Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Eric Wohlgethan

Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Eric Wohlgethan

Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuende Prüferin: Prof. Dr. Ulrike Steffens
Zweitgutachter: Martin Behrmann geb. Knoblauch

Eingereicht am: 15. Mai 2018

Eric Wohlgethan

Thema der Arbeit

Entscheidungshilfe für die Webentwicklung anhand des Vergleichs von drei führenden JavaScript Frameworks: Angular, React and Vue.js

Stichworte

JavaScript, Angular, React, Vue.js, Frontend, Webentwicklung

Kurzzusammenfassung

Das Gebiet der Webentwicklung hat sich in den vergangenen Jahren stark verändert. Diese Thesis gibt einen Einblick in drei der aktuell populärsten JavaScript Frameworks: Angular, React und Vue.js. Dabei wird jedes einzelne auf Vor- und Nachteile anhand von vordefinierten Kriterien hin untersucht und bewertet. Abschließend wird eine Einschätzung abgegeben, welche Technologien für bestimmte Szenarien geeignet sind.

Eric Wohlgethan

Title of the paper

Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js

Keywords

JavaScript, Angular, React, Vue.js, frontend, web development

Abstract

The area of web development has changed a lot in the past years. This thesis provides an insight into the currently most popular JavaScript frameworks: Angular, React and Vue.js. Each one of them will be investigated and evaluated based on pre-defined criteria. Ultimately, a recommendation will be given on which technology is most appropriate for certain situations.

Contents

List of Tables

List of Figures

List of Code Examples

List of Abbreviations

API	Application Programming Interface.
BSD	Berkeley Source Distribution.
CDN	Content Delivery Network.
CLI	Command Line Interface.
CMS	Content Management System.
CSS	Cascading Style Sheets.
DI	Dependency Injection.
DOM	Document Object Model.
ES	ECMAScript.
GPL	GNU Public License.
HTML	HyperText Markup Language.
HTTP	HyperText Transfer Protocol.
IDE	Integrated Development Environment.
JS	JavaScript.
MIT	Massachusetts Institute of Technology.
MVC	Model-View-Controller.
MVVM	Model-View-Viewmodel.
NPM	Node Package Manager.
PWA	Progressive Web App.

SemVer	Semantic Versioning.
SPA	Single-Page Application.
TS	TypeScript.
URL	Uniform Resource Locator.

1 Introduction

> *'Most JavaScript developers have heard of or experienced JavaScript fatigue. JS fatigue is the overwhelming sense that we need to learn most of the hottest emerging technologies in order to do our jobs well. This is unattainable and the stress we feel to achieve it is unjustified; so how do we manage and combat JavaScript fatigue?'*[1]

History JavaScript started out as a small scripting language for the use in Netscape Communicator back in 1995 when the World Wide Web was still a fresh invention and the modern-day phenomenon of JavaScript fatigue was unfamiliar. The creators of the Communicator decided that the web had to become more dynamic which led to the development of the language Mocha by Brendan Eich[2]. Mocha is based on Scheme which is a dialect of Lisp, a functional programming language. The requirements for Mocha were manageable:

- Dynamic
- 'Easy-to-grasp' syntax
- Powerful

Shortly after its completion in May 1995, it was renamed to LiveScript for marketing purposes[3]. About six months later, a deal between Netscape Communications and Sun led to the final name which is globally known as JavaScript.

---

[1] Cf. Maida, *How to Manage JavaScript Fatigue*.
[2] Cf. Severance, *JavaScript: Designing a Language in 10 Days*.
[3] Cf. Peyrott, *A Brief History of JavaScript*.

Today More than 20 years have passed since then. Nowadays, JavaScript is the most widely used programming language in the world as recently shown in a study issued by the website Stack Overflow which is shown in figure 1.1 (for further information see section 6.3):

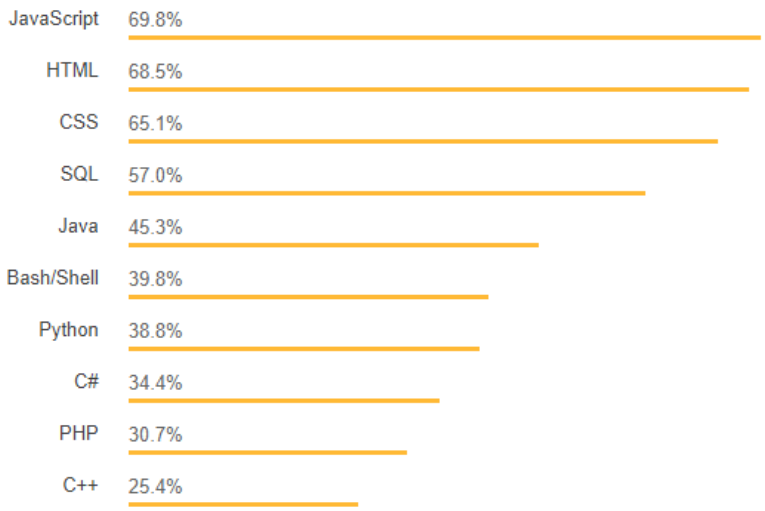

Figure 1.1: *Most popular technologies in 2018*

With this increase in use and popularity, it was inevitable that a large amount of tools and suchlike were published. They can be divided into different categories such as IDEs and editors, package managers, compilers, bundlers, libraries and frameworks.

While it would exceed the scope of this thesis to take a look at every single category, there are two that this thesis will cover in its course: Libraries and frameworks. Still, the other categories will appear in and contribute to the respective topics as additional information.

1.1 JavaScript Frameworks

> *"JavaScript UI frameworks and libraries work in cycles. Every six months or so, a new one pops up, claiming that it has revolutionized UI development. Thousands of developers adopt it into their new projects, blog posts are written, Stack Overflow questions are asked and answered, and then a newer (and even more revolutionary) framework pops up to*

usurp the throne.'[4]

The quote above verbalises exactly what JavaScript fatigue, which was mentioned at the very beginning of chapter 1, is about. The release cycle for (new) frameworks is relatively short which results in a large quantity of frameworks. Although the lack of a distinct definition complicates determining how many JavaScript frameworks exist in total, there are more than 50 listed on the website JSter[5] (only accounting for Model-View-Controller (MVC) frameworks). Most of them are not even developed by large firms like Google or Facebook but are community-driven, often by individuals.

1.1.1 Definition

As mentioned above, there is no distinct definition which criteria a software has to fulfill to be considered a framework. At times, frameworks are being confused with libraries. The following section will attempt to remove ambiguity.

Library A (third-party) library '[...] generally consists of pre-written code, classes, procedures, scripts, configuration data and more.'[6]. Mostly, it can be integrated in an existing project with ease and used to shorten development time. This is due to the fact that many issues concerning basic algorithms and functions have already been solved by another expert programmer in the community. When these experts decide to share their code as open-source (for more information see chapter 6), it spares time for the developer to focus on more business-related issues rather than fundamentals. But using a third-party library can also pose a potential risk to an application[7]. Developers should always check first who published the code and how safe it is to integrate. This can be done by checking download numbers and issues on Node Package Manager (NPM) or GitHub, for example.

[4] Cf. Allen, *The Brutal Lifecycle of JavaScript Frameworks.*
[5] http://jster.net/
[6] Cf. *Definition for software library.*
[7] Cf. Wisseman, *Third-party libraries are one of the most insecure parts of an application.*

However, it cannot be stressed enough that libraries only enhance a software in terms of specific functions but never cover the complete stack needed for development.

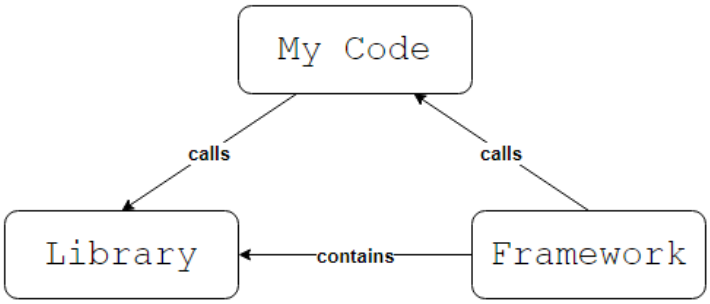

Figure 1.2: Differentiation of library and framework (own visualisation)

Framework In comparison to libraries, frameworks indeed offer a complete stack of helpful functions and take responsibility for decisions that otherwise the developer would have to make prior to actually writing the application's code. This includes strategies for in-app routing of URL paths, state management, bundling and others. Furthermore, frameworks provide workflow improvements which include best practices for basic development aspects like the overall structure of an application or generating boilerplate code (e.g. compare *Angular CLI* in section 3.2). Most frameworks in the MVC world are component-based. The thesis will cover how each of the discussed technologies handles this part. The logic behind the concept of components is important because they alone describe the user interface. When changes occur to the underlying data, the framework rerenders the complete UI component. The displayed data should be up-to-date at any point in time. This approach can be summarised as 'UI as function'.

It should be emphasised that libraries are being integrated and also used in frameworks. The visualisation[8] in figure 1.2 conflates the above statements: The code is called by the framework and it can make use of the integrated libraries. Meanwhile, the framework is also responsible for providing the libraries as it handles the logic of

[8] Based on this blog entry (https://medium.com/datafire-io/libraries-vs-frameworks-626cdde799a7, visited on 05/03/2018)

loading. This is known as Dependency Injection (DI) (for more information see section 3.2).

The 'obstinacy' of some frameworks, which seemed convenient at first, might yet result in problems in the course of the project: This includes decisions with respect to, e.g., state management, data binding or template handling. The selection of a framework should be elaborate. Therefore, a team or even a single developer has to evaluate the following question foresightful: Is the purpose or the extent of a framework adequate for the task?

1.1.2 Overview of Status Quo

According to Riehle[9], '[...] frameworks promise higher productivity and shorter time-to-market of application development through design and code reuse (than possible with non-framework based approaches).' But still, there are situations where utilising a framework might result in an overhead with regard to the development. For example, more static content can be displayed with a usual website or a Content Management System (CMS) like WordPress[10] or TYPO3[11].

In case the use of a framework is suitable for the project, a new problem arises. Considering the past years in the JavaScript world, the market for frameworks is highly competitive. However it has to be noted that competitive is rather meant in a quantitative than in a monetary way. This is due to the fact that almost all frameworks are open-source (for more information see Licensing in section 2.2.1) and therefore are not connected to direct costs. Figure 1.3 only shows an excerpt of frameworks and related technologies (compilers, bundlers, etc.) but foreshadows the wide range of the JavaScript ecosystem.

[9] Cf. Riehle, *Framework Design: A Role Modeling Approach*, page 1f.
[10] https://de.wordpress.com/
[11] https://typo3.org/

Figure 1.3: *JavaScript ecosystem*

Being a developer nowadays does not only include the knowledge required for the coding, but developers are increasingly forced to keep up-to-date with the current trends. With more and more software being realised with JavaScript - be it in frontend applications or in backend *Node.js*-based APIs - the job gets more complex along the way. It also complicates the decision making regarding which technology should be used in a project. Heads of IT are being confronted with a growing ecosystem that is hard to oversee. This aspect leads to the aim of this thesis formulated in the following chapter.

1.2 Aim of Thesis

This thesis aims to give an overview of the three technologies that currently have the highest traction in the market[12]: *Angular*, *React* and *Vue.js*. While this does not suffice for a thorough study of the whole JavaScript ecosystem, it does provide an indication to decision making as these three frameworks all follow different core principles. Ultimately, the following question should be answered: Does one of the

[12] Cf. *npm trends*.

aforementioned frameworks stand out so that it can be recommended for the majority of use cases in terms of web development?

In chapter 2, the criteria that will be used for the analysis of the respective framework will be presented. Also the process of how the criteria was chosen will be discussed.

The subsequent chapters deal with the earlier mentioned frameworks in no particular order. First *Angular* (chapter 3) then *React* (chapter 4) and after that *Vue.js* (chapter 5) are described and analysed. Each of these chapters first provide background information after which the general structure of the technology will be explained. The closing analysis presents an important part of this thesis as the frameworks are assessed based on the aforementioned criteria.

Chapter 6 contains the main comparison. While already various aspects are examined in the respective chapters for the frameworks, this chapter will discuss further similarities as well as differentiations between the technologies. Moreover, the influence and the standing of the open-source community and their related websites like GitHub and Stack Overflow will be shown.

Chapter 7 will summarise the result of the thesis. It will provide an elaborate answer to the question mentioned at the beginning of this section.

In the last chapter 8, an outlook for the upcoming years in the JavaScript scene will be provided. Also, possibilities for a future work will be discussed.

2 Framing the Comparison

2.1 Identification Stage

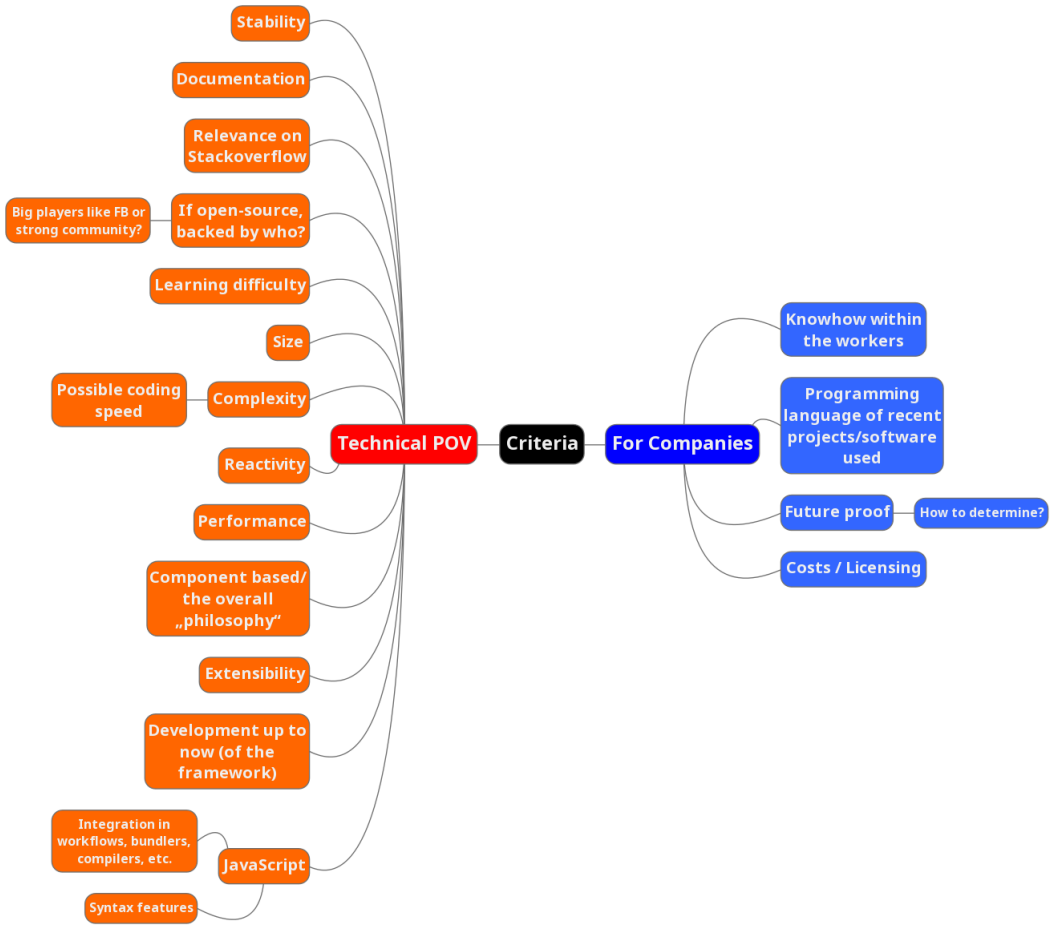

Figure 2.1: Mindmap of the criteria-related keywords (own visualisation)

As a first foundation for this thesis, a mindmap was created (see figure 2.1). This mindmap helped choosing the appropriate criteria to be used in the analysis later on. Every keyword in there is connected to an aspect of JavaScript frameworks or the ecosystem in general. To ease the handling, the information of the mindmap was condensed by grouping the keywords into their respective higher category. This resulted in three superior areas, each dealing with a completely different aspect of the analysis.

2.2 Criteria for Analysis

The following sections will present the aforementioned areas. Each area will be split up into several keywords including a reasoning for their involvement.

2.2.1 Stability

The first criterion for evaluating frameworks is summarised in the term *Stability*. It includes the following parts:

Versioning Version numbers can tell a lot about a software if assigned properly. Even with rather unusual concepts (see end of this paragraph), the most important factor in versioning is consistency. The users of software (or with regard to this thesis the developers who use third party libraries and frameworks) should be able to rely on new releases and plan ahead if there is a breaking change to come. This is part of why the project Semantic Versioning (SemVer) was introduced in 2011[1]. Currently (as of March 2018), SemVer is at version *2.0.0*. The main principles[2] of it are comprehensive and limited in number:

- First number: Major release ⇒ Breaking changes
- Middle number: Minor release ⇒ New features, no breaking changes
- Last number: Patch release ⇒ No new features, mostly bug-fixes

[1] https://github.com/semver/semver/tree/v1.0.0
[2] https://semver.org/spec/v2.0.0.html

The more developers follow these principles, the more reliable libraries and suchlike will get. As criterion for the analysis later on, this will be used as an indicator as to how stable the release policy of the respective developer is. However, versioning is sometimes very opinionated and more of a philosophy. SemVer is not the only possible approach to this topic. The opinions amongst developers are diversified[3].

Release Policy It can be determined how consistent new releases are both announced and then finally released. The analysis will include the historical releases, too. Furthermore, it should be evaluated how drastic breaking changes were and how the overall 'updatability' was perceived.

Maintainability The included frameworks in this thesis will, of course, be viewed from a technical stance. With regard to this, topics like structure, third party reliance and the file concept will be discussed. Maintainability plays an important role in the development as it can determine, e.g., how consistent project structures are or how efficiently already engineered components can be re-used in future works.

Licensing Most often, JavaScript frameworks are published under the MIT license that was originally developed at the renowned Massachusetts Institute of Technology. The Berkeley Source Distribution (BSD) license, which was developed by the Berkeley University, is resembling the MIT license in many parts[4]. Especially in terms of liberality, these two licenses are different to the popular GNU Public License (GPL), which is the most widely used open-source software license.

Licensing is a relevant term for software as it defines how an external piece of code may be implemented in and distributed with the company's own software. This is also connected to the aforementioned reliance on third party libraries.

[3] Based on this discussion on StackExchange (https://softwareengineering.stackexchange.com/questions/3199/what-version-naming-convention-do-you-use, visited on 27/03/2018)

[4] Cf. *Open-source licenses.*

2.2.2 Learning Curve

The second criterion for the evaluation is summarised in the term *Learning Curve*. It includes the following parts:

Available Documentation A well written and comprehensive documentation is often the key to the adoption of a library or framework. The more extensive, the better. The analysis will cover of what quality the provided documentation is and also if it is well locatable, structured and up-to-date. Moreover, it will deal with available tutorials and open source examples on, e.g., GitHub.

Knowhow Requirements This part determines how well the frameworks can be adopted. Depending on a developer's background, the initial time to learn and understand a certain technology can vary. In the world of web development, it is expected that everyone is familiar with at least the 'basics' namely:

- HTML ⇒ For structure
- CSS ⇒ For styling
- JavaScript ⇒ For logic

A thorough knowledge of these is essential. But still, for some frameworks this might not suffice. Some require or are based on a more specific language (e.g., TypeScript) or use special syntax enhancements (e.g., *Angular*).

Knowing the limitations and requirements of frameworks is crucial for a company in terms of recruiting.

Human Resources and Recruiting Depending on the knowhow requirements and the overall structure of a technology, companies should be aware of their own team's knowledge base. Also when acquiring new work forces, the amount of time a developer needs to get familiar with a project can vary. More opinionated frameworks can offer a more consistent project structure and best practices rather than liberal ones where strategic decisions can diverge heavily. This might result in a long adoption process. Again, all this depends on the size and longevity of a project or application.

2.2.3 JavaScript Integration

The third criterion for the evaluation is summarised in the term *JavaScript Integration*. It includes the following parts:

Best Practices and Stacks Some frameworks already offer a lot of tools and structures automatically, for example by using a Command Line Interface (CLI) for scaffolding the project structure. Others are more liberal: They allow the developer to decide upon routing, state management, compilers, bundling tools, linting and more. Also, the combination and compatibility of the aforementioned is relevant for efficient developing. With this freedom of choice, the responsibility also increases and so does the risk. Developers who can choose anything strategic on their own need a wholehearted knowledge of the processes and technologies involved. The analysis of the frameworks will cover how the technology behaves and what it provides. Furthermore, the advantages and disadvantages of the behaviour will be discussed respectively.

Development Languages Especially in the world of web development, many different programming languages can be used interchangeably. Often, the selected one is a personal preference of the developer. In regard to the observed frameworks, some have a 'main' development language that can also be counted towards the best practices. Others are more liberal in terms of choice. However, the amount of considerable programming languages is overseeable:

- Vanilla JavaScript (ES5/6/7)
- TypeScript
- JSX
- Dart

Syntax Features Depending on the main development language, different syntax features of JavaScript can be used. This is relevant in terms of browser support. Not all browsers (especially older versions of the Internet Explorer) support the latest features

of ECMAScript. ECMAScript is a standard published by the ECMA International[5] organisation. JavaScript is the most popular implementation of that standard. For example, if one decides to use features of ES6 it is recommended to use a compiler like Babel[6] to transform the code to ES5. This ensures a wide platform and browser compatibility.

TypeScript, for instance, is a superset of the ECMAScript standard. Its placement among the other mentioned versions can be seen in figure 2.2:

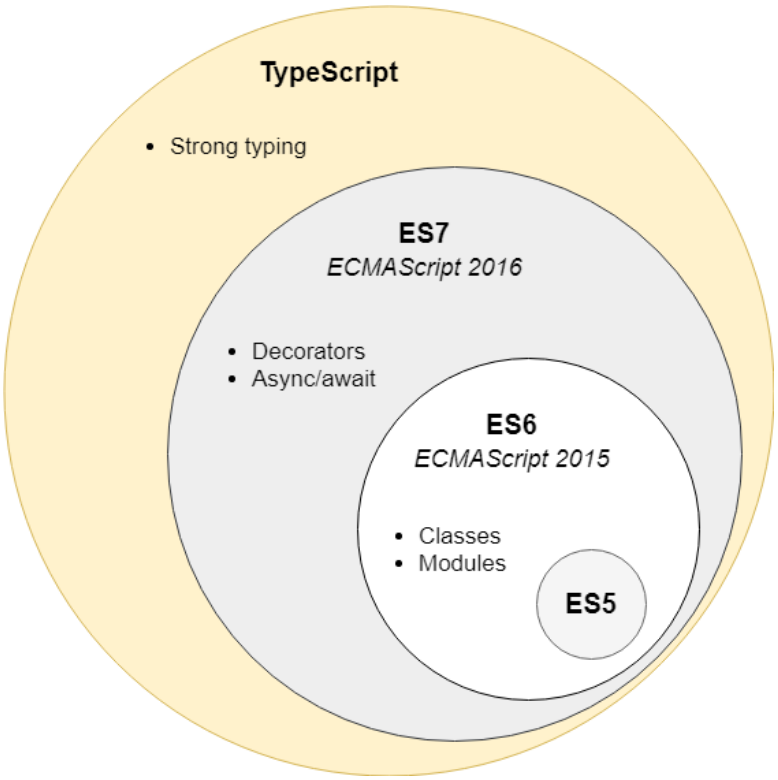

Figure 2.2: Placement of TypeScript (own visualisation)

The following three chapters will be addressed based on these criteria. The aggregate of all advantages and disadvantages will validate the standing of the respective framework.

[5] http://www.ecma-international.org/
[6] https://babeljs.io/

3 Angular

This chapter centers on the framework *Angular* and it contains three parts. The first deals with general information about the development history while the second is about the overall structure. The last part analyses the framework with regard to the criteria formulated in chapter 2.

3.1 Background Information

Angular was originally created by Google employees Misko Hevery and Adam Abrons in 2008[1]. Back then it was referenced to as *AngularJS* and developed in plain JavaScript. This was at a time when the majority of websites where based on the multi-page application approach: When a user clicked on a link, the browser had to retrieve the requested HTML document from the server. Depending on the internet connection and the responsiveness of the server, it could take a fair amount of time until the user could view the new page. Gradually user devices increased in overall performance so that application logic could be executed in the browser. This led to the approach of Single-Page Applications (SPAs).

AngularJS was one of the first frameworks for the development of SPAs. It was able to supersede *jQuery* by offering developers features like two-way data binding (for further information see Data Binding) and the possibility to organise modules for importing external scripts[2]. One of its main advantages over most of the competitors was its approachable nature. By simply inserting the CDN link into the HTML document and adding the `ng-app` directive to the `<body>` tag, the application was ready.

[1] Cf. Gudelli, *History Of AngularJS*.
[2] Cf. Metnew, *History of SPA frameworks: AngularJS 1.x and nostalgia.*

Furthermore, the documentation and tutorials provided by the developer team were very comprehensive and straight to the point.

In summer 2014, *Angular 2* was announced. *Angular 2* meant a complete re-write of the framework. Alongside this re-write, a lot of the core concepts of the framework changed as well. While *AnjularJS* was focused on scopes and controllers[3] as architecture pattern, *Angular 2* relies completely on a hierarchy of components.

TypeScript Another important novelty was the introduction of TypeScript (TS) as successor to JavaScript as the main development language. TypeScript is being transpiled into plain JavaScript code at compilation time. It is a superset of ECMAScript (ES)6 that was originally introduced and is still maintained by Microsoft. It is notable that TypeScript - as well as *Angular 2* and *AngularJS* - are open-source. One of the merits of using TypeScript is the ability to make use of advanced language features[4] of ES6:

- 'for...of'-loops
- Lambdas (more commonly known as arrow functions)
- Decorators

Especially Decorators play an important role in the Angular context (see code example 3.4). They are used to add meta information which is indicated with the `@` sign, e.g. `@Component` or `@Input`.

TypeScript has gained popularity because it adds a strong typing for the code. On one hand, a basic class definition in JavaScript would look similar to example 3.1.

```
1  var Dog = (function () {
2    function Dog(colour) {
3      this.furColour = colour;
4    }
5    Dog.prototype.bark = function () {
6      return "Wooof";
```

[3] https://docs.angularjs.org/guide/controller

[4] see *Official TypeScript Homepage*, State of the art JavaScript.

```
7     };
8     return Dog;
9   }());
```

Code Example 3.1: Basic JavaScript class

On the other hand, the same class written in TypeScript has a more definitive structure. The attributes are typed and TypeScript will also detect the `return` type of the `bark()` function as `string` (compare example 3.2).

```
1   class Dog {
2     furColour: string;
3     constructor (colour: string) {
4       this.furColour = colour;
5     }
6     bark() {
7       return "Wooof";
8     }
9   }
```

Code Example 3.2: Basic TypeScript class

As can be seen in code example 3.2, TypeScript establishes class-based, object-orientated programming. This opens the web development for developers that work mainly with C# or Java because TypeScript's structure is familiar to them[5].

Past Releases Alongside the release[6] of *Angular 2.0.0* in September 2016, the development team also proclaimed changes to the future release policy: They will be following SemVer and the release cycle is pre-determined for the upcoming years, releasing two major versions of the framework per year. More on this can be found in the Analysis section of this chapter.

Current Version As of March 2018, *Angular* is at version *5.2.7*.

[5] Cf. Boyer, *JavaScript - TypeScript: Making .NET Developers Comfortable with JavaScript*.
[6] Cf. *Versioning and Releasing Angular*.

3.2 Structure

Angular, like many other frameworks, is component-based. This means that components are the main building blocks. They can display information, render templates and perform actions on data. The best practice suggests that components consist of three separate files: A HTML file for template, a CSS file for styling and a TS file for controlling. By following this approach, a separation of concerns is implemented. Also, it adds to a more organised project structure and code.

Components are organised hierarchically: Information can flow between parent and child nodes, between two or more child nodes as well as between two or more completely decoupled ones. One special component is `app-root`. This represents the top level node of the component tree and this is the entry point where the framework initialises the application.

Component Interaction *Angular* has to handle the following situations in regard to internal communications:

- **Case 1**: From parent to child component[7]
- **Case 2**: From child to parent component[8]
- **Case 3**: Between two unrelated components[9]

While this does not cover all possible scenarios, it does cover the most common ones in terms of a developer's regular work. Also, the examples are simplified to present only the core concepts. This approach will be used throughout the thesis as a more (technical) in-depth approach would exceed the aim of this thesis.

Case 1 The parent passes the property `selectedDog` to the `dog` property of the child component which is enclosed in box brackets (compare example 3.3).

[7] Cf. *Master/Detail Components.*
[8] Cf. *Parent listens for child events.*
[9] Cf. *Parent and children communicate via a service.*

```
1  <dog-detail [dog]="selectedDog"></dog-detail>
```

Code Example 3.3: Parent component (case 1)

In the child component, there has to be an import from the `core` package first (see line 1 in example 3.4). After that, a property is declared using the imported `@Input` decorator.

```
1  import { Component, OnInit, Input } from '@angular/core';
2  [...]
3  @Input() dog: Dog;
```

Code Example 3.4: Child component (case 1)

Case 2 Children can pass events (i.e., data) to their parent component via `EventEmitter` and an imported `@Output` decorator (compare example 3.5).

```
1  import { Component, OnInit, Output } from '@angular/core';
2  [...]
3  @Output() onBark = new EventEmitter<boolean>();
```

Code Example 3.5: Child component (case 2)

The parent can react with the method `handleBarking(event)` upon the fired event `(onBark)` from the child as presented in the example 3.6.

```
1  <dog (onBark)="handleBarking(event)"></dog>
```

Code Example 3.6: Parent component (case 2)

Case 3 The last case contains the use of injectables to communicate between two unrelated components. This is one of the core concepts of *Angular*: Services handle this aspect of the application. Code example 3.7 shows the implementation of a simple service that manages an array of `Dogs`.

```
1  import { Injectable } from '@angular/core';
2  @Injectable()
3  export class DogService {
4    private dogs: Dog[];
5    setDogs(dogs: Dog[]): void {
6      this.dogs = dogs;
7    }
8    getDogs(): Dog[] {
9      return this.dogs;
10   }
11 }
```

Code Example 3.7: Simple service (case 3)

Components can interact with this service by importing it (see line 1 in example 3.8), instantiating it in their constructor (line 3) and finally using it by calling any the provided methods (line 5).

```
1  import { DogService } from 'dog.service';
2  [...]
3  constructor(private _dogService: DogService) {}
4  [...]
5  this._dogService.getDogs();
```

Code Example 3.8: Component using the service (case 3)

Data Binding The data binding within a component is also mentionable. Essentially it concerns the data interchange between the view (i.e. the HTML template) and the model (i.e., the TypeScript file). Again, there are three different types:

- **Property binding**:
  Data flow from the component to the template, i.e. [property].
- **Event binding**:
  Data flow from the template to the component, i.e. (event).

- **Two-way binding**:
  Combination of both aforementioned types, i.e. [(...)].

The internal application's communication as well as the data binding are fundamentals to understand when working with *Angular*. The following figure shows a summarised overview of the explained keywords:

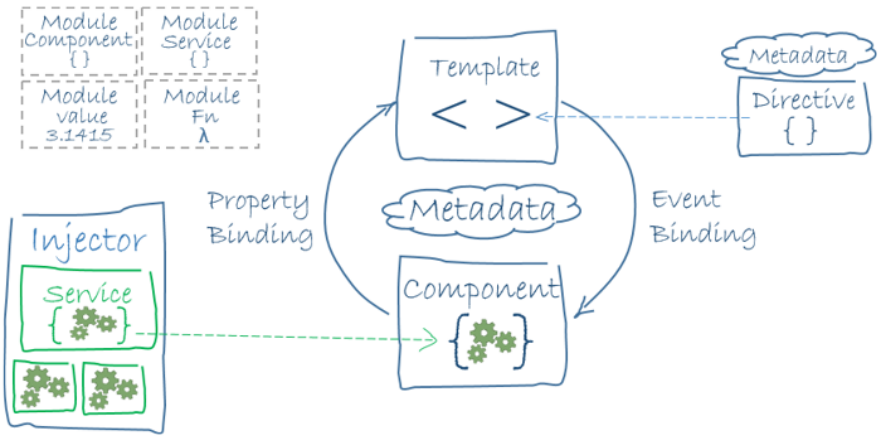

Figure 3.1: *Angular binding types*

Angular keeps track of changes by adding watchers internally to components. Moreover, it has an extensive lifecycle hook system implemented. This system runs permanently in the background and is quite complex (the related part[10] of the official documentation gives an impression of this). In many cases, components only implement the `ngOnInit` interface which is also a default for generating components with the CLI.

Command Line Interface The CLI is the 'backbone' of the framework. It serves as an entry point for almost every *Angular* application. It can be installed with NPM, for example: `npm install @angular/cli -g`. Creating a new project with the CLI is straight forward too and scaffolds an entire project structure including all relevant files: `ng new <my-application-name>`. The tool also offers a lot of useful commands to enrich the active developing process:

[10] Cf. *Lifecycle Hooks*.

- **ng serve**:
  Test the application locally on a simple, built-in webserver (including hot-reloading[11]).
- **ng build –prod/–dev**:
  Build the application with different environment variables.
- **ng generate service/component/module path/to/name**:
  Generate blueprints of *Angular* fundamentals.

Routing Routing plays an important role for the usability of a SPA. By defining routes in a separate file or module, *Angular* handles the logic of which component should be displayed depending on the currently active URL path. An example file would look like the following:

```
1  [...]
2  import { Routes, RouterModule } from '@angular/router';
3  import { HomeComponent } from './home.component';
4
5  const routes: Routes = [
6    { path: '', redirectTo: '/home', pathMatch: 'full' },
7    { path: 'home', component: HomeComponent }
8  ];
9  [...]
10 export class AppRoutingModule {}
```

Code Example 3.9: Routing module

The routing module handles an array of `Routes` which contain rules for different URL paths. Depending on the matching path, the defined component is loaded in the `<router-outlet></router-outlet>`. As stated in the documentation, '[t]he <router-outlet> tells the router where to display routed views.'[12] This tag can be placed

[11] The CLI detects saved changes in the files and re-compiles the application automatically.
[12] Cf. *RouterOutlet*.

anywhere in the application. Furthermore, the `RouterOutlet` can be marked with a `name` so that multiple tags can be addressed individually[13].

3.3 Analysis

The following section deals with various aspects that define the *Angular* framework. Both positive and negative points will be discussed and assessed. The criteria was defined in section 2.2.

Stability When looking at this criterion in regard to *Angular*, the first thing that has to be mentioned is the naming confusion beginning with the announcement of version 2.0.0[14]. While the original framework was called *AngularJS*, the re-write was suddenly called *Angular 2*. Many third-party libraries also started labelling their software with an `ng2` prefix (which caused problems in some cases which will be explained later). But with this announcement in September 2016, the team introduced a new versioning scheme and a release policy too: Versions will be released under the SemVer approach and there will be two major releases per year - one in March and the other one around September/October. SemVer as well as the new policy could be seen as plannable for companies that are interested in using the framework. Since then, the *Angular* team almost[15] kept to their promises as can be seen in table 3.2.

Version	Planned release	Actual release
2.0.0	September 2016	14 September 2016
4.0.0	March 2017	23 March 2017
5.0.0	September/October 2017	01 November 2017

[13] Cf. *Angular 2/4 Named Router Outlet*.
[14] Cf. *Versioning and Releasing Angular*.
[15] Cf. *Angular Version 5 Release was Delayed*.

Version	Planned release	Actual release
6.0.0	March 2018	not yet released

Table 3.2: Angular releases

However, it has to be criticised that the developer team made a jump and skipped the version number *3.0.0*. This is due to an inconsistency[16] at one point where the `@angular/core` package was at version *2.x.x* while the `@angular/router` package was already at version *3.x.x*. They unified this with the release of *Angular 4.0.0*.

One mentionable aspect is that *Angular 4* was backwards compatible with *Angular 2*. This made it easy for developers to update their projects. However, as it was mentioned above, there have been problems and confusions with third party libraries: Many packages from the NPM repository come with a prefix for the targeted software/framework. For example, a date picker for *Angular 2* would be called `ng2-datepicker`. With the step to version *4*, suddenly it was not obvious anymore if a certain package with `ng2` prefix would work in the new version despite the backward compatibility. This is one of the reasons why the *Angular* team proclaimed that new releases and the framework in general should be referred to as simply *Angular*[17]. Also the prefixes for packages should follow this logic, i.e. `ng`.

Since the introduction of SemVer, updating is mostly fluent. The developer team publishes information like release announcements and similar on their official blog which can be found under `https://blog.angular.io/`. They also provide an online update helper[18]. On this page, it can be stated which version is currently used and which version should be installed. Depending on this setting and a selection of the application's complexity, the tool gives a more or less detailed guide what to keep in mind when executing the update process. However, the differentiation between the three complexity levels 'basic, medium, advanced' is neither obvious nor described

[16] Cf. *Angular 4 and 5-6-7 Release Dates & Features.*
[17] Cf. *Angular Presskit: Brand Names.*
[18] https://angular-update-guide.firebaseapp.com/

on the page which might result in a bad user experience, especially for inexperienced developers.

Another important aspect in terms of stability is the driving force behind the framework. While many frameworks on GitHub and in the general open source world are published and maintained by individuals and/or the community, *Angular* is promoted and developed by Google. This is a huge benefit and some kind of an insurance for the users because Google itself plans to use the framework more widely in and for their own applications[19]. Even prestigious and highly valuable projects like the AdWords portal are being developed with *Angular*[20].

From a more technical point of view, *Angular* ensures a high maintainability by using components as main building blocks. Also, concepts like services and two-way data binding enforce a loose coupling between them. Furthermore, the multiple files approach results in a clear code structure and emphasises the framework's philosophy of separation of concerns. One point of criticism, though, is that the 'syntactic sugar' used in the HTML template can get complicated and hard to overlook.

```
1  <input type="text"
2         class="input-search"
3         placeholder="Last Name"
4         [(ngModel)]="name">
5  [...]
6  <button type="submit"
7          class="clear-input style-button-shadow"
8          (click)="clearForm()"
9          *ngIf="isFormValid">
10 </button>
```

Code Example 3.10: Simple form elements

In code example 3.10, there are elements of a form (`input` field and a submit `button`. Both hold a special syntax from *Angular*:

[19] Based on answer at Quora (https://www.quora.com/What-Google-products-make-use-of-AngularJS/answer/Aaron-Martin-Colby?srid=33NW, visited 16/03/2018)

[20] Cf. *The new AdWords UI uses Dart — we asked why.*

- `[(ngModel)]` is responsible for the two-way binding (for further information see Data Binding).
- With `*ngIf` the button's instantiation can be controlled via a boolean value.
- The `(click)` part is an event handler which calls a method of the component when the button was clicked.

While this example is rather simple, it becomes obvious that in more complex components the HTML is full of additional syntax. HTML tags can have various properties and directives. Based on this situation, *Angular* is often accused of '[...] continu[ing] to put JS into HTML'[21] which can be seen as one of their greatest weaknesses: If you want to learn *Angular* you are forced to learn its special syntax.

Another aspect of this analysis is the 'courage'[22] of *Angular* to be so heavily dependent on third party technologies at the core of the framework's functionality. First there is TypeScript: While it is being developed by Microsoft and it is not foreseeable to be dropped in near future[23], the risk of using it is relatively small. Secondly, there is RxJS. RxJS is '[...] a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code.'[24] *Angular* uses the concept of `Observables` especially in the context of HTTP requests and similar which poses a highly contemporary take on this topic. `Observables` are essentially the successor to the long-time used `Promises` with the main advantage that the data response flow can be handled and altered more efficiently and powerful in many different ways.

Learning Curve In comparison to other technologies, *Angular* has a rather steep learning curve. Although not right from the start. Setting up an initial project with the CLI only takes a few minutes (assuming that the prerequisites are met including an installed version of Node.js and NPM). From there on, new learners can take the

[21] Cf. House, *Angular 2 versus React: There Will Be Blood*.
[22] Based on this blog entry (https://medium.com/@cyrilletuzi/i-disagree-with-all-you-mention-da02cb783040, visited 16/03/2018)
[23] On the contrary: According to Chand, who is a Regional Director at Microsoft, the team behind TS '[...] has even bigger plans [for it]'. (https://www.c-sharpcorner.com/article/angular-2-or-react-for-decision-makers/, visited on 16/03/2018)
[24] http://reactivex.io/rxjs/

tutorial on the official *Angular* homepage[25]. While the course covers lots of the most common aspects, such as master/detail components, services, routing and HTTP, in real world applications this knowledge will not suffice. Shortly, topics like route guards, pipes, lifecycle hooks and several other start to surface. It has to be mentioned that the documentation[26] is extensive as well as comprehensive.

As *Angular* is a complete framework it provides a full set of homogeneous APIs. It takes time to gain an overview of all possibilities that are offered. The framework predetermines many decisions for the developer on how to handle certain situations. This may be seen as limiting for some developers. 'Opinionated' is a keyword that is often used to describe the nature of 'thinking in *Angular*'. However, one could argue that especially this argument poses an advantage for, e.g., companies that hire new developers. In the *Angular* world, a new coworker, who has worked on *Angular* projects before, will get familiar with the code base of the company relatively fast as almost all *Angular* project structures are comparable. This is due to the majority of developers using the scaffolding functionality of the CLI. Furthermore, developers with a background in object-orientated programming will find TypeScript to be close to, e.g., Java or C# and easy to access.

JavaScript Integration By using TypeScript as the main development language, advanced features of ES6 can be utilised. This might be one of the reasons why TypeScript was among the top five most loved languages in the 2018 edition of Stackoverflow's survey[27]. Although it has to be mentioned that *Angular* can be used with Vanilla JavaScript[28] and Dart[29], too. Using TypeScript also means that web developers with a background of only the basics (e.g., HTML, CSS and JS) may take longer to get acquainted with the framework.

As mentioned earlier, *Angular* is an 'opinionated' framework. This continues in terms of how it handles the static module bundling: Webpack[30] is used for this purpose. Webpack is a module bundler released under the MIT license and widely backed in

[25] https://angular.io/tutorial
[26] https://angular.io/docs
[27] https://insights.stackoverflow.com/survey/2018#most-loved-dreaded-and-wanted
[28] Cf. Johnson, *How to do Everything in Angular 2 using vanilla ES5 or ES6.*
[29] https://webdev.dartlang.org/angular/
[30] https://webpack.js.org/

the community. Currently (as of March 2018) it is at version *4.1.1*. One of its main advantages is that developers may write their own loaders to define what should be included in the bundling process.

It has to be mentioned that in the course of the early *Angular 2.x.x* releases, the developer team first used SystemJS[31] internally but later switched to Webpack[32]. This caused some confusion back then, because the change resulted in inconsistent tutorials and third party libraries' documentations. Again this posed a problem especially to inexperienced developers.

So far for the analysis of *Angular*. In chapter 6, the framework will be compared to the other two frameworks that are contained in this thesis. Furthermore, numbers from the open source community will be presented later on including popularity indices and download statistics.

[31] https://github.com/systemjs/systemjs
[32] Cf. Jorge, *Angular 2 CLI moves from SystemJS to Webpack*.

4 React

This chapter centers on the framework *React* and it contains three parts. The first deals with general information about the development history while the second is about the overall structure. The last part analyses the framework with regard to the criteria formulated in chapter 2.

Preface Before getting to the actual chapter, it has to be mentioned that the information and, especially, the analysis including all code examples are based on version *15.4.2* of *React*. This is due to the circumstance that when the research for this thesis started, the above mentioned version was the current. But in the course of research, version *16* was released which meant a re-write of the framework. Section 4.4 will cover the changes in more detail. The following chapter will therefore stick with the prior major version as documentation and other resources were scarce for the newly released one. Also for the sake of the comparative nature of this thesis, *React* will often be referred to as framework. Actually, *React* is only a view library which will be explained later on.

4.1 Background Information

React[1] is a JavaScript library developed by Facebook which can be used to create user interfaces for the web. It has been published as open source software in 2013 and has gained a lot of traction in the developers world since then. Some of the most popular use cases of *React* include Instagram and WhatsApp[2]. While *React* is often connected to web development, its core (the `react` package[3]) is a standalone library and can be

[1] https://reactjs.org/
[2] Cf. Warcholinski, *10 Famous Apps Using ReactJS Nowadays*.
[3] https://reactjs.org/docs/cdn-links.html

used in a variety of scenarios including native applications (iOS and Android). Only in combination with the `react-dom`, UIs for the web can be developed. JavaScript is an inherent part of the framework because its the main development language[4].

Due to *React* being only responsible for the view part of an application, the development process requires a stack of various technologies to be effective:

- Compiler (for JSX[5])
- Modules (and an appropriate loader) for the application structure
- Build process
- Routing
- State management

As a compiler, Babel is the most often recommended tool, e.g. by the Facebook developers themselves[6]. By using a compiler, new language features of not yet implemented ECMAScript specifications can be used in a today's application. When analysing a comparative table for browser support of different vendors, it can be observed that ES5, for instance, is widely accepted[7] with 100% of its features in almost all major desktop browsers (i.e., Internet Explorer, Edge, Mozilla Firefox, Google Chrome). ES6 gets close but fails to reach 100% compatability with any of the afore-mentioned browsers[8]. These numbers should always be viewed parallel to the browser usage, as can be seen in figure 4.1.

[4] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 3.
[5] https://reactjs.org/docs/introducing-jsx.html
[6] https://reactjs.org/docs/add-react-to-an-existing-app.html
[7] http://kangax.github.io/compat-table/es5/
[8] http://kangax.github.io/compat-table/es6/

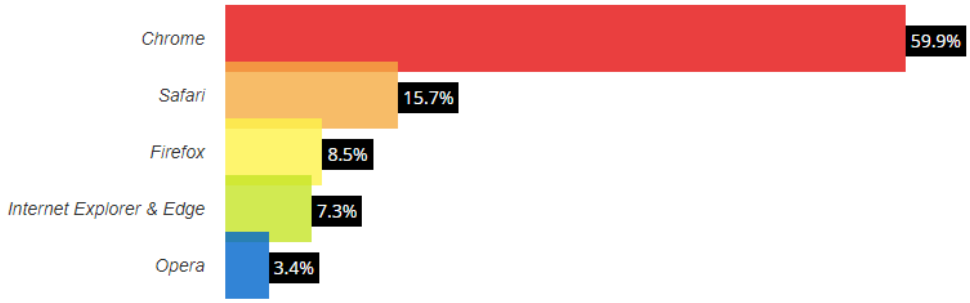

Figure 4.1: *Browser usage in February 2018*

In regard to modules and bundling, Webpack is one of the best solutions. One special merit is hot reloading which is an extremely important feature for an optimised developing process. Options for routing and state management can be found in the following section.

Current Version As of March 2018, *React* is at version *16.2.0*. But as mentioned in the preface of this chapter, the analysis is based on *React 15.4.2*.

4.2 Structure

The core of *React* is made of components and their composition. The overall goal is to transform a certain state of the application to a view which can be displayed in the browser. It is possible to write components with two different approaches[9]: Components as functions and components as ES6 classes.

Components as Functions Here, there are pure functions that return exactly one `ReactElement`. The name of the component is as well the name of the function. This approach, however, has its limitations: Neither can the state be altered nor can lifecycle methods (e.g., `componentDidMount`) be used.

Components as ES6 Classes The name of the component is represented by the name of the class. The class always has to extend the super class `React.Component`.

[9] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 80ff.

Furthermore, the visualisation logic should be contained in a `render()` method which as well should only return one root element. The aforementioned limitations of the first approach do not apply here: Classes can utilise states, lifecycle hooks and more.

In general, the *React* developers recommend using the function components as much as possible. That should support an efficient re-usability of simple but often-needed components. In practice, class components are placed at a high position in the hierarchical tree of the application. They work as container components and handle the various states that are then delegated to the lower level function components.

Props *React* components can be configured externally with properties and have a mutable state internally[10]. However, properties inside a component are immutable but can be set from an external point of the application at all times. Code example 4.1 shows the usage of `props`.

```
1  class Animal extends React.Component {
2    constructor(props) {
3      super(props);
4    }
5
6    render() {
7      const {type, name, legs} = this.props;
8
9      return (
10     <div className="animal">
11       <h1>Hello {name}</h1>
12       <p>He is a {type} and has {legs} legs</p>
13     </div>
14     );
15   }
16 }
17 <Animal type='dog' name='Rufus' legs={4} />
```

Code Example 4.1: Props provided for and used by component

[10] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 22.

The code example 4.1 contains the information about an `Animal` component. All class components have to extend the `React.Component` base class (see line 1). This concept will be explained further in later sections. Lines 2-4 contain the `constructor` where the properties are registered. The only method in this component is the `render` method which is mandatory. Also, every `render` method should return exactly one element[11]. This is the reason why the `h1` and the `p` tags have to be enclosed with a `div` tag (see lines 10-13). In line 17, the component is being initialised with the specific `props`. Furthermore, this example shows that *React* uses one-way data binding to forward information to the display unit by updating the model first and then rendering the UI element (more on this topic can be read in section 6.1).

State The `state` is responsible for the data management within a component. An example can be seen in 4.2.

```
1  class Dog extends React.Component {
2    constructor() {
3      super();
4      this.state = {
5        mood: null,
6        hunger: false
7      }
8    }
9    [...]
10   if(this.state.hunger == true) {
11     this.setState({
12       mood: angry
13     });
14   }
15 }
```

Code Example 4.2: Initial state and setState()

Here, the `Dog` component contains an initial state (lines 4-7) which is integrated in the `constructor`. The `state` properties can be accessed in a familiar behaviour.

[11] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 6.

The one speciality in the above example is the function `setState(...)` (see line 11) which is called in the `if` statement block. It has to be noted that this function is a core principle of *React*. When this function is executed, a re-rendering of the UI component is triggered (for more detailed information see ReactDOM.render()). Furthermore, the update of the `state` does not have to alter each property: Partial updates are possible (line 12). However, it has to be noted that the `setState(...)` function is asynchronous which means that the updated values are only available as soon as the re-rendering is completed[12]. This forces the developer to design the usage very carefully.

JSX As the previous code examples show, *React* does not use a specific template language but the JavaScript code can be enriched with HTML snippets. This extension is called JSX which roughly translates to 'Extensive JavaScript'. When using this approach, a compiler is required to transform the JSX code to plain JavaScript. Code example 4.3 shows a way where the `const` is neither just a `string` nor just HTML but a combination of both.

```
1  const element = <h1>Hello, world!</h1>;
```

Code Example 4.3: Assigning HTML to a JS element

ReactDOM.render() This is one of the core functions of the framework. It takes a `ReactElement` as first parameter (see line 2 in example 4.4) and mounts it behind an element from the native DOM which is provided in the second parameter (line 3).

```
1  ReactDOM.render(
2    <Animal type='dog' name='Rufus' legs={4} />,
3    document.getElementById('root')
4  );
```

Code Example 4.4: ReactDOM.render(): Code

This process is visualised in figure 4.2.

[12] Cf. Carnecky, *Beware: React setState is asynchronous!*

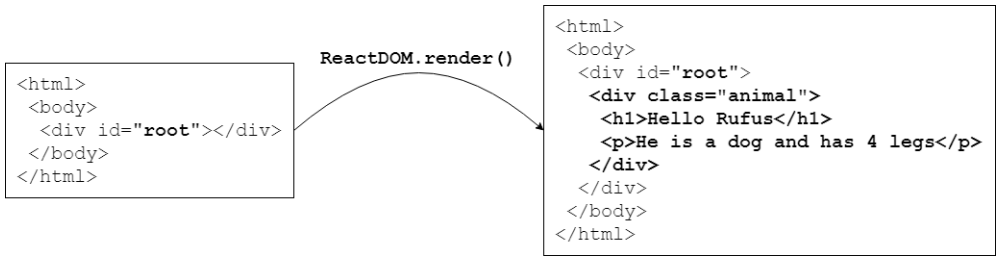

Figure 4.2: ReactDOM.render() (own visualisation)

After an alteration of the state, e.g. when the user enters a character in an `input` field, or of the properties, *React* re-renders the complete UI of the component. However, these changes are not applied to the native DOM. They are applied to what is called a 'Virtual DOM' which is a representation of the entirety of all `ReactElements`[13]. To avoid building the whole DOM again, the framework compares the new virtual DOM to an earlier snapshot. Based on the deviations, *React* generates operations that are being executed on the native DOM. This results in a minimal stress on the native DOM which is the reason for *React*'s swiftness because operations on the native DOM are expensive[14].

As mentioned before, *React* on its own only handles the 'V' part of MVC. To develop a complete application, more technologies are needed. As the framework's approach is quite liberal in terms of the decision responsibility of the developer, various different libraries can be used interchangeably. But as for most parts of the JavaScript ecosystem, best practices and stacks have emerged. Two of these, routing and state management, are mentioned in the following section.

Routing Routing can be handled with the *React Router*[15]. Although *React* itself was not influenced by the Ember framework[16], the *React Router* is based on the same principles as the Ember-Router[17]. The configuration is straight forward as can be seen in the code example 4.5.

[13] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 60.
[14] Based on paragraph 'DOM inefficiency' from this blog entry (https://medium.com/@hidace/understanding-reacts-virtual-dom-vs-the-real-dom-68ae29039951, visited on 21/03/2018)
[15] https://github.com/ReactTraining/react-router
[16] https://www.emberjs.com/
[17] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 183.

```
1  [...]
2  import {Router, Route, Redirect} from 'react-router';
3  import createHashHistory from 'history/lib/createHashHistory';
4  import {Home, About} from './components';
5  [...]
6  const router = <Router history={history}>
7    <Redirect from='/' to='/home' />
8    <Route path='/home' component={Home} />
9    <Route path='/about' component={About} />
10 </Router>
```

Code Example 4.5: Router setup

Three imports from the `react-router` module are necessary alongside a `router` element which contains the definitions of the routing logic, an element to store the page history (`createHashHistory`) and, of course, the components to be displayed (line 4). The `Route`, for instance, takes a `path` and assigns a component to it (lines 5 and 6).

State Management This aspect of developing an application increases in significance the more a project grows in size and complexity. With the idea of one centralised state logic (based on the architectural pattern call Flux[18]), *React* applications often use Redux[19]. According to their homepage, Redux is '[...] a predictable state container for JavaScript apps.' The processing of data works solely via pure functions (also known as Reducers). This approach is based on the programming language Elm[20].

4.3 Analysis

The following section deals with various aspects that define the *React* framework. Both positive and negative points will be discussed and assessed. The criteria was defined in section 2.2.

[18] https://facebook.github.io/flux/
[19] https://redux.js.org/
[20] http://elm-lang.org/

Stability With the release of *React 15*, *React* changed its versioning scheme[21]. While the prior versions were called *0.14* and *0.13*, the developer team adjusted to Semantic Versioning. It has always been a misconception with the release numbers as a version starting with *0.x.x* is often treated as non-stable or non-production-ready. In terms of *React*, this does not hold true. The framework has been stable since its initial release in 2013[22]. One big point of criticism and adaption blocker has always been the BSD-derived licensing of *React*. Facebook has often been criticised for releasing the library under this kind of license because it contains some questionable passages[23]. More on this topic can be read in section 4.4.

However, one disadvantage might be that there is no distinct release version for the whole *React* framework as most libraries involved are loosely coupled. This results in a constant struggle for developers to cross-check for dependency issues throughout the project. For example, for implementing a routing structure into an application there have to be two separate modules: `react-router` and `history`. The latter is important to enable back and forth clicking of pages which is definitely a wanted functionality.

For one thing *React* has an overseeable API and for another thing changes in the API are being introduced carefully. Furthermore, incompatible or even breaking changes are communicated beforehand on a reliable basis and APIs are just abolished when they have a surrogate which results in a robust API[24]. The architecture behind *React*, where the application state is centralised and data flow is only one-directional, caters for even larger projects beeing still readable and understandable[25].

In regard to updatability, several sources state that there have not been larger problems[26]. This is most probably connected to the aforementioned stability of the official API. Migrations should also be simple. It has to be noted, however, that an eventual strong dependency of third party libraries from smaller development teams

[21] https://reactjs.org/blog/2016/02/19/new-versioning-scheme.html
[22] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 4.
[23] Cf. Jorge, *Your license to use React.js can be revoked if you compete with Facebook.*
[24] https://reactjs.org/docs/design-principles.html#stability
[25] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 7.
[26] Based on comments (https://www.reddit.com/r/reactjs/comments/5a45ai/is_react_a_good_choice_for_a_stable_longterm_app/?st=jesl2tg7&sh=fd94a4c0, visited on 29/03/2018)

might hinder the update process because they can not ensure a fast compatibility adjustment of their software.

One merit in terms of future release and bug fixing reliability is the fact that Facebook is responsible for *React*. Similar to *Angular*, where Google is the driving force, Facebook pushes the use of its own software by implementing it in their own applications as well (for examples see Background Information).

The overall maintainability of *React* is given because of its use of components, although this does not need to be valued too much as most of the other major frameworks follow a similar approach. What indeed does need to be mentioned is that *React* excels at handling 'dumb' components. This is due to the core principles of `state` and `props` (for further information see section 4.2). Furthermore, the one-way binding approach of *React* results in a better data overview because data only flows in one direction which also positively affects the debugging capability.

Learning Curve Due to using the virtual DOM approach, *React* is not only fast but also offers a certain simplicity to the development process once the main principles of the framework are understood. Developers do not have to deal with how the UI transforms from one state to another one but rather the developer can describe the anticipated state and *React* handles the steps that are needed to get there. This facilitates both the developing and testing of an application[27].

Developers who attempt to learn *React* do need to have a profound knowledge of JavaScript because it is the main programming language for this framework. One could say that *React* entails more JavaScript. This might become relevant when a company has designers that work close to the code. In the real world, it could eventually be hard to find designers that know how to modify JSX code in order to change structural or styling related parts. For this task working with HTML templates would be much easier. In a way, *React* breaks with long-term best practices: E.g., developers were trying to separate UI templates and inline JS logic but the usage of JSX causes them to be intermixed again[28]. Although this can be seen both positive and negative. As stated in a talk by Pete Hunt in 2013, building components instead of templates results

[27] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 24.
[28] Cf. Neuhaus, *Angular vs. React vs. Vue: A 2017 comparison.*

in a better reusability and testability[29]. However, when switching jobs or projects, developers might need more time to adjust as *React* projects structures have a high probability of being different.

The 'anti-opinionation' of *React* requires development teams to have experienced leaders who can design a robust and well-proven setup for the application. The flexibility of *React* starts with the simplicity of just adding the `react.js` and `react-dom.js` to a base HTML file and the minimum setup is done. It goes even further with being able to reduce a certain SPA down to micro-services as an act of reusability or project re-focusing. This absence of rules and limited guidance can, however, pose a potential risk to the successful outcome of an undertaking.

The official documentation[30] is comparably small but nevertheless good as a starting point. It has to be noted that the documentation in question offers a large collection of various examples and code snippets to the different areas of *React*. In addition to that, the complete API is displayed too (including more specific examples)[31].

JavaScript Integration *React* focuses on the use of JavaScript with ES5 and ES6. This leads to an inconsistency in some tutorials and similar online resources as it sometimes is not obvious to which version of ECMAScript the code snippets refer to. For advanced and recent syntax features, at least ES6 or ES8 is needed. To compile the code to a browser compatible state, a compiler like Babel has to be used.

As this framework is very open to developer decisions, the amount of different stacks and best practices can not be covered in this thesis. However, one of the most popular stacks for a project contains: *React Router* and *Redux*. As mentioned before, *Redux* is a predictable state container for the use with *React* and it is popular among the community. However, it is not the only implementation of Flux: There are at least 16 different packages for state management to choose from in *React* alone[32]. It can be hard to determine what the differences or merits of each of them are. Also, the usage of *Redux* adds a lot of complexity to the code and project structure. Because of that, it might not be needed in the first place as even its creator, Dan Abramov, stated in

[29] Cf. Hunt, *React: Rethinking best practices*.
[30] https://reactjs.org/docs/
[31] https://reactjs.org/docs/react-api.html
[32] https://github.com/voronianski/flux-comparison

a comment on Medium[33]. This constitutes a difficulty especially for inexperienced developers at what point in time or to what extent of complexity *Redux* is feasible.

To start of a project with pre-configured libraries and similar, the NPM package `create-react-app` has to be mentioned which scaffolds a proven file structure and creates a build pipeline[34]. However, this package cannot be treated as a fully fledged CLI because it only facilitates the initial setup but offers no functionality regarding pre-configured boilerplate code.

4.4 React 16

Also referred to as *React Fiber*, *React 16* is a complete rewrite of the framework and it was released in September 2017[35]. According to their announcement, upgrading from 15 to 16 will work flawlessly, stating: 'With minor exceptions, if your app runs in 15.6 without any warnings, it should work in 16.' Also, breaking changes should only occur in a small number of uncommon cases. Dan Abramov also stated that '[React 16] is quite literally 99,9% [backward compatible].'[36] The main features of version *16.0* consist of:

- New core architecture (codename *Fiber*)
- New render return types: Fragments and strings
- Reduced file size
- Support for custom DOM attributes

The release in terms of the licensing follows a great discussion and backlash[37] among the developer community. When initially released in 2013, Facebook published *React*

[33] Based on this comment (https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367, visited on 29/03/2018)
[34] https://reactjs.org/docs/add-react-to-a-new-app.html#create-react-app
[35] https://reactjs.org/blog/2017/09/26/react-v16.0.html
[36] Based on this blog entry (https://medium.com/@dan_abramov/hey-thanks-for-feedback-bf9502689ca4, visited 29/03/2018)
[37] Cf. Hughes, *Facebook re-licenses React under MIT license after developer backlash.*

under the Apache V2 OSS license[38]. Later, in 2014, the license was switched to a BSD one. While these two licenses do not differ too much in terms of permission and general idea, it was hard to comprehend Facebook's motives. Finally after years of backlash and criticism of the community, Facebook decided to switch the license once again to the MIT license.

So far for the analysis of *React*. In chapter 6, the framework will be compared to the other two frameworks that are contained in this thesis. Furthermore, numbers from the open source community will be presented later on including popularity indices and download statistics.

[38] https://www.apache.org/licenses/LICENSE-2.0

5 Vue.js

This chapter centers on the framework *Vue.js* and it contains three parts. The first deals with general information about the development history while the second is about the overall structure. The last part analyses the framework with regard to the criteria formulated in chapter 2.

Preface Similar to *React*, *Vue.js* will often be referred to as a framework. Again, it actually is only the view part of a complete application. The following sections will show which technology stacks are needed in order to accompany *Vue.js* properly.

5.1 Background Information

Vue.js (in the following short: *Vue*) can be considered the newest framework in this comparison although its initial release happenend two years before *Angular 2*. But as the latter was built on various concepts of the first iteration (i.e., *AngularJS*), one could argue that *Vue* indeed is the most recent in terms of currentness. It was released in 2014[1] by Evan You who is a former employee of, e.g., Google where he worked a lot with *AngularJS*. However, the *1.0.0* version only arrived in October 2015[2].

Vue is often described as a progressive framework which can be used to build user interfaces for the web. While it is not strictly associated with the Model-View-Viewmodel (MVVM) pattern, the design principles of *Vue* were partly inspired by it. According to their website[3], the framework can be used both for small projects where the core library is used among other technologies and for full-blown SPAs.

[1] http://blog.evanyou.me/2014/02/11/first-week-of-launching-an-oss-project/
[2] https://vuejs.org/2015/10/26/1.0.0-release/
[3] https://vuejs.org/

The scalability is one of the main merits amongst others that will be discussed in the analysis later on (see section 5.3).

One specialty about *Vue* is that it is entirely developed by the open source community and not a large enterprise. It started out as a hobby project by You until he decided to quit his job and work full-time on it. In this context it has to be noted that the financing was completely realised with the help of Patreon[4]. Patreon is a community hub for content creators in various fields (e.g., game development, music, writing, photography and science) where people can become a subscriber for a specific project that they want to support with a monthly payment. After a short period of time, the supporters of the *Vue* project already contributed more than 4,000$ in total per month. As of April 2018, this figure has increased close to 15,000$[5].

This rise in popularity can be understood by looking at other related community pages: On GitHub[6], an open source platform for hosting code in repositories to share with other developers, success and popularity is displayed by the number of stars a repository or project has. *Vue* has gained more than 40,000 stars in the course of 2017 making it the highest rising project overall[7]. Part of this derives from the strong bonding and extensive support from the PHP community, especially Laravel[8], where *Vue* is used as the default view engine.

Past Releases The earliest releases in 2014 and 2015 were no major versions as they were all labelled with *0.x.x*. *Vue* started to gain traction after its *1.0* release in October 2015. This marked the point of production readiness which made the framework more interesting for larger companies.

Current Version As of April 2018, *Vue* is at version *2.5.16*.

[4] https://www.patreon.com/
[5] https://www.patreon.com/evanyou
[6] https://github.com/
[7] Cf. Motroc, *The rising star of JavaScript: Vue.js takes control of the game.*
[8] https://laravel.com/

5.2 Structure

Vue is a component based framework. The code example 5.1 shows a simple implementation of such a component.

```
1  Vue.component('dog-info', {
2    data: function () {
3      return {
4        legs: 4
5      }
6    },
7    template: '<h1>A dog has {{ legs }} legs.</h1>'
8  })
```

Code Example 5.1: Simple component in Vue

As can be seen in the above example 5.1, the `data` was not provided directly whereas in code example 5.10 it is provided directly as an object. Instead, the `legs` property is part of the `data` function. This is needed because each instance of a component has to manage an independent copy of information. If this rule would not exist, components would alter the `data` of other instances. The reason why the first example does not have to obey this rule is because it does not deal with being a component but with being an instance. Furthermore, all *Vue* components are also instances which means that they accept the same options object[9]. On creation of a *Vue* instance, all properties that can be found in its data object are added to the frameworks underlying reactivity system.

It has to be mentioned that *Vue* components are very similar to Custom Elements which are part of the Web Components Spec[10]). This relatively new approach to web development allows developers to create new HTML tags or modify existing ones with additional functionality only using vanilla JavaScript, HTML and CSS. *Vue* is loosely shaped after this draft but differs in some key aspects and offers advantages: The approach for Custom Elements is currently still in draft status, not final and therefore not supported in every browser. *Vue* works reliable in all supported browsers

[9] https://vuejs.org/v2/guide/instance.html#Creating-a-Vue-Instance
[10] https://www.w3.org/wiki/WebComponents/

(including IE9 and above) and for this to be working, components do not rely on polyfills[11]. Furthermore, *Vue* components add functionality that Custom Elements cannot provide, i.e, custom event communication and cross-component data flow.

In terms of structuring the components, the *Vue* guide suggests using a single file approach (compare example 5.2).

```
1  <template>
2  [...]
3  </template>
4
5  <script>
6  [...]
7  </script>
8
9  <style>
10 [...]
11 </style>
```

Code Example 5.2: Single file components

The above example visualises this approach. Files like these are saved under the `.vue` extension. However, this is only a suggestion by the developer team. A separation of concerns is also possible as can be seen in example 5.3.

```
1  <!-- component-a.vue -->
2  <template>
3  [...]
4  </template>
5  <script src="./component-a.js"></script>
6  <style src="./component-a.css"></style>
```

Code Example 5.3: Separation of concerns

Another important aspect is that every component must have a single root element in the template section[12].

[11] "Replicate an API using JavaScript [...] if the browser doesn't have it natively" - Remy Sharp (https://remysharp.com/2010/10/08/what-is-a-polyfill, visited on 10/04/2018)

[12] https://vuejs.org/v2/guide/components.html

Internally, the templates are compiled into Virtual DOM render functions. In combination with the reactivity system, *Vue* is able to efficiently determine the minimum amount of components to re-render and apply the necessary number of DOM manipulations when the application's state changes.

Directives Similar to *Angular*, *Vue* uses directives in its templates. They can be used for data binding, event handling and more. In the template, they are visually marked with a `v-` prefix. The `v-bind` directive, for example, is used to reactively update an HTML attribute (compare example 5.4).

```
1  <a v-bind:href="url"> ... </a>
```

Code Example 5.4: Example for v-bind

Events like click actions by the user can be handled with the directive shown in the code snippet 5.5.

```
1  <a v-on:click="doSomething"> ... </a>
```

Code Example 5.5: Example for v-on

Vue offers special shorthands for the two mentioned directives as they are heavily used in applications (compare example 5.6).

```
1  <!-- shorthand for v-bind -->
2  <a :href="url"> ... </a>
3
4  <!-- shorthand for v-on -->
5  <a @click="doSomething"> ... </a>
```

Code Example 5.6: Shorthands for directives

Another specialty of *Vue* is the approach of modifiers[13]. These are special postfixes which are denoted by a dot. This indicates that a directive, for example, should be bound in a way differing from the default. Furthermore, multiple modifiers can also be

[13] https://vuejs.org/v2/guide/events.html#Event-Modifiers

chained. Code example 5.7 shows two possible options to leverage the modifier feature.

```
<!-- only fire an event when the 'Enter' key is pressed -->
<input @keyup.enter="onEnter">

<!-- execute the function of this button only one time -->
<button @click.once="submit"> ... </button>
```

Code Example 5.7: Modifiers on directives

Data Binding *Vue* provides two options to bind data between model and view:

- One-way: `v-bind` (compare code example 5.4)
- Two-way: `v-model`

The latter synchronises view and the model no matter where the change initially occured. An `input` field, for example, can be altered in the view when the user enters something but also the value for the field can be set and modified within the script. An example for `v-model` can be seen in 5.8.

```
<template>
  <input v-model="name" placeholder="How do you name your dog">
  <p>My dog is called {{ name }}</p>
  <button @click="setDefaultName">Set default!</button>
</template>

<script>
  [...]
  methods: {
    setDefaultName: function () {
      this.name = 'Rufus'
    }
  }
</script>
```

Code Example 5.8: Example for v-model

The example above shows the registration of functions within a component, too. Similar to `data` of *Vue* instances, functions are arranged in an object that is passed to the component. This approach to registering methods is relatively unique amongst the JavaScript framework ecosystem. Furthermore, in addition to the `methods` object, *Vue* provides the possibility to define a `computed` object which also contains functions[14]. The difference between those two is the fact that the results of the latter are cached and can be re-used. The merit of this is that, for example, computing-intensive operations only have to be executed once and can later be accessed faster.

Props This aspect of *Vue* reminds of *React*. To send data from parent to child components, one option is to make use of `props`. They can be passed in either a static or a dynamic manner[15]. However, it has to be noted that any kind of value can be passed to a prop (including type checking), as the example 5.9 shows.

```
1  <template>
2    <dog-detail name="Rufus" age="7"></dog-detail>
3  </template>
4
5  <script>
6    [...]
7    Vue.component('dog-detail', {
8      props: {
9        name: String,
10       age: Number
11     },
12     template: '<h3>My dog {{ name }} is {{ age }} years old</h3>'
13   })
14   [...]
15 </script>
```

Code Example 5.9: Example for props

[14] https://vuejs.org/v2/guide/computed.html
[15] https://vuejs.org/v2/guide/components-props.html#Static-and-Dynamic-Props

Command Line Interface *Vue* provides an official CLI to enhance the development process. NPM can be used to install it: `npm install -g vue-cli`. To scaffold a new application, one simply has to execute the following command: `vue init <template> <my-application-name>`. One specialty of this CLI is the fact that the developer team offers a set of pre-configured templates for various use cases. What kind of options are available and more can be found in section 5.3.

As mentioned before, the core library is focused on the view layer only. Therefore, additional libraries are needed to enhance the functionality of the application. Similar to *React* (both in terms of approach and naming), *Vue* recommends the use of the following two libraries for the respective parts: *Vue-Router* and *Vuex*. These two will not be discussed in this section as they are both similar to approaches by *Angular* and *React*. Instead, they will be analysed in the following section where the inspiration for the development of *Vue* will be discussed, too.

5.3 Analysis

The following section deals with various aspects that define the *Vue* framework. Both positive and negative points will be discussed and assessed. The criteria was defined in section 2.2.

Stability Back in 2015 alongside the release of version *1.0.0*, Evan You stated that '[Vue] is a personal project. So if you are looking for an enterprise backed dev team, Vue is probably not the one.'[16] However, he also mentioned that the statistics for *Vue* in the first year were solid:

- Code coverage[17] of 100% on every commit since 0.11 release
- More than 1,400 issues on GitHub closed
- Issues on GitHub were closed within an average of 13 hours

[16] http://blog.evanyou.me/2015/10/25/vuejs-re-introduction/
[17] Measurement of how many lines of the code base are called while running automated tests against it.

Especially in regard to the adaption willingness of larger companies, it is crucial for such a small project in terms of man power to be consistent. This includes regular and transparent releases including extensive testing. The numbers presented by You back in 2015 suit these premises. In comparison, the numbers of the current version *2.5.x* in 2018 do not seem to show a sign of degradation:

- Still a code coverage of 100%[18]
- More than 6,200 issues closed[19]
- Average closing time for issues reduced to six hours[20]

Other indicators for the stability of the framework are the consistent release cycles[21] and the highly supported Patreon campaign that was mentioned earlier. Overall, it can be noted that *Vue* is very active in development and evolving. Also according to their lead developer, 'there is really no incentive for [them] to just suddenly stop [with] that.'

While it can be seen as a downside that *Vue* is not backed by a large company of the technology industry, this fact can also have its own advantages. Decisions to where the framework is heading in the future are not tied to corporate interests. Only the community in consultation with the development team determines changes and new features. However, this can also pose a risk to the adaption of *Vue*: It is always possible that another fresh framework is published that all of a sudden catches the interest of the community. In that case, the popularity could be slowed down or even stagnate. Google or Facebook on the other side can almost ensure a long-term commitment both in terms of active development and financial aspects to their respective frameworks. In the JavaScript world, adapting a certain technology can always be considered a bet of how well and long the technology will persist. Again, decisions regarding this of course depend on various circumstances like projects sizes, for example.

[18] https://codecov.io/gh/vuejs/vue
[19] https://github.com/vuejs/vue/issues
[20] Number is assumingly from 2017 but still a good indication (http://issuestats.com/github/vuejs/vue, visited on 11/04/2018)
[21] https://github.com/vuejs/vue/releases

Learning Curve *Vue* has a small learning curve and is very easy to integrate. As it uses mainly ES6 for developing, the most simple example (as can be seen in 5.10) is straight forward and understandable.

```
1  <div id="app">
2    <p>{{ message }}</p>
3  </div>
4
5  <script src="https://unpkg.com/vue"></script>
6  <script>
7    new Vue({
8      el: '#app',
9      data: {
10       message: 'Hello Vue.js!'
11     }
12   })
13 </script>
```

Code Example 5.10: Minimal Vue setup

In this form, it does not require any additional compilers or transpilers as it only utilises the basic web technologies.

The documentation is extensive and has an 'Essentials' section that covers the most important concepts in a very concise and elaborate way[22]. It has to be mentioned that the guide and the API are also part of the open source project and therefore open for everyone who is interested to contribute and improve. While this option is also available for the other two frameworks, *Vue* and its creators put a lot more emphasis on this feature and encourage the people to actively participate.

As mentioned earlier, *Vue* provides a CLI. With the command `vue init <template> <my-application-name>`, it is simple to generate a new project from scratch with many best practices automatically set up. Furthermore, the `<template>` represents a valuable feature as it accepts a number of pre-configured setups developed by the development team. They include setups for applications with module bundling

[22] https://vuejs.org/v2/guide/

(either Webpack or Browserify[23], including both a fully-featured setup or a simple one for prototyping) as well as a configuration for a Progressive Web App (PWA). This in particular is an interesting aspect because the popularity of PWAs is steadily increasing[24]. In short, PWAs are web applications that can be installed on a smartphone through the browser, they feel almost like a native app and are able to even work offline, which is due to the concept of service workers. Furthermore, push notifications are also possible. Especially due to PWAs being a relative novelty it definitely helps their propagation if popular frameworks like *Vue* emphasise their adaption by offering a template for it.

JavaScript Integration Regarding best practices and stacks, *Vue* took quite a few ideas and incentives from the big players: *Angular* and *React*. The single file concept, for example, derives in parts from the latter while the general idea of splitting up JS, HTML and CSS to ensure a separation of concerns derives from the first. An advantage, however, is that *Vue* manages to just recommend approaches to the developer but also accepts different takes (e.g., organising the aforementioned technologies in separate files).

Furthermore, additional libraries can also be traced back to the other frameworks. As mentioned earlier, the suggested routing library on the official *Vue* website is *Vue-Router*[25]. Looking at the simple example 5.11, the proximity to the *Angular* counterpart becomes obvious.

```
1  [...]
2  import Router from 'vue-router';
3  import Home from '@/components/Home';
4
5  Vue.use(Router);
6
7  export default new Router({
8    routes: [
9      { path: '/', name: 'Entry point', component: Home, },
```

[23] http://browserify.org/
[24] Cf. Roy, *Progressive Web Apps: What they are and why you should care.*
[25] https://router.vuejs.org/en/

```
10      ],
11    });
```

Code Example 5.11: Vue-Router implementation

In regard to a solution for state management in larger applications, *Vuex*[26] is the recommended library. *Vuex* is a Flux-like implementation which is comparable to *React*'s *Redux*. Again, the developers took an existing, well-proven idea and built an own solution on the same approach. However, this should not be considered as 'stealing' of mindset because all technologies involved are open source and therefore people are encouraged to build upon existing ideas and projects. *Vue*'s success is partly based on the fact that Evan You took his knowledge of web development and filtered out the best approaches to various aspects of modern-day applications to create a wholesome new framework. The next chapter will show that this strategy left an impact on the JavaScript community.

Vue uses mainly JavaScript ES5 or ES6. However, with the release of version *2.5.0* the developer team announced that TypeScript will be officially supported by the framework[27]. According to this, components in *Vue* can be implemented as displayed in example 5.12.

```
1   import Component from 'vue-class-component'
2
3   @Component({
4     template: '<button @click="bark">Click!</button>'
5   })
6   export default class Dog extends Vue {
7     name: string = 'Rufus'
8
9     bark (): void {
10      window.alert(this.name + ' barks!')
11    }
12  }
```

Code Example 5.12: Vue with TypeScript

[26] https://vuex.vuejs.org/en/
[27] https://vuejs.org/v2/guide/typescript.html

This possibility adds yet another merit to *Vue* and makes it a very flexible framework to fit any kind of use case and developer background knowledge.

So far for the analysis of *Vue*. In chapter 6, the framework will be compared to the other two frameworks that are contained in this thesis. Furthermore, numbers from the open source community will be presented later on including popularity indices and download statistics.

6 Comparison

The following sections will cover various aspects of decision making in terms of adapting a new framework. While the preceding chapters contained many, more technical related parts of the technologies, this comparison chapter will pick the most relevant ones and furthermore include insights to community statistics. As all of the evaluated frameworks are open source, the community aspect plays an important role.

6.1 Features and Technical Aspects

In terms of general approaches, the frameworks do not differ very much. All of them are component based which already counts for a relevant part of their overall philosophy. However, there are also differences that are often, of course, subjective. This includes strategies in regard to file concepts (single vs. multiple) or the main development language. While *Vue* is liberal in this case offering developers more than one way to write their code (e.g., ES5/6 or TypeScript since *2.5+*) and also emphasises doing that, *Angular* is more restrictive: Although it is possible to develop an application with vanilla JS or even Dart, all official resources as well as most of the available tutorials and code snippets require the usage of TypeScript. While this also means an advantage in terms of being familiar for developers with any object-orientated background, it may make simpler projects more complex. This goes on with *Angular* being dependent on Dependency Injection, a concept which is not wide-spread among the JavaScript ecosystem[1]. *React*, for example, does not rely on it. However, *React* also comes with a quasi restriction by enforcing the use of JSX for developing. It definitely is the most diverse approach of all three frameworks as JSX implies that HTML is strongly intermixed with JS. Also, the concept of states can be a barrier to adaption.

[1] Cf. Zeigermann and Hartmann, *Die praktische Einführung in React, React Router und Redux*, page 37.

Vue and *Angular* share a relatively similar approach to structuring their components: Both split up template (HTML), style (CSS) and logic (JS). While both offer the option to handle these parts either in one file or in three separate, *Angular* definitely prefers the separation while *Vue* emphasises the single file approach even offering a special file extension for this purpose: `.vue`. Single file components have the advantage as they 'enforce' the developer to write components as slim as possible to ensure simplicity and re-usability. Both frameworks also use special syntax enhancements for the template files (i.e., `*ngFor`/`v-for` or `*ngIf`/`v-if`). *Vue*, however, offers additional shorthands for the most used directives which can slightly tidy the code. This gains importance as projects move forward and template files grow accordingly.

Another aspect in terms of data handling is the approach to binding. *React* uses one-way data binding only. *Angular* and *Vue* offer both ways by recommending the implementation of `ngModel` and `v-model`, respectively. However, two-way binding may seem convenient at first but can cause difficulty the more the application grows in terms of size and complexity. With two-way binding it can sometimes be hard to track which data gets updated where. Also, side effects may occur more often[2]. Therefore it is recommend to use one-way binding as extensively as possible even though this results in a generally higher coding effort for the developer.

One important thing that has to be mentioned at this point is the general classification of the presented technologies: While *Angular* is the only fully-featured technology in this comparison which indeed lives up to being called a framework, *React* and *Vue* are just view libraries that are often named in the same contexts as full-blown frameworks because they offer best practices for a complete development setup. However, both are very similar in this regard especially when looking at suggested external libraries for routing and state management:

- State management: *Vue* ⇒ *Vuex*, *React* ⇒ *Redux*
- Routing: *Vue* ⇒ *Vue-Router*, *React* ⇒ *React-Router*

This classification results in another comparison aspect: Opinionated vs. liberal. While *Angular* on the one hand provides a full set of homogeneous features and helpful

[2] Cf. Greene, *Two-Way Data Binding: Angular 2 and React*.

strategies for development, it also leaves less room for own decisions. When a team or a company tends to adapt this framework, it means that a lot of mindset has to be adjusted to cover *Angular*'s philosophy. On the other hand, *React* and *Vue* are very liberal and flexible. They can be used in various scenarios and technology stacks as they can, e.g., only be integrated to handle the view part of an application. This freedom of choice, however, also requires a high level of responsibility and experience from the project leaders. All decisions have to be made on an elaborate basis and accounting for future proof, too.

6.2 Support and Accessability

The biggest difference regarding the topics support and accessability has been discussed extensively in the preceding chapters but it has to be noted again as it may be one of the top reasons to lean towards a technology or not: Who is behind the framework? *React* is controlled by Facebook and *Angular* is controlled by Google. While there are merits when companies are financially supporting a certain software, it always has to be considered that these companies are still based on profit which, of course, impacts their decision making. Their concern does not ultimately have to match the needs and requests of the user base. Although they can possibly ensure a longer support and less probability of stagnation. In *Vue*'s case, the project is completely funded and supported by the open source community. Here, the concerns of the developers have a higher a chance of aligning with the users[3]. However, due to this project being dependent on few people in terms of responsibility, the risk of failure is higher in comparison with larger companies. More on community and open source numbers can be found in the following section.

In terms of accessability, *Vue* has a small learning curve. It focuses on well proven practices and offers a great documentation online which is co-developed with the community. *React* is easy to adapt for developers with a background in or an extensive knowledge of the JavaScript language as almost everything is adjusted to using JSX. Otherwise it can be difficult to get acquainted with *React*. Furthermore, the state management especially for larger projects can have a high complexity. *Angular* has

[3] https://blog.hackages.io/https-blog-hackages-io-evanyoubhack2017-cc5559806157

the steepest learning curve among the discussed frameworks. While setting up new projects with the CLI is concise and the fundamentals are easy to learn, the logic can get complicated quickly. Concepts of services, pipes and interceptors are powerful but also hard to master. However, for a company *Angular* can still be a viable choice because project structures around the scene are often very similar and therefore new coworkers can be integrated without further complications.

Job offerings can also indicate where the interest of companies lies. The following numbers are based on the business portal LinkedIn[4]. The evaluation has been limited to Germany only (search parameters are provided):

- *React*: 1,840 offerings[5]
- *Angular*: 1,081 offerings[6]
- *Vue*: 307 offerings[7]

React is by far the most mentioned technology in this context; Angular is a decent amount behind. *Vue* can not quite compete with the numbers of the other two frameworks. This can partly be explained with their time-on-market: *Vue* released its *1.0.0* version in 2015 while *React* was released in 2013. The current *Angular* has not been released until 2016 but its predecessor *AngularJS* has been around since 2012 which already made the brand more established. However, the numbers for *React* align with several aspects of community statistics which will be discussed in the following section.

6.3 Community Statistics

GitHub When it comes to the success story of open source, GitHub is among the first websites that have to be mentioned. In terms of statistics, the star system of GitHub is particularly of interest for this comparison as they indicate the popularity

[4] https://www.linkedin.com/
[5] Query: React OR ReactJS OR React.js
[6] Query: Angular4 OR "Angular 4" OR Angular2 OR "Angular 2" OR Angular NOT AngularJS NOT Angular.JS
[7] Query: Vue OR VueJS OR Vue.js

of a project. Furthermore, the change over time has to be discussed, too.

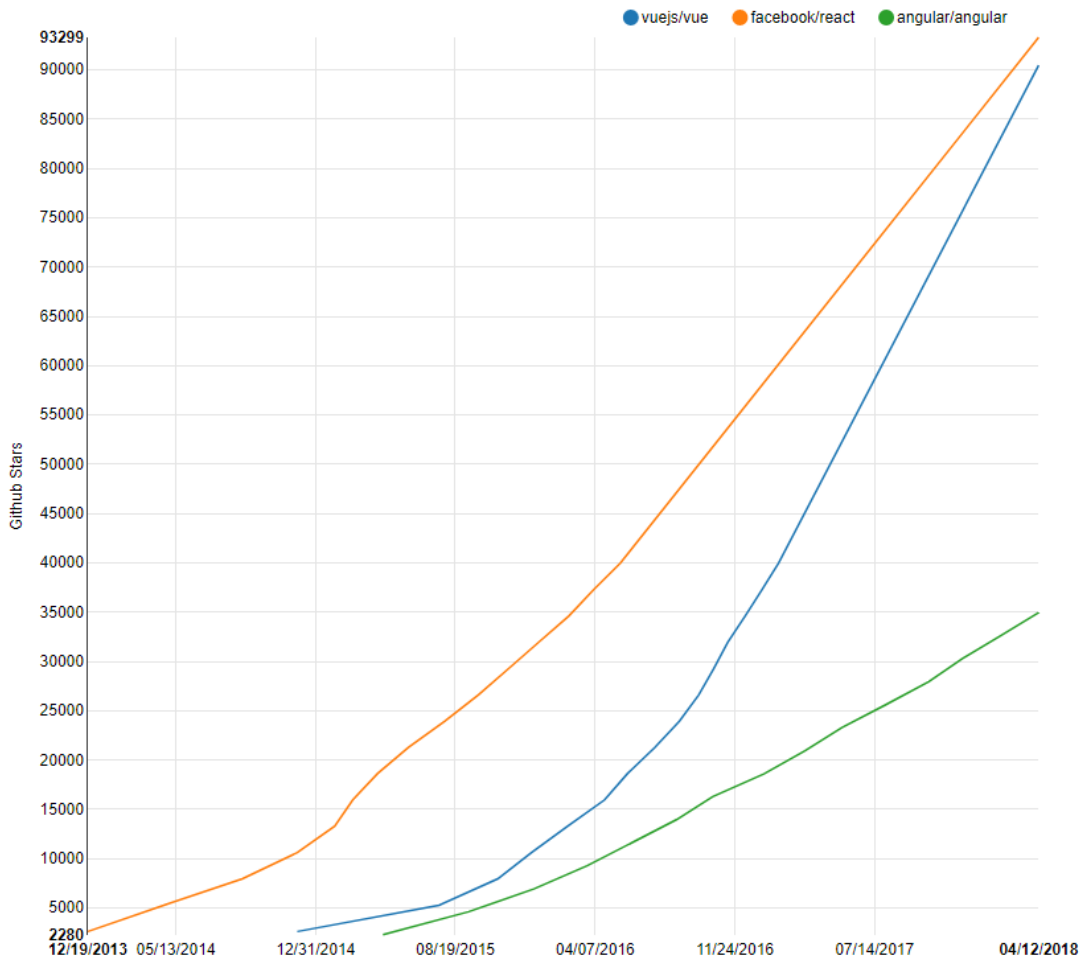

Figure 6.1: *GitHub stars over time*

As can be seen in figure 6.1 above, *React* leads the field again. However, *Vue* is just shortly behind but what is most noticeable is the historic development of it: Only in the last year (2017), *Vue* has gained more than 40,000 stars making it the highest rising framework[8]. It also does not show any indication of slowing down. *React*'s progress does not have to be diminished either: Considering the amount of time it has been on the market, it still ranks second in the competition with more than 27,000 stars

[8] https://risingstars.js.org/2017/en/#section-all

added. *Angular* can not reach these regions at all making it the least favoured project among the three. Furthermore, the historic development does not indicate any sudden increase at all. This is supported by the fact that *Angular* has the highest number of open issues at the moment (see footnotes):

- *React*: 346 issues[9]
- *Angular*: 1,962 issues[10]
- *Vue*: 107 issues[11]

It seems as if the users encounter far more problems with *Angular* than with the other two. A further interpretation can also be that *React* and *Vue* work more actively on resolving issues.

NPM Another source to assess the standing of a framework are the download statistics on NPM. With NPM being the most popular place to install third party libraries and suchlike from, its download numbers (including history) can provide viable information on how well a technology is perceived and used among the community. The data of the figure 6.2 is again based on the last twelve months to align with the previous one.

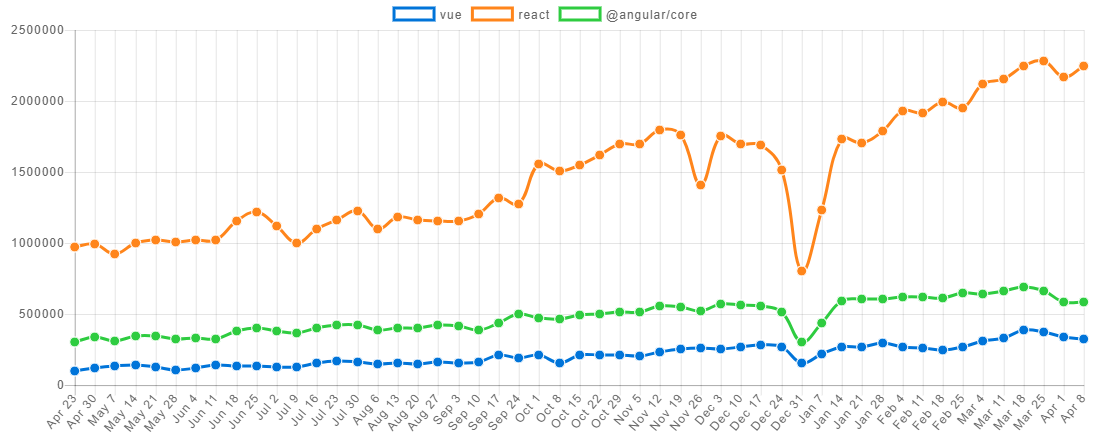

Figure 6.2: *NPM downloads over time*

[9] https://github.com/facebook/react/issues, visited on 12/04/2018
[10] https://github.com/angular/angular/issues, visited on 12/04/2018
[11] https://github.com/vuejs/vue/issues, visited on 12/04/2018

React has undeniably the most downloads and this by far. While *Angular* and *Vue* stay close to the 50,000 download mark, *React* manages to even surpass 200,000 downloads per day. The history proves that this is no snapshot but there has been a consistent increase over the past twelve months. The other frameworks struggle with their download numbers to reach relevant increases.

Stack Overflow This popular website among developers does a yearly study on everything related to favourite technologies, work preferences and coding habits[12]. The results of the study which is renewed every January are presented in the following. In regard to this thesis, one interesting part of the study is the section about the most popular technologies (see figure 6.3) and within this section the 'Most Loved, Dreaded, and Wanted Frameworks, Libraries, and Tools'. Both contain valuable insights to the frameworks discussed in the preceding chapters.

Figure 6.3: *Most popular frameworks*

Figure 6.3 shows which framework was most popular representing the general usage amount. As can be seen, *Angular* was the most intensively used frontend framework voted with 36.9% by the participating developers (excluding *Node.js* as it is a backend framework). *React* places second with 9.1% behind. However, figure 6.4 shows that although *Angular* was widely implemented it does not have the top standing in the community.

[12] https://insights.stackoverflow.com/survey/2018

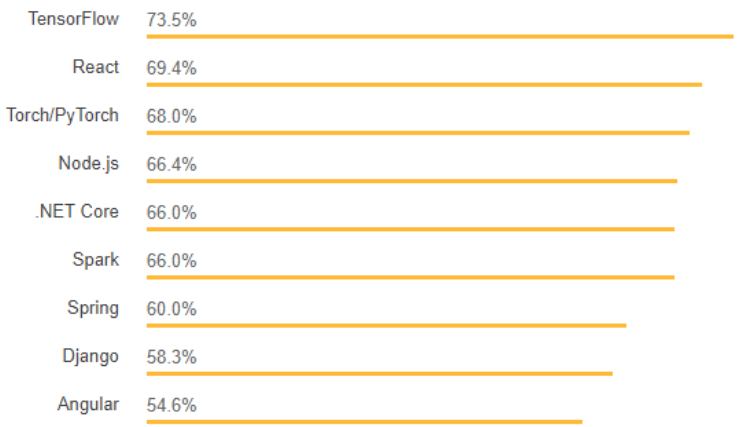

Figure 6.4: *Most loved frameworks*

The figure above represents the appreciation the developers had for the technologies they used in the course of the year. It becomes obvious that the ranking of the first graph has switched: While many developers appreciate using *React* (only second to TensorFlow[13] which has gained a lot of traction in regard to the machine learning 'hype'), they do not quite seem to enjoy *Angular* that much in relation to the high market share. Figure 6.5 continues with this aspect as *Angular* is among the top four most dreaded frameworks (first in terms of frontend) which means that developers who already worked with it would not want to use it again.

Figure 6.5: *Most dreaded frameworks*

The last figure 6.6 displays the opinions of developers in terms of which framework they would like to use in future projects resulting in the first place for *React* as being the most wanted. However, the interest in *Angular* is still present whereas not as strong as in *React*.

[13] https://www.tensorflow.org/

Figure 6.6: *Most wanted frameworks*

One thing that stands out regarding all these statistics is that *Vue* is completely missing. This is because the rise discussed earlier in this chapter just happened in 2017 and is therefore not reflected in the developer survey which covers the same year. Assumably, *Vue* will show up in the next iteration as it is already in heavy use at, e.g., the three largest Chinese technology companies (i.e., Alibaba, Tencent and Baidu)[14].

[14] Based on answer at Quora (https://www.quora.com/How-popular-is-VueJS-in-the-industry-Will-becoming-a-Vue-expert-be-useful-career-wise, visited 13/04/2018)

7 Conclusion

As this thesis presented extensive comparisons and statistic insights for the regarded frameworks, this conclusive chapter will return to the question formulated at the end of the introduction chapter: Does one of the frameworks stand out so that it can be recommended for the majority of use cases in terms of web development?

The answer clearly is no. Decisions for and against a certain technology depend heavily on the use case and other varying circumstances: Sizes of projects, worker knowledge, previous experiences and terms like these affect this process. As especially the last but also the three framework chapters have shown, *Angular*, *React* and *Vue* do not stand far apart from each other. While the first two differ mainly in their preferred development language and file separation philosophy, they have the similarity of a large tech company sponsoring their development which ensures a certain level of trust for the user base or those that plan on using them. *Vue* in this regard is the complete opposite as it is developed with mainly the community interest in mind. Also it took some of the best features from already existing frameworks and managed to create something new that saw an enormous rise in popularity in 2017 and ongoing.

To summarise a few relevant use cases and their respective recommendations in terms of framework choice, see the following collection:

- High TypeScript appreciation ⇒ *Angular*
- Emphasising guidance and structure across projects ⇒ *Angular*
- Coming from an object-orientated programming background ⇒ *Angular*
- High importance of flexibility ⇒ *React* or *Vue*
- Large scale of applications ⇒ All three

- Shallow initial learning process ⇒ *Vue*
- Emphasis on using the newest, most popular technologies ⇒ *Vue*
- Large ecosystem ⇒ *React*
- Separation of concerns in one file ⇒ *Vue*
- Designers required to work with HTML code ⇒ *Angular* or *Vue*
- Strong focus on using JavaScript ⇒ *React*

Again, it can not be emphasised enough that the above collection only represents recommendations and not strict rules. Deciding to adopt a new framework always has to be an elaborate process as it probably determines the success of future projects. However, in the JavaScript world where new frameworks are published on a regular basis it may not be smart to wait too long with the adoption of a new technology as it might be outdated by then. Although this, of course, depends largely on the size of a company: Whereas smaller development teams can test and adopt a new framework more quickly, larger companies need more time for the assessment as a wrong decision might result in financial struggles in hindsight.

8 Outlook and Future Work

As of now, the three frameworks discussed in the course of this thesis seem to play an important role for the frontend development. Of course, nobody can tell where they are heading in the future or if another framework follows the example of *Vue* and rises to the top. However, what can already be viewed is the evolution of the JavaScript language and its superior specification ECMAScript: As the consortium plans on releasing a new major version every year, the flow of interesting features continues. So far, the newest among them is the ES2017 version which was released in June 2017[1]. It contains new major features like async functions. However, while this release is still very fresh, only few of the major browsers support the specification natively so far[2]. Future ES versions are often called 'ES.Next' which is a dynamic term and a reference for the respective next major release of ECMAScript. It has also to be noted that JavaScript is not the only implementation of the standard. Further examples are V8[3], ActionScript[4] and SpiderMonkey[5].

In terms of the future of JavaScript frameworks, their general lifecycle can be an indicator for how long they will be around. Stack Overflow is a good measurement tool to visualise this lifecycle as can be seen in figure 8.1. Both of the following graphs present the percentage of all questions asked on Stack Overflow over time with the framework tag, respectively.

[1] https://www.ecma-international.org/ecma-262/8.0/index.html
[2] http://kangax.github.io/compat-table/es2016plus/
[3] https://developers.google.com/v8/
[4] https://www.adobe.com/devnet/actionscript/learning.html
[5] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

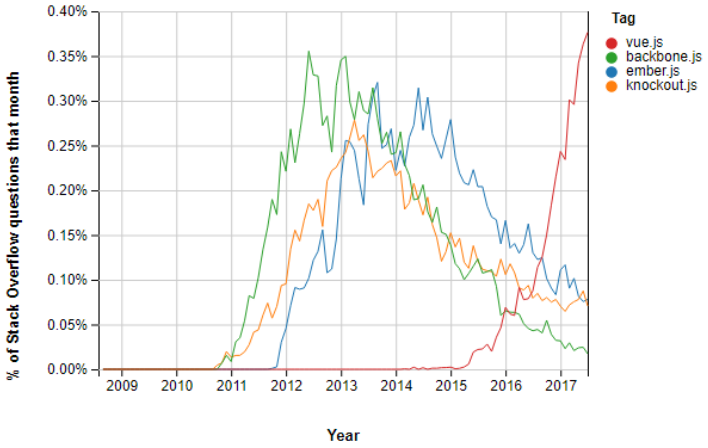

Figure 8.1: *Stack Overflow trends*

For this comparison, there are three frameworks with *Ember.js*[6], *Knockout.js*[7] and *Backbone.js*[8] that have almost completed their lifecycle. *Vue* is part of this graph to show a framework that has just begun its lifecycle. For completeness purposes, figure 8.2 visualises the current lifecycle standing for the most popular frameworks, *Angular* and *React*.

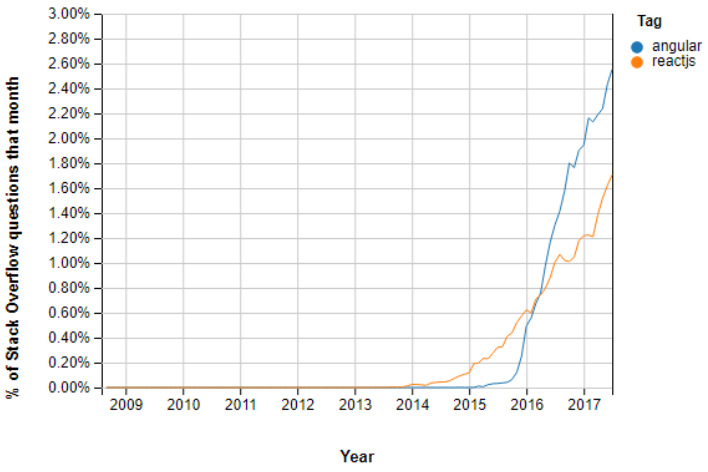

Figure 8.2: *Stack Overflow trends*

[6] https://www.emberjs.com/
[7] http://knockoutjs.com/
[8] http://backbonejs.org/

Future and additional work on the whole topic of JavaScript frameworks can contain the assessment of more relevant frameworks. Also, the analysis part of the chapters can be deepened as a lot of the mentioned keywords were only touched on a flat level to keep this thesis at a reasonable length. Aspects like further explanation of core mechanics and speed comparisons are imaginable. However, the approach may vary for a future work as the requirements for web applications and their development are under constant change.

Bibliography

Allen, Ian. *The Brutal Lifecycle of JavaScript Frameworks*. Jan. 2018. URL: https://stackoverflow.blog/2018/01/11/brutal-lifecycle-javascript-frameworks/ (visited on 15/02/2018).

Angular 2/4 Named Router Outlet. URL: https://www.concretepage.com/angular-2/angular-2-4-named-router-outlet-popup-example (visited on 15/03/2018).

Angular 4 and 5-6-7 Release Dates & Features. Jan. 2017. URL: http://www.newsjox.com/angular-4-5-6-7-release-date-features/141/ (visited on 16/03/2018).

Angular binding types. URL: https://angular.io/guide/architecture (visited on 13/03/2018).

Angular Presskit: Brand Names. URL: https://angular.io/presskit#brand-names (visited on 16/03/2018).

Angular Version 5 Release was Delayed. Sept. 2017. URL: https://dormoshe.io/articles/angular-version-5-release-was-delayed-21 (visited on 16/03/2018).

Boyer, Shayne. *JavaScript - TypeScript: Making .NET Developers Comfortable with JavaScript*. Jan. 2013. URL: https://msdn.microsoft.com/en-us/magazine/jj883955.aspx (visited on 15/02/2018).

Browser usage in February 2018. URL: https://www.w3counter.com/globalstats.php (visited on 21/03/2018).

Carnecky, Tomas. *Beware: React setState is asynchronous!* Feb. 2016. URL: https://medium.com/@wereHamster/beware-react-setstate-is-asynchronous-ce87ef1a9cf3 (visited on 28/03/2018).

Definition for software library. URL: https://www.techopedia.com/definition/3828/software-library (visited on 07/05/2018).

GitHub stars over time. URL: http://www.timqian.com/star-history/#vuejs/vue&facebook/react&angular/angular (visited on 12/04/2018).

Greene, Eric. *Two-Way Data Binding: Angular 2 and React.* Nov. 2016. URL: https://www.accelebrate.com/blog/two-way-data-binding-angular-2-and-react/ (visited on 12/04/2018).

Gudelli, Arunkumar. *History Of AngularJS.* Mar. 2017. URL: http://www.angularjswiki.com/angularjs/history-of-angularjs/ (visited on 07/02/2018).

House, Cory. *Angular 2 versus React: There Will Be Blood.* Jan. 2016. URL: https://medium.freecodecamp.org/angular-2-versus-react-there-will-be-blood-66595faafd51 (visited on 16/03/2018).

Hughes, Matthew. *Facebook re-licenses React under MIT license after developer backlash.* 2017. URL: https://thenextweb.com/dd/2017/09/25/facebook-re-licenses-react-mit-license-developer-backlash/ (visited on 29/03/2018).

Hunt, Pete. *React: Rethinking best practices.* 2013. URL: https://www.youtube.com/watch?v=x7cQ3mrcKaY (visited on 29/03/2018).

JavaScript ecosystem. URL: https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f (visited on 27/03/2018).

Johnson, Nicholas. *How to do Everything in Angular 2 using vanilla ES5 or ES6*. 2016. URL: http://nicholasjohnson.com/blog/how-to-do-everything-in-angular2-using-es6/ (visited on 16/03/2018).

Jorge. *Angular 2 CLI moves from SystemJS to Webpack*. Aug. 2016. URL: https://react-etc.net/entry/angular-2-cli-moves-from-systemjs-to-webpack (visited on 16/03/2018).

– *Your license to use React.js can be revoked if you compete with Facebook*. July 2016. URL: https://react-etc.net/entry/your-license-to-use-react-js-can-be-revoked-if-you-compete-with-facebook (visited on 29/03/2018).

Lifecycle Hooks. URL: https://angular.io/guide/lifecycle-hooks (visited on 13/03/2018).

Maida, Kim. *How to Manage JavaScript Fatigue*. Mar. 2017. URL: https://auth0.com/blog/how-to-manage-javascript-fatigue/ (visited on 15/02/2018).

Master/Detail Components. URL: https://angular.io/tutorial/toh-pt3 (visited on 07/03/2018).

Metnew, Vladimir. *History of SPA frameworks: AngularJS 1.x and nostalgia*. Apr. 2017. URL: https://medium.com/@vladimirmetnew/history-of-spa-frameworks-angularjs-1-x-and-nostalgia-2e4a00df5ee2 (visited on 14/02/2018).

Most dreaded frameworks. URL: https://insights.stackoverflow.com/survey/2018#technology-most-loved-dreaded-and-wanted-frameworks-libraries-and-tools (visited on 17/04/2018).

Most loved frameworks. URL: https://insights.stackoverflow.com/survey/2018#technology-most-loved-dreaded-and-wanted-frameworks-libraries-and-tools (visited on 17/04/2018).

Most popular frameworks. URL: https://insights.stackoverflow.com/survey/2018#technology-frameworks-libraries-and-tools (visited on 17/04/2018).

Most popular technologies in 2018. URL: https://insights.stackoverflow.com/survey/2018%5C#technology-programming-scripting-and-markup-languages (visited on 27/03/2018).

Most wanted frameworks. URL: https://insights.stackoverflow.com/survey/2018#technology-most-loved-dreaded-and-wanted-frameworks-libraries-and-tools (visited on 17/04/2018).

Motroc, Gabriela. *The rising star of JavaScript: Vue.js takes control of the game*. Jan. 2018. URL: https://jaxenter.com/vue-js-journey-to-big-leagues-140489.html (visited on 10/04/2018).

Neuhaus, Jens. *Angular vs. React vs. Vue: A 2017 comparison*. Aug. 2017. URL: https://medium.com/unicorn-supplies/angular-vs-react-vs-vue-a-2017-comparison-c5c52d620176 (visited on 15/03/2018).

NPM downloads over time. URL: http://www.npmtrends.com/react-vs-vue-vs-@angular/core (visited on 12/04/2018).

npm trends. URL: http://www.npmtrends.com/react-vs-vue-vs-@angular/cli-vs-ember-cli-vs-polymer-cli (visited on 05/03/2018).

Official TypeScript Homepage. URL: https://www.typescriptlang.org/ (visited on 15/02/2018).

Open-source licenses. URL: http://whatis.techtarget.com/definition/MIT-License-X11-license-or-MIT-X-license (visited on 05/03/2018).

Parent and children communicate via a service. URL: https://angular.io/guide/component-interaction#parent-and-children-communicate-via-a-service (visited on 07/03/2018).

Parent listens for child events. URL: https://angular.io/guide/component-interaction#parent-listens-for-child-event (visited on 07/03/2018).

Peyrott, Sebastián. *A Brief History of JavaScript*. Jan. 2017. URL: https://auth0.com/blog/a-brief-history-of-javascript/ (visited on 15/02/2018).

Riehle, Dirk. *Framework Design: A Role Modeling Approach*. ETH Zürich, Switzerland, 2000.

RouterOutlet. URL: https://angular.io/tutorial/toh-pt5#add-routeroutlet (visited on 13/03/2018).

Roy, Soumik. *Progressive Web Apps: What they are and why you should care*. Feb. 2018. URL: http://techwireasia.com/2018/02/progressive-web-apps-care/ (visited on 11/04/2018).

Severance, Charles. *JavaScript: Designing a Language in 10 Days*. 2012. URL: https://www.computer.org/csdl/mags/co/2012/02/mco2012020007.pdf (visited on 15/02/2018).

Stack Overflow trends. URL: https://insights.stackoverflow.com/trends?tags=ember.js%5C%2Cknockout.js%5C%2Cbackbone.js%5C%2Cvue.js (visited on 17/04/2018).

Stack Overflow trends. URL: https://insights.stackoverflow.com/trends?tags=angular%2Creactjs (visited on 17/04/2018).

The new AdWords UI uses Dart — we asked why. Mar. 2016. URL: https://news.dartlang.org/2016/03/the-new-adwords-ui-uses-dart-we-asked.html?m=1 (visited on 16/03/2018).

Versioning and Releasing Angular. Oct. 2016. URL: https://blog.angularjs.org/2016/10/versioning-and-releasing-angular.html (visited on 15/02/2018).

Warcholinski, Matt. *10 Famous Apps Using ReactJS Nowadays*. URL: https://brainhub.eu/blog/10-famous-apps-using-reactjs-nowadays/ (visited on 28/03/2018).

Wisseman, Stan. *Third-party libraries are one of the most insecure parts of an application*. Apr. 2016. URL: https://techbeacon.com/third-party-libraries-are-one-most-insecure-parts-application (visited on 05/03/2018).

Zeigermann, Oliver and Nils Hartmann. *Die praktische Einführung in React, React Router und Redux*. dpunkt.verlag, 2016.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15. Mai 2018 Eric Wohlgethan