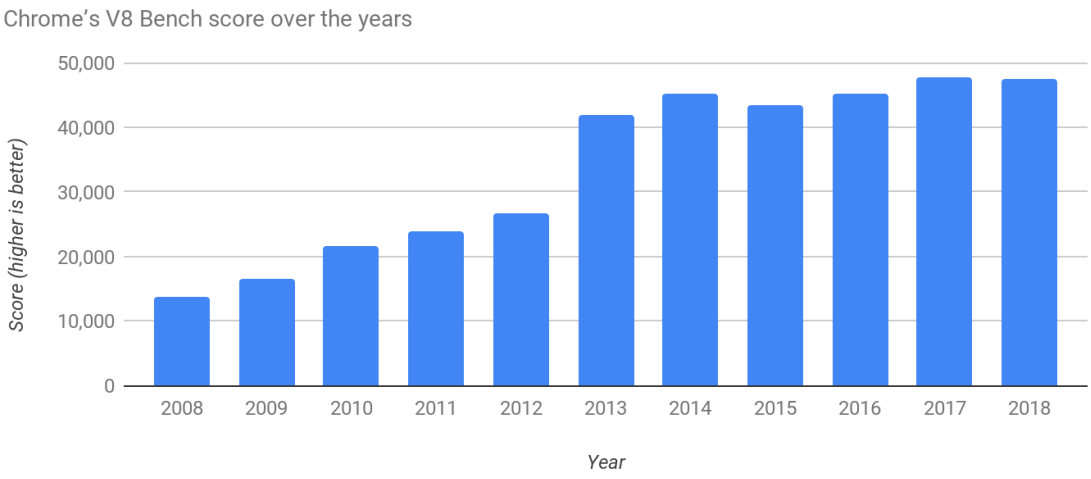

Figure 2. The performance of Google's V8 JavaScript engine over time. [10]

As the performance of JavaScript increased, so did its popularity. According to the annual Stack Overflow developer survey from 2018 [11], JavaScript has been the most used programming language since 2013, and in HackerRank's Developer Skills Report 2019 [12] JavaScript overtook Java and became the most popular language of 2018.

Since 2015 the digital service industry has been pushing towards the web platform and Progressive Web Applications or PWAs. Progressive Web Applications is a name for web applications utilizing a group of newly established technologies and practices, and it was coined by Google developer Alex Russell and his friend Frances Berriman in 2015 [13]. The word progressive refers to progressive enhancement, a term coined in 2003 by Steven Champeon and Nick Finck. The idea of progressive enhancement was to make sure web pages "deliver information to the widest audience possible" and "leave no one behind" [14] no matter what platform or browser they are using. In other words, web experiences should progressively enhance and gracefully degrade depending on the user's browser. Progressive enhancement recognized and tackled the problem of browser feature mismatch which stems from the JavaScript standardization process. In his article from 2015, Russell eloquently articulates:

> "It happens on the web from time to time that powerful technologies come to exist without the benefit of marketing departments or slick packaging. They linger and grow at the peripheries, becoming old-hat to a tiny group while remaining nearly invisible to everyone else. Until someone names them." [13]

Russell goes on to postulate that this development may be an "inevitable consequence of a standards-based process and unsynchronized browser releases" [13]. Combined with the modern techniques and practices, they make a powerful combination.

Russell defined progressive web applications with a list of nine key attributes:

- responsiveness: user interface adapts to any form factor
- connectivity independence: functional while not connected to the internet
- application-like interactiveness: interactions do not induce a full-page refresh
- freshness: always up-to-date via service worker
- safety: served securely using TLS / HTTPS
- discoverability: identifiable as applications via W3C Manifests and search engine optimized
- re-engageability: notifications

- installability: add to home screen prompt
- linkability: easy to share, no manual installation, i.e. zero-friction [13]

Today, Google's developer guide characterizes PWAs as reliable, fast and engaging [15]. Since 2015, Google has made notable additions to Russell's list:

- fast first load times via performance budgets
- cross-browser functionality
- increased perceived performance via a fast and performant user interface
- linkability to each page
- continuity of task flows and navigation
- cache-first networking via service worker
- notification control, customization, timeliness, precision and relevance
- modern API usage, including: Credential Management API, History API and Payment Request API

The beating heart that makes PWAs possible is the service worker, shown in figure 3.

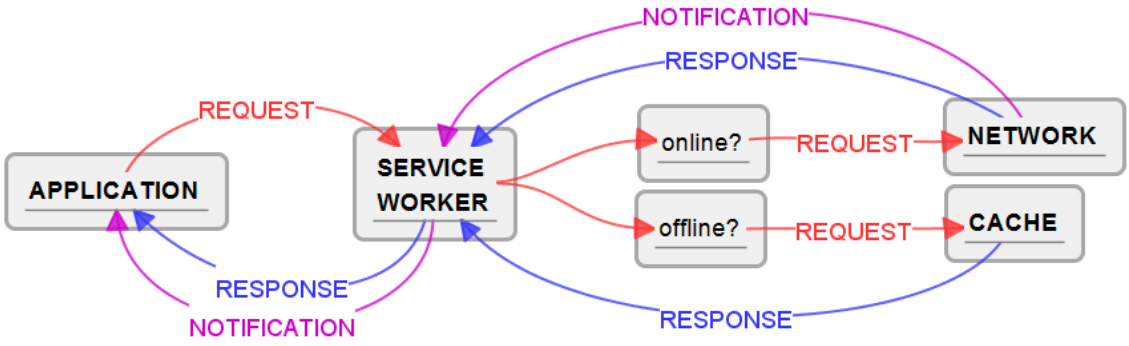

Figure 3. The functionality of the service worker

The service worker is a piece of software that handles all the user actions, network requests and responses, the application cache, notifications and background synchronization. PWAs have many benefits that native applications do not, illustrated by table 1.

Table 1. Comparison between traditional web pages, native mobile applications and progressive web applications. [13] [15] [16]

Statement	Traditional web pages	Native mobile applications	Progressive web applications

Network connection reliant	yes	no	no
Has an application-shell	no	yes	yes
Up-to-date by default	yes	possibly	yes
Accessible via URL	yes	no	yes
Supports push notifications	no	yes	yes
Marketplace reliant	no	yes	no
Works in most browsers	yes	N/A	yes
Requires full application to use one part of the application	N/A	yes	no
Maintenance of multiple versions for different platforms	no	yes	no
Indexable by search engines	yes	no	yes
Must be installed	no	yes	no
Can be added to home screen or desktop	yes	yes	yes
Browser feature reliant	yes	no	yes

PWAs combine all the good aspects of websites and native applications and do not suffer from the same disadvantages. The only disadvantage PWAs have is their dependence on web browser implementations.

2.4 The tools and their history

The technologies presented in the following chapters have become corner stones of modern web application development.

Every programming project needs **version control**. There are several tools that handle versioning in different ways, but the most common tool is git. It was created by Linus Torvalds in 2005 [17] and gained popularity especially among open-source developers ever since. Git is a simple command line tool which stores all versions of the project files to a .git -folder. Changes to files are committed with the 'git commit' -command which creates a new version of the code repository. Git supports all the necessary functions, such as returning to old versions and merging commits, and handles projects of all sizes.

HTML, CSS and JavaScript are the core of every web application. When JavaScript was famously designed within six months in 1995, it was intended to be used as a scripting language for small client-side tasks in the browser [2]. After 2010, as JavaScript increased in popularity, developers began expanding its intended use case by building full scale applications. Managing these large code bases was increasingly difficult as the size of the projects grew. The tooling, such as, code editors and extensions of the time were simply not ready to accommodate building large applications with JavaScript. The biggest issue with incorporating JavaScript into large-scale application development was the dynamic type system. For tooling to be able to point out the errors in development, it must know the intent of the developer when assigning variables, function arguments, return values and so on. The dynamic type system made this impossible because, in JavaScript, there is no way to specify types. This is why a team at Microsoft started developing TypeScript and the complementary **tooling**, namely, the Visual Studio Code -editor in 2011 [18].

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript [19]. Its goal is to help catch errors as early in the development process as possible. It does this with the help of its type system, a static type checker and type definition files. The type system is briefly described as: erasable, gradual, structural, generic, inferable, expressive, object-oriented and functional [18]. The type checker analyzes the code in real time while editing and emits errors or warnings when it comes across erroneous code. TypeScript can also infer types based on static analysis if an element's type is not defined and is inferable. The strictness of TypeScript's type checker is very flexible and can be adjusted from not checking anything to checking every possible detail with the “--strict” flag. To run TypeScript code, it must be compiled to JavaScript with the TypeScript compiler, which simply removes the type information. The compiler also has options to transform newer versions of JavaScript into older versions, such as Babel. All in all, TypeScript in an excellent tool for developing web applications because it has the necessary tools to make JavaScript development manageable.

The success of TypeScript can be attributed to three things: the merits of the language itself, the Visual Studio Code -editor [20] and the Definitely Typed Github repository [21]. Visual Studio Code is a highly customizable IDE, which has a large community and extensions for most programming languages. As increasing numbers of developers discovered TypeScript its popularity grew but transitioning projects to TypeScript was not as simple as it may have seemed. Most popular packages for libraries and frameworks did

not include type definition files for TypeScript, so developers could not take advantage of TypeScript's most basic purpose while using these packages. The answer was an open-source Github repository of type definition files. The Definitely Typed Github repository is the home to all the TypeScript type definition files for the libraries that do not include type definitions. In 2017, it was the most active repository on Github with over 42 000 commits [18]. Since then, an increasing amount of open source projects have begun including official type definition files reducing the need for community created definitions. In the 2019 Stack Overflow developer survey, TypeScript is the third most popular language and Visual Studio Code is the most used development environment among all respondents [22].

Historically, the **backend** of web sites and services was mainly responsible for serving HTTP requests and handling the web service logic, often with PHP. The Apache web server and Microsoft's IIS have been the most popular tools in the past, but other solutions have been gaining market share in the recent years as shown in figure 4.

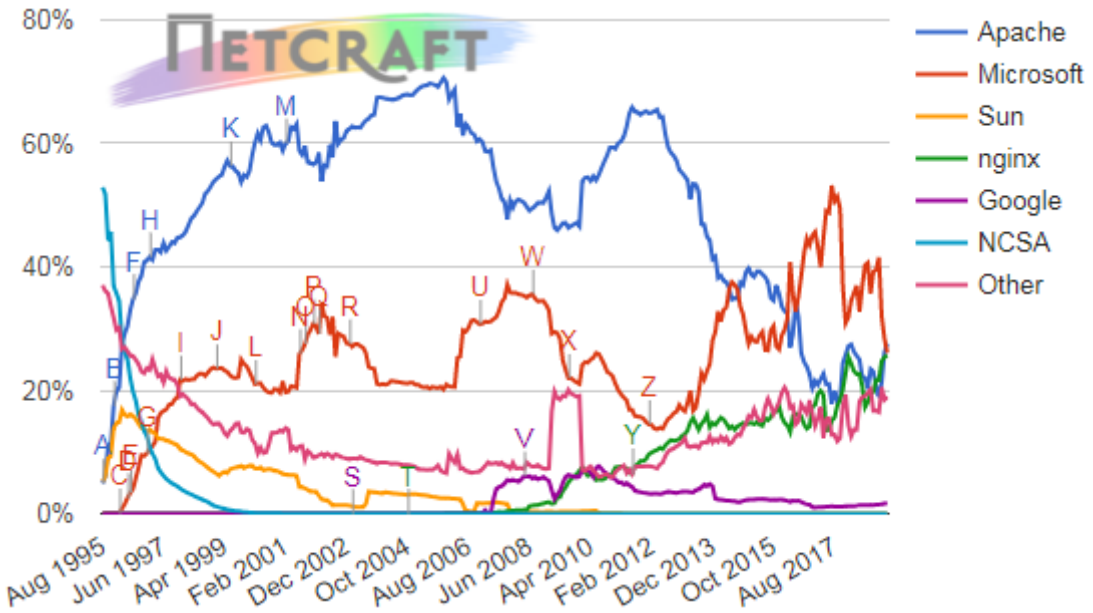

Figure 4. Web server market share of all web sites [23]

Different backend pieces of software represent different approaches and solve the same problem differently while also offering different or additional functionality. PHP had been the de facto language for handling server-side logic for years along with the Apache web server, but the benefits of Node.js in web application development has drawn the attention of many developers.

Node.js is a **JavaScript runtime environ**ment built on Chrome's V8 JavaScript engine [24]. In a nutshell, it is described as follows:

> "Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices." [25]

The first version of Node.js was released in May 2009 [26]. Node.js is commonly used for server-side JavaScript for web applications and as a platform for desktop applications. With PHP it is common having to wait for the application to do things, which is due to synchronous code execution blocking any other actions before an active task is complete. This is experienced by users as decreased perceived performance. If the user cannot make further actions while a time-consuming task is being handled, they feel they are not in control. The biggest benefits of Node.js compared to PHP are:

- its asynchronous event-driven nature, which enables building fast and responsive applications,
- modularity,
- and the fact that it enables developers to use one language in development.

The module system of Node.js enables developers to write JavaScript as modules that can be loaded and used by other modules. Modules define their imports and exports, which makes it possible to build tree-structures of modules.

Modules and packages are very similar, but they serve the same purpose. Packages must include a package.json -file, which defines the dependencies of the project. Projects depend on other modules or packages when they import and use their exported features, such as, functions or variables. The file can also define scripts for any kinds of tasks, such as running the application and tests.

The Node Package Manager, npm [27], is both a registry for packages and a command line tool, and it was released in January 2010 [28]. It allows for downloading all the package.json specified packages from the online registry with a single command, which makes beginning development very fast. By 2015, the npm registry had surpassed all the other popular package registries of different languages and surpassed 100 000 packages. JavaScript's popularity exploded after 2015, and recently the number of packages available on the npm registry surpassed 800 000 packages, as seen in figure 5.

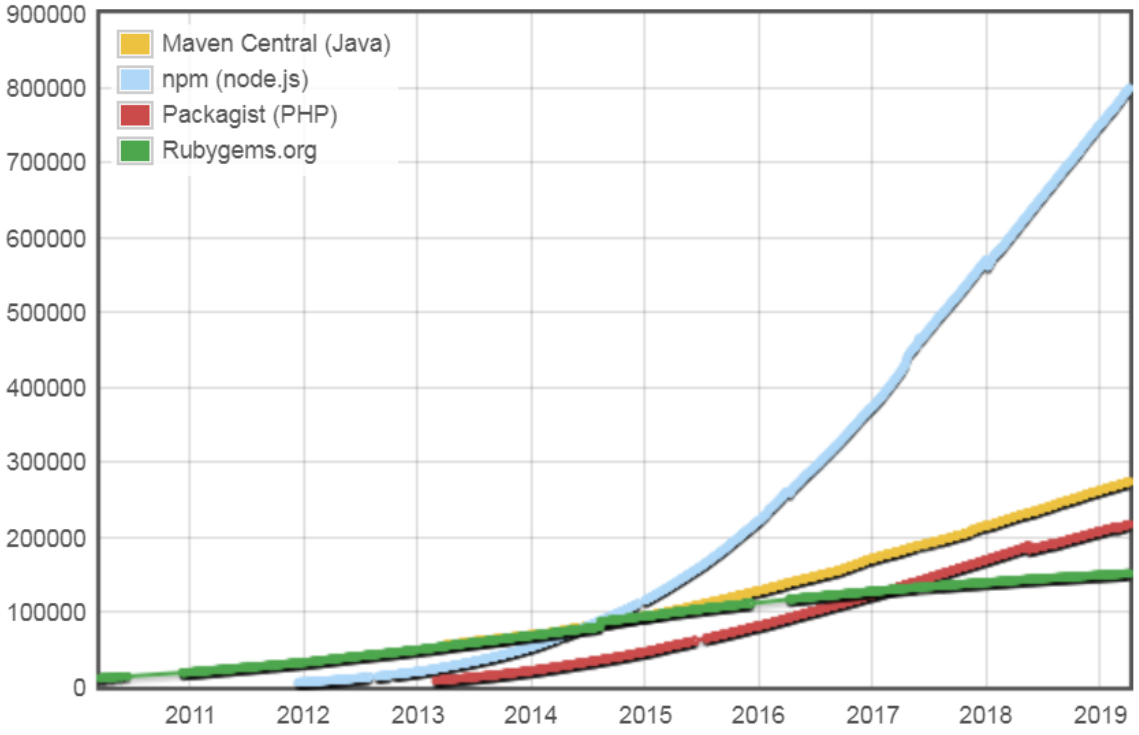

Figure 5. The number of packages available in different registries from 2010 to 2019. [29]

Node.js and npm slowly grew and were established as the basis for JavaScript development. Today, Node.js is the most commonly used tool according to the Stack Overflow Developer Survey 2019 [30].

Node.js started a new modular era for JavaScript. It used the CommonJS require-function to define modules. The idea of the require-function is that a file requires or depends on some external resources to be able to run, hence, the code can be split into several different files and required in many different places. The problem was that browsers did not support modules, so a tool was needed to run the modular code and build the application into a non-modular browser compatible bundle. Several different tools were developed for this purpose from 2011 through 2013, including webpack.

In general, **module bundling** is an operation where a project's modular source code is built i.e. the dependencies are determined and placed into the resulting bundle. Tobias Koppers started developing webpack in March 2012 [31] with the main purpose of bundling CommonJS modules into browser compatible JavaScript files. In addition to bundling, webpack's functionality can be extended with loaders and plugins, which enable bundle optimization, compatibility and bundling performance features.

Loaders are transformations applied to modules, and they are typically used to compress or compile code into earlier JavaScript versions for browser compatibility. Upon running webpack, it searches for specified file names in the project directory, e.g. all files with ".js" file extension and applies the specified loaders. Plugins, on the other hand, have access to webpack's entire compilation lifecycle, enabling them to extend the compilation functionality in various ways [32]. In fact, webpack's main process itself operates as a plugin. An example of a bundle optimization plugin is the SplitChunksPlugin. It intelligently splits the resulting output into chunks based on configurable conditions. The point of splitting the code into smaller units is to optimize the user experience of the application. For example, when the user loads the application for the first time, they should not have to download the entire application for it to work. The user may never visit some parts of the application, so the parts that can be loaded later should be loaded when necessary. Lastly, webpack can eliminate unused code based on an analysis of the module structure and statements, resulting in optimal bundles.

React is Facebook's library for building **user interfaces** [33], open-sourced in May 2013. Traditionally, the content, presentation and behavior of a web page are handled separately by HTML, CSS and JavaScript respectively [34]. This model works well for building traditional web pages and small web services, but for large dynamic and interactive applications it quickly becomes cumbersome and difficult to manage. React's approach recognizes this and solves the problem by embracing JavaScript as the primary language and implements a model for creating modular applications with components [35].

React applications are essentially trees of nested components. Components represent individual parts of the application which manage their own state. React components are JavaScript functions or classes that render out a virtual representation of the UI into a virtual DOM, which is synchronized with the actual DOM [36]. Components can take arguments as inputs which can determine the component's state and what it should render [37]. React recognizes that the user interface logic is inherently coupled with rendering logic, which is why it utilizes an HTML-like syntax called JSX [35]. JSX is a way of representing components and what they take in as arguments. Although JSX is not strictly required it is a good visual aid in development. Components can also have lifecycle methods which have to do with the moments they are created or removed from the DOM to manage memory resources [38].

Other user interface libraries or frameworks like Angular or Vue have similar approaches, but React's approach has proven to be the most popular as shown in figure 6.

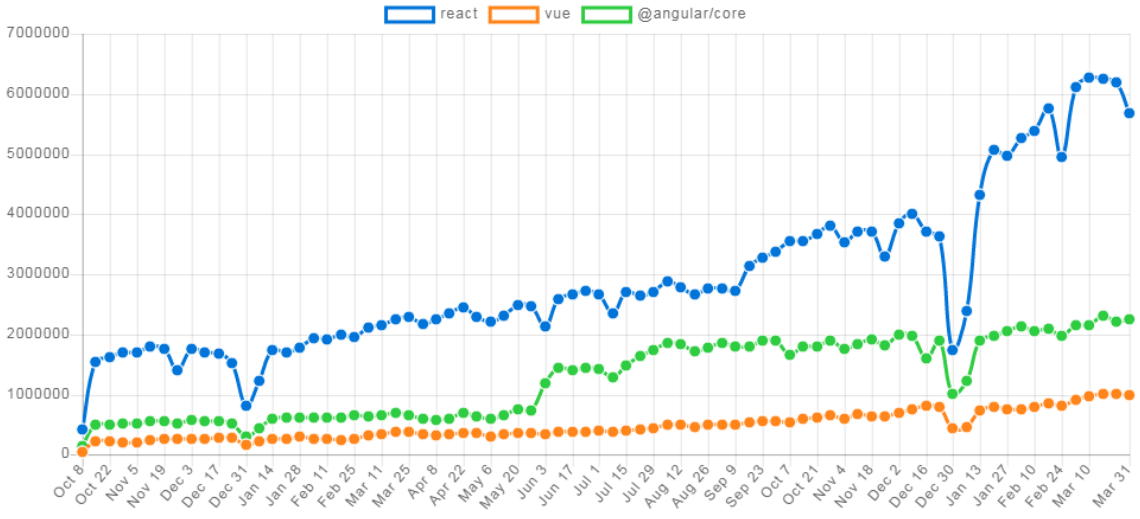

Figure 6. The number of weekly downloads over the past two years. [39]

Although React had built-in **state management**, it was not suitable for large applications. The problem was application state becoming unpredictable, non-deterministic and difficult to reason about when the application logic became increasingly complicated involving server responses, caching and complex UI state [40]. The answer was the Flux pattern, shown in figure 7, for which Facebook started developing tools in 2014 [41].

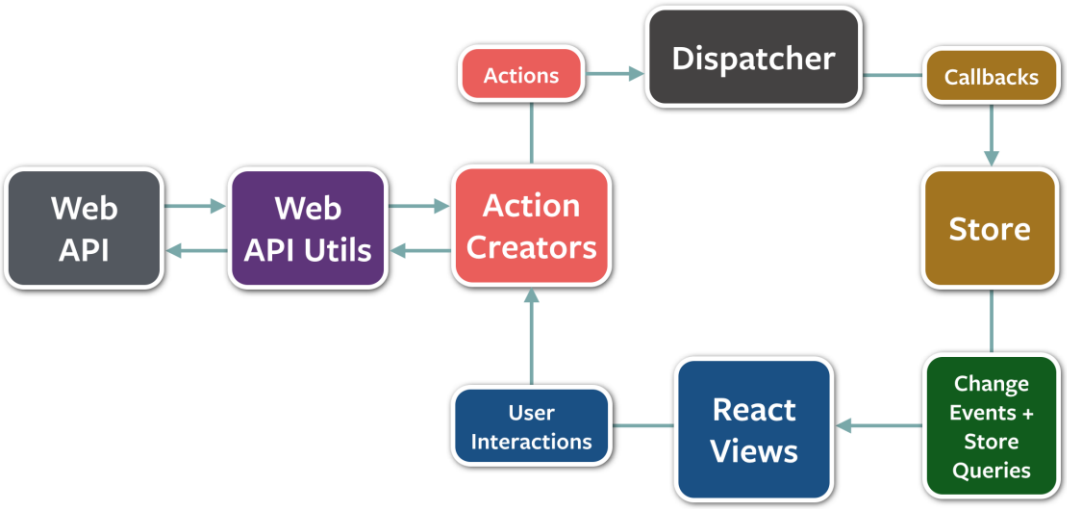

Figure 7. The Flux pattern [42]

The main point of the pattern was to create a unidirectional data flow to make the application state predictable, but Facebook's own implementation of the pattern turned out not to be the best solution to the issue [43]. The pattern can be summarized into three parts:

- user interactions create actions,
- actions transform a store,
- and changes to the store reflect the application state.

Redux introduced the best solution in 2015 [43] with three principles [44]:

- a single source of truth
- immutable state
- pure functions determine state

Where Flux has an approach with one action dispatcher and multiple stores, Redux uses one store as the "single source of truth" to determine the application state [45]. A Redux application's state is immutable and can only be replaced with a state produced by reducer functions. The store transforming reducer functions must be pure functions i.e. functions whose output relies solely on the arguments passed to the function. This means that for a given input the reducer function always produces the same output; hence, the output is predictable [44]. It is also possible to time travel to a previous state because the state can never be altered, only replaced. Deterministic application state benefits large applications because any bugs that occur can be tracked down, replicated and fixed much easier.

Although the Redux library is very useful, it may be replaced in the future. Recently, a feature called hooks was included in React. It was inspired by the separation of concerns inside React. Currently, code related to one concern must be managed in multiple lifecycle methods. Hooks aims to consolidate the management of lifecycle methods and produce cleaner abstractions. The Redux pattern can be applied with hooks in a cleaner way. [46]

Cascading Style Sheets, CSS is the language for defining how elements look on the web. CSS has always been problematic in the context of web applications due to its cascading nature in combination with dynamic views. The problems with CSS are like those JavaScript had: differences in browser implementations, scoping and optimization.

These issues were not trivial even for traditional web pages, but they were less important. Throughout years, CSS has evolved to meet some of the needs of modern web applications, e.g. with the advent of media queries, but not all of them. Language extensions like SASS [47] added features like nesting, while frameworks like Bootstrap offered robust responsive grid systems and prebuilt components [48]. While these were good solutions for traditional pages, they were inadequate for large web applications. As modular JavaScript was growing and bundlers such as webpack were becoming common, CSS had to be fit into the model as well.

The Block Element Modifier (BEM) [49] naming convention by Yandex was among the first CSS scoping solutions which became a best practice by achieving an abstraction equivalent to components. BEM, however, was only a convention and did not eliminate the possibility of global naming collisions. Eventually, the solution was CSS modules which implemented automated hash-based scoping as a webpack loader and was added to webpack in 2015 [50]. As webpack became an established workflow, to meet the shortcomings of bundlers in CSS, PostCSS, a tool for CSS syntax transformations was developed [51]. Many CSS tools operate as PostCSS plugins including CSSnano, Stylelint, Autoprefixer and cssnext, which address compression, syntax checking, different browser vendor implementations and converting new CSS to older syntax. PostCSS catered to most of the needs of modular web applications.

The first CSS-in-JS libraries, e.g. JSS were developed in 2014 [52]. JSS and many others scoped styles to components in various ways, but their implementations contained several compromises, such as not supporting certain CSS features. The additional advantage CSS-in-JS offered compared to CSS modules was the ability to leverage JavaScript functions to determine styles. Styled-components, released in 2016, solved most of the compromises by using ECMAScript 2015 tagged template literals to wrap CSS into JavaScript [53] [54]. This approach made it possible to use all of CSS in styled-components.

It included theming, automatic critical CSS, class naming and prefixing [55] and ultimately became the most used CSS-in-JS library as shown in figure 8.

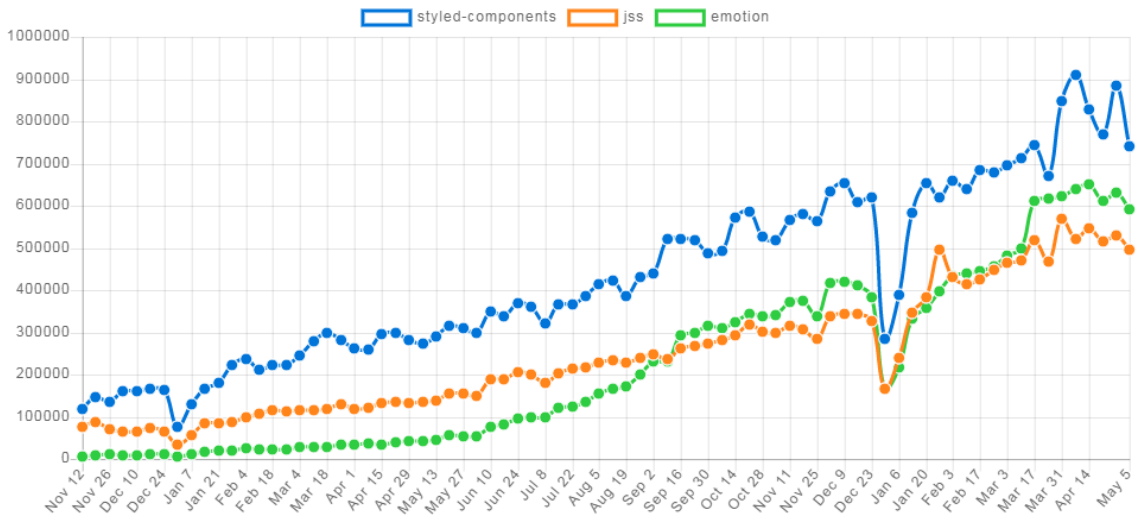

Figure 8. The popularity of component styling libraries [56].

Unit testing is a practice used for testing individual functions and classes of an application to make sure they work as intended. Unit tests should be a normal part of the build process to detect errors before integration into an existing code base or deployment. Well-crafted unit tests are fast, independent, repeatable, self-validating and timely. Each test should test only one thing and either pass or fail. Tests should not depend on each other or slow down development by being slow or difficult to run. There are two ways to approach unit testing. The first way is to implement a feature and write tests afterwards. The second stems from test-driven development practices where tests are written ahead of time or simultaneously with the implementation. This increases development efficiency because the requirements, design and implementation must carefully be taken into consideration. Unit tests provide a clear list of items to fix when changes are made, or dependencies updated. Code coverage and data coverage measure how much of the code is being tested and the types of data used for the testing. Ideally, all lines of code should be covered by tests with valid, invalid, too much and too little data, and all cases should be handled gracefully. [1, p 193-200]

Testing JavaScript web applications is typically done with the help of frameworks or libraries like Jest [57], Mocha [58] or Chai [59], which are currently the most popular JavaScript testing tools. Jest surpassed Mocha as the most popular testing tool in 2018, as shown in figure 9.

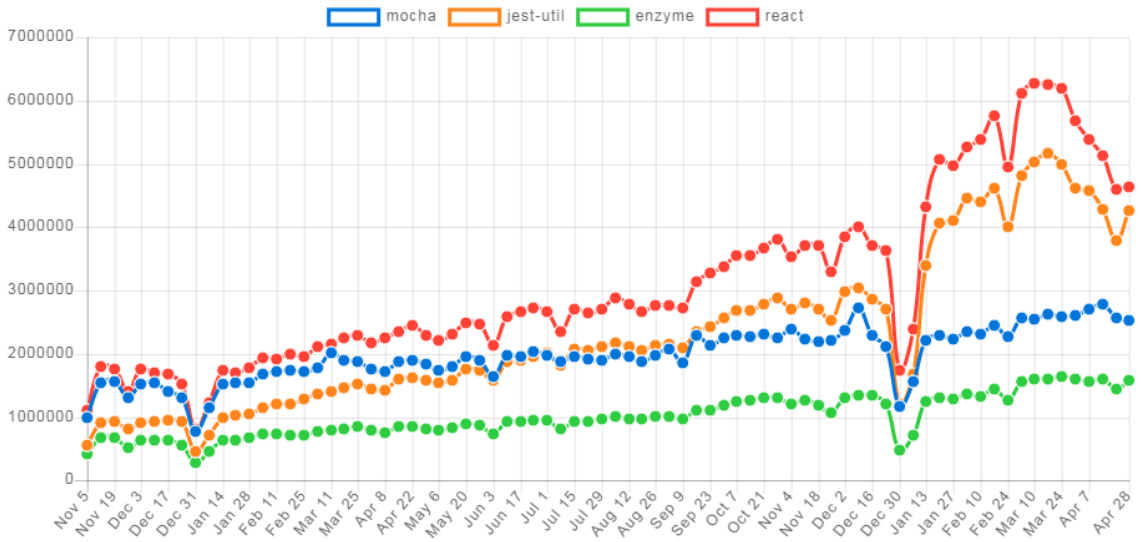

Figure 9. The popularity of JavaScript testing libraries. [60]

Jest supports all the typical testing features like assertations, mocking and code coverage reports. Mocking is used to mimic functions or APIs and test against them. Testing against real APIs should be done in the system testing stage.

Large modern web services rely on **distributed database systems** serving as persistent data storage. Historically, relational databases have been the most used databases in large enterprise applications, but the increasing requirements of modern web applications and services have spawned new approaches to data storage and management in the form of NOSQL databases [61, p 183]. Relational databases store data in tables and use the structured query language, SQL to query the data, while NOSQL describes databases that typically do not rely on SQL and use a wide range of data models. Because different data models benefit different use cases, an application's data domain should be examined as carefully as possible in the early stages of development.

According to the CAP theorem [62] and its extension, PACELC [63], distributed database systems must make tradeoffs between consistency, availability, latency or partition tolerance, as shown in figures [10] and [11].

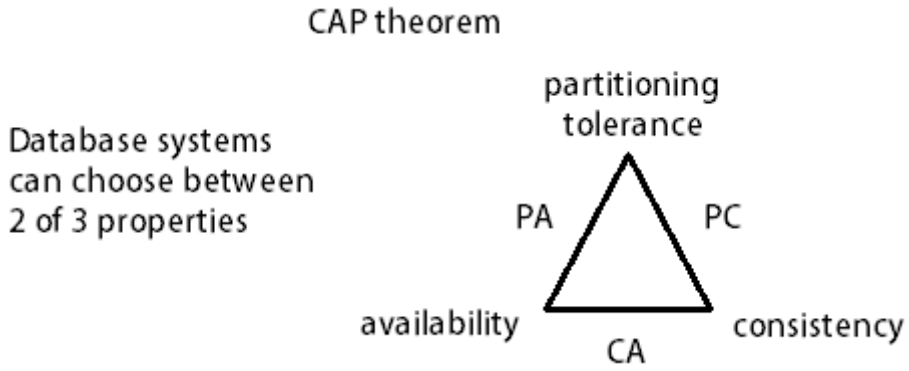

Figure 10. The CAP theorem.

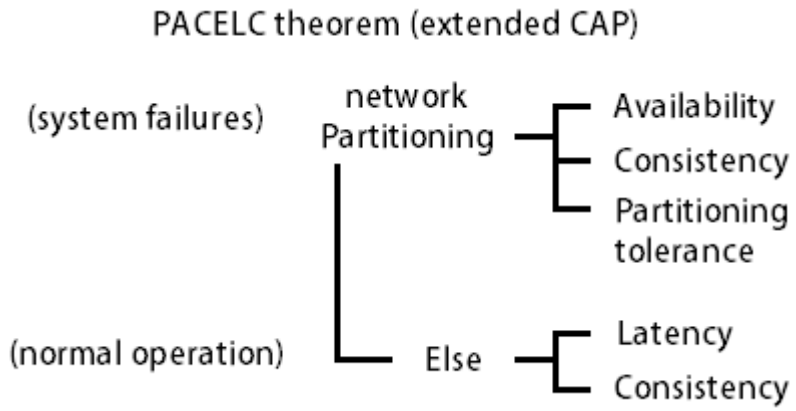

Figure 11. The tradeoffs in database systems according to the PACELC theorem.

Partitioning refers to system failures or unusual network circumstances that inevitably occur in real world environments, e.g. hardware or network failures. Because database systems can be designed to operate in the ways described by CAP and PACELC, application developers must choose which database characteristics are most important in their particular use case.

Many use cases require ACID transactions which are atomic, consistent, isolated and durable, and therefore, guarantee a safe environment for data operations. A more optimistic approach BASE offers basic availability, soft state and eventual consistency. BASE data stores compromise on consistency to provide better scalability and resilience without consistency guarantees. [61, p 185]

As data becomes increasingly connected it becomes harder to model and query relationally and, therefore, is best modeled as a graph. Graphs consist of nodes and relationships. There are many ways to describe graph data, but the most used are property

graphs and triples. In property graphs, nodes and relationships have properties as key-value pairs and relationships have a direction, whereas, triples store data in a subject-predicate-object data format. Graph databases excel in storing and querying highly connected data and usually have a flexible schema in contrast to the rigid schemas of relational databases. They are typically built with create, read, update and delete (CRUD) functionality for transactional systems and optimized for consistency and availability. The way graphs are queried depends on the implementation and the query language, but the patterns are similar. [61, p 196]

The most important factor in graph databases is the underlying data model and implementation. Some graph databases use relational, document or key-value stores as the backing service and expose a graph model, while others have a native graph model or a hybrid multi-model. Currently, the most popular approach to graph data storage and processing is Neo4j which is a property graph database [64]. It stores nodes, relationships and properties and connects them using doubly linked lists under the hood. The structure can be traversed by chasing pointers which eliminates the need for a global index, hence the name index-free adjacency [61, p 146 & 196]. It makes querying millions of nodes extremely efficient as opposed to relationally joining massive tables. However, Neo4j is based on a single server architecture which limits its potential to serve as a database for distributed systems. Rather than scaling horizontally it must scale vertically, which is not sustainable for large web applications. All approaches are advantageous in the use cases they were designed for, and the application requirements determine which approach is the most suitable.

Another consideration in large web applications is caching which is used for storing common query results. If a web service does not use a caching layer, it must constantly query the database for common results, which is potentially much slower and creates unnecessary database load. Redis is currently the most used in-memory data structure store that can be used as a database, cache and a message broker. [64] [65]

Containers are lightweight operating environments that are used for running software in isolation. Multiple containers can run on a single machine. Where virtual machines virtualize hardware, containers virtualize the operating system and share the host operating system kernel. They are an abstraction that appears to be an independent machine from both the container's point of view and from the outside. [66]

Containerization of web services was popularized when Docker, a software for building and running containers was launched in 2013 [67]. It was based on existing Linux features such as chroot, namespaces and cgroups which were originally designed to protect applications from each other. In 2013, containers had been in wide use by the likes of Google internally, but their potential was discovered after Docker introduced the concept of including the application code and all its dependencies in a container. The concept created an abstraction that isolates the application from the host operating system, enabling the easier and more efficient separation of application and infrastructure. Docker focused on the needs of developers and systems operators and enabled data centers to shift focus from machine-oriented to application-oriented by making containers the standard unit of software. [66] [68]

Developing and deploying containerized applications is beneficial in many ways compared to virtual or physical machines:

- Development is easier, faster and more efficient.
- The decoupling of application and infrastructure eliminates the underlying infrastructure as a concern for developers.
- The application environment is consistent in development, testing and production, and facilitates rapid development, integration and deployment, reliable and frequent updates or easy rollbacks.
- Containers are portable, lightweight, fast to start up and shut down, and hence more resource efficient and have a smaller attack surface.
- The hardware resources provided to a container can be limited via configuration, resulting in predictable performance.
- Containerization simplifies application monitoring and logging. [68] [69]

Dockerfiles are sets of instructions which describe how docker images should be built, and docker images become containers at runtime. Dockerfiles can build on top of other Dockerfiles, enabling easy versioning and differentiation of containers. Docker Hub is a library of Docker images, and currently it hosts over 100 000 open-source Docker images from software vendors and the community [70].

Large scale applications require **automatic container management** to achieve demand based horizontal scaling. Kubernetes is Google's open-source system for containerized application management, automatic deployment and scaling [71]. Its design goal was to make it easy for developers to deploy and manage complex distributed systems, while utilizing containers [68].

Kubernetes defines a cluster as a set of machines, called nodes, that run containerized applications. A cluster can have several worker nodes and at least one master node. The worker nodes communicate with the master via the Kubernetes API that it exposes. The master manages the cluster and coordinates all its activities, including scaling, maintaining a desired application state or rolling out updates. A pod is the smallest deployable object, and it is essentially a wrapper around a container or a set of containers. The pod provides the containers a local network environment or a shared storage volume as configured. [72] [73]

When a containerized application pod is deployed to a cluster, the master schedules it to run on the worker nodes according to a deployment configuration. Once running, the state of the deployment is monitored by a deployment controller. If a worker node goes down due to a hardware failure or maintenance, a new node is assigned to maintain the application's desired state. New nodes are started or shut down based on CPU utilization thresholds. [74]

Continuous integration, delivery and deployment, or CI / CD is a set of modern development practices. They build on top of one another and form a development culture whose aim is to produce working code continuously. If all three are embraced, the result is an automated release process where each working commit goes through an automated testing pipeline and is deployed to production. [75]

The goal of continuous integration is avoiding massive merge conflicts by constantly merging code back into the project's main branch while gradually fixing small conflicts. This helps keep the code unified and the development effort synchronized. However, continuous integration requires a CI server, e.g. Jenkins [76] or Travis [77] that runs unit tests to prevent erroneous code from being pushed. [75]

Continuous delivery relies on a strong culture of continuous integration and a high degree of test coverage. It requires the master branch to always be ready to release and automates the deployment process, but the team decides when to release. Continuous deployment automates the whole process by automatically testing, deploying and pushing each passing commit to production. [75]

2.5 Security

Security should be a top concern for web application developers. Developer teams should always aim to follow the current security best practices and stay up to date on security breaches and vulnerabilities. Most technology related security considerations fall under a few key topics and can be prevented by implementing and enforcing modern security measures and practices.

There are many types of threats on the web with different targets and goals. Those targeting users, seeking to compromise their credentials include cross-site scripting (XSS) and phishing sites. Those targeting web services include denial-of-service attacks seeking to make services unavailable, SQL injection attacks seeking to compromise the web service and those making use of stolen credentials. The biggest security problem on the web today is phishing. According to Google, 81% of hacking-related account breaches in 2017 were caused by weak or stolen passwords, and multi-factor authentication has been the industry's collective response to the issue. However, traditional two-factor authentication one-time passwords are vulnerable to phishing sites that implement man-in-the-middle attacks (MITM) if users are not careful. [78]

The universal 2nd factor, U2F was developed by Google and Yubico in 2014, and its aim was to provide a strong verification factor via a physical USB-key [79]. To make U2F more compatible with web services, Google, FIDO Alliance and the W3C developed a new recommendation for web authentication called WebAuthn [80]. WebAuthn includes the domain name of the party attempting to authenticate in the authentication mechanism which eliminates the possibility of successful MITM attacks. Additionally, once a device has been registered with the USB-key, it can be used to provide fast authentication with biometrics-based re-authorization. Providing access to these improved authentication mechanisms is the first step toward a more secure web and should be leveraged in application development in the future. [78]

Although U2F keys are a step in the right direction, the majority does not have access to them, hence, other means of creating secure web services must be prioritized. Communicating over HTTPS and other secure protocols is the most important security measure. A strict content security policy [81] should also be in place to mitigate cross-site scripting and data injection attacks. All user inputs should be assumed to be malicious and treated accordingly. Although many modern libraries, such as React and Angular

handle user inputs securely by default [82] [83], steps should be taken to ensure this. Due to the nature of open-source it is impossible to be completely certain that open-source dependencies are secure, but measures can be taken to reduce the effects of possible vulnerabilities. Tools like npm audit [84] and snyk [85] should be used to scan dependencies for known vulnerabilities. Also, running containers and Node.js as non-privileged users is essential to prevent unauthorized actions in case of a security breach.

Passwords and other user data should be handled with extreme caution using bcrypt or Argon2 as the recommended hash functions. Argon2 won the password hashing competition in 2015 [86] and bcrypt [87] has remained secure since 1999. Dropbox published their bcrypt-based solution in 2016 [88] shown in figure 12 which is still considered best practice by OWASP [89].

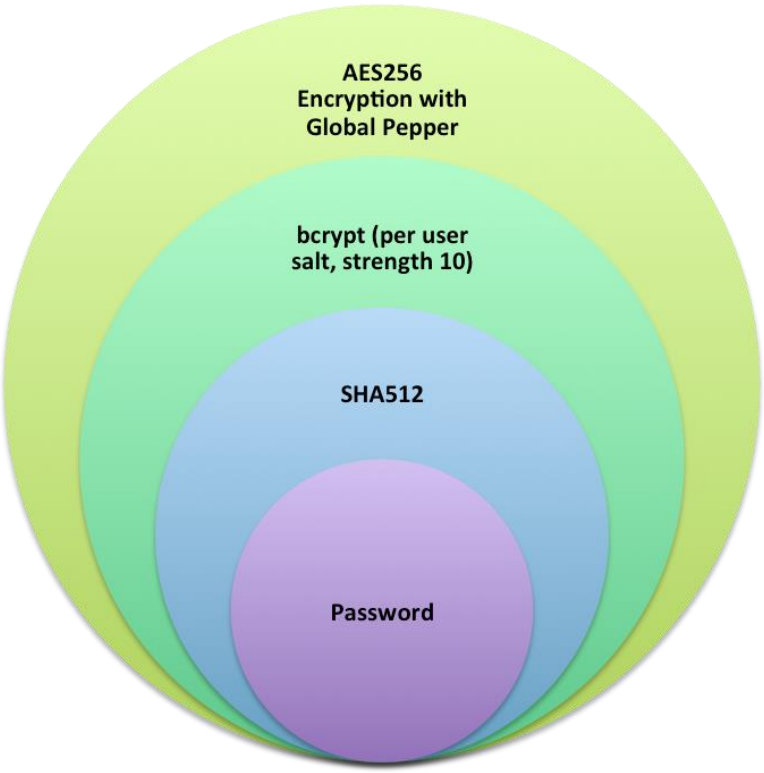

Figure 12. The layers of encryption in Dropbox's password storage mechanism. [88]

However, Argon2 was designed to be a highly secure password hashing algorithm for the modern age. Because today it is only a matter of how much money can be used towards cracking passwords on a data center scale, Argon2 implemented a mechanism that requires a high amount of processing and memory resources to produce a single

guess. An Argon2-based npm package called secure-password [90] was implemented by Emil Bay in 2017 and should be considered for new JavaScript projects.

3 The development process of the case project

Over the last several years I have been working on a large-scale web application development project. Figure 13 illustrates the different phases of the project.

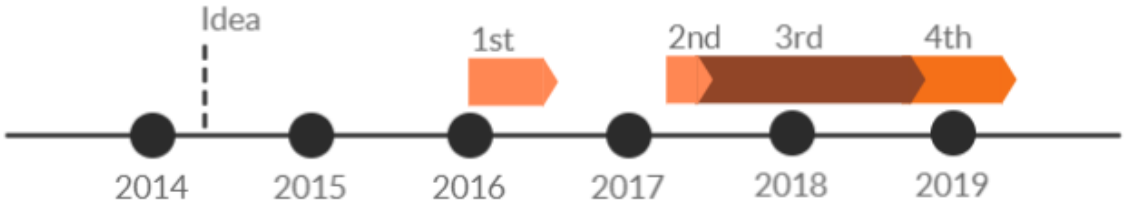

Figure 13. A timeline of the different phases of the project.

The goals of the project have been to design, implement and deploy a large web service based on the idea from 2014.

3.1 The project begins as an innovation project, first iteration

We started working on the project's first iteration with my peer Ilpo Oksanen when the Innovation project course started in January 2016. Although I had designed and refined the idea for over a year between studies when I had time, it was to no extent refined. At this point, many things were not thought through at all, including many features, their technical implementation, and the data model. My friend Ilpo was miles ahead of me with his knowledge and passion for programming, and he helped me understand many of the concepts, practices and code libraries we used. The goal was to implement some of the application's main functionalities. This definition of the goal was not specific enough, so it had to be narrowed it down to something manageable. I had listed my ideas for the application's features and different functionalities, but I knew it would be impossible to implement all of them during the course, so we decided upon the bare minimum features one would expect from a modern web service like creating a user profile, logging in, editing the profile and other basic interactive features. We set up a git repository for version control and began working.

First, we had to decide what the technology stack should be, i.e. what languages, libraries, server and database the application should use so we started with the most familiar area to both of us: the languages and the server. The natural first picks were JavaScript, HTML, CSS, PHP and the Apache server with which we had the most experience. We also needed a place to deploy the application to make it available online. Many of our peers and Ilpo himself had used Heroku's cloud application platform before so we hosted the project there.

Our knowledge about databases was limited to what we had learned on a course on information management. The course had mostly focused on relational data management with tables and joins using MySQL and MariaDB as examples so the natural first pick was a relational one. Our first option was MySQL, but we decided to test PostgreSQL because it was natively supported by Heroku. We chose a development stack consisting of Windows, Apache, PHP and PostgreSQL

We knew that Facebook's React was gaining momentum as a UI library but used Google's AngularJS, jQuery and VelocityJS due to familiarity. We also used SASS as a CSS extension to make use of its nesting feature.

The data model was a pain point because we were using a relational database. The problem was my lack of planning for the data model, and that relational data is not flexible. This was a much bigger problem than I realized at the time because when modeling relational data, the domain of the data should be fully known and understood to avoid future problems. In other words, you must know the questions you want to ask the data before you design the data model. With a relational model, massive data refactoring will be necessary if the requirements change for the application's data.

We knew that the data the application would eventually produce would be complicated to handle in a relational way, but we did not know of any alternatives other than NoSQL document databases. We were busy with the application's implementation and did not want to spend our time researching a topic we knew next to nothing about, so we asked for guidance from our information management teacher Aarne Klemetti. We described the application's data requirements and he pointed us toward graph databases, namely Neo4j and OrientDB. I researched both and was immediately excited about Neo4j and its query language Cypher. Neo4j uses a property graph model where data is modeled as nodes and relationships between nodes. The query language was also immediately

easy to understand. The data modeling capabilities of Neo4j had powerful implications for the application. OrientDB seemed much too cumbersome with its SQL-like queries. It was clearly designed by and for SQL-minded people which I was not. We decided to move to Neo4j in the future but for now we did not have the time for it.

The semester was coming to an end and the project was incomplete, but we had to present the results of the project. We put together a presentation and a demonstration of the project's current state and presented our results. By the end we had managed to make a working login form and a very rough version of some of the application's functionalities. We were not happy with the results and agreed to return the final report when we had at least reached the defined requirements.

I was enthusiastic about the project's progress and pressed on with the design but Ilpo was unavailable for the entire summer, so the project lost all momentum. The following semester's schoolwork turned out to be very demanding and as a result our work on the project was halted until March 2017 when we resumed working on it. Table 2 shows the results of our work during the first iteration.

Table 2. The results of the first iteration

Requirement	**Solution**
Version control	git
Languages	HTML, CSS, JavaScript, PHP
Web server	Apache
UI	AngularJS, jQuery, VelocityJS
Styling	SASS
Database	PostgreSQL
Cloud platform	Heroku

3.2 The project continues a year later, second iteration

Our work on the project resumed when most other demanding schoolwork was over in March 2017 and continued until the beginning of June as shown in figure 13. During the year, Ilpo had learned to use React, Node.js, npm, webpack and PostCSS and was

enthusiastic about porting the project over to these technologies. He showed me some examples of the technologies and I was quickly convinced. We began working on the second iteration.

Along with Node.js and React I was introduced to modular JavaScript programming. None of the code we previously wrote was modular. Modularization is among most powerful concepts in programming but the situation with JavaScript modules was particularly confusing and complicated. ECMAScript 2015 modules were recently standardized [4], but the CommonJS module syntax was the most common way to handle modules at the time.

Switching to Node.js and managing packages with npm was a bliss. The way dependencies were managed with a package.json file helped the project grow and made it easy to understand and manage. Understanding webpack's bundling and configuration logic opened my eyes to what was possible. We configured webpack loaders to transform and optimize JavaScript and CSS, although, at this point it was not very important. During the first iteration we considered React but chose Angular instead. Having learned about the merits of React, it was the only option going forward.

These software stack additions made developing the application more efficient and manageable. We ported our existing Angular code to React components and worked on React routing. Previously we used SASS as the CSS language extension, but we were aware of the complications we would face with it especially in a web application as complicated as this. Once we learned how hash-based class name scoping was configured the problem was gone.

Over a few weeks we included all the new techniques and successfully implemented React routing. In early April we began considering Meteor as an application framework. Meteor [91] is an opinionated and managed framework whose promise of "building great apps faster than ever" was enticing. We learned to use Meteor during a few development sessions and it seemed very promising. Nevertheless, we ran into problems very soon after we began porting the application over. The package for PostCSS did not work and we wondered whether that would be the case for more packages. Making decisions based on a framework's support was not sustainable, so we left Meteor behind. The Meteor community simply was not large enough.

Towards the end of spring we had a working local application running and we were looking into the data model. I had worked on a rough version of the database for a while, but we never quite reached the point where we could run the application alongside the database. The user interface and basic functionality was beginning to look good and I was eager to continue but summer was upon us again. Our summer jobs took most of our time and our cooperation on the project was halted again ending the second iteration. Table 3 shows the results of our work during the second iteration.

Table 3. The results of the second iteration

Requirement	**Solution**
Version control	git
Languages	HTML, CSS, JavaScript
Server	Node.js + ExpressJS
Package manager	npm
Bundler	webpack
UI framework	React -> Meteor -> React
State management	React
Styling	PostCSS
Database	Neo4j

3.3 The project enters a slow phase, third iteration

Throughout the summer of 2017 I kept researching and discovered many new technologies and techniques that I wanted to include in the project. Ilpo was busy with his work even after the summer and phased out of the project at this point. The project's overall design needed further development and refinement so Ilpo's department in a way freed me from the rush to make potentially bad design decisions as soon as possible just to get the application implemented fast. In a sense, it was both a setback and a relief.

First, I discovered Visual Studio Code and TypeScript. The examples given by Anders Hejlsberg in a presentation at Microsoft Build 2017 [92] made me realize the power and potential of TypeScript. Previously I had used Atom as my code editor, but Visual Studio

Code was miles ahead of it in terms of performance and features. It was exactly what I wish I had from the day I started programming.

Soon after, I discovered react-boilerplate, a maintained starting point for new React based web applications [93]. It included many important libraries and taught me some best practices which steered the project towards a good direction. I knew about Facebook's create-react-app [94], but the way it structured its files by functionality (e.g. actions, components, containers and reducers in different folders) put me off to it. However, react-boilerplate's way of grouping files by feature (e.g. folders like App, Menu, Sidebar, Footer, etc.) was much more appealing, so I began studying how it works. In programming, many things are a matter of preference, and the project structure is one of them.

React-boilerplate also used Redux, a library for state management in large applications. I had heard about it before and knew it was important, so I learned how it works and realized how important it would be for the project.

For CSS react-boilerplate used styled-components. CSS had always been a topic I had very little interest in mostly due to the complications and side effects the global scope introduced in web application development. Discovering styled-components and CSS-in-JS in general was both exciting and a relief.

React-boilerplate also included libraries for two important areas large applications should address: accessibility and internationalization. In general, for a web site to be considered accessible, its structure must be semantically correct. This is to help people with disabilities use the website as intended. For example, a screen reader software often cannot extract meaning from code and it is up to the developers to convey that meaning via the web page's structure. For example, buttons must be button HTML elements instead of div elements with a JavaScript onClick attribute and so on. Internationalization, on the other hand, has a much clearer meaning. In web development it is easy to forget that the semantics of HTML matters for many people. Being aware of these two areas broadened my view on web development.

By the end of the summer of 2017 I had discovered Docker and was trying to use it as a development environment. I was interested in the prospect of being able to develop the project in a containerized environment to be as close to a production environment as possible while developing. Ultimately, I realized that other aspects are much more

important at this stage of development and shifted my focus but the knowledge I had gathered about Docker would not go to waste.

React-boilerplate also had many quality-of-life features like templates to help create components and the required scaffolding faster. Though, the big downside react-boilerplate had was that it was written in JavaScript. I was enthusiastic about TypeScript because it helps discover errors early and forces thinking through requirements. It also helps new developers joining the project later understand the code faster. I looked for good alternatives written in TypeScript but none of them were as good as react-boilerplate, so I started converting it to TypeScript myself. The TypeScript compiler gave hundreds of compile errors and I was unable to solve all of them. The types of different function return values and variables provided by different libraries were hard to determine without type definition files for some of the libraries. The overarching problem was that writing React applications with TypeScript was not yet completely supported. I stopped my conversion efforts and started writing my own version. I worked on it for a while until I realized I need to learn React and Redux properly to be able to build the application.

While my learning efforts continued I was also designing the application's features and trying to define how it should work. The way I approached designing the features was user centered. I found user stories to be the most effective method for defining features for the application. I asked design questions to reflect on a feature: how it should work and what are the consequences of a design. A good design is as simple as it can be while being intuitive. A good design does the intended task, stays out of the user's way and satisfies all the relevant design questions. The questions it cannot answer are the most important ones because that is where design decisions must be made. I made many of these decisions to define the limits and functionality of many of the features but because the application has potential to be extremely large, I found it difficult to answer all the questions decisively.

Over the spring and summer of 2018, I learned about React and Redux as well as the details of TypeScript via their respective documentation and examples, and in September, I started building the application's component structure and routes. In October 2018, I had most of the component structure planned for multiple major features some of which had many sub features and many of them had to work in combination. As the list of components grew, I was dealing with increasing amounts of complexity. Each time I thought about how an interconnected feature should work it was hard to fully take into

consideration all the aspects of the feature. The problem was the incomplete design of the data model. I had thought that the graph data model would arise from the requirements naturally and that fitting it into Neo4j would be trivial but that was not the case. I realized I need to properly define the data model of the application to be able to answer the design questions and define how each feature should work. I decided to stop working on the application code until the data model was satisfactory. Table 4 shows the results of my work during the third iteration.

Table 4. The results of the third iteration

Requirement	**Solution**
Version control	git
Languages	HTML, CSS, TypeScript
Server	Node.js + ExpressJS
Package manager	npm
Bundler	webpack
UI framework	React
State management	Redux
Styling	styled-components
Database	Neo4j
Containerization	Docker

3.4 The project's fourth iteration and current state

I began working on the fourth iteration in November 2018 by mapping all the required node and relationship types and how they are connected. Having to think about all the features from the point of view of the data really helped me clarify my thoughts on how the features work and interact with each other. Thinking about the required queries even sparked new feature ideas because seeing the data model made me realize what was possible. In November, I thought the modeling would not take longer than a few weeks but three months later the process was still incomplete. During this phase, I learned that Neo4j would not be able to handle the application's requirements by itself. The graph structure was perfect for describing the shape of the data, but the actual data had to be

stored elsewhere due to the technical limitations of Neo4j. Eventually, I had to put development on hold as I started working on my thesis.

Currently the project is defined by 70 topics of user stories with multiple sub topics which describe 40 interconnected features with sub features. The project's architectural design is on a solid foundation apart from the database. The data model is nearly complete with 35 different node types and 25 different relationship types each with several properties and information about which node types can be connected via which type of relationship. The application's features still require further development with the help of the completed data model. I have a strong vision for the application and its different sections, but several design questions require informed answers for which more research needs to be done.

4 Conclusion

The project began in 2016 as an innovation project with limited goals. The initially defined requirements were a good starting point, but they were nowhere near the actual requirements to fulfill even a tenth of the vision for the application. As I listed the user stories describing the required features to fulfill the vision, I realized that I must extensively define the core features and their data model to be able to implement them. I knew very clearly what the service would look like, but the details of its functionality were covered in unanswered design questions that required more research and information to be answered. Towards the end of 2018 it became clear that the goal of implementing the project would not be reached on time. In the end the goals of the project were:

- learning to create modern web applications using open-source libraries and tools, and
- designing and implementing the case project, a large-scale web application.

The goal of designing and implementing a working large-scale web application was not met, although, a great deal of progress was made, but the skills needed to create modern web applications have been acquired. The project was too ambitious, and the bar was set too high for one person to implement in a reasonable amount of time. Because several design questions are still open and waiting for informed answers, in the future I will continue with the project's development by answering all such design questions, by

designing the data model based on the decisions made, and finally, by implementing the features based on the design and the data model.

References

1 Dooley, J. 2011. Software Development and Professional Practice.

2 Peyrott, S. 2017. A Brief History of JavaScript. Blog. Online. <https://auth0.com/blog/a-brief-history-of-javascript/> Published 16.1.2017. Read 7.4.2019.

3 TC39 Programme of work. Online. <https://www.ecma-international.org/memento/tc39.htm#Programme%20of%20work> Read 7.4.2019.

4 ECMAScript 2015 Language Specification. Online. <https://www.ecma-international.org/ecma-262/6.0/> Read 7.4.2019.

5 The TC39 process for ECMAScript features. Online. <http://2ality.com/2015/11/tc39-process.html> Published 15.11.2015. Read 7.4.2019.

6 Hejlsberg, A. 2017. Microsoft Build 2017. Presentation. Online. <https://channel9.msdn.com/Events/Build/2017/B8088> Published 8.5.2017.

7 Can I use -service. Online. <https://caniuse.com/> Read 7.4.2019.

8 Babel compiler. Online. <https://babeljs.io/docs/en/> Read 7.4.2019

9 Zuckerberg, M. 2012. TechCrunch Disrupt conference interview. Online. <https://www.youtube.com/watch?v=o2wPzjH2xwA&t=653> Published 21.9.2012.

10 Osmani, A. 2018. 10 years of Speed in Chrome. Chromium Blog. Online. <https://blog.chromium.org/2018/09/10-years-of-speed-in-chrome_11.html> Updated 11.9.2018. Read 17.3.2019.

11 Stack Overflow developer survey. 2018. Online. <https://insights.stackoverflow.com/survey/2018#technology-_-programming-scripting-and-markup-languages> Read 5.4.2019.

12 HackerRank Developer Survey. 2019. Online. <https://research.hackerrank.com/developer-skills/2019> Updated 28.1.2019. Read 5.4.2019.

13 Russell, A. Progressive Web Apps: Escaping Tabs Without Losing Our Soul. Article. Online. <https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/> Updated 15.6.2015. Read 4.4.2019.

14 Champeon, S. Finck, N. 2003. Inclusive Web Design For the Future. Presentation. <http://hesketh.com/publications/inclusive_web_design_for_the_future/> Read 6.4.2019.

15 Progressive Web App Checklist. 2019. Google Developers. Web documentation. Online. <https://developers.google.com/web/progressive-web-apps/checklist> Updated 12.2.2019. Read 4.4.2019.

16 Modi, A. 2018. Progressive Web Apps - Blurring the Lines between Web and Mobile Apps. Article. Online. <https://www.topsinfosolutions.com/blog/progressive-web-apps-web-mobile-apps/> Updated 20.7.2018. Read 4.4.2019.

17 Git release v0.99. Github. Online. <https://github.com/git/git/releases?after=v0.99.2>

18 Heijlsberg, A. 2018. What's New in TypeScript. Microsoft Build 2018. Online. <https://www.youtube.com/watch?v=hDACN-BGvI8&t=704> Published 8.5.2018. Read 8.4.2019.

19 TypeScript home page. Online. <https://www.typescriptlang.org/index.html> Read 8.4.2019.

20 Visual Studio Code -editor home page. Online. <https://code.visualstudio.com/> Read 8.4.2019

21 Definitely Typed repository. Github. Online. <https://github.com/DefinitelyTyped/DefinitelyTyped> Read 8.4.2019.

22 Stack Overflow developer survey. 2019. Most Loved, Dreaded, and Wanted Languages. Online. <https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted> Read 18.4.2019.

23 Netcraft web server survey. March 2019. Online. <https://news.netcraft.com/archives/2019/03/28/march-2019-web-server-survey.html>

24 Node.js home page. Online. <https://nodejs.org/en/> Read 5.4.2019.

25 Gackenheimer, C. 2013. Node.js Recipes.

26 Nodejs repository. 2009. Github. Online. <https://github.com/nodejs/node-v0.x-archive/tags?after=v0.0.2> Read 5.4.2019.

27 Node Package Manager home page. Online. <https://www.npmjs.com/> Read 5.4.2019.

28 NPM repository. 2010. Github. Online. <https://github.com/npm/npm/releases?after=v0.0.2> Read 5.4.2019.

29 Modulecounts -service. Online. <http://www.modulecounts.com/> Read 5.4.2019.

30 Stack Overflow developer survey. 2019. Other Frameworks, Libraries, and Tools. Online. <https://insights.stackoverflow.com/survey/2019#technology-_-other-frameworks-libraries-and-tools> Read 18.4.2019.

31 Webpack initial commit. 2012. README.md. Github. Online. <https://github.com/webpack/webpack/commit/2e1460036c5349951da86c582006c7787c56c543> Read 8.4.2019.

32 Webpack documentation, glossary. Online. <https://webpack.js.org/glossary/> Read 20.4.2019.

33 React library home page. Online. <https://reactjs.org/> Read 6.4.2019.

34 Flanagan, D. 2011. JavaScript: The Definitive Guide.

35 React documentation, JSX. Online. <https://reactjs.org/docs/introducing-jsx.html> Read 23.4.2019.

36 React documentation, Internals. Online. <https://reactjs.org/docs/faq-internals.html> Read 23.4.2019.

37 React documentation, Components and Props. Online. <https://reactjs.org/docs/components-and-props.html> Read 23.4.2019.

38 React documentation, State and Lifecycle. Online. <https://reactjs.org/docs/state-and-lifecycle.html> Read 23.4.2019.

39 NPM Trends, React vs. Vue vs. Angular. Online. <https://www.npmtrends.com/react-vs-vue-vs-@angular/core> Read 6.4.2019.

40 Redux documentation, Motivation. Online. <https://redux.js.org/introduction/motivation>. Updated 3.12.2018. Read 9.4.2019.

41 Flux initial commit. 2014. Github. Online. <https://github.com/facebook/flux/commit/45c856e78783f132f491d9a374a3895a5ee1076c> Read 9.4.2019.

42 Facebook Flux repository. Github. Online. <https://github.com/facebook/flux> Read 23.4.2019.

43 Redux initial public release. 2015. Github. Online. <https://github.com/reduxjs/redux/releases?after=v0.2.1> Read 9.4.2019.

44 Redux documentation, Three Principles. Online. <https://redux.js.org/introduction/three-principles> Read 23.4.2019.

45 Redux documentation, Prior Art. Online. <https://redux.js.org/introduction/prior-art> Read 23.4.2019.

46 React documentation, Hooks. Online. <https://reactjs.org/docs/hooks-intro.html> Read 23.4.2019.

47 SASS CSS language extension home page. Online. <https://sass-lang.com/> Read 24.4.2019.

48 Bootstrap CSS framework home page. Online. <https://getbootstrap.com/> Read 24.4.2019.

49 Block Element Modifier naming convention. Online. <https://en.bem.info/methodology/naming-convention/> Read 24.4.2019.

50 Dalgleish, M. 2015. The End of Global CSS. Article. Online. <https://medium.com/seek-blog/the-end-of-global-css-90d2a4a06284> Read 24.4.2019.

51 PostCSS architecture. Github. <https://github.com/postcss/postcss/blob/master/docs/architecture.md> Read 24.4.2019.

52 JSS initial commit. 2014. Github. Online. <https://github.com/cssinjs/jss/commit/f3ff17912c4d6d283a1bfc7d66ca1ac89ea3c49d> Read 24.4.2019.

53 Styled-components documentation, tagged template literals. Online. <https://www.styled-components.com/docs/advanced#tagged-template-literals> Read 24.4.2019.

54 Template literals. Mozilla MDN web docs. Online. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals> Read 10.4.2019.

55 Styled-components documentation, basics. Online. <https://www.styled-components.com/docs/basics> Read 24.4.2019.

56 NPM Trends, component styling library comparison. Online. <https://www.npmtrends.com/styled-components-vs-jss-vs-emotion> Read 7.5.2019.

57 Jest JavaScript testing framework. Online. <https://jestjs.io/> Read 1.5.2019.

58 Mocha test framework. Online. <https://mochajs.org/> Read 1.5.2019.

59 Chai assertion library. Online. <https://www.chaijs.com/> Read 1.5.2019.

60 NPM Trends, testing libraries. Online. <https://www.npmtrends.com/mocha-vs-jest-util-vs-enzyme-vs-react> Read 1.5.2019.

61 Robinson, I. Webber, J. Eifrem, E. 2015. Graph Databases, 2nd edition. Online. <https://neo4j.com/graph-databases-book/>

62 Fox, A. Brewer, E. 1999. Harvest, Yield and Scalable Tolerant Systems. Proceedings of the Seventh Workshop on Hot Topics in Operating Systems IEEE CSTCOS.

63 Abadi, D, J. Consistency Tradeoffs in Modern Distributed Database System Design. 2012. IEEE Computer Society.

64 DB-Engines Ranking. Online. <https://db-engines.com/en/ranking> Read 30.4.2019.

65 Redis documentation. Online. <https://redis.io/topics/introduction> Read 30.4.2019.

66 Docker Resources. Online. <https://www.docker.com/resources/what-container> Read 2.5.2019.

67 Hykes, S. The future of Linux Containers. PyCon US 2013. Online. <https://pyvideo.org/pycon-us-2013/the-future-of-linux-containers.html> Published 15.3.2013.

68 Burns, B. Grant, B. Oppenheimer, D. Brewer, E. Wilkes, J. 2016. Borg, Omega, and Kubernetes Lessons learned from three container-management systems over a decade. ACMQueue.

69 Kubernetes documentation, Overview, What is Kubernetes. Online. <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/> Read 3.5.2019.

70 Docker Hub. Online. <https://hub.docker.com/> Read 2.5.2019.

71 Kubernetes home page. Online. <https://kubernetes.io/> Read 3.5.2019.

72 Kubernetes documentation, terminology glossary. Online. <https://kubernetes.io/docs/reference/glossary/?all=true> Read 3.5.2019.

73 Kubernetes documentation, Concepts. Online. <https://kubernetes.io/docs/concepts/> Read 3.5.2019.

74 Kubernetes documentation, horizontal pod autoscaler. Online. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/> Read 3.5.2019.

75 Pittet, S. Continuous integration vs. delivery vs. deployment. Article. Online. <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment> Read 3.5.2019.

76 Jenkins, open source automation server. Online. <https://jenkins.io/> Read 4.5.2019.

77 Travis CI, continuous integration service. Online. <https://travis-ci.org/> Read 4.5.2019.

78 Langley, A. 2018. Enabling Strong Authentication with WebAuthn. Google Developers. Article. Online. <https://developers.google.com/web/updates/2018/05/webauthn> Updated 1.5.2019. Read 4.5.2019.

79 Yubico FIDO U2F. Online. <https://www.yubico.com/solutions/fido-u2f/> Read 4.5.2019.

80 Web Authentication: An API for accessing Public Key Credentials Level 1. W3C Recommendation. 2019. Online. <https://www.w3.org/TR/webauthn/>

81 Content security policy. Mozilla developer documentation. Online. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP> Read 4.5.2019.

82 React documentation, JSX prevents injection attacks. Online. <https://reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks> Read 4.5.2019.

83 Angular documentation, Security. Online. <https://angular.io/guide/security> Read 4.5.2019.

84 NPM documentation, Audit. Online. <https://docs.npmjs.com/cli/audit> Read 4.5.2019.

85 Snyk vulnerability scanner. Online. <https://snyk.io/> Read 4.5.2019.

86 Password hashing competition. 2015. Online. <https://password-hashing.net/> Read 4.5.2019.

87 Provos, N. et al. 1999. A Future-Adaptable Password Scheme. Proceedings of 1999 USENIX Annual Technical Conference.

88 Akhawe, D. 2016. How Dropbox securely stores your passwords. Article. Online. <https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/>

89 Manico, J. 2019. AppSecCali 2019 Keynote. The Unabridged History of Application Security. YouTube. Online. <https://www.youtube.com/watch?v=L07IjIFbRCY&t=1127> Published 13.2.2019.

90 Secure-password repository. Github. Online. <https://github.com/emilbayes/secure-password> Read 4.5.2019.

91 Meteor JavaScript framework. Online. <https://www.meteor.com/> Read 10.4.2019.