DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte

Mattias Levlin, 38852
Master’s Thesis in Computer Science
Åbo Akademi University
Supervisors: Annamari Soini & Dragos Truscan

Abstract

One of the most used tools for creating interactive, advanced, and easily maintainable websites in 2020 is the programming language JavaScript. Over the last decade, many front-end frameworks have been built on top of JavaScript, which makes creation, design, and maintenance of interactive websites easier. As of 2020, the most popular front-end JavaScript frameworks were, by a large margin, React and Vue, followed by Angular. A relatively new framework called Svelte was also gaining in popularity and raised developer interest. This study will evaluate the performance of these front-end JavaScript frameworks. The criteria for evaluation are primarily based on speed of completing certain operations related to DOM elements in the web browser, such as DOM element addition, editing, and removal. Non-technical evaluation points include differences in architecture, development experience, popularity, maturity, and availability.

To set the context for the study of these frameworks, the study begins with an outline of the history and development of JavaScript. Its current status and versions are also described. Surrounding technologies relevant to the study are presented, such as DOM and HTML. Following this, the key features of front-end JavaScript frameworks are investigated, and the development setup process for a generic framework is documented. The criteria for selecting frameworks for evaluation is presented, and the four selected front-end frameworks are then investigated and evaluated. A benchmark JavaScript application is described and created for each of the frameworks. With this reference application, a number of technical benchmarks are performed, where the frameworks are rated according to how well they perform various DOM data updates. Finally, a recommendation is made on which frameworks are best suited for use, and how the future landscape of front-end JavaScript frameworks is likely to develop.

Contents

1 Introduction

This study aims to compare and evaluate some of the most popular JavaScript frameworks with a series of DOM performance benchmarks, to find out which one is best suited for web development, and what strengths and weaknesses each of the frameworks have. DOM is an abbreviation of Document Object Model, which is a web browser representation of current elements displayed on a certain web page. Furthermore, other factors are discussed for each framework, such as their respective history, usability, popularity, and maturity. This study may be of general interest to web developers, especially developers focused on front-end technologies, as it could potentially make the selection of an appropriate front-end JavaScript framework for a certain project easier. The study may also serve as a general introduction to the domain of JavaScript frameworks. Furthermore, focusing on the technical metrics may be of special interest to stakeholders concerned with efficiency and web browser speed. A practical situation where these metrics become relevant is one where a person or company is looking to start a new web project where a large number of visual web elements are handled. In this case, handling a web application with resource-heavy DOM manipulation, the technical metrics would be helpful for estimating scalable measurements. The best-performing framework(s) will be recommended for general web development usage.

An outline of this study is as follows: In chapter 2, JavaScript's surrounding technologies are outlined; in chapter 3, JavaScript, its history, and current status is overviewed; in chapter 4, the general idea of a JavaScript front-end framework is presented; in chapter 5, technical tools related to setting up a JavaScript project are discussed; chapter 6 contains a presentation of the JavaScript framework landscape, and a more thorough description of a smaller number of popular front-end JavaScript frameworks. In chapter 7, the described frameworks are evaluated in practice, according to a number of technical DOM benchmarks. Some non-technical evaluation points are also discussed. Finally, chapter 8 contains a discussion of results, usage recommendations, and a conclusion.

2 The web environment and JavaScript's surroundings

In 2020, the Internet was more ubiquitous than ever, exemplified by a statistical report assembled in late 2019 by the International Communication Union, which reported that more than half of the world population, 53.6%, had access to the Internet (4.1 billion people), either through mobile or broadband connections [1]. To navigate the Internet, people commonly use the World Wide Web information system. Navigation and access to web pages on the World Wide Web is usually done with the help of web browsers, available in both mobile and desktop versions. Within a web browser, information can be accessed in the form of web pages, which are most commonly built with HTML, CSS, and JavaScript. These are the three fundamental building blocks of web pages. HTML is used to create elements on web pages, such as menus, texts, and boxes; CSS is used to style, design, and place these elements on the web page; and JavaScript enables interaction with and manipulation of these elements. JavaScript has been described as a glue language, used for assembly of other components. Out of these three, HTML and CSS are less complex, and are mostly used for static design purposes. For developers who want to create dynamic web content, JavaScript is one of the most important building blocks, and for this reason, the scope of the language, including the tools, libraries, and frameworks found within the language, is much larger and more complex than that of HTML or CSS. While some static web pages use HTML and CSS only, JavaScript web page interactivity has become a de facto standard used on most web pages.

In this chapter, the history of the Internet, the World Wide Web, and the web browser is presented in a condensed form, followed by the development of the initial web content technologies, HTML and CSS.

2.1 The web browser

The most used web browsers in February 2020 were Google Chrome (64.5% global usage share), Safari (17.6% usage share), and Mozilla Firefox (4.6% usage share) [2]. To develop web pages and other content for these web browsers, JavaScript, and JavaScript-based frameworks and libraries, are central tools. The different versions of the web browser allow for quick and easy navigation through different pages, and have become one of the backbones of the Internet. The first web browser, which serves as the first ancestor to all the latter ones, was the World Wide Web browser, developed and released in December 1990. Tim Berners-Lee served as the lead developer for the project, while working at the European Nuclear Research institute, more commonly known as CERN, located in Switzerland. Berners-Lee simultaneously developed and released the Hypertext Transfer Protocol (HTTP), which serves as an application protocol for the World Wide Web, used for indexing and navigation of web sites. Several technological breakthroughs in the previous decades had enabled the creation of a technology such as the web browser: TCP/IP, the Domain Name System (DNS), and the Uniform Resource Locator (URL), which were all part of the early Internet [3].

The World Wide Web browser was an innovative project, though limited in scope at first. Another innovation that would enable wider proliferation and spread of the Internet, was the graphical web browser. The first graphical web browser, Mosaic was released in 1993. Mosaic was developed by the American state-owned company National Center for Supercomputing Applications and its lead developers were Marc Andreessen and Eric Bina. The browser was successful thanks to its user-friendly interface, easy installation process on the operating system Microsoft Windows (which was growing in usage share at the time), and support for multiple internet protocols. Mosaic would serve as a template for companies to follow. One of the more notable follow-ups was the Netscape Navigator browser, developed by the company Netscape Communications, and released in 1994. It took over the market after Mosaic and became the most used web browser a few months after its release. On these early web browsers, HTML content was standard, to which CSS was later added as a styling language. Besides JavaScript these are, even in 2020, the two fundamental web standards for creating web pages.

2.2 HTML and CSS

HTML was developed and released in 1993, some years before JavaScript, while CSS was released in 1996, one year after JavaScript. These three languages have come to form a technology stack referred to as “the triad of technologies that all Web developers must learn”, by David Flanagan [4]. HTML is an abbreviation of Hypertext Markup Language, defined by the World Wide Web Consortium as “the Web’s core language for creating content for everyone to use anywhere" [5]. Hypertext refers to text that contains references, or hyperlinks, to other text segments or text pages. Markup refers to text containing annotations, specifying data properties belonging to a certain text, beyond the visible text content itself. Furthermore, it is a descriptive markup language, which means that it is used for labeling the text, not giving instructions on how to process the text.

An HTML document is recognized by its initial declaration `<!DOCTYPE html>`. All HTML elements follow the same syntax (`<element>`). Basic HTML syntax, with common root elements such as `<html>, <head>`, and `<body>`, is described in Figure 1. This type of syntax contains nested elements, where the `<head>` and `<body>` elements are child elements to their parent element `<html>`.

```
<!DOCTYPE html>
        <html>
                <head>
                        <!-- Head (title) content here -->
                </head>
                <body>
                        <!-- Body (page) content here -->
                </body>
        </html>
</html>
```

Figure 1: Example of HTML syntax. Identical or similar syntax to this is often used when developing with the JavaScript frameworks evaluated in this study.

The latest major version of HTML is HTML5, initially released in January 2008, but fully released in October 2014, in a complete version recommended for usage by the World Wide Web Consortium (W3C). HTML5 seeks to improve multimedia capacities and easier manipulation of the Document Object Model, abbreviated as DOM, which is a key action in advanced web applications. HTML5 also introduced many new HTML elements, such as `<article>`, `<canvas>`, `<footer>`, and `<header>`. The `<canvas>` element can be used for drawing a large number of elements on the screen. In terms of multimedia elements, the `<audio>` and `<video>` elements replace the `<object>` element. This has been seen as an attempt to provide Adobe Flash-like functionality, and ultimately replace it, since Adobe Flash is a proprietary technology. These elements are ready-made components which will help developers construct advanced web applications more quickly. HTML5 also discontinued certain elements, notably the `font` element.

CSS, an abbreviation of Cascading Style Sheets is defined by the Word Wide Web Consortium as “a simple mechanism for adding style (e.g., fonts, colors, spacing) to Web documents” [6]. CSS is one of the most common tools used as an addition to HTML, and is found in many basic introductory courses to programming and computer science, forming a simple toolkit for designing basic user interfaces. The primary developers of CSS were Håkon Wium Lie and Bert Bos. The former was working with Tim Berners-Lee at CERN, and so had the opportunity to discuss what kind of styling technology was needed for web content. The impetus of the development of CSS came from the fact that there was no easy way to style documents on the Web in the early 1990s. While working on the first web browser, Tim Berners-Lee had not specified a syntax for styling HTML documents, though he had envisioned a separation of document structure and document layout. Early browsers introduced various browser-specific style languages, such as DSSSL and Pei Wei’s Viola browser language. CSS was designed to be simple, and to create a balance between the author and the user. Crucial to the success of CSS was browser support; the first commercial browser that supported CSS was Microsoft’s Internet Explorer 3, released in August 1996. This browser supported most of the standard CSS elements, such as color, background, font and text properties. Soon after, Netscape Navigator and Opera announced support for CSS, and most subsequent browsers have supported CSS, including the top-used browsers in 2020 [7].

The relationship of HTML to JavaScript today is that it can work either as a complement to JavaScript, as with jQuery, or, as often more recently, as integrated, HTML-like syntax that is compiled into HTML elements. An example of this is the JSX syntax, recommended for use when developing with React, which is neither purely JavaScript nor HTML, but a combination of them both, integrating the basic element creation and structural functionality of HTML with the dynamic capabilities of JavaScript. CSS as a technology is also commonly used in tandem with JavaScript. One option is to implement CSS properties in separate .css files, which is the more traditional format. Another option, similar to how HTML has been combined with JavaScript, is to integrate CSS into the JavaScript frameworks themselves, through specialized libraries such as CSS-in-JS and styled-components. Libraries such as these enable the developer to write JavaScript code that styles visual elements with CSS-like syntax. This styling code is then usually compiled into pure CSS in the browser [8].

3 JavaScript

This chapter presents a general overview of JavaScript; the background, history, and development of JavaScript are outlined, followed by a description of the current features of JavaScript. The status and version history of ECMAScript, which JavaScript is an implementation of, is outlined, and then the different flavors of JavaScript are presented. The current structure and features of JavaScript are important to understand, as all the frameworks discussed are built upon the core JavaScript language. References to different ECMAScript versions and different JavaScript flavors are common within JavaScript's developer community, and it is useful for any JavaScript developer to achieve an understanding of these different versions.

3.1 Creation and development of JavaScript

In 1995, NetScape Communications, the developers of the then-popular web browser NetScape Navigator, hired the programmer Brendan Elch to create a new dynamic scripting language for web pages and client-side manipulation of data. Having established themselves in the web browser market, Netscape saw the need for creating dynamic websites instead of using just HTML, which had been the standard up until then. Most early websites were designed in a computationally inefficient way using only HTML: for each user action or click, a request was sent to the server, and then a new HTML page was sent back to the client.

Marc Andreessen, founder of Netscape Communications, believed that there existed a fundamental need for a simple web scripting language, targeted for DOM manipulation. The scripting language was intended not only for experienced developers, but also for designers and people with less programming experience, something that would function as an add-on to HTML. The project was inspired by the functionality and syntax of Java, though fundamentally different. Brendan Elch and his team initially called the project Mocha, later switching to LiveScript, before finally settling on the name JavaScript, which has caused much confusion in terms of the similarity between Java and JavaScript (little similarity exists beyond a minor syntactic resemblance) [4]. The project was prototyped during 1995, and released officially in

March 1996. Already at launch, JavaScript enabled new functionality on web pages that HTML alone could not handle, such as responding to user input, changing colors of elements, and showing pop-up windows [9]. As JavaScript was developed by NetScape, it was not envisioned as being a future web standard, which it became. One reason for its later popularity was that NetScape was bought by America Online (AOL) and later turned over their browser's code to Mozilla, including JavaScript-based functionality, which contributed to its growth [10].

3.2 ECMAScript and standard JavaScript

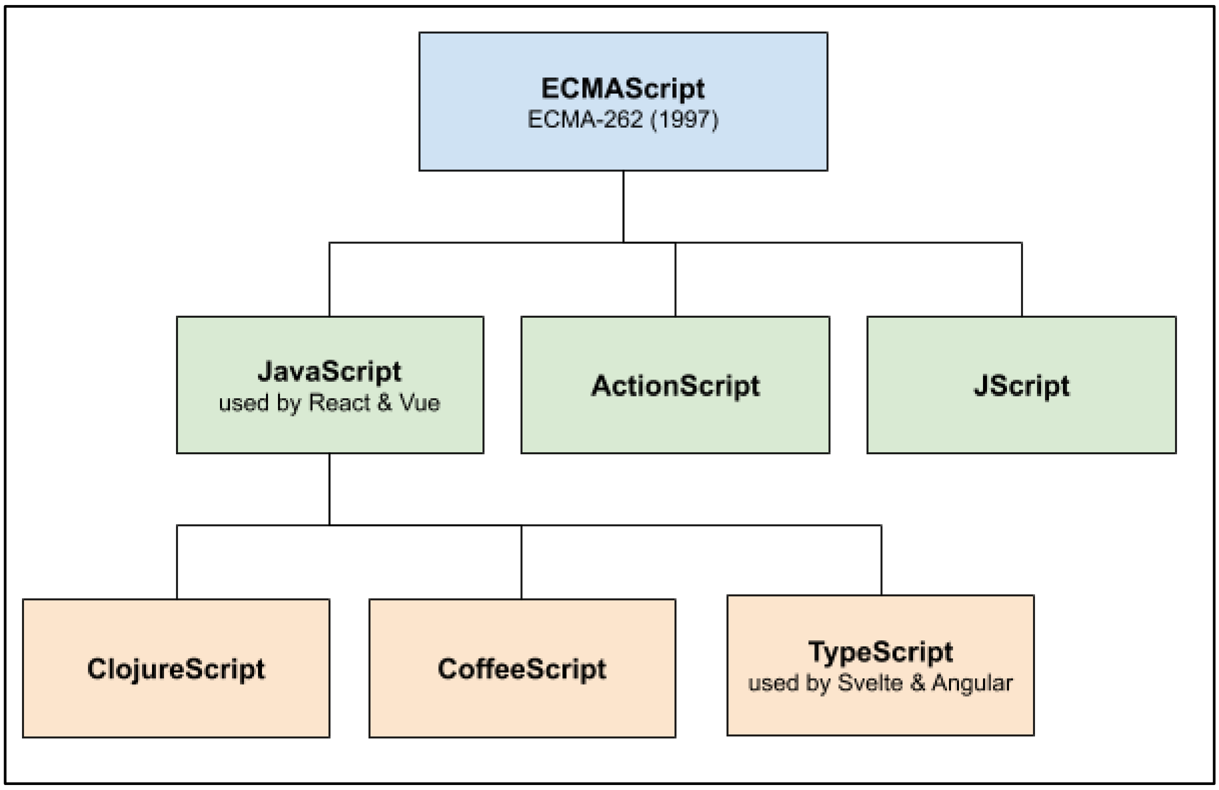

Figure 2: ECMAScript implementations or dialects in green (one of them being JavaScript), and JavaScript flavors depicted in orange.

After the initial release of JavaScript in 1997, the developers, headed by Brendan Elch, saw the need for a language standardization, so as to foster growth, prevent fragmentation of the JavaScript developer community, and make the language accessible across browsers. This was done through the ECMAScript language standard, defined in 1997 by the standards organization Ecma International. ECMAScript has the standard ID ECMA-262. ECMAScript is additionally defined as an ISO standard (ISO/IEC 16262, later revised and updated to ISO/IEC 22275 in 2018) [11].

Figure 2 describes the relationship between the ECMAScript standard, the different implementations or “dialects” of ECMAScript (JavaScript, ActionScript, and JScript), and the JavaScript flavors ClojureScript, CoffeeScript, and TypeScript (the last of which is used by Svelte and Angular, frameworks evaluated in this thesis). JavaScript is the most well-known implementation of ECMAScript, but there are several other implementations or “dialects” of ECMAScript besides JavaScript. One of these is JScript, developed by Microsoft in 1996 as their own in-house alternative to JavaScript, primarily used within Microsoft’s Internet Explorer browser. Another notable dialect is ActionScript, developed by Macromedia Inc., a company later bought by Adobe Systems. The ECMAScript standard has been under continuous development since its first standardization (version 1) in 1997. ECMAScript 6, also known by the name ECMAScript 2015, is the sixth edition of the ECMA-262 standard and is an often-used version. From ECMAScript 2015 version onwards, Ecma shifted to an annual release system, updating ECMAScript once a year, thus making each version thereafter known both by its version number and its version year. The most recent ECMAScript edition, the tenth, was defined in June 2019, as ECMAScript 2019 [12].

When compatibility with older browsers needs to be ensured, more modern JavaScript versions need to be converted into older versions. Conversion can also be done between JavaScript versions, or from one JavaScript flavor into another, such as from TypeScript to JavaScript. This process is called transcompiling. The most widely supported JavaScript version corresponds to ECMAScript 2015 and this is a commonly used target version for transcompiling purposes. The different JavaScript flavors, marked in orange in Figure 2, are sometimes called transcompiled languages.

```
function additionExample(left, right) {

        return left + right

}
```

Figure 3: Example of JavaScript code. The input variables left and right are weakly typed, meaning that their data types do not need to be specified.

JavaScript is an object-oriented, high-level scripting language containing both dynamic and weak typing. Dynamic typing means that definition of data types is not strict; a value assigned to a string may be reassigned to a number. Weak typing means that data types are implicitly defined, not explicitly, and automatic data type conversions may happen depending on the operation, sometimes in an unpredictable fashion. This can be seen in Figure 3, where the variables `left` and `right` do not have a specified variable type. This means that an unpredictable addition of a number and a string, such as `3 + "3"` could be performed with this example function, which would return `"33"` as a result.

JavaScript implements standard programming syntax introduced by the language C, such as `if`, `while`, and `switch`, and makes extensive use of functions. Furthermore, just like C, JavaScript uses curly bracket syntax to define statement blocks (`{ … }`). The ECMAScript 2015 standard of JavaScript introduced numerous changes: `let` and `const` were introduced to enable block scoping, whereas previously, function scoping was the only scoping variant available. The arrow function, a type of anonymous function, was also introduced with ECMAScript 2015, which enabled a shorter function definition. Semicolons are allowed for termination of statements, but can be omitted.

3.3 JavaScript flavors and TypeScript

JavaScript contains several flavors that differ in syntax, but are still considered variants of JavaScript. React and Vue, two of the frameworks evaluated in this study, are based on standard JavaScript, while the other two, Svelte and Angular, are based on TypeScript. TypeScript was developed by Microsoft as a superset of JavaScript, and aimed to create a language that would be more suitable for large-scale applications. This is exemplified also by the slogan of TypeScript: "JavaScript that scales" [13]. TypeScript is an extension of ECMAScript, more specifically ECMAScript 6. While TypeScript is fundamentally different from standard JavaScript, it is still popular; in the 2019 State of JavaScript survey, TypeScript was the JavaScript flavor that, except for the standard version, had the highest awareness (100%), satisfaction (89%), and interest rating (66%) [14]. The satisfaction with TypeScript has also increased over time, in the same survey, the "satisfied users" category for TypeScript has risen from 20.8% in 2016 to 58.5% in 2019. Worth noting is TypeScript's original release date, 2012.

```
function additionExample(left: number, right: number): number {

        return left + right;

}
```

Figure 4: TypeScript equivalent to the JavaScript code seen in Figure 2. This is an example of static typing, where the data types of the variables `left` and `right` are specified as numbers.

While TypeScript is based upon JavaScript, there are several key differences. An example of TypeScript code syntax is seen in Figure 4. In this figure, the variable types of the variables `left` and `right` are specified as `number` in the function definition. This convention, known as static typing, is an additional verification step which makes TypeScript differ from JavaScript. Static typing enables variable type checking, which makes the development process more secure and easier to debug while coding: the required input and return variable types for functions can be specified. TypeScript is thus stricter but also potentially more secure than standard JavaScript, where the variable types do not have to be defined and can be passed to a function regardless of variable type. TypeScript files are denoted with the suffix .ts, instead of .js, for JavaScript files. Furthermore, TypeScript is designed to be an object-oriented language, while JavaScript is a scripting language. One drawback with TypeScript is that it cannot be run directly, it has to be transcompiled to JavaScript, and this usually means some extra load times at some part of the development process.

In the 2019 State of JavaScript survey, the numbers of satisfied users (meaning users that have both used the JavaScript flavor in question and would use it again) for each of the most used JavaScript flavors other than standard JavaScript were as follows: TypeScript at 58.5%, Reason at 3.5%, Elm at 4.7%, ClojureScript at 2.0%, and PureScript at 1.6% [14]. As can be seen, TypeScript is by far the most used and well-liked JavaScript flavor.

3.4 XML, AJAX, and the Single Page Application

XML is a markup language similar to HTML, but is more often used for data representation instead of content display. Both XML and HTML are today widely used on the Internet, and both also derive from the earlier markup language SGML (Standard Generalized Markup Language), which was in use as a dynamic information language in the 1980s. XML was developed by the World Wide Web Consortium and initially released in 1996, while the latest standard edition, the fifth, was defined in 2008. The developers of XML intended the language to be usable over the Internet, easy to write and read, encoding documents in a format readable both to humans and computers [15]. As can be seen in Figure 5, XML is similar to HTML, in that they both use opening and closing tags to define elements, and content or text can be defined between the tags. Another similarity to HTML is the nesting of properties, as `city` and `country` are sub-properties of `location` in Figure 5. This example code represents a person with the properties `firstName`, `lastName`, and `location`, and the properties `city` and `country` are nested properties of within the parent property `location`.

```
<person>
        <firstName>Mattias</firstName>
        <lastName>Levlin</lastName>
        <location>
                <city>Espoo></city>
                <country>Finland></country>
        </location>
</person>
```

Figure 5: Example of XML syntax, the similarity to HTML can be seen.

XML was to become relevant within the JavaScript sphere with the invention of the technology collection AJAX, an abbreviation of Asynchronous XML and JavaScript. AJAX provides the developer with a way to update parts of an HTML page without downloading its entire content. AJAX is not one single technology or tool, but rather a collection of several technologies, bundled together as a whole; Garrett defines these technologies as XHTML, CSS, DOM, XML, XLST (eXtensible Stylesheet Language Transformations), `XMLHttpRequest`, and JavaScript [16]. Within AJAX, JavaScript is the tool binding all the other ones together. In the early 2000s, a prominent source of frustration for web users were the slow server responses and long data transmission times, often exacerbated by the low-speed Internet connections during the era. A marked improvement came with the development of the AJAX technology stack.

The first step towards AJAX and the appearance of asynchronous elements on the web happened with the introduction of the `iframe` (inline frame) HTML tag, introduced in 1996 in Microsoft's Internet Explorer. Another important technology, which is a central part of the

AJAX technology stack, is the `XMLHttpRequest`, developed in 1998. The `XMLHttpRequest` is a scripting object, used for sending XML data to and from the server, instead of HTTP data. The AJAX system was first prototyped in 1999. Using web applications before AJAX usually involved long periods of waiting on the user end: each time a user request was sent through a click in the interface, the user inevitably had to wait for the synchronous server response and the data transmission. Within the AJAX system, every user action that would normally require a server request, is directed instead to the AJAX engine, which is located on the client side. Certain simple actions can be handled on the client side exclusively, and for things that require server communication, asynchronous XML data is used, instead of synchronous HTTP data. Thus, the web application operates seamlessly from the user's point of view, and eliminates waiting times. Displayed in Figure 6, the key innovations of the AJAX system are the JavaScript call and the AJAX engine, both located on the client side, and the communication through XML data, replacing HTML/CSS data from the server [16].

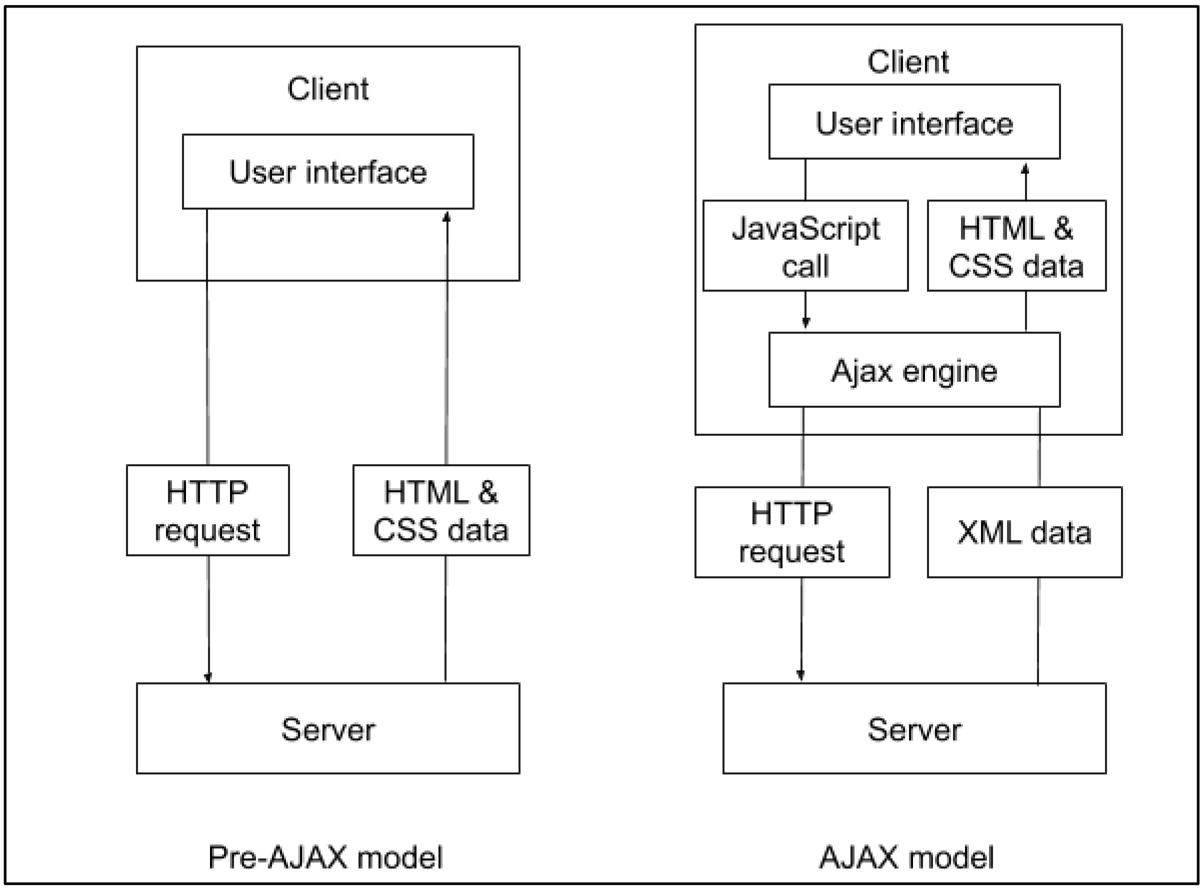

Figure 6: Diagram showing the difference between pre-AJAX and AJAX web applications.

All of the JavaScript frameworks presented in this thesis have adopted the single page application principles, a design philosophy that has become a well-known standard among web application programmers. A Single Page Application (SPA) is an application composed of individual components, loaded into memory upon first page visit, that can then be replaced or updated independently, so that the entire webpage does not have to be reloaded on every user action. Another advantage with single page applications is that components can be reused, and thus the amount of code needed can be drastically reduced. The single page application was implemented and patented for the first time in 2002, with the patent specifically mentioning JavaScript as an example of a target language for the implementation [17]. Single-page applications can be contrasted with the alternative multi-page application, which can have some marginal advantages, including easier search-engine optimization, as each page on a multi-page application is treated as an individual page by search engines. The success of single page applications, which would be popularized later on, was largely thanks to the preceding AJAX technology and its innovations in server communication.

3.5 Current status of JavaScript

Throughout the 2010s, JavaScript has grown to become one of the most used programming languages for web development purposes. According to a survey done by the code hosting site GitHub.com, JavaScript was the most used language on GitHub in 2019 [18]. In addition, it was the language that had the most GitHub code commits in the first quarter of 2020 [19]. While some other web development languages have declined in usage or stalled in growth, such as PHP or Ruby, JavaScript's growth in usage has been steady during the last decade [20]. A large contribution to this growth came with the 2009 release and subsequent popularization of Node.js, a server-side implementation of JavaScript, which extended the JavaScript domain to the back-end, helping JavaScript become a full-stack language, sometimes described as the "JavaScript everywhere" paradigm.

On the site Stack Overflow, JavaScript has been the most commonly used programming language since 2013, used by 69.8% of all the site users, followed by the related languages Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS) at 68.5% and 65.1% respectively, in 2018 [21] [22]. JavaScript is supported by all mainstream browsers, including

Google Chrome, Mozilla Firefox, Safari, Opera, and Microsoft Edge. This wide support has helped wide adoption: according to a survey done by Web Technology Survey, JavaScript was used on 95% of all websites, and is by far the most popular client-side scripting language [23]. In David Flanagan's book *JavaScript: The Definitive Guide*, Flanagan summarizes the language as being "a lightweight, interpreted programming language with object-oriented capabilities". In terms of programming syntax and object inheritance, JavaScript has a vague resemblance to some other major programming languages, such as C, C++, Java, and Perl. JavaScript is mostly used client-side, and not server-side, to the point that the term "JavaScript" has come to refer almost exclusively to the client-side implementation [4]. Notable additions to the core JavaScript technology stack in recent years have been Node.js, JSON, jQuery, and ES6 Generation [24].

3.6 Comparison with PHP

A language worth mentioning and briefly comparing to JavaScript is PHP, which has been used for many of the same purposes as JavaScript, with regard to web development. PHP is a general-purpose programming language, though originally designed, and commonly used, as a web programming language. It was developed by Rasmus Lerdorf, who intended it to be a personal project for his own website, naming it Personal Home Page (PHP) and published the first version of it in 1995 [25]. The syntax and functionality of PHP is partially inspired by C and Java. As its usage grew, the language's full name was later changed to Hypertext Preprocessor.

Earlier in the history of JavaScript, the language was used together with PHP, with JavaScript taking care of browser details and front-end functionality, and PHP used for server-side scripting, as JavaScript was unable to handle it. During the 2010s, this relationship has changed: the most notable change was the development of server-side JavaScript in the form of Node.js, something that essentially rendered PHP obsolete in the JavaScript development stack [26]. This has turned JavaScript and PHP into competitor languages more than associated languages. Furthermore, PHP web projects often use the database MySQL as back-end technology, while JavaScript rarely makes use of MySQL. One notable advantage that JavaScript has over PHP, is that JavaScript can easily handle the data formats XML and asynchronous functionality, while these tasks are significantly more difficult to handle with PHP [27]. Despite the success of JavaScript and the development of Node.js, PHP has remained one of the most used web programming languages, though its popularity has declined somewhat in recent years.

According to the TIOBE index, a well-known measure of programming language popularity, PHP was the 8th most popular programming language in June 2019 [28]. The TIOBE index is calculated from the number of search engine queries containing the programming language. In June 2019, PHP had a TIOBE interest share of 2.57%, while JavaScript was slightly more popular than PHP, ranked 7th and having an interest share of 2.72%. Worth noting is also that the popularity of JavaScript increased 0.22% from June 2018 to June 2019, while the interest in PHP decreased by 0.31%. Another similar index called PYPL, developed as an alternative to TIOBE and based on language tutorial interest, ranked JavaScript as the third most popular language in June 2019 with 8.29% interest share and PHP as the fifth most popular with 6.96% interest share [29].

3.7 The Document Object Model

In this study, the Document Object Model (DOM) is important, since it is extensively used in the technical benchmarks section, where the performance of each JavaScript framework is measured using different DOM operations. An example of control flow in the DOM model is seen in Figure 7. This figure outlines how communication is done between JavaScript, the DOM, and the HTML web page.

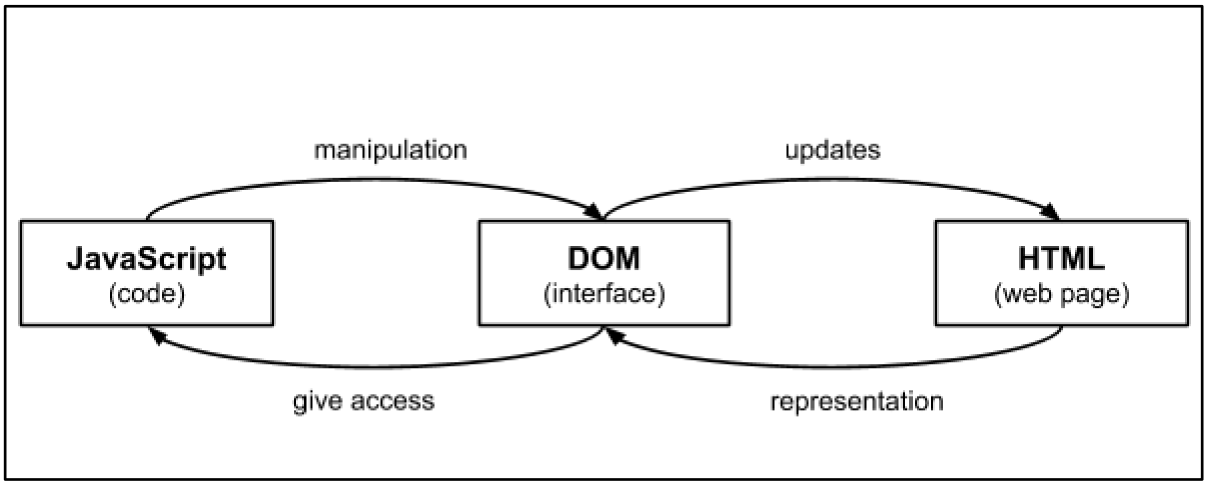

Figure 7: Interactions between JavaScript, the DOM interface, and the HTML document.

The DOM is a programming interface that allows for dynamic access to the content, structure, and style of HTML web page documents. Beyond HTML documents, the DOM can also handle XML and XHTML documents. However, in combination with JavaScript, the most common type of document edited is HTML (in the technical benchmarks section of this study, the web page documents are exclusively in the HTML format). Most browsers natively implement the DOM, as defined by the W3DOM standard, which means that the DOM representation of a web page will automatically come into existence and be available for editing once an HTML document is parsed by the web browser. The DOM can then be manipulated using JavaScript or any other language, and does not need a separate setup or installation [30]. In practice, whenever a certain web page is accessed through a web browser, an HTML Document Object Model of the page is created. In Google Chrome, for instance, the DOM representation of a web page can be accessed by right clicking and selecting “Inspect”, which opens the Google Chrome web console. The DOM representation of the current web page is then found by navigating to the “Elements” tab. The DOM can be imagined as an intermediate data representation of a web page that converts the webpage into an interactive model.

An entire HTML web page loaded into the web browser is represented in the DOM as a `document` object. This `document` object is the entry point for JavaScript and other scripting languages, which gives access to dynamic and programmatic manipulation of the elements found in the HTML (or XML) web page. Most interactive DOM functions (available to JavaScript) start with the `document` syntax, such as `document.createElement()` or `document.getElementsByName()`. In addition to the root access point, the `document` object, several other DOM data type objects are available; `Node`, `NodeList`, `Element`, `Attribute`, and `NamedNodeMap` [31]. By performing JavaScript manipulations on these DOM objects, the corresponding HTML elements on the web page can be updated. The `node` is a general, abstract, commonly seen object type, since all the `elements` and `attributes` on a webpage are represented by a `node`. `Element` is usually the most important object type from a JavaScript perspective, since it represents all the visual elements on the page. Retrieving an `element` using JavaScript gives access to interactive operations (for instance through the function `document.getElementById(id)`), `NodeList` lists an array of `elements`, while the `attribute` node can be used to access the attributes of `element` nodes [32].

Architecturally, the DOM treats an HTML document as a collection of objects organized into a tree structure, where each `Node` in the DOM tree is an HTML object, such as a `<body>`, `<text>`, or `<html>` element. For instance, beyond the first `Document` object, which is the JavaScript entry point, the following root `Node` element is usually an `<html>` element, with the `<head>` and `<body>` elements being subtrees of that root element, each consisting of further subtrees. Within this tree structure JavaScript can interact with the DOM in a number of ways: it can change and manipulate the HTML elements and attributes, change the CSS styles, remove and add HTML elements and attributes, and create and react to HTML events [33]. The DOM tree itself can also be traversed along available paths, navigating from parent nodes to child nodes (for instance using the `ParentNode` functionality). For an example of a DOM tree structure, see Figure 8.

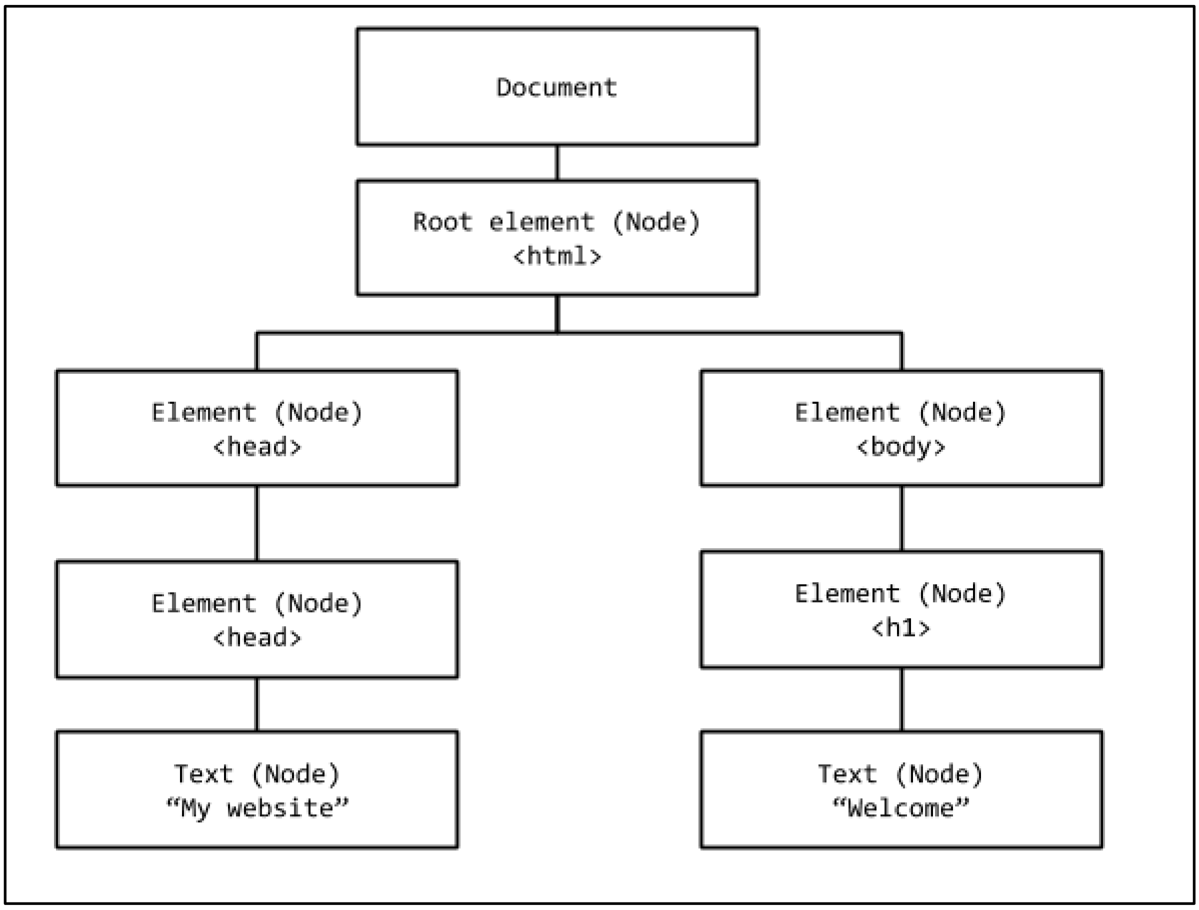

Figure 8: Illustration of a simple HTML DOM tree, showing the document entry point and nodes, which can be manipulated and edited using JavaScript.

The DOM is an important element in the evaluation of front-end JavaScript frameworks, since different frameworks have different approaches to how they deal with the DOM; this is one of the fundamental differences between them, which is described in more detail later in this study. Certain utility JavaScript libraries, such as jQuery, are mainly used for manual DOM interaction and manipulation, in order to create dynamic web functionality. Within the context of the DOM, another concept called “shadow DOM” is sometimes encountered: this refers to the hiding or encapsulation of CSS styles and child elements into a single parent element. DOM was standardized by the World Wide Web Consortium (W3) in 2004, and the latest version, DOM version 4, was released in November 2015.

3.7.1 Virtual DOM

One example of an alternative JavaScript framework interaction with the DOM is through the creation of a virtual DOM, which matches and updates the content on the actual DOM with that of a virtual one, and offers the virtual DOM as an intermediate medium for the developer to interact with, instead of forcing the developer to manually handle the DOM. This is what React and Vue do, and the popularity of these frameworks has also popularized the usage of a virtual DOM. The motivation for implementing a virtual DOM is to improve the efficiency and speed of DOM updates, since manual DOM updates can be costly. How well this works in practice is studied in more detail in the benchmarks section of this study, where the frameworks implementing a virtual DOM, React and Vue, are compared to the frameworks without a virtual DOM, Angular and Svelte. Furthermore, manual DOM updates using jQuery are also included in the benchmark study. Within the context of the virtual DOM, the process of synchronizing a virtual DOM with the actual DOM is known as a reconciliation.

Angular does not use a virtual DOM. The main reason for this is because it is designed to be a rewrite of AngularJS, which was released in 2010 and at that point did not make use of a virtual DOM. The usage of the virtual DOM spread afterwards, with React and Vue. The fact that Svelte is a relatively new framework that nevertheless does not use a virtual DOM represents a break with the paradigm popularized by React and Vue. If Svelte becomes more widely used, this may signal a shift in DOM handling for JavaScript frameworks.

3.7.2 DIV, SVG, and Canvas elements

Benchmarking of data in a web application presents a question regarding what elements to use for testing. HTML elements accessible through the DOM can be of different types. One common element for drawing simple visual arrangements is the `<div>` element, but for larger visualizations and complex graphs, the `<div>` option can become resource consuming [34]. Thus, if a web developer wishes to create interactive graphs and charts from scratch, specialized elements can be useful; two of the most commonly used charting elements are `<svg>` and `<canvas>`. A third, quite complex but resource-efficient option is to use a separate technology called WebGL, though this technology is beyond the scope of this thesis [35].

SVG and the `<svg>` element stand for Scalable Vector Graphics, and are based on XML. This makes `<svg>` elements different from other standard HTML elements, which are typically handled in JavaScript applications. SVG was initially released in 2001, envisioned as a tool for handling interactive, animated 2D graphics. Using SVG has long been considered as a good alternative for visualization purposes, having a comparatively long history, good performance, being flexible, and widely available [35]. SVG provides support for event handlers, since the graphical element structure created using SVG is preserved as a DOM tree. This is also a potential weakness with SVG element visualizations, since the DOM structure may become very complex and thus slow down the entire application [36]. SVG is resolution independent, so a zoom-in operation will not affect the quality of the visualization.

Canvas elements are one of the main alternatives to SVG elements. Canvas elements are a relatively new technology, being introduced in HTML5. Unlike the XML-based SVG elements, Canvas is based on HTML. However, Canvas differs from most HTML and JavaScript elements in that it does not make use of the DOM. This has certain advantages; Canvas has been cited as faster than SVG in some cases, especially when displaying a larger number of visual elements (on the scale of thousands of elements). Canvas objects are drawn immediately on the screen, using the "Immediate Mode", without the usage of any intermediate DOM route or other saving mechanisms [36]. This is also a potential drawback, since there is no way to interact with elements without redrawing the canvas. Furthermore, since there is no DOM information, event handlers are not available and CSS edits are more difficult. The initial setup of a simple graph is also a more advanced process than using SVG elements. However, for dynamic, quickly-changing content with many variables, such as web browser games, canvas is most likely the better choice [34]. Both SVG and and Canvas elements are library agnostic, and can thus be handled with any JavaScript framework, or web technology in general, though they may require more manual alterations than simply adding a standard HTML element.

There are many existing JavaScript libraries developed specifically for handling visual elements in the browser with internally implemented DOM manipulations. Examples include the SVG-based alternatives d3js and Highcharts. Canvas visualization libraries also exist, such as Chart.js, which avoid the DOM altogether. In this thesis, however, all DOM manipulation is done manually. For the scope of this thesis, and for studying elements, the HTML element `<div>` and related text elements have been chosen; the practical part of this study concerns the DOM behavior and speed when handling these HTML elements, using the different JavaScript frameworks. While Canvas is a viable technology especially for larger visualization datasets, the fact that it does not work with the DOM makes it superfluous to this study. For instance, implementing a benchmark using Canvas would cause difficulties in trying to track or analyze how the frameworks handle the DOM interactions and events.

SVG elements are also excluded from this study, since they are based on XML and not HTML, thus causing difficulties with most frameworks, representing a less common use case, and being hard to make use of in technical benchmarks in a consistent way. The `<svg>` element can be handled using JavaScript frameworks, but requires special considerations: for instance, the core library of React does not easily handle the `<svg>` element as it does `<div>`; there are instead specialized libraries such as React-svg and Svgr created for handling and conversion of `<svg>` elements, but these are outside the scope of this study. `<svg>` elements are also sometimes handled as picture elements. For these reasons, standard HTML elements such as `<div>` and `<p>` have been chosen as benchmark elements.

3.8 jQuery

jQuery is a core JavaScript library which is mainly used for DOM traversal and manipulation. It is described by the developers themselves as a fast, small, and feature-rich JavaScript library [37]. Other notable functionality supported by jQuery is event handling, animations, JSON parsing, as well as AJAX and other asynchronous operations. jQuery's DOM manipulation is handled by a selector engine, called "Sizzle". With the help of this engine, jQuery allows for easy access to all elements available on the DOM. jQuery is supported by all major desktop and mobile browsers in their current versions.

```
$("#button_1").click(function() {
        $("p").show("slow");
});
```

Figure 9: Basic DOM interaction, performed with jQuery.

jQuery uses the dollar sign ($) as a shorthand for "jQuery", which is one of the central commands in the library; within JavaScript code, jQuery functionality can often be identified by the dollar sign. The code in Figure 9 selects a button with the ID *button_1*, using the ID selector (#), found in the DOM and attaches a click function to it. If the button then is clicked by the user, a new paragraph element (p) will be shown on the screen.

jQuery's original developer was John Resig. The impetus for the development of the library came from the fact that cross-browser development with JavaScript in 2006 was perceived as difficult, and there were few libraries handling JavaScript DOM interaction. Resig stated that jQuery was aimed to improve the interaction between JavaScript and HTML, mainly by manipulating DOM elements. Resig initially released jQuery in January 2006, and the library was licensed and standardized under an MIT license later the same year [38]. According to data from the site BuiltWith.com, jQuery was used on 79.2% of the top 1 million websites in the world [39], thus being the most used JavaScript library by an overwhelming margin. For comparison, the second most used library was Bootstrap.js, with 16.9% usage share. The same situation is seen in w3Techs.com data from May 2019: here, jQuery was used on 97.3% of all JavaScript-based websites indexed in the survey. By contrast, the second most used JavaScript library was once again Bootstrap.js, which was used by 24.7% of all indexed websites [40].

jQuery is important for historical reasons, and is also helpful for understanding the behavior of the DOM. It has also been used directly or as inspiration in several frameworks; the predecessor library of Angular 2+, AngularJS, makes use of a built-in version of jQuery (in AngularJS this function is known as `angular.element`) [41]. However, other JavaScript frameworks, such as React and Vue, replace the functionality of jQuery altogether with their own respective virtual

DOM implementations. These DOM implementations are administered automatically by the frameworks themselves, and functionality exists for always making sure the DOM is updated according to the state of the code. This removes the need for manual tracking of events and user-directed updates, effectively presenting a viable alternative to jQuery altogether. Still, jQuery can be useful for one-off and smaller dynamic operations, when a developer does not wish to make use of a whole library or framework.

4 Front-end JavaScript frameworks

After reviewing JavaScript as a whole, in this chapter, the overall function and architecture of a general front-end JavaScript framework is evaluated. React, Vue, Angular, and Svelte conform to the characteristics defined in this chapter, either through their core functionalities, or through commonly used extension libraries. Some of the most common features of front-end frameworks are a synchronization of state and view, routing, a template system, and reusable components [42].

4.1 Terminology: Frameworks versus libraries

The terminology contrasting frameworks with libraries can be fuzzy: React is sometimes referred to as a library, and at other times as a framework, especially in online discussions and articles. For the sake of clarity, it would be useful to separate frameworks from libraries, and to use these terms in clearly separate, consistent ways.

A *library* is a passive collection of non-volatile resources where the developer is given control over how to use the resources. JavaScript libraries conform to this standard definition. An example of a JavaScript library is jQuery, used for functionality such as DOM manipulation, event handling, and AJAX functionality. Libraries that are very simple and only perform one certain task can be classified as tools, such as JSLint, used only for syntax checking, or Mocha, used only for testing. *Frameworks* are designed to be less passive and force the developer to do things in a certain way by providing a skeleton for development purposes and enforcing a control flow. A web application framework, for instance, provides the developer with a set way to develop and set up a whole web application, while a web application library does not contain any such overall philosophy, instead providing simpler sub-domain operations, such as network requests or styling operations that the developer has more control over [43]. In terms of architecture, one of the key defining characteristics of a framework is an element called the inversion of control. Within the domain of libraries, methods and functions are generally called explicitly by the programmer. In contrast, within the domain of frameworks, methods and functions are called by the code itself, such as a windowing system [44].

While Vue and Angular are considered frameworks, there seems to be some disagreement regarding whether React is a framework or a library. The developers of React refer to React as a library, it is designed as one and can be used as such, but it is more commonly used as a framework. For all intents and purposes, whenever React is used as a library, it can be referred to as a library, but when it is used as a framework, as in this study, it can be called a framework. A typical framework context is building a web application from scratch, and so within the context of this thesis, React will be referred to as a framework.

4.2 Templates and reusable component files

Within the context of single page applications, *templates* are HTML-like files, which contain additional syntax and elements, often representing dynamic JavaScript-like variables which can be changed through user interaction. The common denominator regarding the frameworks evaluated in this study is that they all use HTML as a base template, with their own element style defined on top. Templates describe the appearance of the document object model, and help the developer visualize the user interface while coding. The advantages of using templates is that they are natural to write and read for developers used to HTML and require little extra learning in order to be able to be use the additional dynamic functionalities. Templates are usually the most common component in a typical web application, compared to other non-presentational files, such as data state management files, configurational files, and logical files.

There are several languages developed specifically for templating purposes, interface design and component creation. For instance, React recommends using the self-developed, XML-like language extension of JavaScript called JSX, an abbreviation of JavaScript XML, though standard JavaScript syntax can also be used with React. JSX integrates HTML with JavaScript: for instance, JavaScript expressions can be mixed with HTML elements, as long as the expressions are defined inside curly brackets. Though JSX takes inspiration from HTML, XML, and JavaScript, it is most reminiscent of JavaScript, and uses the `camelCase` naming convention, instead of HTML attribute naming conventions [45]. In Figure 10, an example of a template written using JSX is seen. Here, an `openPositions` array is defined programmatically, and that array is then combined dynamically with HTML elements to create the `openPositionsList`, which can be reused as a dynamic template.

```
const openPositions = [‘Software developer’, ‘Graphic designer’, ‘Project manager’]

const openPositionsList = (
    <div>
        <p>Our current available positions are:</p>
        <p>
            {openPositionsList.map(position => <p>{position}</p>)}
        </p>
    </div>
);
```

Figure 10: An example of a React template file written using JSX.

Reusable components are another important feature within front-end JavaScript frameworks. The behavior, appearance, and characteristics of a certain component are typically defined and constructed in one file, and then imported and used in another. This makes the development of larger web applications easier, since each component can be imagined as a module, or as a building block, that can then be added or removed without causing problems for the surrounding functionalities. Reusable components, when containing HTML or HTML-like elements, are usually written as templates, combining HTML elements with interactive JavaScript-based syntax. Within the context of reusable components, it is common to see component file state contained in one component only. Component files can also be created without state, then being known as stateless functional components. These components are usually simple components that function as sub-modules to one or several complex components. The process of importing a component file into another component file is seen in Figure 11. The component `UserInfoBox` is defined in its own component file, but is imported and reused in two different component files, `EditUser` and `AddUserToProject`.

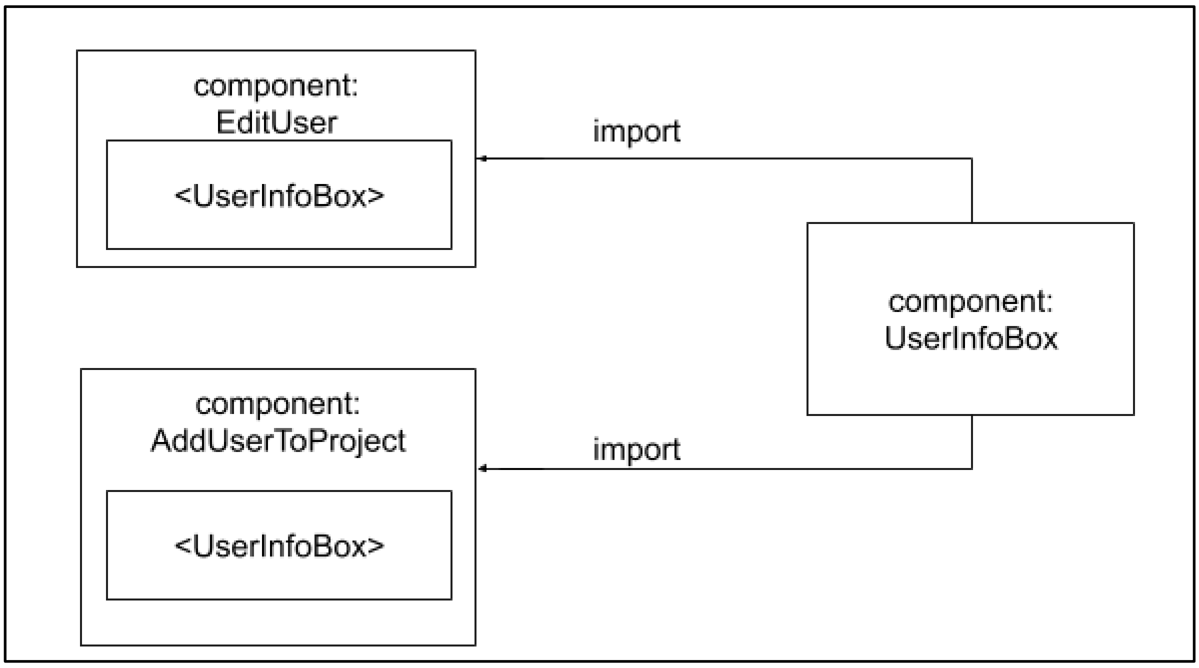

Figure 11: Example of a reusable component. The component UserInfoBox is imported and used in two other components, CreateUser and AddUserToProject.

A general, framework-agnostic standard for creation of components was defined in 2011, known as the Web Components standard. This standard includes an API to define new HTML elements, DOM handling including the usage of the shadow DOM, and HTML templates. The Vue framework has based its component syntax and creation on the Web Components syntax, and most JavaScript frameworks in this study treat components similarly to how they are defined in the Web components standard [46].

4.3 The model-view-controller pattern

The model-view-controller pattern, displayed is Figure 12, is an abstract software design pattern which is relevant within the context of most web applications, including this study on JavaScript frameworks and web applications built with them.

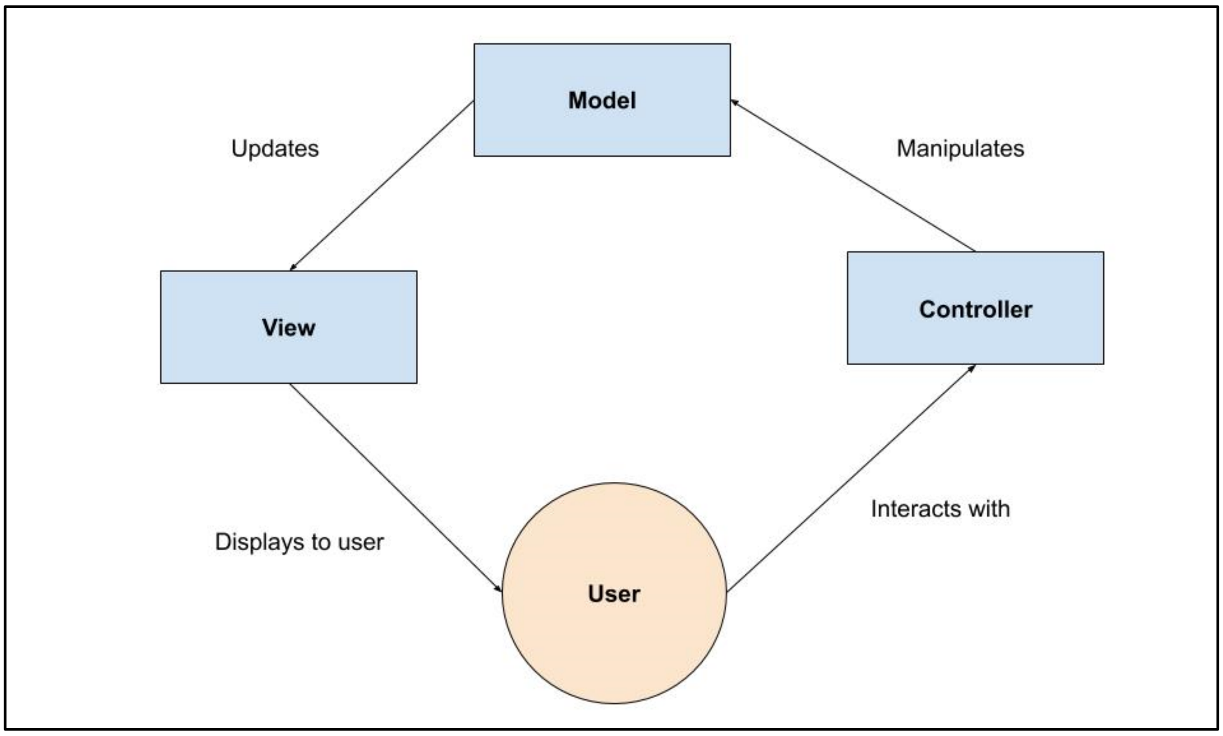

Figure 12: A simple example of a model-view-controller pattern

This pattern is a model of how internal interactions are done within most web applications and how the user interacts with the application. The pattern consists of three elements: the *model* represents the dynamic data structure and application logic, the *view* is the user interface that is displayed to the user, and the *controller* is what the user makes use of to interact with the model. The user interacts with the controller, which manipulates the model, which in turn updates the view, which is displayed to the user. In this model-view-controller context, most of the relevant functionality of the frameworks studied in this thesis is located in the view portion of this pattern. Dynamic JavaScript functionality that lets the user interact with some type of model can be imagined as the *controller*, defining interactions with the model. The model-view-controller pattern was developed for desktop contexts, but has since been used for web applications as well [47].

4.4 The global state, the data store, and props in SPA

The implementation of global application state was introduced with the arrival of single page applications; instead of having the user navigate through different web pages, the user navigates through different application states. Within single page applications, the global application state is usually handled by something called *data store*, which can be imagined as the model part in the model-view-controller pattern. In a single page application built with a JavaScript framework, the user sees the framework user interface as the *view*, and when the user clicks a function, the *controller* (typically JavaScript or JavaScript-like code) sends a request or data to the data store (the model part in the MVC-pattern). The data store presents a centralized way of storing, updating, and accessing application-wide data. Using a data store eliminates the need of having to always pass data between components; with a data store, data can be accessed globally by all components. When implementing a data store, edited data is passed one way to the store [48]. The global state is always recorded, can change depending on user action, and determines what components to show and to hide, and what actions to allow and disallow. The global state can also be accessed by all components (usually implemented as component files), in this way the global state is also a shared state for the entire application. Figure 13 describes how the state is shared across component files within the application. There is one global state, shaded in orange in the figure, which can be accessed by all the components in the application (shaded in blue and green). Component files can, in addition to implementing the global state, implement their own local state. The local state contains variables specific to that file only.

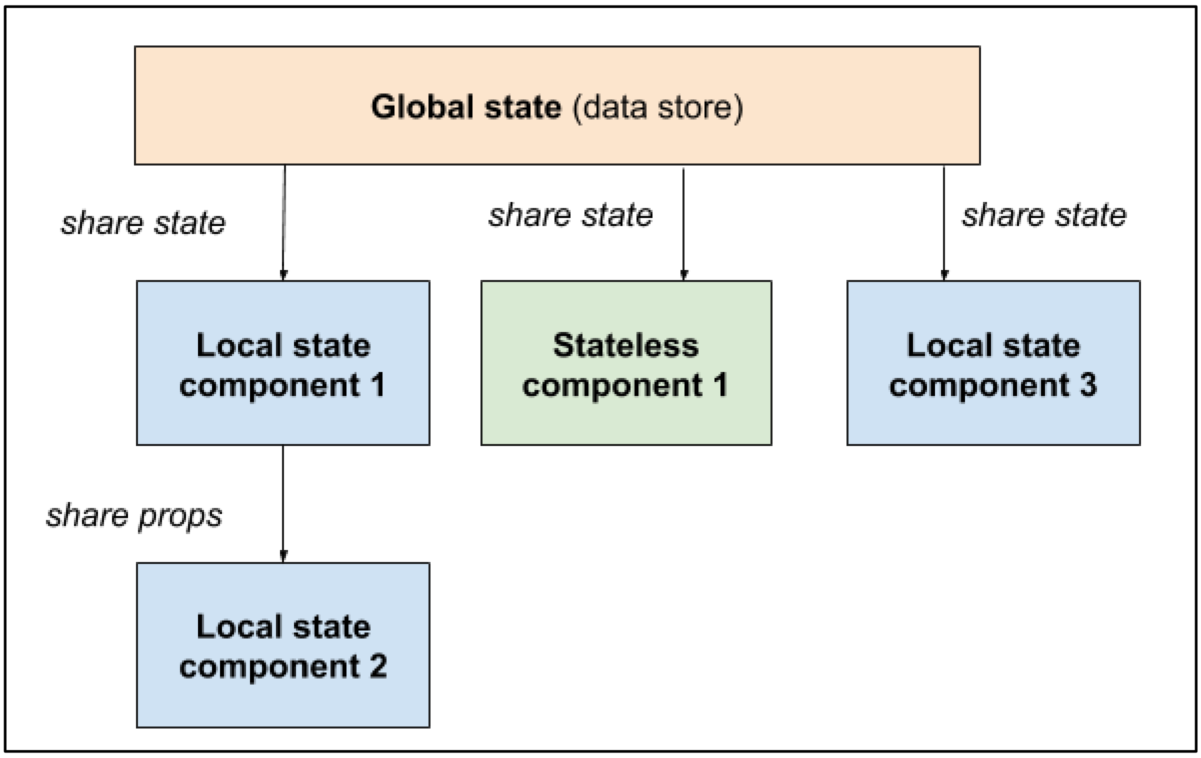

Figure 13: The global state being shared with local component files that can implement their own local state in addition to the global one.

An initial global state variable in a single page application might be called `login_status` with the initial value `’logged_out’`. This global state variable could then be retrieved from the data store by a root component file, which would determine what to show. If the variable `login_status` should contain the value `’logged_out’`, the component could decide that a login screen should be shown. After the login function has been performed and verified through some authentication system, the global stat variable `login_status` could then be updated to the value `’logged_in’`. In this way, the root component file could then hide the login screen and instead show a main menu component file. Further states could be created manually, such as `editing_user_settings` or `in_transaction_mode`. In this way, there is only one HTML page in use, but many different content components that are shown and hidden on the HTML page, depending on the current state.

The data store typically handles one global app state. In addition to this, each component file (shaded in blue and green in Figure 13) can contain a local state, managing variables in that component only. This is useful in order to separate data from each other, and keep more important, more widely used data in the global store. The local state found in a certain component contains local variables that can be accessed and edited by any function in that particular component. In this way, functions existing in the same component file do not need to explicitly pass data to each other, and can pass it through the local state instead, much in the same way the data store functions, but locally within the component file. When building a more complex web application, managing data in a uniform way becomes important, otherwise data has to be handled separately in each component file, which may cause issues with data management for larger applications. This was one of the problems with web applications prior to the data store and single page application model.

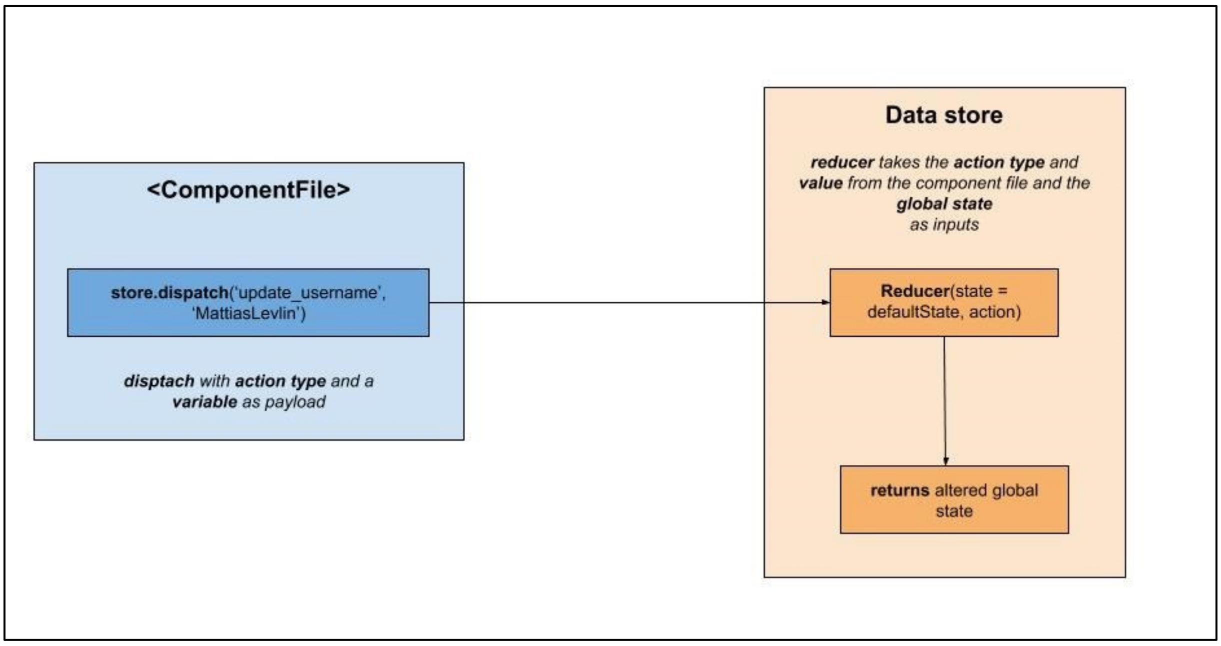
<ComponentFile>
store.dispatch('update_username', 'MattiasLevlin')
disptach with action type and a variable as payload
Data store
reducer takes the action type and value from the component file and the global state as inputs
Reducer(state = defaultState, action)
returns altered global state

Figure 14: Example of a dispatched action to the data store, which is received by a reducer.

Dispatching data to the store can be done in the form of an *action*, a payload which contains data and specifies the type of store alteration that should happen in the data store. For instance, an action type might be called `update_username` with an attached value such as `'MattiasLevlin'`. This process is seen in Figure 14. The dispatched action data and action type are then received by a *reducer*. A reducer is a state evaluation function that typically takes two arguments: the current global state of the application and an action. Based on these inputs, the reducer evaluates whether any global state variables should be updated and returns the altered global state. Since the action type is dynamic, the reducer could handle other types of actions such as `update_email` or `add_item`, with attached values. In addition to the dispatched action and its attached value, a reducer takes the current global state as an input, and then evaluates whether the any state variables should be altered, and what their value should be changed to [49]. In Figure 14, a global state variable username would be updated to the value `'MattiasLevlin'`, but other state variables could be updated as well, at the same time.

Some data store implementations use *mutations* instead of reducers. This is done in Vuex, a library commonly used as a data store implementation in Vue. Mutations are very similar to reducers, but directly modify the existing the data store variables, instead of reassigning and returning them. Data can then be retrieved from the store in various ways, but it is always read-only. For instance, a view can be used, which displays the data in the application. Several views can use the same state: the user login status may be retrieved in order to evaluate whether to run or to disable certain functions in several different files. Retrieval of global state variables can also be implemented as a simple fetch or get statement, or by using selectors: functions used to obtain various parts of the current data store state. Usually, using selectors, several recent state transitions can be retrieved.

With the introduction of state management, the old server-communication paradigm, where each HTML page was retrieved from the server as it was needed, gradually becomes obsolete; the new state management paradigm is more efficient, safer, and easier to scale. Single page applications rewrite the page contents instead of loading entirely new web content from a server upon a user request. Page loads are much faster than the traditional request-response cycle. For these reasons, the global application state is one of the key defining characteristics of single page applications, simplifying the creation of web applications, and giving the developer more resources to control the application.

4.4.1 Data store implementations

While there are many libraries that handle data store implementation, none of the frameworks studied in this thesis contains a data store implementation natively, instead, the framework developers have all opted for a modular approach to preserve modularity and minimize the size of the respective framework. In the documentation of each framework, different companion data store libraries are recommended: the data store Redux is mentioned in React's documentation, while Flux is another commonly used alternative [50]. Vuex is mentioned in Vue's documentation, and other alternatives exist such as Ngrx [51] [52]. These data store implementations all work in a similar way, and while some data store implementations have originally been developed for use with a certain framework, data store libraries are generally platform agnostic in relation to frameworks, which means that any data store can be implemented together with any JavaScript framework. For instance, while Vuex is recommended for usage with Vue, Vue can also use Redux or Flux as its data store implementation. The concepts become more relevant the bigger and more complex the web application is. However, as data stores are not implemented natively in any of the evaluated frameworks, they have been excluded from evaluation. As a suggestion for an extended study, an implementation of the data store could be done, using the same data store library for each of the frameworks.

4.4.2 Navigation: props and routing

All data handling in a single page application does not need to go through the data store. For certain operations, *props* can be used. Props, an abbreviation of properties, are values or objects used to pass data between two component files, often in a hierarchical manner (parent-to-child), instead of going through the data store and altering the application state. When the value of a prop is updated in a parent component, its value will automatically be updated in all the child components to which it has been passed. For security reasons, when a prop is passed to another component, it usually requires some type validation, to ensure that the correct type of value has been passed to the child component. In Figure 13, props are shared between two component files (marked with the text "share props").

An example scenario of this could involve a variable edited locally in its own component file only, such as `edited_username`. If that variable should be needed outside its component file and the variable had not yet been committed to the data store, props could be used to pass the variable to a child component file. A JSX syntax example of this is seen in Figure 15, where the value of `edited_username` is assigned to the variable `edited_name_to_display`. That variable is then passed to the `ChildComponent` file and displayed inside the `ChildComponent` file.

```
// Parent component
Class ParentComponent {
    <ChildComponent
        edited_name_to_display=edited_username
    >
    </ChildComponent>
}

// Child component (separate file)
Class ChildComponent {
    return() {
        <p>{this.props.edited_name_to_display}</p>
    }
}
```

Figure 15: Example of passing props to a child component file in JSX syntax.

One important distinction between data store variables and props is that data store variables can be edited through commit action sent to the data store, but props are read-only when passed to another component file, and changes cannot be committed back to the component file that sent the props variable. In the practical example in Figure 15, it means that the variable `edited_name_to_display` variable would be read-only. Props are generally useful for smaller, localized operations between components. For simple web applications, props may even replace the data store. However, when a web application grows complex enough and starts to handle many different modules, using props only for data transfer between files can become unnecessarily cumbersome and hard to manage. In this case, the data store becomes relevant.

Another navigational concept is *routing*, which is useful when certain components need to be shown depending on the navigation path in the browser. Routing also makes possible traversing backwards in the navigation path. Routing controls the URL navigation in a dynamic way, and inserts a variable in the URL address in order to show different views to the user. In comparison to the original HTML-style navigation where a click in the user interface translated into a server request and a returned, reloaded web page, routing is a much more efficient way to handle site architecture and is also easier to develop, especially when an application grows in scale. Within the context of single-page applications, routing has become a key feature.

```
const Index = { template: ‘<div>Index</div>’ }
const User = { template: ‘<div>User</div>’ }

const routes = [
        { path: ‘/index’, component: Index },
        { path: ‘/user’, component: User },
]

const router = new VueRouter({
        routes
})
```

Figure 16: Example of routing with simple components in Vue, using the Vue router

A typical use case for routing is the navigation in a menu, where different variables are passed to the router depending on what menu tab is clicked. Routing is often connected to the global application state. For a simple example of this using Vue’s router, see the syntax in Figure 16. Here, two templates containing HTML `<div>` elements, `Index` and `User`, are connected to two separate routes, `/index` and `/user`, which are defined in the `routes` object. The `routes` object is then connected to a `VueRouter`, a utility function defined in Vue’s `VueRouter` library, which makes sure that whenever a user navigates to the subroute /index or /user in the web browser, the corresponding template, containing a <div> element is displayed.

5 Technical environment of front-end JavaScript frameworks

In this chapter, the elements needed to set up and run a JavaScript application in practice are discussed. Setting up a JavaScript web application usually involves similar technical details. To start the development of a basic application, a runtime environment is needed to execute the application. In addition, a toolchain is needed to set up the development environment. These tools are not part of the comparison in this thesis, but are relevant as a base for setting up any kind of JavaScript development project, whether it is built with React, Vue, Angular, Svelte, or some other framework. Thus they are of strong relevance to the thesis, and will be referred to in the development evaluation.

5.1 The runtime environment

A runtime environment handles most things needed to execute a JavaScript application, including the front-end applications evaluated in this thesis [53]. Node.js is the most used JavaScript runtime environment, used for running JavaScript code outside of the web browser environment. Most often, this is used for server-side functionality. The creators of Node.js sought to develop a more efficient alternative to the then-popular Apache HTTP Server, which was most often used together with PHP on the front end as a full stack development environment [54]. Node.js is built using Google's V8 JavaScript engine, which itself is built using C++. One of the technical advantages of Node.js is the way it handles input and output communication, such as server requests and local file handling: it is classified as an asynchronous, non-blocking input-output system, allowing for parallel database requests [53]. In the documentation of React, Vue, and Angular, Node.js is recommended as a run-time server environment [55].

5.2 Toolchains

There are many ready-made toolchains for each of the JavaScript frameworks, to enable a quick setup of a working development environment. A JavaScript toolchain generally consists of a package manager, a bundler, and a compiler. A bundler is used for the assembly of written code into packages to optimize loading times; webpack is a commonly used bundler. Finally, a compiler is used to create code that functions in older browsers; Babel is often used for this purpose [55]. Ready-made JavaScript toolchains are often administered using the command line, thus effectively acting as command line interfaces. A command line interface, abbreviated CLI, is used for administrative purposes, such as starting the JavaScript application, running tests, running different builds and specifying execution options. React, Vue, Angular, and Svelte all have their official or semi-official tool chains functioning as command line interfaces. For a local setup of Angular, the Angular CLI tool is required. React's CLI tool is known as create-react-app, and Vue has its own CLI tool as well.

5.2.1 Package managers: npm and Yarn

Package managers are used to install, maintain, edit, and remove packages. Packages are third-party libraries that are used within JavaScript applications to import functionality, instead of writing it from scratch. Packages used within a web application are often called *dependencies*, since the functionality of a web application is said to depend on a certain package, when its functionality is used within the application. The JavaScript package landscape has grown immensely thanks to the open-source culture of sharing code on GitHub and other Internet sites. Some of the most used package manager are npm, an abbreviation of Node Package Manager, and Yarn [56].

Npm was introduced in 2010 to support installation and updates of Node.js libraries. It is a part of Node.js as its official package manager and is thus one of the most used JavaScript package managers. Npm also includes npx, used for execution of npm packages, being included in npm since version 5.2.0. While npm itself is used for package management, npx simplifies the usage of command line interface tools and executable files. Npx is used in the setup of Create React App, one of the recommended toolchains for developing a React application [57]. Npm's syntax for installing a package is `npm install package-name`. Some of npm's popularity as a package manager can be attributed to the fact that it is included as part of the standard installation package when installing Node.js. Another contributing fact is that npm was released in 2010, and has thus been part of the web application ecosystem for a comparatively long time. The popularity of npm is further illustrated by the fact that most other npm alternatives, including Yarn, are developed to be compatible with the npm package registry and to use it by default.

Yarn is another commonly used package manager within JavaScript applications. Yarn was developed by Facebook for some time internally, but released as open-source in October 2016, six years after npm was released. Yarn has been developed with a focus on improved performance, in addition to its open-source nature [56]. Yarn contains its own registry, registry.yarnpkg.com, though it is mostly used as a proxy, to retrieve packages from the main npm package registry. Package installation with Yarn is handled through its add syntax: `yarn add packagename`, which is essentially the same function as npm's `npm install`. One technical difference that sets Yarn apart from npm is that Yarn uses its own lockfile, `yarn.lock`. This file contains all the version numbers of each installed dependency [58].

Within a web application project folder, packages downloaded through a package manager are found in a folder called `node_modules`. This folder is in most cases excluded from the git repository, since it can be very large, contain many required packages, and does not provide any beneficial functionality to the developer. Instead, the dependencies are defined in the `package.json` file and are installed locally in the `node_modules` folder, using a command such as `npm install`. A related file is the package-lock.json, which exists in both npm and Yarn and is typically auto-generated based on the `package.json` file, to track exact versions of packages.

5.2.2 Bundlers: Webpack

JavaScript bundlers are used to compile all existing JavaScript code in a certain project into one single JavaScript file. This significantly improves performance and dependency handling. The need for JavaScript bundlers stems from the JavaScript import syntax, which can quickly get complicated if handled manually. Bundlers solve this issue, by automating a JavaScript project's global import process. In a typical bundler, all dependencies are put into one file, so that in the case that one dependency is dependent on another, no problems are created. This is similar to how Node.js handles dependencies on the server side [59]. Webpack has, for the last couple of years, been one of the most used JavaScript bundlers. Webpack is defined as a static module bundler for JavaScript applications, which creates an internal dependency graph that maps every module needed in a JavaScript project to create a bundle. Webpack starts the bundling process from a certain file, typically from some kind of index file, known as the entry point. From this file, Webpack then looks through each needed import and constructs the dependency graph accordingly, finally outputting the finished bundle to a defined location. Webpack also contains loader functionality, for handling various types of files and converting them into valid modules [60]. React, Vue, and Angular all cite Webpack as a bundler in their respective documentations.

5.2.3 Transcompilers: Babel

A transcompiler is a type of compiler used to compile and convert JavaScript code into a version that can be run on older versions of JavaScript engines. Compiling usually refers to conversion of code between different abstraction levels (such as higher level developer code to lower level machine code), while a transcompiler converts code on the same abstraction level (such as between different JavaScript versions or flavors). For instance, some functionality in newer versions of JavaScript is incompatible with older versions; an example is the arrow function (`() => { … }`), introduced in ECMAScript version 6 in 2015, and certain code functions may have to be converted in order to be run on different browsers. Transcompilers are used so as to allow the user to use newer functionality and run it on an older engine. Babel is one of the most widely used JavaScript transcompilers, used to convert ECMAScript 2015+ code into older JavaScript versions that can be executed on older JavaScript engines [61].

5.3 Setup and project structure

While setting up a JavaScript-based website application, some elements are usually found in the main folder: the folders, `node_modules`, `public`, and `src` (abbreviation of source), as well as the files `package.json`, `package-lock.json`, and `README.md`. The `node_modules` folder contains required libraries that have been installed using the selected package manager, such as npm or Yarn, when running the command `npm install` or `yarn install`, respectively. What packages to store in the `node_modules` folder is tracked using the `package.json` and the `package-lock.json` files. When an `npm` package is installed or uninstalled, the package folders are automatically updated. The `src` folder contains the actual source code, and can be divided into several more folders, depending on the project type. When handling reusable components, each component will usually be defined in its own file, and then exported for reuse in other components, or in the root file. Other commonly used folders are utilities, tests, and translations, though these are not used in this study. `README.md`. is a standard format for providing read-me information on repository sites.

6 The frameworks

The four JavaScript frameworks presented, evaluated and discussed in this thesis can all be categorized as client-side, general-purpose website frameworks, thus excluding server-side back-end frameworks, as well as smaller libraries created for more narrow purposes, such as data visualization or utility function libraries. Furthermore, the frameworks discussed here can be connected to a server-side application, as part of a full-stack application. All of the frameworks presented follow the design philosophy of single page applications.

6.1 Framework selection

As there exist a large number of JavaScript frameworks for front-end development, it is important to define some selection criteria. Reliable, quantitative survey data is notably found in The State of JavaScript surveys, published yearly since 2016 [62]. The 2019 edition had 21,717 developer respondents. In order to select frameworks for evaluation, two data dimensions from this survey were considered: first, how many developers were actively using the framework in question, and second, how much developer interest the framework had been generating. Qualitative data on these dimensions is found in The State of JavaScript surveys, from 2016 to 2019. Yearly usage data, during the time span 2016 to 2019, for the top six frameworks in 2019 is depicted in Figure 17.

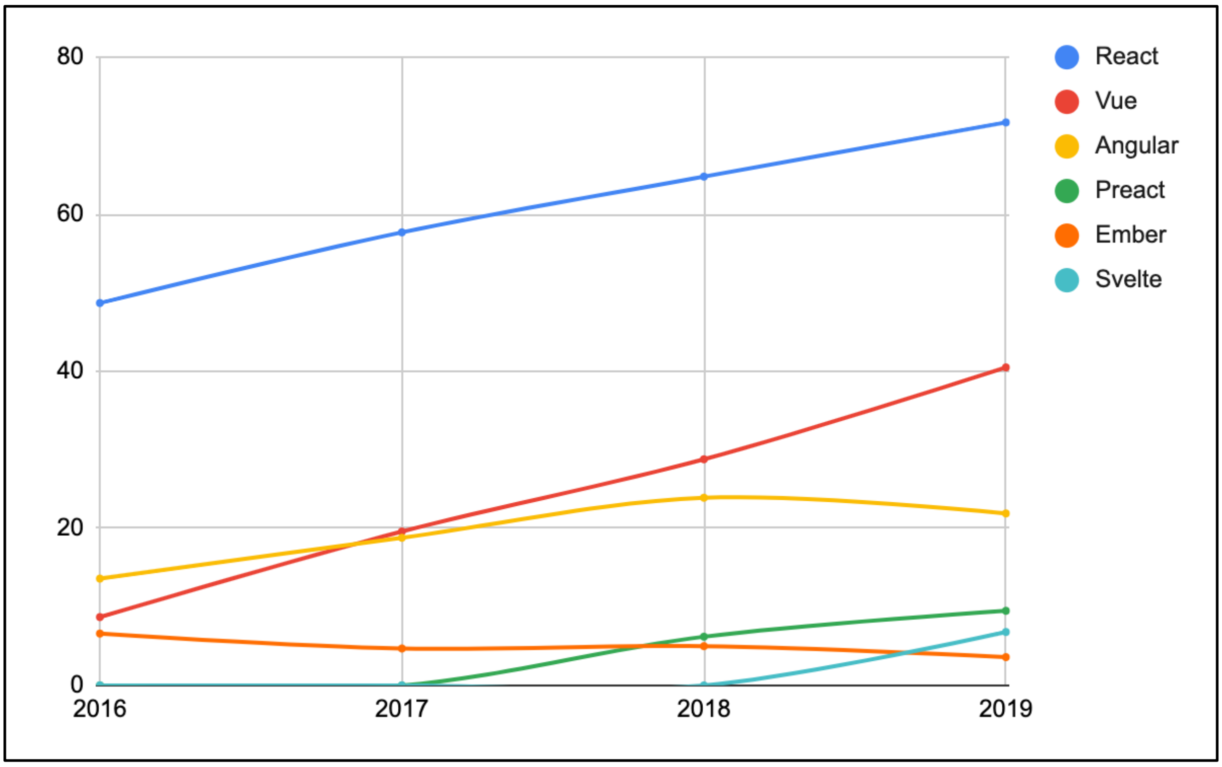

Figure 17: Percentage of respondents who agreed with the statement "I've used it before and would use it again", State of JavaScript Surveys 2016 - 2019.

In this figure, which plots the percentage of developers who said "I've used it before and would use it again", with regard to each of the top six most used frameworks, it can be seen that React has for several years been the most used JavaScript framework, and is still gaining in terms of preference, while Vue is a solid second, followed by Angular, Preact, Svelte, and Ember [63]. Angular has declined somewhat in popularity, but is still recommended by over 20% of developers. Based on the data in Figure 17, it would seem appropriate to select React, Vue, and Angular for evaluation. However, the other important data dimension in the 2019 State of JavaScript survey measures developer interest in learning a framework: Figure 18 shows the percentage of developers who agreed with the statement "I've heard of it, and would like to learn it", for each framework.

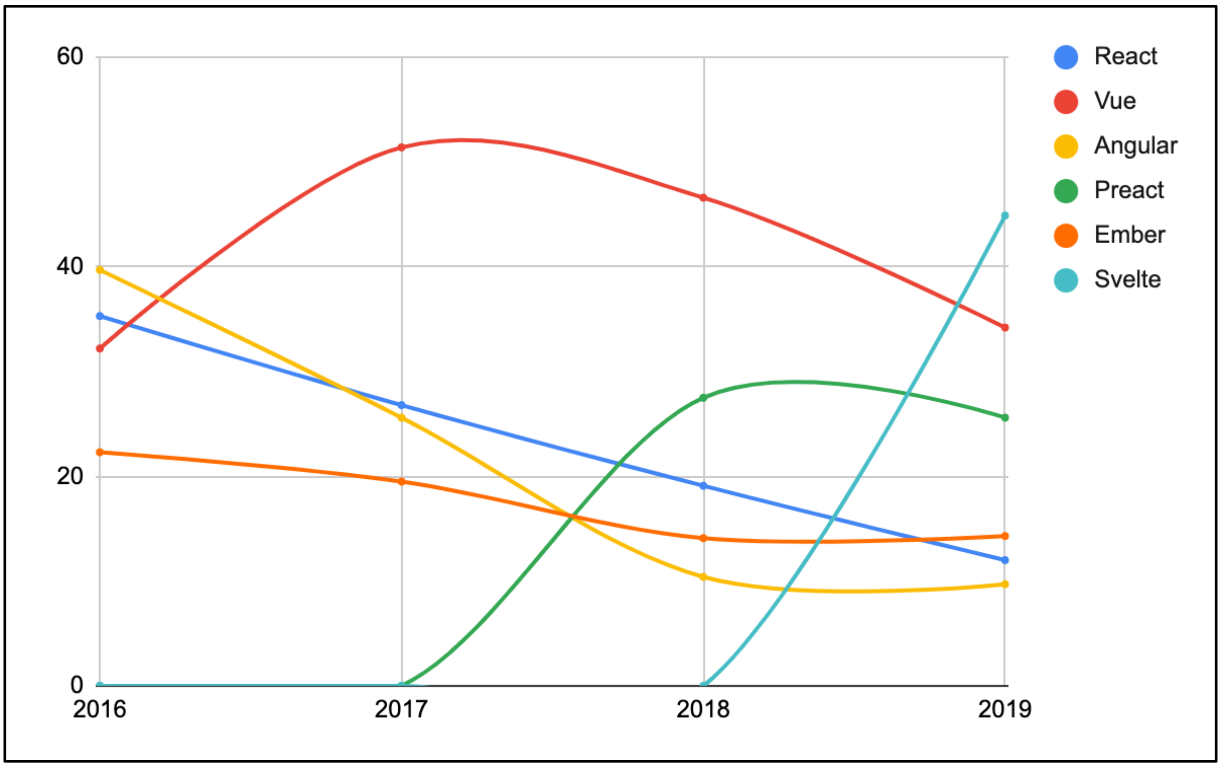

Figure 18: Percentage of respondents who agreed with the statement "I've heard of it, and would like to learn it" for each framework, State of JavaScript Surveys 2016 - 2019.

After evaluating the data in Figure 18, it becomes clear that the newcomer framework Svelte has raised the interest of many JavaScript developers during 2019. While it was not included among the top six frameworks of 2018, it has in one year become the framework which most developers want to learn, overtaking Vue. For this reason, Svelte has been included in this study. Furthermore, Svelte is TypeScript-based, which serves as a comparison to Angular, another TypeScript-based framework. Another quantitative metric for measuring the popularity of a framework is found on the code hosting site Github.com: on Github.com, users can mark a repository they like, or want to save, as 'starred'. This metric is public and the number of stars a certain repository has shows its popularity. The 'star' action is similar to a generic 'like' or 'save' action. The result can be seen in Figure 19.

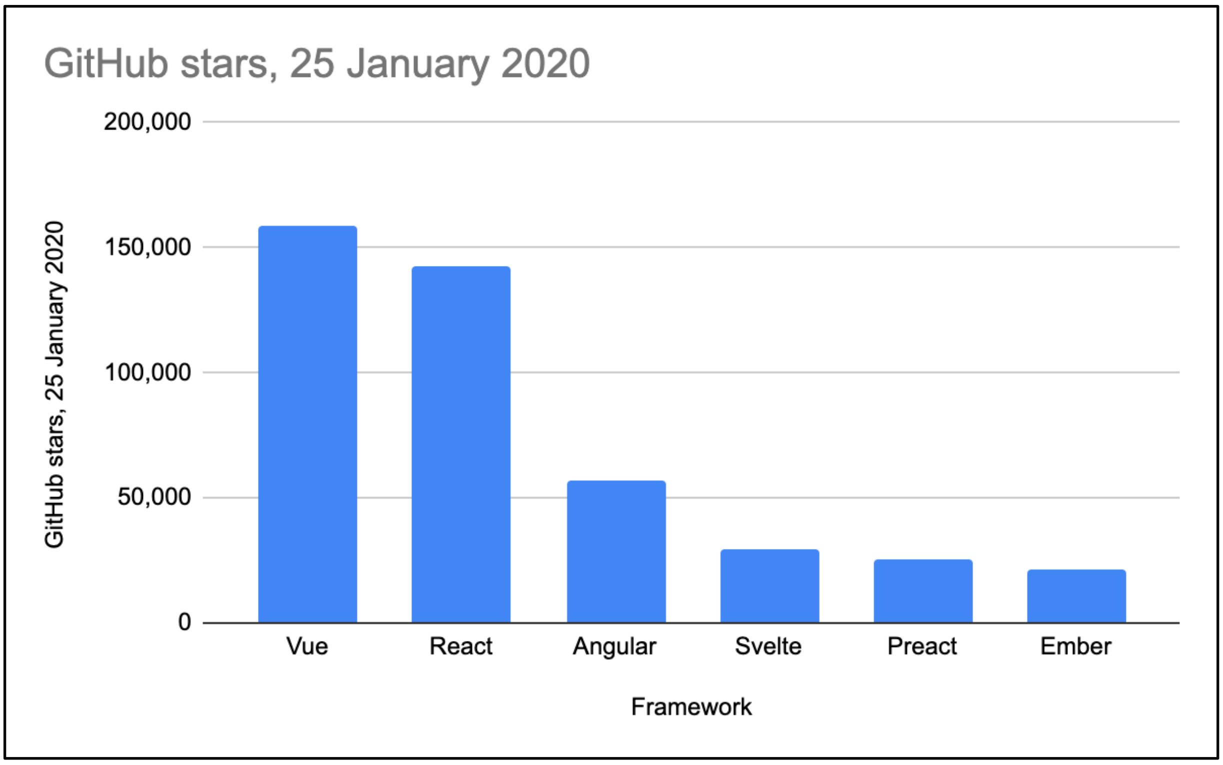

Figure 19: GitHub stars for each of the top six frameworks in the State of JavaScript survey 2019.

In Figure 19, in contrast to the State of JavaScript Survey 2019, Vue is slightly more well-liked than React, with 159,091 stars [64]. React had 142,850 stars, as of 25 January 2020 [65]. React and Vue are both by far the most popular frameworks, with Angular being a distant third, at 56,789 stars [66]. Finally, the rise of Svelte can be seen here too, as it already in January 2020 had 29,756 stars, more than either Preact (25,245) or Ember (21,338) [67] [68] [69].

To summarize, the selection criteria can be defined as follows; the JavaScript framework had to be one of the top six most popular frameworks in the State of JavaScript survey 2019. Furthermore, the framework had to have more than 20,000 stars on Github. In the aforementioned survey, React, Vue, and Angular were the top three most popular frameworks, while Svelte, Vue, and Preact were the top three frameworks that were deemed to be most interesting by developers. Preact is essentially a re-imagined version of React optimized for a faster performance. While it could be included in an expanded study, it was not included here due to its similarity to React and lower popularity. Regarding Ember, the framework has been important in the development of the JavaScript ecosystem, but also seen as superseded in performance and popularity by other frameworks [70]. It is also an older framework, released in 2011, and does not have a dedicated major company supporting its development, like React or Angular has. For these reasons, Ember has not been included in the comparison, though it would be a contender for addition in an expanded comparison.

6.2 React

React.js, or simply React, is a JavaScript library developed by Facebook. It has been described as a declarative, efficient, and flexible framework [71]. The first version of React was released in May 2013. React has a more narrow scope than other frameworks in this list, only rendering the application user interface. The benefit of this is the lightweight structure of the library, being less costly to learn and use. However, this has also meant that React in certain contexts has been referred to as a user interface library, not a framework. Generally, however, it can be considered a framework, as it is used for the same purpose as Vue and Angular 2+ [72]. React was initially developed as a JavaScript port of XHP, a PHP library created by Facebook. XHP was a modification of PHP that allowed for custom component creation, something React also is capable of. This development can be seen as an important step in the overall shift in web development, where JavaScript is chosen as a core web technology instead of PHP, which was the dominant standard during the 2000s. XHP was a library that aimed to prevent malicious user attacks, and out of the JavaScript porting project grew the language JSX (JavaScript XML), which has become a common standard language for React, together with standard JavaScript [73].

One reason for the success of React is that it was the first framework to optimize its functionality according to the DOM: since DOM manipulation is quite costly in terms of computing resources used, React is designed to perform as little DOM manipulation as possible, using state management and the virtual DOM to control this manipulation [74]. The usage of the virtual DOM makes React update faster, at the expense of being more memory intensive: in order to perform fast updates to the browser DOM, React keeps a copy of the virtual DOM tree in memory, which is consumes additional memory. React’s popularity is exemplified by its numerous spinoff libraries: a mobile development form exists, called React Native. The main developer of React, Facebook, has used React Native for the development of parts of its own mobile Facebook application [75].

There are several options for testing out React, which contributes to usability. For simple tests, online code editors are available through React's website. For a complete setup of React, however, the JavaScript package manager npm and the run-time environment Node.js are required as a toolchain. Create React App (CRA) is a commonly used, ready-made React toolchain. It uses Babel as a compiler and webpack as a bundler. The CRA toolchain requires Node.js 8.10.0 or later and npm to run [76]. CRA is very easy to set up, with a complete folder setup being created by a single command: `npx create-react-app <application_name>`. A default folder is set up with this command, containing `.git`, `node_modules`, `public`, and `src` folders, as well as files .gitignore, `package.json` and its related `package-lock.json`, and `README.md`. Create React App is quite a narrow and simple tool, designed specifically for single page applications only, to keep it lightweight and simplify its functionality.

6.2.1 DOM interaction in React

The document object model is the element to which React sends all user interface elements written by the developer. The approach is declarative; the developer defines what state the UI should be in, and React makes sure the DOM is displayed in that state. This effectively replaces the attribute manipulation and event handling approach which is used in jQuery, which has previously been a widely popular approach. By default, React uses a library known as `react-dom` to render things onto the DOM, and contains DOM-specific methods that help with DOM interaction. An element in React's code (as written using JSX or other syntax) is different from a DOM element; React elements can be imagined as simple objects in the code that are given to the `ReactDOM` for translation purposes, in order to be rendered onto the actual DOM in the browser. This is done with the `ReactDOM.render()` function. The manual manipulation of `ReactDOM` is discouraged, since React's state updates keep track of what should be rendered. However, the manual can be used for debugging purposes, where usage of `ReactDOM` is similar to jQuery operations; the `findDOMNode()` is comparable to jQuery's `.get()` function.

6.2.2 Templating, components and syntax

```
function App() {
  <div>
    <p>Welcome!</p>
  </div>
}

export default App;

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Figure 20: Basic React example which renders a welcome message.

In Figure 20, a basic example of a React application is displayed. The function `App()` is defined in which simple HTML elements are used to display a welcome message. That function is then exported to the `ReactDOM.render()` function, which takes it as an input parameter for display. React can be used without JSX, although the coding syntax then has to be altered; HTML elements, such as the `<div>` and `<p>` elements , cannot be directly defined in the code and must instead be created using the `React.createElement` syntax [77]. Dynamic variables can be included in the middle of JSX using the bracket syntax (`{ … }`). As a templating tool, JSX has become a somewhat popular standard, and is supported in Vue as well, though not as a standard choice [78].

6.3 Angular

Angular is a framework that exists in two versions, commonly referred to as AngularJS and Angular 2+. AngularJS is the older JavaScript-based version, which is no longer under active development, while Angular 2+ is newer and based on TypeScript. The version evaluated in this study is Angular 2+.

Angular 2+ was released in 2016 and differs from both the predecessor AngularJS and most other frameworks in that it is based upon the JavaScript flavor TypeScript, being designed as a TypeScript rewrite of AngularJS. It is possible to use Angular without TypeScript, but this choice has been cited as challenging, and is generally not recommended [78]. Nowadays, Angular 2+ is the more popular Angular version, containing performance improvements and other advantages compared with AngularJS [79]. Due to their similarities, and the fact that AngularJS is no longer under active development, AngularJS is not evaluated in more detail in this study. Being designed for larger application development, Angular is one of the larger, more full-featured JavaScript frameworks, both in terms of programming features and file size. In the State of JavaScript survey 2018, the most commonly cited positive aspects of Angular were its amount of features, the programming style, and the documentation. The most commonly cited negative aspects of Angular were its perceived bloatedness, complexity, and heavy style of development, not being recommended for smaller development projects. It was also cited as having a somewhat steep learning curve [80].

6.3.1 DOM interaction in Angular

The way Angular handles the DOM is different from React and Vue, since Angular does not make use of a Virtual DOM, handling instead only direct DOM manipulations. This is similar to how jQuery is used. Angular uses create functions, such as `createCustomElement()` for user-defined components, in order to convert these existing components into a class that can be registered and displayed onto the DOM. The process is somewhat more advanced than with React and Vue: first, the app registers a custom element with the browser (the syntax is `customElement.define(“tag”, Class)`). This element is implemented together with a tag and its class in an intermediate registry called `CustomElementRegistry`, located in the browser. This intermediate registry is then used to instantiate the particular element, and that instance of the element is then translated onto the DOM [81].

6.3.2 Templating, components, and syntax

Angular makes extensive use of its command line interface `ng`. Though components can be created manually, like with React and Vue, Angular allows for the usage of a command called `ng create` to create components. The console results of an execution of this command, `ng create hello-world`, where `hello-world` is the name of the component, are seen in Figure 21.

```
CREATE src/app/benchmark-container/hello-world.component.css (0 bytes)
CREATE src/app/benchmark-container/hello-world.component.html (34 bytes)
CREATE src/app/benchmark-container/hello-world.component.spec.ts (713 bytes)
CREATE src/app/benchmark-container/hello-world.component.ts (320 bytes)
UPDATE src/app/app.module.ts (619 bytes)
```

Figure 21: Log results of running ng create hello-world in the Angular CLI tool.

Running this command sets up an Angular component divided into four files: a `.html` file for displaying HTML elements, a `.css` file for styling purposes, a `.ts` file for dynamic scripting content, and a `.spec.ts` file for testing. This command also sets up some basic functionality within the files themselves; a constructor and a ngInit function are created in the TypeScript file and basic test cases are created in the testing file. The heavier and more thorough development style of Angular is apparent already in the tutorial, where the `ng create` command is recommended, and the usage of the CLI makes the basic Angular workflow different from the general developing style of the other frameworks.

```
// .TS FILE:
import { Component} from '@angular/core';

@Component({
    selector: 'app-hello-world',
    templateUrl: './hello-world.component.html',
    styleUrls: ['./hello-world.component.css']
})

export class HelloWorld {
   name:string;
    constructor() {
        this.name="Mattias Levlin"
}
// .HTML FILE:
<p>Welcome {{name}}</p>
```

Figure 22: Basic Angular example which renders a welcome message. Note that even basic functionality is split into separate files, a .html file and a .ts file.

In Figure 22, a basic component is defined in Angular, and then exported. The functionality is divided between a .HTML file and a .TS file, which is a different way of approaching the component creation process compared to React, where everything relevant to a certain component is usually located in one file only.

6.4 Vue

Vue.js, or simply Vue, was created by Google employee Evan You, who was inspired by AngularJS, but wanted to create a more streamlined, improved version of it; Vue can thus be seen as a lightweight version of AngularJS. Vue's core library is focused on the view layer only. The first version of Vue was released in 2014. Vue has since then grown to become one of the top three most popular JavaScript frameworks, together with React and Angular. In the State of JavaScript survey 2018, the most commonly cited positive aspects of Vue were that it is easy to learn, lightweight, has a nice programming style, documentation, and fast performance [82]. By a large margin the most common negative aspect was its clumsiness.

The developers of Vue released their framework at a time when React and the first version of Angular were dominating the JavaScript framework landscape. For this reason, the Vue developers opted to include a page comparing their own framework to others. The article discusses differences and similarities to React, first and foremost, but also includes comparisons to both versions of Angular, as well as other frameworks [78]. In this documentation, there is an emphasis on the similarities between React and Vue; the documentation authors note that they both utilize the virtual DOM and provide reactive and composable view components. Another similarity is that the core of both frameworks is quite narrow in scope, to maintain focus and enable the users to utilize the framework modularity. Elements that could have been included in the core framework, such as routing and global state management, are instead handled by companion libraries (popular alternatives are Vue-router and VueX). Vue supports all ECMAScript 5 compliant browsers.

6.4.1 DOM interaction in Vue

Vue handles the DOM much in the same way React does, building and using a separate virtual DOM to handle the management of the real DOM [83]. One difference here is that Vue does not use a separate, modular library for this purpose. In practice, if a line like `createElement(‘p’, this.title)` is found in a *.vue file, Vue will create an internal node description to keep track of what information that particular node should display (a `<p>` HTML element with `this.title` as its value), and what child and parent nodes exist, if any. This is known as a `VNode`, or a “virtual node”, and the entire virtual DOM in Vue consists of a tree of `VNodes`, which corresponds to the Vue component structure, as defined by the developer in JavaScript code. This virtual DOM in Vue interacts with the actual DOM, which then updates the HTML web page contents. In Vue, all elements are implemented as virtual DOM nodes; all types of HTML elements, including text elements, and even code comments, are implemented as `VNodes`. The typical function used to create a `VNode` is `createElement`, which creates a node description, sending the information to Vue, which takes care of the virtual DOM translation into an HTML insertion, through the actual DOM. Data is declaratively rendered to the DOM using a `template` syntax.

Vue has a runtime build, which is responsible for creating Vue instances, rendering, and patching the virtual DOM. There is also an alternative full build, which includes the runtime plus a compiler. The runtime build is more lightweight, and also recommended by Vue's documentation. For Vue's standard syntax, which includes functions such as `render()`, the compiler is not needed, though it is needed if a developer wishes to pass HTML encoded in strings to a `template`. A Vue application starts with the creation of a new Vue instance with the `Vue` function. Upon creation, an `options` object containing data is passed to the new instance. The properties found in the `options` object are added to Vue's DOM reactivity system. When the values of these properties change, the view reacts, updating to match the new values.

6.4.2 Templating, components, and syntax

For components, Vue makes use of proprietary `.vue` files, as template files, similar to how React uses `.jsx` files. These files combine HTML elements and dynamic JavaScript functionalities. Components are usually constructed in a small, self-contained, reusable format and then combined into bigger, cohesive modules. The modular components are usually defined in a tree structure of files and folders. Vue components are similar to Custom Elements, which are part of the Web Components Spec, and have been modeled after them.

```
<template>
  <div id="app">
    <p>Welcome!</p>
    <p v-if=loggedIn>You are logged in.</p>
  </div>
</template>

<script>
export default { name: app }
</script>

new Vue({
  render: h => h(App),
}).$mount('#app')
```

Figure 23: Basic Vue example which renders a welcome message.

Vue’s syntax is identified by its heavy usage of directives, prefixed with `v-`. Examples are `v-bind`, used to bind data to a component; `v-if`, used for conditional functionality; and `v-on`, which handles user input and can take an argument such as a function to be run. An example of `v-if` usage is seen in Figure 23, where it is used as part of a simple welcome message. These directives apply special reactive behavior to the DOM, and keep the associated attributes up to date with the specified elements in the code. Certain commonly used directives can also be used in an alternative shorthand format, for instance, `v-on:click` can be written as `@click` in the shorthand syntax. Another distinct feature of Vue is the usage of the “mustache syntax” (double curly braces (`{{ … }}`)). This syntax can be inserted into HTML templates; variables inside it are then interpreted as JavaScript code, instead of HTML. This allows for defining JavaScript expressions inside the mustache syntax.

6.5 Svelte

Svelte is a relatively new JavaScript framework, developed by Rich Harris. It was initially released in 2016 and was gaining in popularity throughout 2019. Its third version was released in April 2019. This version introduced several improvements to the framework that further boosted its popularity, such as changes in local state handling. Svelte is written in TypeScript, just like Angular. There are dedicated build tools for Svelte, such as rollup-plugin-svelte and svelte-loader.

6.5.1 DOM interaction in Svelte

The DOM interaction in Svelte is more reminiscent of Angular than of React or Vue: much like Angular, Svelte has no virtual DOM [84]. Svelte converts written code into JavaScript at build time, instead of run time, and avoids converting declarative elements to the real DOM. While the rationale for implementing a virtual DOM is typically to make DOM updates faster, the developers of Svelte have argued against this in an article titled “Virtual DOM is pure overhead”, explaining their reasons for avoiding the virtual DOM altogether [85]. One cited reason in the article is that any virtual DOM operations are done in addition to the actual DOM updates, which still must be done in any event. Another potentially costly operation that must be done is comparing the previous state of the real DOM to the virtual DOM, and then deciding on whether to update (or "reconcile") the real DOM or not (an operation called diffing). Citing the potential performance drawbacks in performing operations such as these, the Svelte developers have decided to avoid its implementation. This represents a potential innovation, and in any case a break with the virtual DOM tradition.

6.5.2 Templating, components, and syntax

```
<script>
	let name = Mattias Levlin;
</script>

<h1>Welcome {name}!</h1>
```

Figure 24: Basic Svelte example which renders a welcome message.

The syntax in Svelte is based on TypeScript, but takes inspiration from the popular frameworks React and Vue. Much like the other frameworks, a key element is the HTML-like reusable component, here implemented in the `.svelte` file format. Svelte avoids declarative syntax like React and Vue. Starting with Svelte version 3, the local state handling is greatly simplified; instead of the previously used local state methods like `this.set(...)` (or the equivalent `this.setState(...)` in React), this functionality is written simply using the assignment operator (=). In Figure 24, an example of setting up a basic component using Svelte is shown.

7 Technical benchmarks and evaluation

In this chapter, the core assessment of each framework is conducted using various DOM benchmarks. For websites, performance metrics are of a central practical importance. Better performance means lower loading times, increased user satisfaction, and for commercial websites, revenue increases. A loading time reduction of a couple of milliseconds can increase user interaction and retention: the e-commerce company Zalando found that 100 milliseconds of loading time improvement on their website led to a 0.7% increase in customer revenue [86]. How quickly a certain interactive function is executed is important when dealing with large numbers of users and technical difficulties, such as slow connection speeds. Compilation speed is important from a developer's perspective, especially when handling complex web applications and large datasets. The main part of the assessment consists of the various benchmarks in terms of technical performance. I have developed four testing benchmark applications, one for each of the frameworks [87] [88] [89] [90].

The test applications have been set up with a minimum number of required elements according to the respective documentation of each framework. All four applications contain the same functionality and similar files: a root HTML file, titled `index.html`, and a `src` folder where the relevant test code is found. Non-essential files and functionality, such as test-related files, CSS files, and other settings files have been excluded wherever possible, even if they were included in the basic setup recommended by the framework documentation. The test of DOM performance is done using direct DOM updates and insertions of various HTML elements, such as `<div>` and `<p>` elements. In the benchmark application, any kind of more permanent or complex data storage beyond the local state has been excluded, since that would dilute the scope of the study.

7.1 Benchmark considerations

In order to test the DOM performance equally, the conditions should be the same for all frameworks. In this chapter, the chosen control structure, HTML elements, and browser are discussed. The way each performance metric is measured is also discussed; the performance metrics are tracked using the internal lifecycle, which exists in each framework. In the testing benchmarks, the best practices of each framework have been followed. This means that, in React and Vue, the local state is used. Syntax that the React documentation advises against, such as `this.forceUpdate()`, is avoided. DOM operations in React are mainly conducted through the virtual DOM syntax `this.setState()`.

7.1.1 Control structure and DOM elements

When dealing with more than one HTML element in a testing benchmark, the choice of control structure should not affect performance. This means that, if possible, the same control structure should be used for each framework. For adding many elements to the DOM, several options are available; one of the more well-known and basic control structures is the `for` loop, where the elements are added iteratively. This control structure is available in JavaScript and is easy to implement in a similar way for each framework and library. Other possible options are the `array map`, which maps each defined HTML element in an array onto the DOM, and the `forEach` loop, which functions similarly to the `for` loop but does not need an explicit definition of the amount of elements. While designing the testing benchmarks, the `for` loop syntax was generally easy to implement; with Angular and Svelte, it involved performing iterative jQuery-like updates directly to the DOM. However, while implementing the `for` loop syntax in React and Vue, it was discovered that the `Array.map()` syntax in both frameworks was significantly faster than the native `for` loop implementations; the `for` loop in React was quite slow compared to the `for` loop implementation in Svelte and Angular. In Vue, the native `v-for` syntax was initially used for testing, but it was discovered to be even slower than the `for` loop implementation in React. Due to these reasons, in both React and Vue, the iterative DOM updates have been done using the `Array.map()` syntax.

Another point of consideration is what HTML elements to choose. In order to test the performance equally, the same elements should be added to the DOM with each framework. Suitable basic HTML elements used for these benchmark tests are `<div>` and `<p>` elements, since they are always represented as DOM objects. With these `<div>` and `<p>` elements, there are three basic operations tested for each framework: insertion, editing, and removal.

7.1.2 The lifecycles

One thing that makes it easy to make a development transition between React, Vue, Angular, and Svelte is the existence of the lifecycle concept in all frameworks. This is useful also when considering benchmarks: the lifecycle events, which are similar in each framework, can be used to track when certain events have happened, and if timer functionality is added, the execution time can be measured as well.

The lifecycle provides the developer with expanded control over the DOM; it contains functions that are run when certain DOM events happen. These events can be categorized into three main phases, which are consistent across all four frameworks: first, the initialization or creation phase; second, the update phase; and finally, the destruction phase. Each of these phases contains one or several events, for which behavior can be defined in JavaScript code. The initialization or creation phase contains lifecycle events that are run when a certain component is initially loaded and rendered onto the DOM. The events in the update phase are run whenever there is a change in the component (in other words, when there is a change in the DOM). Finally, the destruction phase contains an event that unmounts or destroys the component and removes it from the DOM, to free up memory. One exception exists: Svelte has a unique lifecycle event called `tick()` that does not fit into this three-part model: `tick()` resolves after any pending state changes have been applied to the DOM (if there are none, it is immediately resolved). In each framework, a lifecycle exists for every component that is used by the application. This helps the developer understand when a certain event will happen, and where to place certain functionality. For instance, if a function should run immediately after a component has been rendered onto the DOM, the developer can use `componentDidMount()` in React, `mounted()` in Vue, `ngOnInit()` in Angular, or `onMount()` in Svelte.

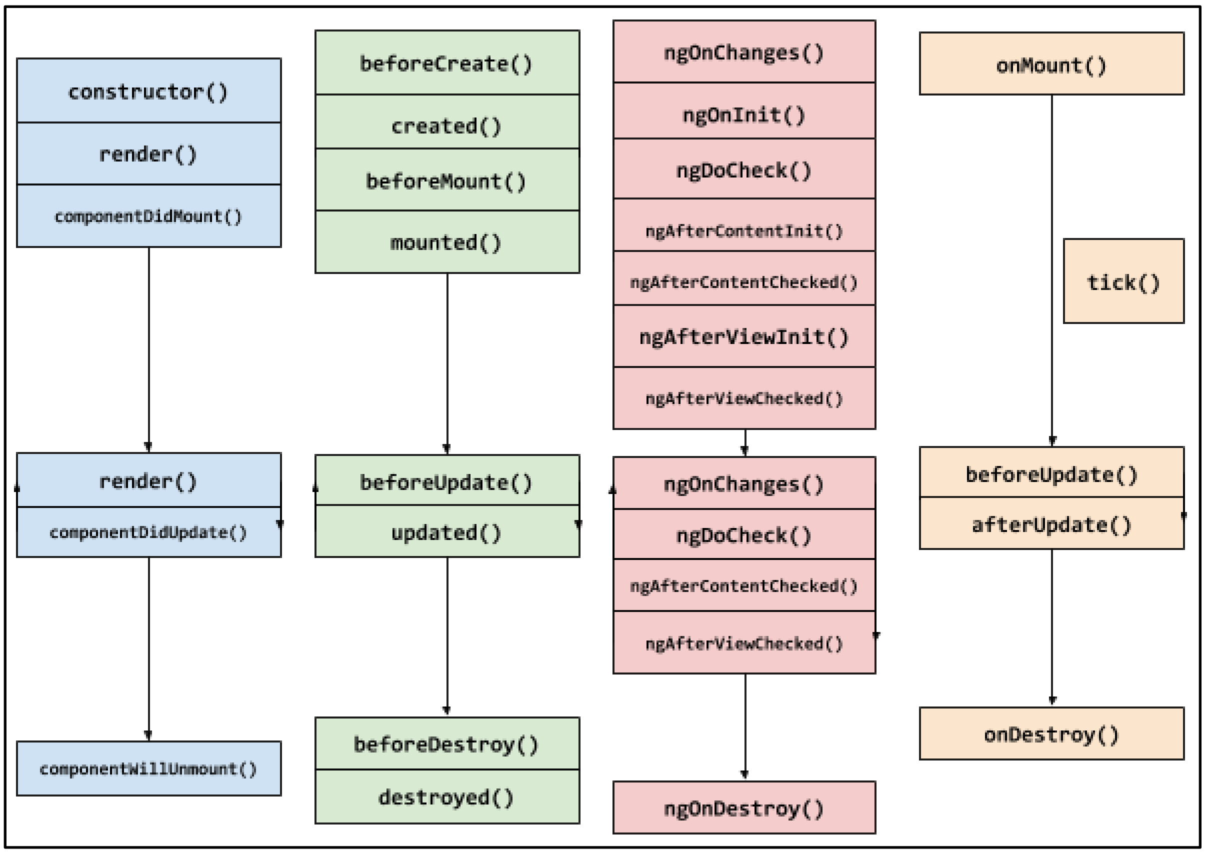

Figure 25: Visual comparison of the lifecycles of React (blue), Vue (green), Angular (red), and Svelte (orange).

The lifecycles of each framework are displayed in Figure 25. As can be seen in the figure, Angular contains more lifecycle events than the others, especially when initiating a component, while Svelte contains the lowest number of lifecycle events [83] [84] [91] [92]. Certain lifecycle events have been altered over time; in React, some events that were previously part of the library have since been removed in newer versions. These lifecycle events are declared unsafe, and usage of these is discouraged for safety reasons. These include `componentWillMount()`, which is now referred to as `UNSAFE_componentWillMount()` and is in practice replaced by `componentDidMount`().

In the next section, the lifecycle events are of practical importance: the event that captures a DOM update is used to track the performance of various DOM operations. The relevant lifecycle event is the last event after each update. In React, this event is `componentDidUpdate()`; in Vue, it is `updated()`; in Angular, it is `ngAfterViewChecked()`; and in Svelte, it is `afterUpdate()`. It can be noted that Angular is the only framework without a unique update event, as the event `ngAfterViewChecked()`, as well as all other Angular update events, is also run also when a component is initialized in the DOM.

7.2 DOM benchmarks

In this section, the DOM benchmarks for each test are outlined. There are three basic operations: insertion, editing, and deletion. For each of these operations, the performance has been recorded with native life cycle events for each framework. The performance for each framework was recorded using the function `performance.now()`, which is a function native to JavaScript and thus available in each framework [93]. The first timestamp was recorded programmatically at the click of a button, and the second timestamp recorded once the DOM event that tracks a DOM update was recorded. The performance has been verified using the Google Chrome developer console.

In this study, the React version benchmarked was 16.12.0, the Vue version was 2.6.11, the Svelte version was 3.20.0, and the Angular version was 8.2.14 (@angular/core version) [87] [88] [89] [90]. In terms of hardware, each test has been run on a MacBook Pro 13-inch model, from 2017. Its processor was a 2.3 GHz Intel Core i5 processor, having 8 GB memory. The browser used for testing purposes was Google Chrome, Version 79.0.3945.130 (in the 64-bit version). All code for each framework has been written using Microsoft Visual Studio Code. Google's developer tools were used to verify and check the performance, specifically Google Chrome's built-in performance recording functionality.

7.2.1 DOM insertion

Outlined in Table 1 is the performance for adding 10000 <div> elements, each containing a <p> element with text data, to the DOM in various ways for each framework. The time for the entire insert operation has been measured according to lifecycle events in each framework (React, Vue, Angular, and Svelte).

Framework Control structure	**React v16.12.0** Array.map()	**Vue v2.6.11** Array.map()	**Angular v8.2.14** for loop	**Svelte v3.20.0** for loop
Attempt 1	27.74	28.73	43.35	33.95
Attempt 2	33.87	24.59	58.48	36.82
Attempt 3	30.76	22.93	42.12	29.56
Attempt 4	28.02	23.42	60.57	30.76
Attempt 5	31.98	24.08	61.60	29.30
Attempt 6	37.27	23.40	58.62	31.13
Attempt 7	35.04	29.16	59.95	30.63
Attempt 8	26.86	24.91	59.87	29.42
Attempt 9	28.54	23.01	41.79	31.07
Attempt 10	29.51	29.37	41.17	29.91
Average (ms)	**30.96**	**25.36**	**52.75**	**31.26**

Table 1: Performance in milliseconds for adding 10,000 <div> elements with a <p> text to the DOM. Best framework performance highlighted in blue.

Evaluating the results in Table 1, it is apparent that Vue has the fastest performance for adding a large amount of HTML elements to the DOM. However, all frameworks performed relatively well in this first testing benchmark, with React and Svelte having a similar performance to Vue. The notable outlier is Angular, which has a slower average performance, at 52.75 milliseconds, while the performance results of the three other frameworks all are found within an interval of six milliseconds (25-31 milliseconds). While Angular is a heavier framework than the other three in terms of development style, the performance for adding a large number of DOM elements is not that much worse: the direct Angular updates to the DOM, omitting the virtual DOM, seem to produce quite a good performance in this benchmark. Svelte uses the same strategy as Angular, but is faster, and reaches a performance similar to React and Vue.

7.2.2 DOM editing

The second benchmark phase looks at DOM editing performance. This is relevant for changing user interface elements. There are two tests in this phase, one where a single element is edited, and one where 10000 elements are edited.

Framework Control structure	**React v16.12.0** reference	**Vue v2.6.11** reference	**Angular v8.2.14** getElementById	**Svelte v3.20.0** getElementById
Attempt 1	16.51	22.25	6.71	0.11
Attempt 2	16.06	21.28	5.64	0.11
Attempt 3	17.48	22.77	5.44	0.11
Attempt 4	16.59	23.34	6.32	0.11
Attempt 5	16.81	21.41	6.43	0.11
Attempt 6	16.55	18.61	6.57	0.12
Attempt 7	15.61	23.60	5.70	0.11
Attempt 8	16.40	22.23	6.12	0.11
Attempt 9	16.35	22.66	5.61	0.12
Attempt 10	17.39	24.16	6.29	0.11
Average (ms)	**16.58**	**22.23**	**6.08**	**0.11**

Table 2: Performance measured in milliseconds for editing one element out of 10,000 <div> elements.

In the second evaluation, editing element one out of 10000 <div> elements was performed. The results are displayed in Table 2. Here, the difference in performance is visible between the frameworks using a virtual DOM (React and Vue) and the ones that do not (Angular and Svelte). Notably, Angular, which was the slowest framework in the first test, performed better than both React and Vue, at 6.08 milliseconds. However, in this test, Svelte was by far the fastest framework, with an average performance for editing one DOM element taking only 0.11 milliseconds. The lightweight implementation of direct DOM interaction in Svelte shows its strength here: for simple, one-off edits, Svelte looks to be the most efficient framework. It is also very easy to implement and handle these kinds of DOM operations in Svelte. The fact that Angular also performed well in this test further shows that direct DOM updates may be the best choice for smaller operations.

Looking at the performance of the virtual DOM frameworks, React and Vue, reveals a slower performance, though the operations were still quite fast, at 16.58 and 22.23 milliseconds respectively. The reason for the slower performance in React and Vue in this benchmark is likely due to the fact that the updates must first travel through the virtual DOM, before reconciling the state of the virtual DOM with the real browser DOM. This additional intermediate step in the virtual DOM seems to require a base amount of time for each update, most likely a couple of milliseconds. Because Angular and Svelte do not need to perform this operation, they enable very fast singular DOM updates, an advantage that Svelte especially seems to utilize and maximize.

Framework Control structure	**React v16.12.0** Array.map()	**Vue v2.6.11** Array.map()	**Angular v8.2.14** for loop	**Svelte v3.20.0** for loop
Attempt 1	17.74	23.40	911.20	918.09
Attempt 2	18.02	21.22	883.82	880.61
Attempt 3	17.78	20.46	886.01	901.07
Attempt 4	17.20	19.86	874.99	882.40
Attempt 5	19.43	20.25	883.20	873.39
Attempt 6	18.28	20.82	878.35	872.37
Attempt 7	17.69	19.98	895.55	881.89
Attempt 8	17.80	20.35	884.95	884.93
Attempt 9	17.64	19.41	877.79	869.33
Attempt 10	17.04	20.64	991.75	886.26
Average (ms)	**17.86**	**20.64**	**896.76**	**885.03**

Table 3: Performance measured in milliseconds for editing each <p> text inside 10,000 <div> elements.

In the third benchmark, an edit of 10000 elements previously added to the DOM was performed. The results of this benchmark are displayed in Table 3. Here, the situation is reversed compared to the previous benchmark. React and Vue were a lot faster than Angular and Svelte. React was slightly faster than Vue, at 17.86 milliseconds on average, while updating 10000 elements in Vue took 20.64 milliseconds on average. In comparison, Angular and Svelte were both very slow. Updating 10000 elements in the DOM using both Angular and Svelte took almost a second on average (896.76 and 885.03 milliseconds, respectively). In these results, the strengths of the virtual DOM in React and Vue can be seen: since the virtual DOM keeps track of what is displayed, its cache-like functionality seems to produce better results than performing direct DOM updates. Angular and Svelte do not seem to handle larger editing operations as efficiently as the virtual DOM frameworks. A base amount of time seems to be needed for each virtual

DOM update in React and Vue, whether updating only one element or updating a large number of elements. However, while this can be detrimental for smaller operations, as seen in the second benchmark, this seems to enable efficient performance for larger updates. In terms of quantitative performance metrics, this benchmark result may be the most prominent argument in favor of using a virtual DOM framework, whether it be React or Vue.

7.2.3 DOM removal

In the third benchmark phase, two tests of removal of DOM elements were performed. These tests can be compared to the editing tests: one test was performed where only one element was removed from the DOM, and one test was performed where all 10000 elements were removed from the DOM. The results of this benchmark are seen in Table 4.

Framework Tool	**React v16.12.0** state assignment	**Vue v2.6.11** state assignment	**Angular v8.2.14** innerHTML	**Svelte v3.20.0** innerHTML
Attempt 1	15.93	22.16	0.09	0.65
Attempt 2	16.47	24.97	0.08	0.51
Attempt 3	16.74	26.20	0.09	0.50
Attempt 4	15.68	26.49	0.09	0.51
Attempt 5	17.98	21.93	0.06	0.50
Attempt 6	16.96	19.14	0.09	0.52
Attempt 7	17.24	26.91	0.09	0.55
Attempt 8	16.07	25.24	0.09	0.51
Attempt 9	16.09	27.06	0.08	0.52
Attempt 10	16.28	25.03	0.10	0.52
Average (ms)	**16.54**	**24.51**	**0.09**	**0.53**

Table 4: Performance measured in milliseconds for removing one <div> element with a <p> text from the DOM.

In this evaluation, one element was removed from the DOM. Here, Angular and Svelte were very fast with direct DOM queries (`innerHTML`). Interestingly, Angular was the fastest framework here, removing a DOM element in only 0.09 milliseconds. This was even faster than

Svelte, which took 0.53 milliseconds. Both these frameworks were a lot faster than React and Vue, which respectively required 16.54 and 24.51 milliseconds to remove a single DOM element. Just like in the benchmark where one element was edited, the strengths of direct, one-off DOM updates are visible here. Again, React and Vue seem to require more time to send the update through the virtual DOM, while Angular and Svelte instantly update the DOM with a very efficient operation. This would confirm the theory that React and Vue always require a certain amount of milliseconds to perform any kind of DOM update, reserved for updating the virtual DOM. This means that single DOM updates faster than a couple of milliseconds seem not to be possible to achieve with React or Vue.

Framework Tool	**React v16.12.0** state assignment	**Vue v2.6.11** state assignment	**Angular v8.2.14** innerHTML	**Svelte v3.20.0** innerHTML
Attempt 1	7.55	32.62	24.09	22.84
Attempt 2	7.42	33.77	24.38	23.71
Attempt 3	7.23	33.04	23.82	23.52
Attempt 4	7.34	32.79	22.25	22.83
Attempt 5	7.41	33.19	24.35	22.76
Attempt 6	7.24	33.00	24.21	22.19
Attempt 7	7.32	33.53	23.80	23.41
Attempt 8	7.49	34.54	23.45	23.08
Attempt 9	7.19	34.18	23.61	22.74
Attempt 10	7.70	32.64	24.36	22.58
Average (ms)	**7.39**	**33.33**	**23.83**	**22.97**

Table 5: Performance measured in milliseconds for each framework while removing 10,000 <div> elements with a <p> text from the DOM.

In the fifth evaluation, a DOM removal operation was performed, removing 10,000 <div> elements with a <p> text from the DOM. The results of this benchmark are seen in Table 5. In this benchmark, React had the fastest performance, at 7.39 milliseconds, while Vue had the slowest performance, at an average of 33.33 milliseconds. However, the all frameworks performed relatively well here. Once again, while React was the fastest framework, it did not manage to achieve a performance faster than a few milliseconds. In contrast to the previous quantitative updates, Angular and Svelte performed well in this benchmark, at 23.83 and 22.97

milliseconds, respectively. This may be due to the fact that a removal operation is quite simple, ideally consisting of a generic remove everything command. The most notable result here is the result of Vue, which surprisingly was the slowest, albeit only around 10 milliseconds slower than Angular and Svelte.

7.2.4 Compilation speed

Table 6 lists the compilation speed for each of the test applications. In this evaluation, all the applications were run in their respective development builds, not in production builds. When all relevant application files are counted together (`index.html` and the contents of the source code folder `src`), the React testing app contains a total of 4 files: 1 JSX file, 2 JavaScript files, and 1 HTML file, for a total of 159 lines of code; the Vue testing app contains a total of 5 files (3 .vue files, 1 JavaScript file, and 1 HTML file) for a total of 201 lines of code; the Svelte testing app contains a total of 3 files (1 .svelte file, 1 JavaScript file and 1 HTML file) for a total of 102 lines of code; while the Angular testing app contains a total of 11 files (5 HTML files and 6 TypeScript files) for a total of 147 lines of code [87] [88] [89] [90]. All projects also contain a `package.json` file and an associated `package-lock.json` file, but no external libraries or dependencies have been added beyond the default ones.

When developing these applications, the documentation was followed for each of the frameworks, thus the setups can be considered as representative of the general development setup of a small application. The compilation speed was measured manually, and averaged out across five test setups. In this case, the time measurement was not automated through code, instead it was started manually when the development setup command was executed in the terminal and stopped immediately once the command line interface indicated that the application was compiled. The compilation tests were run on the same hardware as the technical benchmarks.

	React v16.12.0 (npm start)	**Vue v2.6.11** (npm run serve)	**Angular v8.2.14** (ng serve)	**Svelte v3.20.0** (npm run dev)
Attempt 1	3.98	3.05	8.58	1.57
Attempt 2	4.00	3.06	8.71	1.60
Attempt 3	3.88	3.08	8.58	1.66
Attempt 4	3.97	3.07	8.78	1.63
Attempt 5	3.97	3.07	8.85	1.62
Average (s)	**3.96**	**3.07**	**8.70**	**1.61**

Table 6: Compilation speed in seconds for each of the test applications, using a basic development setup. Development setup command in parenthesis; for instance, (npm start) for React.

The results displayed in Table 6 show that Svelte is the fastest framework in terms of compilation speed; it only took an average of 1.61 seconds to compile the test application. Vue and React are both relatively fast, with an average of 3.07 and 3.96 seconds respectively. Finally the heaviness of Angular is apparent in these tests as well: it took an average of 8.70 seconds to compile the test application. It should be noted that compilation speed is relevant only for developers, and only on startup of the application. The previous technical benchmarks are important both for developers and end users, and DOM updates are usually performed multiple times during an application usage session.

7.2.5 Summary of the technical benchmark tests

In Table 7, a summary of the technical benchmark tests outlined in the previous sections is presented. As can be seen in Table 7, React was the fastest framework in two of the tests, and slowest in none. Svelte was the fastest in two tests, while Angular was the fastest in one and together with Svelte, very slow when updating any DOM elements. Angular was also by far the slowest framework in terms of compilation. Vue performed relatively well in all tests, having the fastest performance for adding 10000 elements, but being beaten by React in four of the benchmarks, even though they both handle the DOM interaction in similar ways.

	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Add 10,000 (ms)	30.96	25.36	52.75	31.26
Edit one (ms)	16.58	22.23	6.08	0.11
Edit 10,000 (ms)	17.86	20.64	896.76	885.03
Remove one (ms)	16.54	24.51	0.09	0.53
Remove 10,000 (ms)	7.39	33.33	23.83	22.97
Compilation (s)	3.96	3.07	8.70	1.61

Table 7: Summary of the previous tests. Best performance marked in blue. Notably slow performances marked in red.

To summarize this table, React has few weaknesses, and performed consistently well in the benchmarks. Svelte would have been comparable to React in performance, were it not for the third benchmark, where Svelte performed quite unsatisfactorily. React is the winning framework in this comparison. Between the two frameworks without a virtual DOM, Angular and Svelte, Svelte is the better performing framework. However, due to the slow performance in the third benchmark, Vue would probably be preferred ahead of Svelte in this comparison. This leaves Angular as the worst-performing framework overall, though it was the fastest framework for removing one element.

7.3 Other evaluations

This chapter evaluates the framework size of each minified library, and then discusses matters from an experiential perspective. There are several factors to consider outside of the technical DOM benchmarks when evaluating these frameworks, though it should be noted that any evaluation which is not based on performance data is going to be of lower reliability and contain a larger risk for personal bias.

7.3.1 Framework size

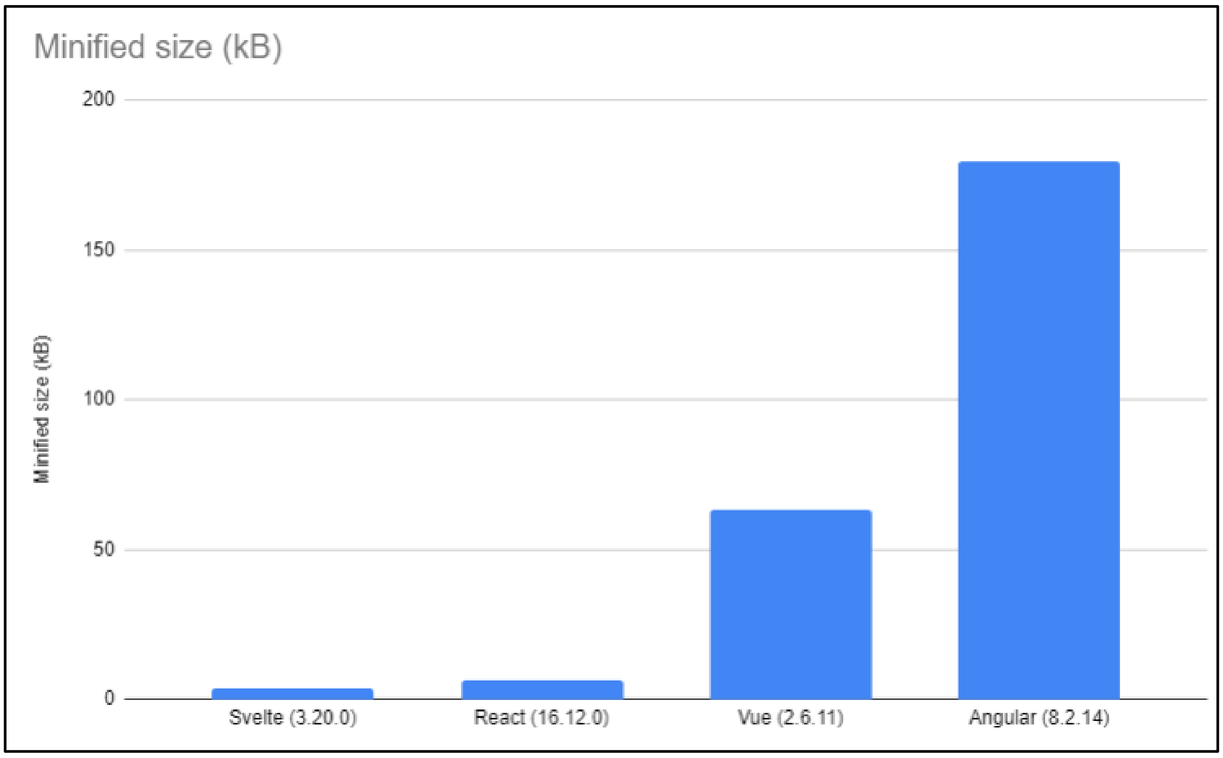

Figure 26: Size of each framework's minified package in the npm registry.

The minified size of the frameworks is displayed in Figure 26 [94] [95] [96] [97]. The frameworks were retrieved in the same versions as the ones studied in the benchmark section. The minified package size was retrieved from the npm registry. While the size of each framework's minified package is unlikely to discourage or encourage any developer from selecting the framework in question for development, it gives an indication of the scope of each framework. The size of each package is below 200 kB, Angular is notably quite a bit larger than the rest (187.6 kB), while Svelte (3.6 kB) and React (6.4 kB) are only a few kB each in minified size. Vue falls somewhere in between, at 63.5 kB. The package size comparison reflects the fact that Angular supports and contains a larger breadth of functionality, while Svelte and React are designed for more streamlined development, giving the developer fewer choices in order to improve efficiency, which becomes especially handy for smaller projects.

7.3.2 Development experience, learning curve, and availability

The initial development setup was very easy for React and Vue. Svelte had a different setup, where the tutorial recommended the user to start off developing in the browser, and then moving to desktop development, which was also quite intuitive. The Angular setup was the most comprehensive and the templates of Angular contained significantly more files and folders than the other three frameworks; Angular has by far the most comprehensive and "heaviest" development process in this evaluation.

In terms of the JavaScript variants, standard JavaScript is used in React and Vue and TypeScript is used for Angular and Svelte. While TypeScript is not in itself much harder or easier to learn than JavaScript, it is not as widely used and contains less libraries and overall documentation than JavaScript, and also has a smaller user base. These factors, combined with the differences in syntax and the time it takes to get used to coding in a slightly different variant of JavaScript, might give a slight advantage to React and Vue. However, TypeScript is still very similar to JavaScript, and has certain advantages in that it is more strict regarding types and syntax than JavaScript, thus preventing some type conversion errors. Thus the division between different JavaScript flavors is unlikely to be of much practical importance for developers when choosing a framework.

In terms of DOM interaction difficulty, Angular and Svelte may be easier to handle; the programmatic implementation of DOM updates is more straightforward in these two frameworks, since neither of them contains an intermediate virtual DOM. Developers with a limited knowledge of JavaScript and the DOM may find it easier to simply perform updates straight to the DOM in the browser, and thus prefer the Angular and Svelte way of doing things. This is exemplified by numerous developer stories and questions on the programming Q&A site StackOverflow.com related to the virtual DOM in React and Vue. A common misconception seems to be that the DOM updates in React and Vue work in a similar way to jQuery. The virtual DOM state updates in React and Vue involve more tweaking of code, and require the developer to become familiar with how the frameworks are designed, and what the purpose of the virtual DOM is. This causes a risk of implementing DOM operations in non-standard ways, which can lead to invalid or slow DOM interaction.

While Svelte and Angular are easier to deal with in terms of DOM handling, development in Angular is more cumbersome due to the fact that functionality is split into several different files for each component. The other three frameworks typically handle everything related to a specific component in one file, but Angular, by default, splits component functionality into separate files (.ts, .spec.ts, .html, and .css). Out of the four frameworks, Svelte was the easiest one to work with in development, having no major drawbacks, implementing a one-file template component syntax and avoiding the virtual DOM. This makes Svelte reminiscent of working with pure JavaScript and jQuery. The syntax of Svelte is also very intuitive, reminiscent of the ease of working with Python.

The availability of documentation was satisfactory for all four of the frameworks. The documentation of React is available in the largest number of languages, 16 [98]. Vue falls behind React, with a documentation available in 8 languages, while the Angular documentation is available in 4 languages [99] [100]. The documentation of Svelte is available in English and Chinese [84]. The framework situation of Angular may present some difficulties for the developer in that it is split between AngularJS and Angular 2+, and the term “Angular” is sometimes ambiguous as to which framework it refers to in online contexts, especially when browsing smaller sites and shorter answers.

In terms of teaching the developer how to use framework, the creators of React, Vue, and Angular had a similar approach with standard documentation and examples, but the structure of Svelte’s initial setup was notable: the guide actually recommends the developer to start out with developing in the browser, using the web development sandbox Svelte REPL. Once the project becomes complex enough, there is an option to download the project in a zipped file, and set it up locally using a few commands. This is a user-friendly approach that may even be inviting for non-coders, and while all four frameworks only require a couple of simple commands, the approach the Svelte developers have taken stands out in its approachability [101].

8 Results, summary, and conclusion

In this thesis, React, Vue, Angular, and Svelte, the four most popular JavaScript frameworks in recent years, have been studied. This chapter contains a final summary, discussion of results, and a summarizing conclusion and recommendation of what framework to use. In Table 8, a summary of all tests and metrics in this study is shown. These results include the metrics already summarized in the benchmarks section, plus the minified size, availability metrics, technical facts, as well as other previously discussed popularity metrics, from Chapter 6 (GitHub stars and sentiment in the State of JavaScript survey 2019).

TECHNICAL METRICS	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Add 10,000 (ms)	30.96	25.36	52.75	31.26
Edit one (ms)	16.58	22.23	6.08	0.11
Edit 10,000 (ms)	17.86	20.64	896.76	885.03
Remove one (ms)	16.54	24.51	0.09	0.53
Remove 10,000 (ms)	7.39	33.33	23.83	22.97
Compilation (s)	3.96	3.07	8.70	1.61
Minified size (kb)	6.4	63.5	187.6	3.5
POPULARITY	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Documentation in number of languages	16	8	4	2
Positive interest in StateOfJs 2019 (%)	83.7	74.7	31.6	51.7
Number of Github stars, January 2020	142,850	159,091	29,756	56,789
OTHER METRICS	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Virtual DOM	Yes	Yes	No	No
TypeScript	No	No	Yes	Yes
Release date	2013	2014	2016	2016
OVERALL RATING	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Placement	1	2	4	3

Table 8: A complete summary of all testing and development metrics.

React is the winning framework of this study. React performed well in the benchmark section, and is also the most used and well-known framework, though Vue had slightly more Github stars. However, if one framework was to be recommended ahead of the others, it would be React, having no discernible weaknesses, and many strengths, winning two of the technical benchmarks and performing very satisfactorily in all of them. The only thing to comment on in terms of drawbacks would be that due to the virtual DOM, React has a performance of a few milliseconds more for simple, singular DOM updates (this is also true for Vue). Vue is similar to React, but performed overall slightly slower than React in the technical benchmarks. Since Vue implements a virtual DOM, much like React, and in other aspects is also reminiscent of React though not as widely used, it may not find a competitive advantage against React. For developers looking to use a framework with a virtual DOM, React is recommended.

The newcomer framework, Svelte, was easy to develop with, fast, and intuitive. The fact that it does not use a virtual DOM is an interesting development, which contributed to a relatively fast performance, apart from quantitative edits. Svelte is easy to start working with also for developers who are novices within the JavaScript domain. The relatively small developer community of Svelte is its greatest weakness, for learning about Svelte, the documentation might have to be consulted directly, while the other three frameworks have quite a large amount of resources available on Q&A websites, as well as a larger number of dedicated forums, tutorials, and communities. Angular is by most standards a bloated framework, though it performed relatively well in the benchmark section, and was the fastest framework when deleting one DOM element. It may find its niche among developers and companies looking for frameworks that support comprehensive, large-scale projects. This also seems to be the consensus among the JavaScript developer community at large. By non-technical metrics, the most popular frameworks are React and Vue by a large margin, with React being well-established as a mature framework, but with Vue seemingly growing in popularity. While the developers of Vue estimated that speed is generally an unlikely factor in deciding what framework to choose, the benchmarks in this study may still be of interest to web developers dealing with large amounts of DOM data [78]. One benchmark metric area that could be studied in an expanded study would be how the different frameworks handle memory allocation and memory usage, which was not included in this study, but has been included in other similar benchmark studies [102] [103].

The future of the front-end JavaScript landscape is hard to predict. The most interesting development will be to see if the non-virtual DOM paradigm of Svelte manages to break through and cause a shift in the landscape. React is still a very powerful framework with a large user base, and is also backed by Facebook. React is likely to be the dominant framework for the foreseeable future, perhaps integrating insights from newcomer frameworks such as Svelte. Other newcomer frameworks may also be launched in the coming years that implement the best ideas from the older frameworks. However, for now, for developers looking to develop applications using a mature, well-liked framework with a good performance and plenty of documentation, React is the best choice.

Svensk sammanfattning

1 Inledning

Ett av de vanligaste verktygen för att skapa interaktiva, avancerade webbsidor år 2020 är programmeringsspråket Javascript. Under det senaste årtiondet har många ramverk byggts upp ovanpå Javascript. Dessa ramverk underlättar skapandet, designen och upprätthållandet av interaktiva webbsidor. År 2020 var de mest populära ramverken React och Vue, följda av Angular. Ett nytt ramverk, Svelte, höll samtidigt på att bli mer populärt och intresserade webbutvecklare. I denna studie evalueras dessa ramverk genom ett antal tekniska tester, för att undersöka vilket eller vilka av ramverken som passar bäst för webbutveckling~~,~~ och vilka styrkor och svagheter vart och ett av ramverken har. De tekniska testerna består av olika hanteringar av DOM-operationer. DOM står för *Document Object Model* på engelska, och är en modell som representerar visuella element i en webbläsare. Ett antal icke-tekniska evalueringar för ramverken beskrivs också, såsom skillnader i arkitektur, utvecklingssätt, popularitet, tillgänglighet och mognad. Denna studie kan vara av intresse för webbutvecklare, speciellt för de som fokuserar på användargränssnitt och design av webbsidor, och studien kan underlätta valet av ramverk för ett webbutvecklingsprojekt. Studien kan även ses som en allmän introduktion till Javascript-ramverkens domän. Vidare kan de tekniska testerna och resultaten vara värdefulla för intressenter som lägger vikt vid prestanda och snabbhet för applikationer i webbläsaren.

2 Javascripts miljö

Den vanligaste miljön där Javascript ofta ses implementerat är på webbläsare. Här kombineras Javascript ofta med teknologierna HTML och CSS. Tillsammans har dessa tre verktyg beskrivits som det tretal teknologier alla webbutvecklare bör bekanta sig med. HTML är på engelska en förkortning av *Hypertext Markup Language* och har definierats av organisationen World Wide Web Consortium som webbens grundläggande programmeringsspråk för att skapa innehåll för alla, för användning överallt. Med hjälp av HTML skapas alla typer av statiska, visuella element på webbsidor. HTML släpptes 1993, tre år före Javascript, och fungerar väl tillsammans med Javascript då Javascript utvecklades som en extension till HTML. CSS är en engelsk förkortning av *Cascading Style Sheets* och har definierats av World Wide Web Consortium som en mekanism för att definiera ett visst webbdokuments stil, så som fonter, färger och mellanrum. CSS används vanligtvis som ett komplement till HTML och hanterar de grundläggande designaspekterna av webbinnehåll. För att summera används HTML för skapande av innehåll på webbsidor, CSS för utsmyckandet av detta innehåll, och Javascript för dynamisk interaktion med innehållet.

3 Javascript

Javascript utvecklades av Netscape Communications och släpptes i sin första version 1996. Målet med Javascript var att skapa ett språk som skulle kunna hantera enklare dynamiska operationer i webbläsaren, som komplement till statiska, HTML-baserade webbsidor. Under de följande åren växte Javascript i popularitet, och kom att bli ett av de mest använda webbverktygen. Javascript är ett objektorienterat skriptspråk med support för dynamiska typer. Detta betyder att en variabels typ inte behöver definieras som till exempel heltal eller sträng, utan tolkas av Javascript och kan enkelt konverteras. Javascript stöder vanliga programmeringsfunktioner som *if*, *while* och *switch*. Javascript standardiserades 1997 genom en standard vid namnet Ecmascript, som uppdateras kontinuerligt. En ofta sedd version är Ecmascript 2015, som var den sjätte lanserade versionen, som introducerade nya egenskaper till Javascript som *let* och pilfunktioner. Utöver standardvarianten av Javascript finns flera deriverade varianter, en känd variant heter Typescript, som är en något mer strikt version av Javascript som till exempel inte stöder dynamiska typer. I denna studie är standardversionen av Javascript och Typescript speciellt relevanta, eftersom att två av ramverken, React och Vue, är baserade på standardversionen av Javascript, och två av ramverken, Angular och Svelte, är baserade på Typescript. Javascript har växt i popularitet sedan 1990-talet, och har sedan 2013 varit det mest använda programmeringsspråket på den populära programmeringssidan Stack Overflow. Vidare har Javascript implementerats på uppskattningsvis 95 % av alla webbsidor. Javascripts typiska miljö är en webbsida med användargränssnitt, men det kan även användas på webbservrar.

Ett viktigt koncept i denna studie är dokumentobjektmodellen, ofta refererad till med förkortningen DOM (på engelska *Document Object Model).* Dokumentobjektmodellen är ett programmeringsgränssnitt som ger tillgång till HTML-webbsidors innehåll, struktur och stil.

Detta gränssnitt används i testerna i den praktiska delen av denna studie. De flesta populära webbläsarna implementerar dokumentobjektmodellen. Detta betyder i praktiken att då en webbsida laddas in av webbläsaren, skapas automatiskt ett DOM-gränssnitt för sidan, vilket kan användas för editering av webbsidans innehåll med hjälp av Javascript eller ett annat skriptspråk. Dokumentobjektmodellen är implementerad som en trädstruktur, där trädets rot består av webbsidans rotdokument och där webbsidans innehåll består av grenar på trädet. Detta träd och dess grenar kan traverseras, editeras och manipuleras med hjälp av Javascript. DOM-gränssnittet kan förstås som en mellanliggande modell mellan användaren och webbsidans innehåll, som ger användaren en utvidgad, programmatisk tillgång till webbsidan.

De olika ramverken i denna studie hanterar dokumentobjektmodellen på två olika sätt. Två av ramverken, React och Vue, implementerar en virtuell dokumentobjektmodell, en egen representation av webbläsarens dokumentobjektmodell. Med en virtuell dokumentobjektmodell hanteras alla inkommande Javascript-förfrågningar först av den virtuella modellen och dess innehåll uppdateras först. Sedan skickas uppdateringarna vidare till den faktiska dokumentobjektmodellen i webbläsaren, där den modellens innehåll synkroniseras med den virtuella dokumentobjektmodellens innehåll. Målet med en virtuell dokumentobjektmodell är att accelerera uppdateringshastigheten i webbläsarens egen dokumentobjektmodell och att förenkla editeringsprocessen för webbutvecklaren. De två andra ramverken, Angular och Svelte, implementerar inte en virtuell dokumentobjektmodell, istället uppdaterar dessa ramverk webbläsarens dokumentobjektmodell med direkta Javascript-förfrågningar.

4 Användargränssnittsorienterade Javascript-ramverk

React, Vue, Angular och Svelte klassas i denna studie alla som Javascript-ramverk (på engelska *framework*). Ett relaterat programmeringsuttryck är bibliotek (på engelska *library*), som består av en samling resurser avsedda för utförande av en viss operation eller en liten domän av operationer. Ett exempel på ett bibliotek är jQuery, som ger en webbutvecklare möjlighet att manuellt utföra uppdateringar av dokumentobjektmodellen. Ett ramverk kan ses som ett mer avancerat bibliotek, definierat som en samling stabila resurser som förser en programmerare med verktyg för att skapa en hel webbapplikation.

Ett relevant uttryck inom Javascript-ramverkens domän är ensidsapplikationen (på engelska *Single Page Application*, förkortat SPA). En ensidsapplikation laddar in alla resurser som krävs då användaren navigerar till en webbsida för första gången; istället för att skicka förfrågningar till webbsidans server för varje användaroperation, laddas det behövda innehållet fram lokalt på webbsidan då det behövs. Detta belastar nättrafiken mindre och ger webbutvecklaren större kontroll över sidans modularitet. Ensidsapplikationens principer växte fram under 2000-talet, och dessa principer blev populära främst tack vare förbättringen i nätverksprestanda. Ramverken i denna studie implementerar alla ensidsapplikationens principer.

En funktionalitet som växt i popularitet samtidigt med ensidsapplikationen är komponentbaserat innehåll. Innehållet på en webbsida utvecklad med ett generiskt, modernt Javascript-ramverk består ofta av komponenter (på engelska *components*). En komponent kan vara till exempel ett textfält eller en inloggningsknapp, och kan återanvändas flera gånger. På så sätt skapas modularitet i applikationen, då en viss komponent bara behöver definieras en gång, men kan användas ett obegränsat antal gånger. En viss komponent kan importeras in till en annan komponent, för att bygga upp en trädstruktur. Data kan även skickas mellan komponenter. En viss komponent definieras vanligtvis i en mall (på engelska *template*), en HTML-liknande fil som ofta innehåller ramverksspecifik syntax och Javascript-funktionalitet. En mall beskriver en viss komponents utseende och funktionalitet. Då en definierad komponent används, skickas dessa egenskaper vidare till dokumentobjektmodellen för att lägga till innehållet i webbläsaren. Fördelen med mallar är att HTML-element och dynamisk Javascript-funktionalitet kan kombineras i samma fil, vilket förenklar utvecklingsprocessen för programmeraren.

Ett annat koncept som populariserats under de senaste åren är applikationstillståndet (på engelska *state*). Detta koncept används också av alla ramverk i denna studie. En applikation kan befinna sig i ett visst tillstånd baserat på vart användaren har navigerat eller vilka operationer som utförts. Applikationstillståndet existerar ofta i två varianter, det globala tillståndet och det lokala tillståndet. Det globala tillståndet kan kontrollera mera fundamentala parametrar, som till exempel om användaren är inloggad eller vilket språk applikationen ska visas på. Det lokala tillståndet implementeras inom mindre komponenter, och kan till exempel kontrollera om en viss informationspanel ska vara expanderad eller minimerad.

5 Teknisk miljö

Ett antal element krävs för att i praktiken påbörja utvecklandet av en webbapplikation med något av de Javascript-ramverk som beskrivs i denna studie. Dessa element inkluderar en exekveringsmiljö (på engelska *runtime environment*) och en verktygskedja (på engelska *toolchain*). En exekveringsmiljö möjliggör exekveringen av en Javascript-applikation. Den mest använda Javascript-exekveringsmiljön är Node.js, som rekommenderas för användning i de flesta populära Javascript-ramverks dokumentation. En verktygskedja är en samling teknologier som försnabbar och förenklar utvecklingsprocessen av en applikation. De flesta ramverk erbjuder färdiggjorda verktygskedjor, men en verktygskedja kan även konstrueras manuellt. En verktygskedja innehåller vanligtvis en pakethanterare (på engelska *package manager*), som hanterar import av funktioner från utomstående programmeringsbibliotek. En annan viktig komponent i verktygskedjan är en paketerare (på engelska *bundler*), som kompilerar ett helt Javascript-projekts kod till en enda fil, vilket förenklar hanteringen av import och export filer emellan. Slutligen inkluderas ofta en transkompilator (på engelska *transcompiler*) som säkerställer att kod skriven med en viss version av Javascript är kompatibel med äldre versioner av Javascript eller mellan olika varianter av Javascript.

6 Presentation av ramverken

De fyra ramverken som evalueras i denna studie kan kategoriseras som klientsideramverk, det vill säga att de körs i webbläsarklienten, och inte på en server. Ramverken kopplas dock ofta vidare till serverfunktionalitet. Vidare är ramverken avsedda för generell utveckling av webbapplikationer från början till slut, inte för en mindre avgränsad domän. De fyra ramverken är React, Vue, Angular och Svelte. Dessa ramverk hade alla i januari 2020 mer än 20 000 stjärnor av användare på källkodswebbsidan Github.com. Alla fyra ramverk nämndes också bland de mest använda och intressanta Javascript-ramverken i undersökningen State of Javascript survey 2019.

React är utvecklat av Facebook och släpptes 2013. React var under senare delen av 2010-talet det mest populära Javascript-ramverket. React introducerade JSX-syntaxen, som står för Javascript XML, en innovativ syntax som kombinerar HTML och Javascript i samma fil (med filändelsen .jsx) och möjliggör enkel definition av återanvändbara komponenter. JSX har använts som inspiration hos efterföljarna Vue och Svelte. React utvecklades med fokus på dokumentobjektmodellens prestanda; utvecklarna strävade efter att utföra så lite DOM-manipulationer som möjligt och introducerade därför en virtuell dokumentobjektmodell som sköter DOM-operationerna, istället för att manuellt redigera DOM-attribut och hantera händelser (på engelska *events*), vilket tidigare var populärt (speciellt hos användare av biblioteket jQuery och ramverket Angular).

Vue utvecklades primärt av en anställd vid Google, Evan You, och är inspirerat både av React och av det äldre ramverket AngularJS. Vue lanserade i sin första version 2014, och har växt i popularitet till att vara det näst mest använda Javascript-ramverket, efter React. Utvecklarna av Vue har noterat likheterna mellan Vue och React i att de båda använder en virtuell dokumentobjektmodell. Vue implementerar sina komponenter i .vue-mallfiler, vilka går att jämföra med Reacts .jsx-filer. En noterbar egenskap Vue har är dess användning av direktiv i komponenterna. Direktiven identifieras med syntaxen v: till exempel v-bind som binder data till en komponent och v-if som evaluerar if-satser.

Angular är ett ramverk som skiljer sig från React och Vue på flera sätt. Ramverket har en föregångare, en äldre version, AngularJS, baserad på standardvarianten av Javascript, släpptes redan 2010. Angular, ibland kallat Angular 2+, är en efterföljare som utvecklats med Javascript-varianten Typescript, och lanserades 2016. Denna nyare version utvecklades för att förbättra prestanda och stödja utvecklingen av större, mer komplexa applikationer, funktioner som sågs vara bristfälliga i AngularJS. Angular implementerar inte en virtuell dokumentobjektmodell, istället skickar ramverket DOM-uppdateringar direkt till webbläsaren. Komponenter skapas vanligtvis via Angulars konsol, där en komponent delas in i flera individuella filer. Angular-komponenters dynamiska funktionalitet definieras i .ts-filer, men definieras också med hjälp av associerade HTML- och CSS-filer. Detta står i kontrast med React och Vue, som oftast placerar en komponents alla funktionaliteter, utseende och egenskaper i samma fil.

Svelte är det nyaste ramverket i denna studie. Det lanserades 2016 och växte i popularitet under år 2019. Svelte har, som Angular, ingen virtuell dokumentobjektmodell. Svelte-kod konverteras till Javascript då applikationen laddas. Utvecklarna av Svelte har argumenterat mot implementeringen av en dokumentobjektmodell, vilket är en potentiell innovation som skulle kunna bryta normen av DOM-användning som React och Vue gjort populär. Gällande komponenter implementerar Svelte funktionaliteterna i en fil, av filformatet .svelte. Svelte har också förenklat hanteringen av applikationstillståndet medan React och Vue ofta använder nyckelordet `this` för att uppdatera applikationstillståndet, uppdateras tillståndet i Svelte endast med hjälp av tilldelningsoperatorn (=).

7 Evaluering och test

De tekniska testerna i denna studie består av ett antal olika jämförelser med hjälp av dokumentobjektmodellen. En testapplikation har utvecklats i fyra versioner, en för varje ramverk, där ett visst antal uppdateringar så som tillägg, editering och borttagning av HTML-element genomförs. HTML-elementet <div> har använts för detta ändamål. De evaluerade ramverken implementerar alla en livscykel (på engelska *life cycle*) som kontrollerar vad som ska renderas av dokumentobjektmodellen, och när ett visst element ska uppdateras, läggas till eller tas bort från modellen. Ramverkens respektive livscykel innehåller liknande funktioner som fångar upp händelser då en viss DOM-funktionalitet har körts färdig, och dessa funktioner har använts för att mäta prestandan för de tekniska DOM-uppdateringarna.

React presterade bäst i de tekniska testerna, med Vue som tvåa. Svelte hade även bra prestanda. Angular klarade sig överraskande bra. Storleken på alla evaluerade ramverk i komprimerad form är mindre än 200 kilobyte. I sina respektive senaste versioner hämtade via pakethanteraren npm är Svelte är det minsta ramverket i komprimerad form, på 3,6 kilobyte. Reacts storlek är 6,4 kilobyte, Vues storlek är 63,5 kilobyte, medan Angular är störst, på 187,6 kilobyte. Gällande den praktiska utvecklingsprocessen var React, Vue och Svelte alla relativt enkla att komma igång med, medan Angulars uppstart tog längre tid. Sveltes dokumentation är den simplaste, där det mesta förklaras via interaktiva exempel och en kodeditor i webbläsaren.

8 Resultat och avslutning

Baserat på testresultaten, rekommenderas React för användning. Med beaktande av alla parametrar, inkluderat popularitet och historia, skulle React kunna beskrivas som förstavalet i denna studie. Vue liknar React, men har något sämre prestanda i de flesta av testerna. Svelte är ett intressant nytt ramverk, men risken finns att det inte lyckas etablera sig i samma utsträckning som React och Vue. Angular är ett mera komplext och tungkört ramverk, och rekommenderas inte. Framtiden för JavaScript-ramverken ser ut att vara dynamisk, och en av de mest intressanta detaljerna att följa är om Sveltes paradigm gällande att inte använda en virtuell dokumentobjektmodell lyckas slå igenom, eller om Reacts och Vues filosofi om att använda sig av den virtuella modellen hålls kvar som en standard.

References

[1] International Communications Union, “Statistics”, January 2020. Retrieved from https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx, 22 January 2020.

[2] StatCounter, “Browser Market Share Worldwide, December 2018”, December 2018. Retrieved from http://gs.statcounter.com/browser-market-share#monthly-201812-201812-map, 4 May 2019.

[3] Berners-Lee, Tim, and Mark Fischetti. Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor. DIANE Publishing Company, 2001.

[4] Flanagan, David. JavaScript: the definitive guide. " O'Reilly Media, Inc.", 2006.

[5] World Wide Web Consortium, “HTML”, April 2019. Retrieved from https://www.w3.org/html/, 4 May 2019.

[6] World Wide Web Consortium, “What is CSS?”, April 2019. Retrieved from https://www.w3.org/Style/CSS/, 4 May 2019.

[7] Lie, Hakon Wium, and Bert Bos. Cascading style sheets: Designing for the web, Portable Documents. Addison-Wesley Professional, 2005.

[8] cssinjs.org, “Introduction”, cssinjs.org. Retrieved from https://cssinjs.org/?v=v10.0.3, 22 January 2020.

[9] Charles Punchatz, “How JavaScript Became the Dominant Language of the Web”, August 2017. Retrieved from https://www.lform.com/blog/post/how-javascript-became-the-dominant-language-of-the-web, 4 May 2019.

[10] Kyla Brown, “JavaScript: How Did It Get So Popular?”. Code Academy, September 2018. Retrieved from https://news.codecademy.com/javascript-history-popularity/, 8 May 2019.

[11] International Organization for Standardization, “ISO / IEC 22275: 2018 (ECMAScript® Specification Suite)”. Retrieved from https://www.iso.org/standard/73002.html, 26 January 2020.

[12] ECMAScript, E. C. M. A., and European Computer Manufacturers Association. “Standard ECMA-262, ECMAScript 2019 Language Specification (10th edition)”. Retrieved from https://www.ecma-international.org/publications/standards/Ecma-262.htm, 26 January 2020.

[13] TypeScript “TypeScript documentation” Retrieved from http://www.typescriptlang.org/docs/home.html, 3 May 2019.

[14] Sacha Greif, Raphael Benitte, “State of JavaScript survey, 2019: JavaScript flavors” Retrieved from https://2019.stateofjs.com/javascript-flavors/, 25 January 2020.

[15] World Wide Web Consortium, “Extensible Markup Language (XML) 1.0 (Fifth Edition”, 2008. Retrieved from https://www.w3.org/TR/REC-xml/#sec-origin-goals, 12 October 2019.

[16] Garrett, Jesse James. "Ajax: A new approach to web applications." (2005). Retrieved from https://courses.cs.washington.edu/courses/cse490h/07sp/readings/ajax_adaptive_path.pdf, 8 May 2019.

[17] Birdeau, Lucas, et al. "Delivery of data and formatting information to allow client-side manipulation." U.S. Patent No. 8,136,109. 13 Mar. 2012.

[18] GitHub, “The State of the Octoverse, 2019”. Github.com Retrieved from https://octoverse.github.com/, 14 May 2020.

[19] Madnight, “GitHut 2.0, Pushes, 2020, Quarter 1”, Github.io. Retrieved from https://madnight.github.io/githut/#/pushes/2020/1, 14 May 2020.

[20] Thomas Elliott, “The State of the Octoverse: top programming languages of 2018”. The GitHub Blog, November 2018. Retrieved from https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/, 8 May 2019.

[21] Piotr Sroczkowski, “100 Most Popular Languages on GitHub in 2019”, Retrieved from https://brainhub.eu/blog/most-popular-languages-on-github/, 3 May 2019.

[22] Stack Overflow, “Developer Survey Results, 2018”. Stack Exchange, Inc. Retrieved from https://insights.stackoverflow.com/survey/2018#most-popular-technologies?utm_source=codecademyblog, 8 May 2019.

[23] Simply Technologies. “Why is JavaScript so popular?”. April 2018, Simply Technologies. Retrieved from https://www.simplytechnologies.net/blog/2018/4/11/why-is-javascript-so-popular, 8 May 2019.

[24] Michael Georgiou, “Why JavaScript Is and Will Continue to Be the First Choice of Programmers. DZone, November 2014. Retrieved from https://dzone.com/articles/why-javascript-and-will, 8 May 2019.

[25] NuSphere, “The History of PHP”, nusphere.com. Retrieved from http://www.nusphere.com/php/php_history.htm, 21 June 2019.

[26] Peter Wayner, “Node.js vs. PHP: An epic battle for developer mindshare”, Infoworld.com. Retrieved from https://www.infoworld.com/article/3166109/nodejs-vs-php-an-epic-battle-for-developer-mindshare.html, 21 June 2019.

[27] Educba, “PHP vs JavaScript”, educba.com. Retrieved from https://www.educba.com/php-vs-javascript/, 21 June 2019.

[28] TIOBE Company, “TIOBE Index (June 2019)”, tiobe.com. Retrieved from https://www.tiobe.com/tiobe-index/, 21 June 2019.

[29] Pierre Carbonnelle, “PYPL PopularitY of Programming Language”, Github.com. Retrieved from https://pypl.github.io/PYPL.html, 21 June 2019.

[30] World Wide Web Consortium, “Document Object Model”, 2005. Retrieved from https://www.w3.org/DOM/, 5 May 2019.

[31] Mozilla, “Introduction to the DOM”. MDN web docs, January 2020. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction, 1 February 2020.

[32] Jonathan Robie, Texcel Research, “What is the Document Object Model?”, w3.org. Retrieved from https://www.w3.org/TR/WD-DOM/introduction.html, 1 February 2020.

[33] w3schools.com, “JavaScript HTML DOM”, Retrieved from https://www.w3schools.com/js/js_htmldom.asp, 4 May 2019.

[34] Simon Sarris, “HTML5 Canvas vs. SVG vs. div”, stackoverflow.com, November 2015. Retrieved from https://stackoverflow.com/questions/5882716/html5-canvas-vs-svg-vs-div, 19 November 2019.

[35] yWorks, “SVG, Canvas, or WebGL? Visualization options for the web”, February 2018. Retrieved from https://www.yworks.com/blog/svg-canvas-webgl.html, 17 November 2019.

[36] Kirupa Chinnathambi, “DOM vs. Canvas”, kirupa.com, October 2015. Retrieved from https://www.kirupa.com/html5/dom_vs_canvas.htm, 17 November 2019.

[37] The jQuery Foundation, “What is jQuery”. The jQuery Foundation, 2019. Retrieved from https://jquery.com/, 9 May 2019.

[38] Khan Academy, “History break: How did John build jQuery”. Khan Academy, 2019. Retrieved from https://www.khanacademy.org/computing/computer-programming/html-js-jquery/jquery-dom-access/a/history-of-jquery, 9 May 2019.

[39] BuiltWith® Pty Ltd, “JavaScript Library Usage Distribution in the Top 1 Million Sites”, BuiltWith® Pty Ltd, May 2019. Retrieved from https://trends.builtwith.com/javascript/javascript-library/, 9 May 2019.

[40] W3Techs.com, “Usage of JavaScript libraries for websites”, W3Techs.com, May 2018. Retrieved from https://w3techs.com/technologies/overview/javascript_library/all, 9 May 2019.

[41] Lokesh Sardana, “Understanding DOM manipulation in Angular”, medium.com. Retrieved from https://medium.com/@sardanalokesh/understanding-dom-manipulation-in-angular-2b0016a4ee5d, 26 January 2020.

[42] Luke Joliat, “Do we still need JavaScript frameworks?”, freeCodeCamp. Retrieved from https://medium.freecodecamp.org/do-we-still-need-javascript-frameworks-42576735949b, 23 May 2019.

[43] Matt Warcholinski, “Top 10 Tools for JavaScript Development”, Brainhub. Retrieved from https://brainhub.eu/blog/top-tools-for-javascript-development/, 14 June 2019.

[44] Fowler, Martin, “Inversion of Control”, June 2005. Retrieved from https://martinfowler.com/bliki/InversionOfControl.html, 3 May 2019.

[45] Facebook, Inc., “Draft: JSX Specification”, GitHub. Retrieved from https://facebook.github.io/jsx/, 12 June 2019.

[46] Mozilla, “Web Components”, MDN Web Docs, November 2019. Retrieved from https://developer.mozilla.org/en-US/docs/Web/Web_Components, 11 March 2020.

[47] Leff, Avraham, and James T. Rayfield. "Web-application development using the model/view/controller design pattern." Proceedings fifth ieee international enterprise distributed object computing conference. IEEE, 2001. Retrieved from https://domino.watson.ibm.com/library/cyberdig.nsf/papers/696CFBA5D4B1E68985256A1E00626E27/$File/rc22002.pdf, 13 May 2020.

[48] Facebook, “flux-concepts”, github.com, 2019. Retrieved from https://github.com/facebook/flux/tree/master/examples/flux-concepts, 29 September 2019.

[49] Petar Vukasinovic, “Redux vs Vuex for state management in Vue.js”. Codementor.io. Retrieved from https://www.codementor.io/@petarvukasinovic/redux-vs-vuex-for-state-management-in-vue-js-n10yd7g2f, 13 May 2020.

[50] React Redux, “Official React bindings for Redux, 2019. Retrieved from https://react-redux.js.org/, 29 September 2019.

[51] VueJS, “What is Vuex?”, 2019. Retrieved from https://vuex.vuejs.org/, 29 September 2019.

[52] ngrx.io, “Getting started”, 2019. Retrieved from https://ngrx.io/guide/store, 29 September 2019.

[53] Priyesh Patel, “What exactly is Node.js”, FreeCodeCamp. Retrieved from https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/, 14 June 2019.

[54] Dmitry Garbar, “PHP vs Node.js”, Belitsoft.com. Retrieved from https://belitsoft.com/php-development-services/php7-vs-nodejs, 12 June 2019.

[55] ReactJs, “Create a New React App”, reactjs.org. Retrieved from https://reactjs.org/docs/create-a-new-react-app.html, 14 June 2019.

[56] npmjs.org, “Hello yarn”, October 2016. Retrieved from https://blog.npmjs.org/post/151660845210/hello-yarn, 9 October 2019.

[57] The npm Blog, “Introducing npx: a npm package runner”, npmjs.org. Retrieved from https://blog.npmjs.org/post/162869356040/introducing-npx-an-npm-package-runner, 14 June 2019.

[58] Gergely Nemeht, “Yarn vs npm - which Node package manager to use in 2018?”, 2018. Retrieved from https://blog.risingstack.com/yarn-vs-npm-node-js-package-managers/, 9 October 2019.

[59] Alberto Gimeno, “How JavaScript bundlers work”, Medium Corporation. Retrieved from https://medium.com/@gimenete/how-javascript-bundlers-work-1fc0d0caf2da, 14 June 2019.

[60] WebPack, “Concepts”, 2019. Retrieved from https://webpack.js.org/concepts/, 12 October 2019.

[61] Babel, “What is Babel?”, 2019. Retrieved from https://babeljs.io/docs/en/, 29 September 2019.

[62] Sacha Greif, Raphael Benitte & Michael Rambeau, “State of JavaScript survey, 2019”. Retrieved from https://2019.stateofjs.com/, 30 March 2020.

[63] Sacha Grief, “State of JavaScript 2019”, kaggle.com. Retrieved from https://www.kaggle.com/sachag/state-of-js-2019, 25 January 2020.

[64] “vuejs / vue”, Github.com, Janurary 2020. Retrieved from https://github.com/vuejs/vue/stargazers, 25 January 2020.

[65] “facebook / react”, Github.com, January 2020. Retrieved from https://github.com/facebook/react/stargazers, 25 January 2020.

[66] “angular / angular”, Github.com, January 2020. Retrieved from https://github.com/angular/angular/stargazers, 25 January 2020.

[67] “emberjs / ember.js”, Github.com, January 2020. Retrieved from https://github.com/emberjs/ember.js/stargazers/, 25 January 2020.

[68] “developit / preact”, Github.com, January 2020. Retrieved from https://github.com/preactjs/preact/stargazers, 25 January 2020.

[69] “sveltejs / svelte” Github.com, January 2020, Retrieved from https://github.com/sveltejs/svelte/stargazers, 25 January 2020.

[70] Sacha Greif, Raphael Benitte & Michael Rambeau, “State of JavaScript survey, 2018: Ember”. Retrieved fromhttps://2018.stateofjs.com/front-end-frameworks/ember/, 23 May 2019.

[71] Sacha Greif, Raphael Benitte & Michael Rambeau, “State of JavaScript survey, 2018: Vue.js”. Retrieved from https://2018.stateofjs.com/front-end-frameworks/react/, 23 May 2019.

[72] John Hannah, “The Ultimate Guide to JavaScript Frameworks”, jsreport.io, January 2018. Retrieved from https://jsreport.io/the-ultimate-guide-to-javascript-frameworks/, 16 May 2019.

[73] TechMagic, “ReactJS vs Angular5 vs Vue.js—What to choose in 2018?”, March 2018. Retrieved from https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d, 5 March 2019.

[74] Chris Dawson, “JavaScript’s History and How it Led To ReactJS” TheNewStack.io, June 2014. Retrieved from https://thenewstack.io/javascripts-history-and-how-it-led-to-reactjs/, 5 March 2019.

[75] Simply Technologies, “Let’s get clear about React Native”. April 2018, Simply Technologies. Retrieved from https://www.simplytechnologies.net/blog/2018/4/4/lets-get-clear-about-react-native, 8 May 2019.

[76] Facebook, “facebook / create-react-app”, GitHub.com. Retrieved from https://github.com/facebook/create-react-app#creating-an-app, 14 June 2019.

[77] ReactJs, “React Without JSX”, Reactjs.org. Retrieved from https://reactjs.org/docs/react-without-jsx.html, 12 June 2019.

[78] Vue.JS, “Comparison with Other Frameworks”, 2019. Retrieved from https://vuejs.org/v2/guide/comparison.html, 12 October 2019.

[79] Manjutah M, “AngularJS and Angular 2+: a Detailed Comparison”, SitePoint.com. Retrieved from https://www.sitepoint.com/angularjs-vs-angular/, 23 May 2019.

[80] Sacha Greif & Raphael Benitte, “State of JavaScript survey, 2019: Angular”. Retrieved from https://2019.stateofjs.com/front-end-frameworks/angular/, 25 January 2020.

[81] Angular, “Elements”, angular.io. Retrieved from https://angular.io/guide/elements, 8 March 2020.

[82] Sacha Greif, Raphael Benitte & Michael Rambeau, “State of JavaScript survey, 2018: Vue.js”. Retrieved from https://2018.stateofjs.com/front-end-frameworks/vuejs/, 22 May 2019.

[83] Vue.js, “Introduction”, vuejs.org. Retrieved from https://vuejs.org/v2/guide/, 30 March 2020.

[84] Svelte, “API Docs”, svelte.dev. Retrieved from https://svelte.dev/docs, 30 March 2020.

[85] Rich Harris, “Virtual DOM is pure overhead”, svelte.dev blog, December 2018. Retrieved from https://svelte.dev/blog/virtual-dom-is-pure-overhead, 8 March 2020.

[86] Shuhei Kagawa, et. al, “Loading Time Matters”, Zalando, June 2018. Retrieved from https://jobs.zalando.com/en/tech/blog/loading-time-matters/?gh_src=4n3gxh1, 12 May 2020.

[87] Mattias Levlin, “dom-benchmark-react”, Github. Retrieved from https://github.com/MattiasLevlin/dom-benchmark-react, 2 February 2020.

[88] Mattias Levlin, “dom-benchmark-vue”, Github. Retrieved from https://github.com/MattiasLevlin/dom-benchmark-vue, 2 February 2020.

[89] Mattias Levlin, “dom-benchmark-angular”, Github. Retrieved from https://github.com/MattiasLevlin/dom-benchmark-angular, 2 February 2020.

[90] Mattias Levlin, “dom-benchmark-svelte”, Github. Retrieved from https://github.com/MattiasLevlin/dom-benchmark-svelte, 2 February 2020.

[91] Reactjs.org, “React Docs”. Retrieved from https://reactjs.org/docs/, 3 May 2019.

[92] Angular, “Lifecycle hooks”, angular.io. Retrieved from https://angular.io/guide/lifecycle-hooks, 8 March 2020.

[93] MDN Web Docs, “performance.now()”, mozilla.org. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Performance/now, 5 April 2020.

[94] Bundlephobia, “react@16.12.0”. Retrieved from https://bundlephobia.com/result?p=react@16.12.0, 26 January 2020.

[95] Bundlephobia, “vue@2.6.11”. Retrieved from https://bundlephobia.com/result?p=vue@2.6.11, 26 January 2020.

[96] Bundlephobia, “@angular/core@8.2.14” Retrieved from https://bundlephobia.com/result?p=@angular/core@8.2.14, 26 January 2020.

[97] Bundlephobia, “svelte@3.20.0”. Retrieved from https://bundlephobia.com/result?p=svelte@3.20.0, 13 May 2020.

[98] ReactJs, “Languages”, reactjs.org. Retrieved from https://reactjs.org/languages, 12 June 2019.

[99] Vue.js, “The Progressive JavaScript Framework, vuejs.org. Retrieved from https://vuejs.org/index.html, 12 June 2019.

[100] Angular, “One framework. Mobile & desktop.”, Angular.io. Retrieved from https://angular.io/, 12 June 2019.

[101] Svelte, “The easiest way to get started”, svelte.dev, August 2017. Retrieved from https://svelte.dev/blog/the-easiest-way-to-get-started, 29 January 2020.

[102] Piero Borrelli, “Angular vs. React vs. Vue: A performance comparison” Logrocket.com. Retrieved from https://blog.logrocket.com/angular-vs-react-vs-vue-a-performance-comparison/, 13 May 2020.

[103] krausest, “js-framework-benchmark”, Github.com. Retrieved from https://github.com/krausest/js-framework-benchmark, 13 May 2020.