A Comparison of the Frontend JavaScript GUI Frameworks Angular, React, and Vue

Matthias Link

Institute of Interactive Systems and Data Science (ISDS),
Graz University of Technology
A-8010 Graz, Austria

09 Aug 2018

Abstract

In recent years, web application development underwent a paradigm shift. The traditional approach of building individual pages was superceded by a component-based approach. To overcome the limitations of static pages, interface frameworks based on JavaScript were developed. This survey examines and compares the different approaches of three currently popular frameworks: Angular, React, and Vue.

A ToDo List Application was created to highlight strengths and weaknesses of each framework. First, a reference implementation was created. Then the user interface was reimplemented using each of the three frameworks to provide a basis for comparison. Code examples are used to demonstrate real-world implementations of different features provided by the frameworks.

Angular seems to be the most corporate-friendly framework, React seems to have the most active community, and Vue seems to be the most friendly for beginners. Hence, the choice of the most suitable framework for a particular project depends on factors such as use case, developer experience, and organisational structure.

© Copyright 2018 by the author(s), except as otherwise noted.

This work is placed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

During recent years, the interfaces of web applications became increasingly interactive and dynamic and therefore more complicated. Plain Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS) are capable of providing only static content. Interactivity is usually provided via JavaScript (JS). All modern browsers are equipped with JS engines and provide crude methods for interfacing with the Document Object Model (DOM). However, interfaces built with only the browser-provided features tend to be complex and error-prone. Graphical User Interface (GUI) frameworks were built to abstract complex interactions with the DOM and to provide extensible interfaces. Amongst many others, such as Svelte [Svelte, 2018] and Inferno [Inferno, 2018], the most widely used are Angular [Google, 2018a], React [Facebook, 2018b] and Vue [Vue, 2018c], which will be compared in this survey.

While there are many small-scale comparisons between each of the frameworks, such as Saha [2018] and Hannah [2018], there is a lack of an in-depth comparison of the particularities of each of the frameworks. This survey closes this gap by studying the techniques provided by each framework and showing their application in a real-world example.

Chapter 2 explores the approach of using independent components for structuring GUIs. The history of web applications and GUI frameworks is presented in Chapter 3. An overview of Webpack, the most common bundler for JS-based applications, is given in Chapter 4.

The process of building an interactive ToDo List Application is documented in Chapter 5. It serves as the basis for the detailed comparison of Angular, React, and Vue in Chapter 6. Aspects such as basic component structure, data handling, and community activity are analysed to provide an in-depth comparison. Additional code examples are provided to further aid the comparison.

Chapter 2

Component-Based Web Design

Using components is a relatively new approach to designing web sites and web applications. Traditionally, web sites were designed with the whole page in mind, with CSS directives used globally and therefore affecting the page as a whole. When not enough care is taken, a misplaced CSS rule can result in unpredictable changes throughout the site. Since CSS rules are interpreted by the browser on a per-page basis, there are no warnings whenever unintended changes take place. In many instances, developers are notified of a bug by their support teams, which has a negative impact on customer satisfaction.

2.1 Introducing Component-Based Web Design

Behind component-based design [Frost, 2016, Foreword] is the premise that every web page is a collection of nested components. A component should not be dependent on its surrounding container or contained children. Strictly isolating pieces of code and allowing them to co-exist independently has several benefits:

- *Improved Stability*: There are many web applications which are used by hundreds or even thousands of users simultaneously. Therefore, changes to such an application need to be carefully planned and executed. Separating the page into smaller chunks makes it easier to estimate the impact of changes to the code base. Bugs in the code base often only affect isolated components rather than the functionality of the whole web application.

- *Separation of Concerns*: SoC is one of the core philosophies of computer science. It states that every part of a system should minimise the dependencies on other parts of the system. Each part should only handle the task it is designed to do and handle that task well. While fulfilling its task, each part should not depend on other parts of the system handling their tasks properly.

  The web implements SoC by separating the structure of a document (HTML), its styling (CSS), and its behavior (JS). However, this separation only takes place at page level. The page itself, even though it also consists of several parts, does not follow that principle by default. With the introduction of components, the SoC principle can be applied on a sub-page level. By keeping the functionality of each component separate, programmers are forced to think about which problem each building block tries to solve. The resulting code is focused on the component itself by design. The communication with other components is only possible through clean interfaces, such as event handlers.

- *Reusability*: Another aspect of using components is reusability. Components can be used multiple times, even on the same page, due to loose coupling with other components. CSS are selector based, which means that it does not make a difference if a component is only used once or several times on a page. Using components on a page can therefore reduce the duplication of CSS as well. Component-based designs usually result in fewer key design elements and components are usually used more than once.

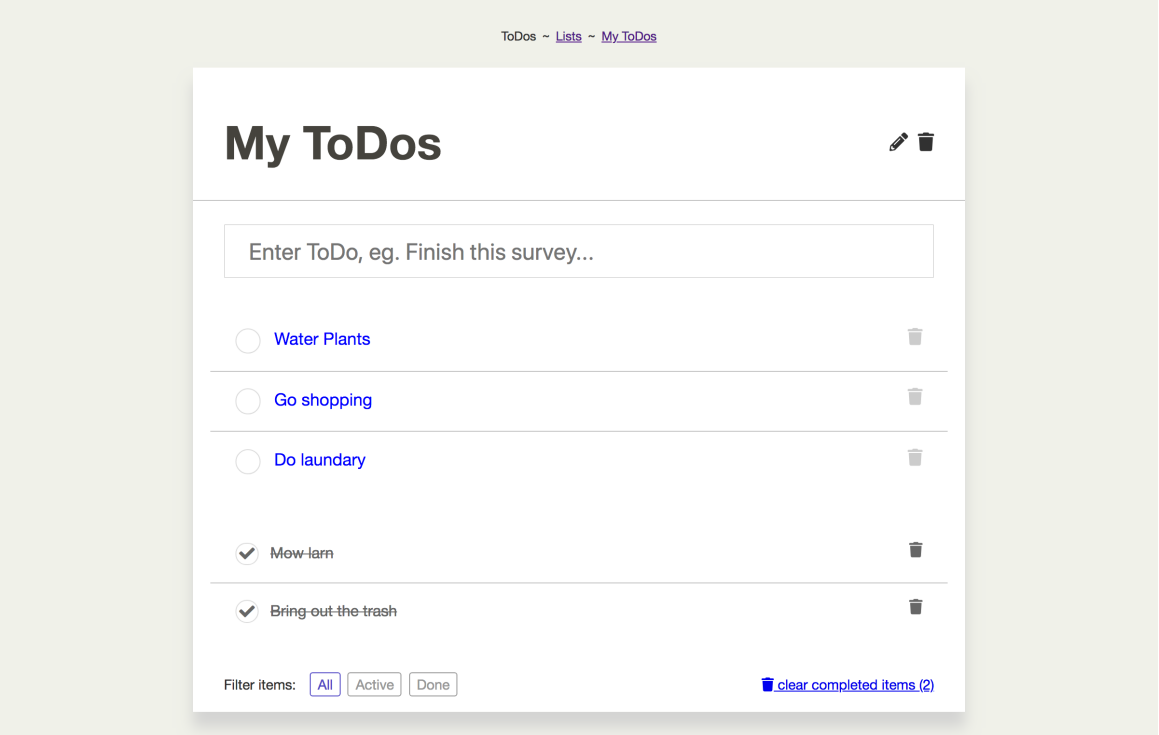

Figure 2.1: The ToDo List Application. Figure 2.2 presents the same screenshot using outlines to highlight the main components.

- *Cleaner Design*: Using recurring components on a web page reduces the effort of the design team as well. Designers only need to lay out a component once. This component can then be placed anywhere on the page and also be reused in several different places. As a result, the overall design is cleaner and more consistent.

- *Predictability*: Reusing components increases the predictability of a web application. A user needs to figure out how a component works only once. The user can make assumptions based on the previously learned behavior of a component, whenever this component is encountered again.

2.2 Practical Example

This section will focus on splitting up a page of a web application using components. This practical example is based on a ToDo List Application which will be discussed in detail in Chapter 5 and is shown in Figure 2.1. The main content of the page is wrapped in a page-like container. This container provides a white background and a box shadow. It is a very prominent part of the page, which will likely be reused in other locations of the application. Therefore, it makes sense to call it a "Layout Component", in our case "*Layout: Page*". The concepts which make up the page are labelled in Figure 2.2.

The content of the layout contains several subcomponents: First, there is the "*Head*" section, consisting of two parts: a title and a set of buttons. The "*Title*", as the name suggests, reflects the title of the page. To the right are two buttons "*Edit*" and "*Remove*" nested inside the "*Buttons*" component. These two buttons are used to navigate to other pages of the application.

Next, there is an input field for entering new todos. An input field is required to be nested inside a `<form>` element. As the form is used to create new items, it makes sense to wrap this behavior into a component, called "*New Item Form*".

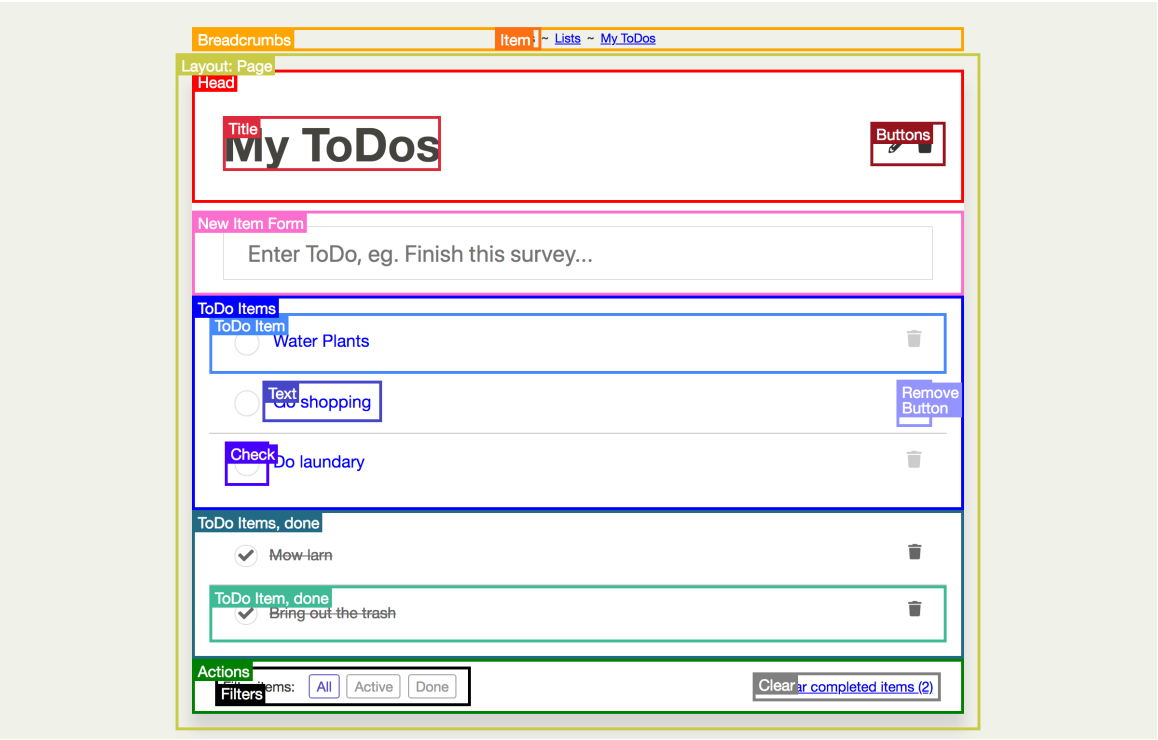

Figure 2.2: The main components of the ToDo List Application are highlighted. Figure 2.1 shows the same screenshot without outlines.

Beneath the form, there are two lists showing todos. The first list shows the active todos and the second contains the completed items. Since both lists contain very similar content, it makes sense to reuse a single component called "*ToDo Items*".

The "*ToDo Items*" component contains several "*ToDo Item*" components. An "*ToDo Item*" itself has several sub-parts. The most obvious one is the "*Text*", which contains the title of each todo. It is preceded by a "*Check*" icon, which reflects the status of the todo. Finally, a todo item provides a "*Remove Button*" at the end of its line. By clicking the button, users can remove an item from the list.

The "*Actions*" component is the last of the content components. It provides a container for two actions. "*Filters*" can be clicked to either show "*All*", "*Active*", or "*Done*" todo items. The second action provided by the "*Actions*" component is the "*Clear*" component. By clicking the link, users are able to remove all completed items from the list rather than having to click the remove button for each item individually.

Above the layout, there is a "*Breadcrumbs*" section. It reflects the usual path a user takes to navigate to the page. Inside, there are several "*Item*" components. The first item contains the application's name, which makes it the default item. It is not linked and therefore not underlined to signal a link to the user. However, all the other items are underlined and can be used to navigate to the corresponding pages.

The deconstruction of the page is complete. Splitting up the page into several independent sub-parts allows for simpler construction of the page and allows developers to change individual components without affecting other components. There are some important take-aways: The "*ToDo Items*" component is used twice on the page, but each has slightly different layouts. The component name "*Item*" is used twice (once in the "*Breadcrumbs*" component and once in the "*ToDo Items*" component). It is important for the developers to take care of this name collision. The "*Actions*" component contains two independent subcomponents with unrelated actions. One would tend to use a layout component for this kind of behavior. However, the application uses this component only once on exactly this page, so a trade-off in favor of simplicity over reusability is taken.

Chapter 3

Frontend GUI Frameworks

When creating interactive frontend GUIs, one of the three main frameworks is often used: Vue.js [Vue, 2018c], React [Facebook, 2018b], or Angular [Google, 2018a]. This chapter describes of each of these frameworks in turn.

The concept of independent and reusable pieces of code itself is rather old. It dates back to the beginnings of Object-Oriented Programming (OOP) in the late 1950s and early 1960s. OOP was introduced in the field of Artificial Intelligence (AI) research at Massachusetts Institute of Technology (MIT). Since then, many popular OOP languages have emerged, including C++, Java, and Ruby [Wikipedia, 2018a].

The utilisation of objects to communicate with parts of a web page was introduced with JavaScript (JS) and the DOM, a tree-like representation of the web page. Brendan Eich added JS to the then very popular web browser Netscape in 1995. In 1997, JS was standardised under the lesser-known name ECMAScript (ES) by European Computer Manufacturers Association (ECMA). The standard has been developed ever since [Terlson, 2018].

By manipulating the DOM, programmers were able to dynamically change the appearance of web pages at runtime. However, in the early beginnings, this approach was rather limited and in the coming years new ways of interacting with the DOM were introduced. In the early 2000s, interactive content was usually provided by browser plugins, such as Flash [Adobe, 2018] and Java Applets [Oracle, 2018]. Many developers were frustrated with the go-to way of creating interactive content on the web and tried several different approaches. Amongst them was Resig [2005], who published his library jQuery in 2006 [Resig, 2006]. Despite its age, jQuery is still used by a large number of web applications and can be considered the first modern JS framework. It was also one of the first JS frameworks to provide cross-browser support. Even though the implementation of a jQuery GUI is based on simple imperative DOM interactions, it allowed developers to build sophisticated web applications.

With the introduction of HTML 5 and the letter “Thoughts on Flash” by Jobs [2010], JS-based solutions gained even more traction, which ultimately resulted in the development several frontend GUI frameworks. The difference between the old and new frameworks is the way the interactions are implemented: While early JS frameworks relied on imperative techniques, many new GUI frameworks took a more modern declarative approach. As a result, developers were able to focus more on the question of what to implement rather than having to think about how this could be done.

The rest of this chapter is split into four sections: one section for each of the three frameworks and an additional section covers virtual DOMs.

3.1 Angular

AngularJS was the first of the modern GUI frameworks. Its initial development was started under the name "Google Feedback" by Miško Hevery in 2009, a developer at Google. Hevery convinced his team leader of the framework's potential by rewriting an application consisting of over 17,000 lines of code. The new application was finished after just three weeks of development and consisted of only 1500 lines of code. Angular JS has since been developed by teams at Google together with a very active open source community [Green and Seshadri, 2013, page vii].

In 2016, Angular Version 2 was released. The new version was rewritten from the ground up and had very few things in common with AngularJS. It introduced many of the modern concepts of Angular including component-based architecture, directives, improved data-binding, and dependency injection. With the release of version 2, the development language of Angular JS shifted from plain JS to TypeScript (TS). The shift in language was a controversial decision which split the community. With this shift, the name of the framework was slightly altered as well: Releases for version one are called "Angular JS", while the later versions are simply called "Angular".

Many companies were already committed to Angular JS (Angular 1) and it was infeasible for them to rewrite their applications using TypeScript (TS). This is the reason why the Angular JS 1.x branch is still under active development to this day. The separation of communities is also reflected in the domains of the online documentation: The documentation of Angular JS version 1 is available at `angularjs.org` [Google, 2018b]. The documentation of Angular versions 2 – 6 is hosted at `angular.io` [Google, 2018a]. The shift to TS since version 2 was motivated in part to help developers familiar with strongly-typed languages, such as Java and C#, feel more comfortable with the framework.

This survey describes version 5 of Angular. An Angular component usually consists of two files. The `[name].component.ts` file contains the TS code of the component, as shown in Listing 3.1. The HTML template of this component resides in the analogously named `[name].component.html` file. A template is usually written in a HTML-like templating language, as shown in Listing 3.2. By convention, every component contains `.component` in its filename. Every component has to be part of a module and a module can contain one or more components. Modules are used to manage dependencies and isolation levels inside an application. A top-level module, usually named `AppModule`, serves as an entry point to an Angular application.

Angular is aimed at large applications. Its high modularity and its use of TS is a double-edged sword. On the one hand, using these techniques usually results in well-isolated and independent code. On the other hand, the modularity and use of TypeScript impose a much steeper learning curve for beginners used to traditional web development techniques. Since this analysis was done, Angular 6 was released with support for Angular Elements, improved Material Design support, Tree Shaking and Animation improvements [Fluin, 2018].

```ts
// greeter.component.ts
import { Component, Input } from '@angular/core';

import template from './greeter.component.html'

@Component({
  selector: 'greeter',
  template: template
})

export class Greeter {
  @Input() name: string;
}
```

Listing 3.1: A greeter component built with Angular 5 using TS and an imported HTML template. The template is shown in Listing 3.2.

```html
<!-- greeter.component.html -->
<h1 class="greeting">{{ this.name }}</h1>
```

Listing 3.2: The HTML template for the Angular 5 greeter component shown in Listing 3.1.

3.2 React

React was originally created by Jordan Walke at Facebook in 2011. Facebook deployed the first version of React to production in 2011. It was used to build Facebook's newsfeed feature. In 2012, Instagram was acquired by Facebook and React was deployed on `instagram.com` as well. According to Papp [2018], the framework was open sourced on 29 May 2013 at JSConf US [Wikipedia, 2018b].

A React component usually resides in one single `.jsx` file, as shown in Listing 3.3. Every component inherits from `React.Component` and provides a `render` function. Every component is written in pure JS, even though that might seem not the case at first glance. The `h1` section of the `render` function is preprocessed by JSX, which converts the HTML-like code section into JS.

However, this conversion comes at a cost: JS defines several keywords, such as `class`, which in turn cannot be used in JSX. As a result, the well-known `class` attribute of an HTML element is called `className` in JSX. Additionally, JSX requires all HTML attributes to be camel-cased and to return only one top-level DOM node, which also needs to be kept in mind. Developers are not required used to use JSX, allthough it is highly recommended. In contrast to other templating engines, JSX uses single curly braces to denote inline expressions.

React is aimed at traditional web developers from beginners to experienced developers. The framework has an immense focus on the developer experience. It provides useful feedback whenever an error occurs. React also follows a very strict data-down events-up approach. The data for each component is provided by its parent. The component itself usually does not alter the underlying data, but instead triggers events which are observed

```
// greeter.jsx
import React from 'react';
import ReactDOM from 'react-dom';

class Greeter extends React.Component {
  render() {
    return (
      <h1 className="greeting">Hello, {this.props.name}</h1>
    )
  }
}

export default Greeter;
```

Listing 3.3: A greeter component built with React 3 using embedded JSX for improved template readability.

by the parent component. By design, there are only a few top-level components responsible for altering the underlying data, with many subcomponents triggering events.

3.3 Vue

Vue.js was published by You [2015] on 11 Feb 2014. According to the interview by Cromwell [2016], Evan You was working at Google and building applications using Angular at that time. He liked the idea of binding data to DOM nodes and letting the framework handle the reflection of data changes to the DOM. However, You was unhappy with the structure and the boilerplate code that came along with Angular. As he put it:

> I figured, what if I could just extract the part that I really liked about Angular and build something really lightweight without all the extra concepts involved? [Cromwell, 2016]

Vue recommends using the single-file-component format, when building Single Page Applications (SPAs). As Listing 3.4 shows, the `.vue` format is written in Extensible Markup Language (XML) and consists of three sections. The `<template>` section contains the component template. Vue uses a proprietary templating language similar to Handlebars [Katz, 2018]. The `<script>` section contains the component's JS code. Components are simple JS objects with pre-defined attributes. Components are not named globally. Instead they are named locally whenever they are used inside another component. Finally, the `<style>` section of the `.vue` format contains the CSS of the component. Vue enables the developers to scope the CSS to the component by specifying the optional `scoped` attribute of the `<style>` tag.

Vue is a progressive frontend framework. Its original intention was to build websites the traditional way and enhance parts of of the website with JS-enabled reactivity. This concept is reflected in the different ways a vue application can be mounted to an application. Vue view templates can either be specified in single-file components or directly inside the surrounding HTML. The latter approach allows developers to incorporate small Vue components in already completed web applications. Additions such as the router [Vue, 2018b] and vuex [Vue, 2018d] allow developers to build complete SPAs.

```
<!-- greeter.vue -->
<template>
  <h1 class='greeting'>Hello, {{ name }}</h1>
</template>

<script>
export default {
  props: ['name']
}
</script>

<style scoped>
h1.greeting {
  text-decoration: underline;
}
</style>
```

Listing 3.4: A greeter component built with Vue.js 2 using the proprietary `.vue` format. This listing also includes an example of isolated CSS rules used only for the greeter component.

3.4 Virtual DOM

Both React [Facebook, 2017] and Vue [Vue, 2016] utilise a virtual DOM to minimise the changes needed to the browser DOM. This practice follows the premise that manipulations to the browser DOM are expensive in terms of performance. In contrast, manipulating an object structure in memory is relatively cheap.

The virtual DOM resembles such an object structure. Both frameworks construct an updated variant of the DOM in memory. Then tree-diffing algorithms are used to identify a patch of minimal changes to be performed on the browser's actual DOM. This practice ensures that expensive DOM operations are kept to a minimum, which in turn increases performance when making many changes to the GUI.

Virtual DOMs, however, come at a cost. The DOM structure needs to be stored twice in memory - once as the browser's DOM and once virtually. Using a virtual DOM markedly increases the performance of web applications with constantly changing content. However, its performance benefits are negligable or sometimes even negative, if an application's content is mainly comprised of static elements.

Angular does not use a virtual DOM to update the browser DOM. Instead, it uses a Service Worker to asynchronously calculate the necessary changes and update the browser DOM accordingly. Angular's technique yields comparable performance to using a virtual DOM [Smiechowski, 2018].

Chapter 4

Bundlers

It is recommended by all three GUI frameworks to use some sort of bundler to package the application. Usually, an application is spread over many files, each containing a component. In addition, applications make use of many assets, such as images, stylesheets, and web fonts. It is very convenient to keep these parts separate for development of the application, but later package them for deployment.

4.1 The Problem with Loading Speed

Serving many files to a browser has a negative impact on performance. With Hypertext Transfer Protocol (HTTP) 1.1, browsers need to request each file individually and wait for each file to be loaded. During this waiting time, the browser window usually stays blank, which frustrates users and might cause them to leave the page. The more requests are needed, the longer the page takes to load. Loading speed is of upmost importance for mobile applications. Due to limited resources of the device, limited bandwidth, and high network latency, it is vital to reduce the number of requests to the server as much as possible.

One seemingly obvious solution to this problem is to concatenate all script files into one large file, which is one of the main goals of a bundler. However, this process needs to be carefully orchestrated. Simply placing one file after the other might lead to a severe problem: Two different components might use the same variable names. In many cases JS would not complain about the reuse of the same name and silently assume that both scripts meant to use the same variable. Unexpected and hard-to-catch bugs would be the result. The same thing holds true for function definitions and redefinitions, which makes the problem even worse.

Moreover, pages do not usually need all of the scripts to be present right from the start of the application. Therefore, simply concatenating all scripts together might not be the optimal solution. Several smaller chunks optimised for specific use cases can lead to better performance. This especially holds true for landing pages, where the loading speed of the page is critical.

JS is an interpreted language which does not need to be compiled in order to be executable. Traditionally, JS files were served to clients in the form they were written. Developers strive to tackle the complexity of programs - amongst many other techniques - by introducing structure to the source code, especially through indentation and line breaks. The browser, however, does not care about the cosmetic structure of a script. The browser solely interprets the scripts in a tokenised form, which does not include whitespaces at all. Simply removing all whitespaces, however, is not a viable solution. The tokeniser needs whitespace to split the code into tokens, but the tokenisation process does not depend on the amount of whitespace between the tokens. Thus, it does not make sense to transmit superfluous whitespace to the users in a production environment. The same applies to CSS files as well.

4.2 Preprocessing and Bundling

In order to tackle the problems mentioned above, several preprocessing tools were developed. Most notable is webpack [Webpack, 2018b] developed by Koppers [2012]. Webpack is recommended by all three frameworks, and is the focus here. In 2012, Tobias Koppers was trying to solve the problem of reduced footprint by using code splitting. Webpack has been in development ever since [Koppers, 2018].

Webpack thinks of individual files as one or more atomic modules. To identify individual components, webpack requires them to use the "Module" notation standardised by ECMA [2015]. Every module is a self-contained file and can be required by other modules. The formats of these files are not required to be the same. As a result, it is possible for a JS to file to require an image file. Webpack uses these dependencies to construct a dependency graph, which is used to assemble optimised bundles. In order to achieve this behaviour, webpack has to be very modular. There are four core concepts, as described in Webpack [2018a]:

- *Entry*: Entry points are the starting points for webpack's internal dependency graph. For SPAs, this is usually the main JS file. These files are used to require additional asset files, resulting in a complete graph.
- *Output*: Webpack assembles bundles of assets, which need to be placed somewhere on the file system. It is impractical to place the output files alongside the source files. Therefore, developers can specify an output directory, where the resulting files will be placed.
- *Loaders*: By default, webpack only provides a basic JS processor. Additional loaders are used in order to enable webpack to process additional input formats. A wide range of loaders are available, including for TS, CSS, images, and web fonts.
- *Plugins*: Plugins are used to further extend the functionality of webpack, for example for additional pre-processing, minification, and post-processing capabilities.

When using a GUI framework, webpack is a large part of the development experience. As stated before, Angular, React, and Vue.js recommend using webpack as their bundler of choice. Developers are able to keep components in separate files and the bundler takes care of assembling all necessary components into one large application file.

Chapter 5

A Demo ToDo List Application

For this survey, an example ToDo List Application was built in each of the three frameworks: Angular, React and Vue. The demo application serves as the basis for the in-depth comparison presented in Chapter 6. Before building the three JS-based GUIs, a reference implementation was created using the traditional GUI techniques provided by Rails [2016]. The reference implementation was used to provide a common starting point for rebuilding the same interface using the three frameworks.

5.1 The ToDo List Application

The ToDo List Application is comprised of four different views. The first is an index page serving as the entry point to the application, shown in Figure 5.1. The page contains a title and a list of todo lists. In addition, there is a button to add new lists located at the top right of the page. The *Add* button is prepended with the `plus` icon from the FontAwesome [Fonticons, 2018] icon font. For every todo list, the number of active items as well as the total number of items is shown. The index page uses a flat look and blends with the background of the whole application.

Clicking the *Add* button at the top of the index page navigates the user to the second view, the *New List View*, shown in Figure 5.2. The view provides a form for adding new todo lists, consisting of a field for entering the title and a submit button. The *New List View* is contained in a paper-like container and with a motivating title for adding a list. On top of the page is a breadcrumb section to provide context. After filling out the title field and clicking the submit button, the user is redirected to the *Detail View* showing the newly created list.

Clicking on the title of a todo list on the index page navigates to the detail view of that list. The *Detail View* is the third view of the todo list application and provides the actual todo list behaviour, shown in Figure 5.3. Similar to the *New List View*, the content is wrapped in a paper-like container, but without the additional padding. At the top of the page, the breadcrumbs section is displayed. The last of the three breadcrumbs indicates the user-defined todo list title.

The *Detail View* consists of several sub-parts. Similar to the previous views, the page contains a title showing the title of the current list. Next, there is a form for entering new todo list items. Most noticeable is that this form does not have a submit button. Instead, it relies on the user pressing Enter to submit a new item. Beneath the form are two lists of todos. The active ones are shown first, then the completed ones. Every todo item consists of a round check box, a title, and a remove link. The remove link is rendered as a trash can, which is another icon provided by the FontAwesome icon font. Todo items are toggled between the active and done state, by clicking on the title. Whenever the todo list does not contain any todos, a short text notice is rendered to indicate that no items are present in the list. At the bottom of the *Detail View* page is an action bar containing actions.

On the left-hand side are a set of filters to switch between showing all todos or only the active or completed ones. On the right-hand side, a *clear completed items* button is presented with a trash can icon in front which

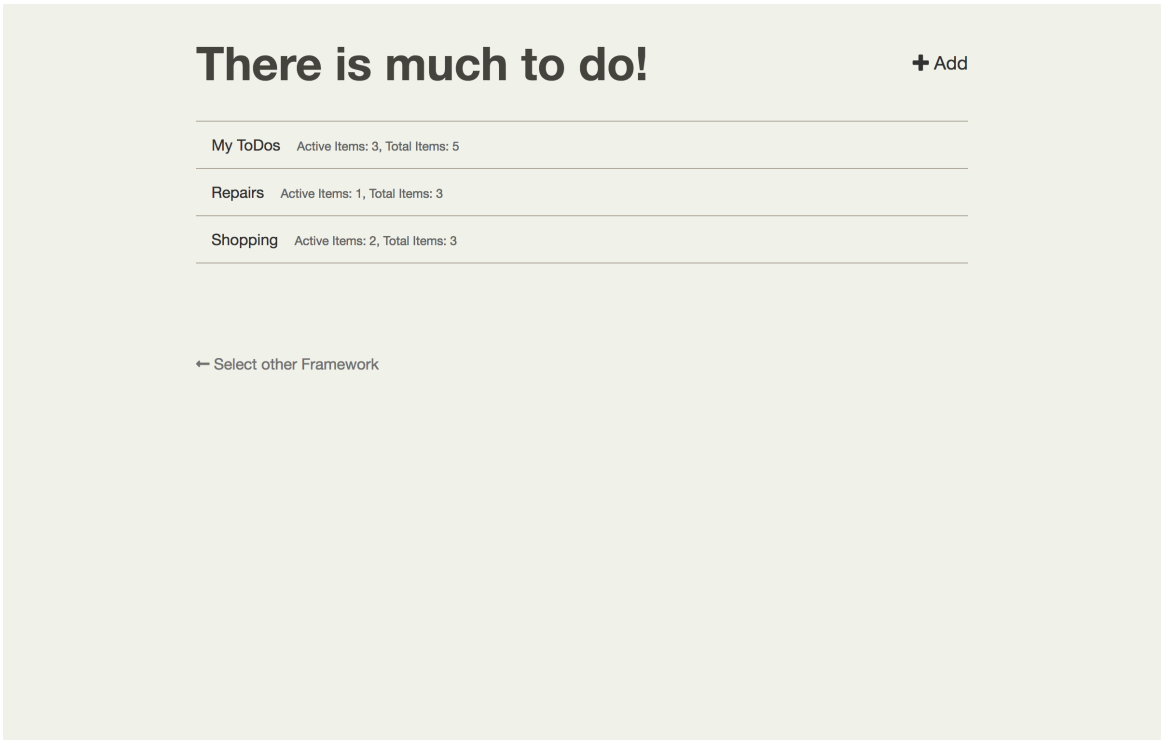

Figure 5.1: The main index view of the ToDo List Application.

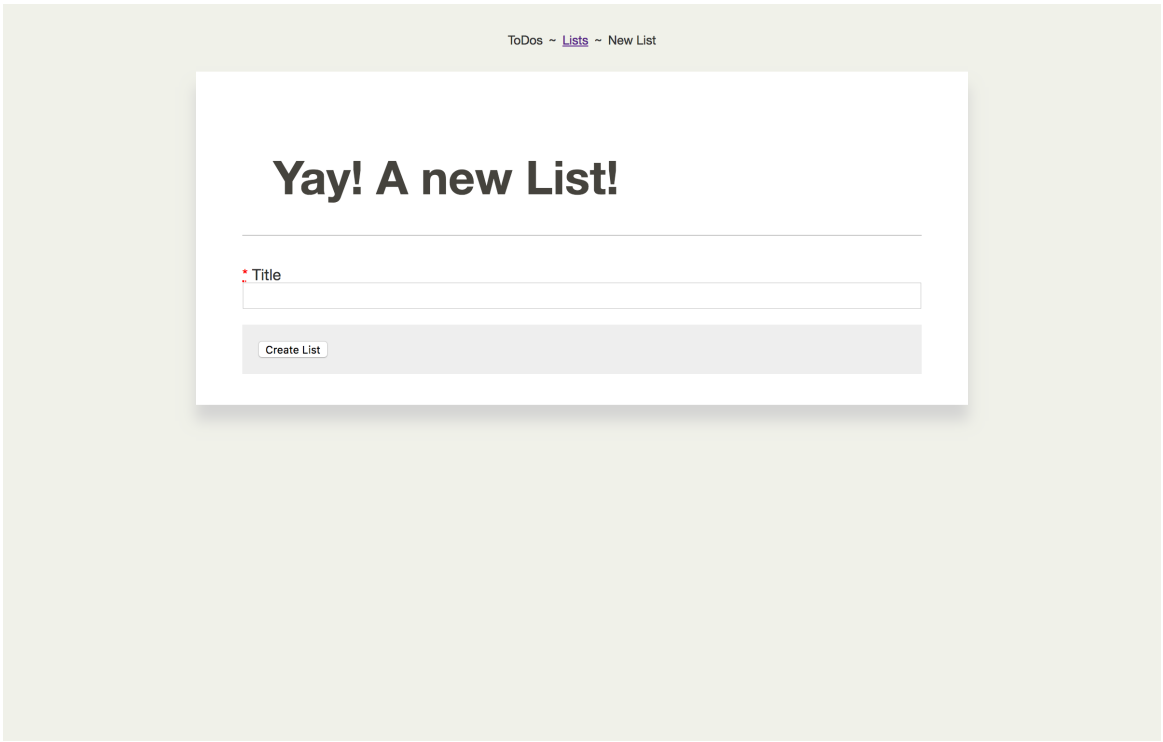

Figure 5.2: The form for creating a new list.

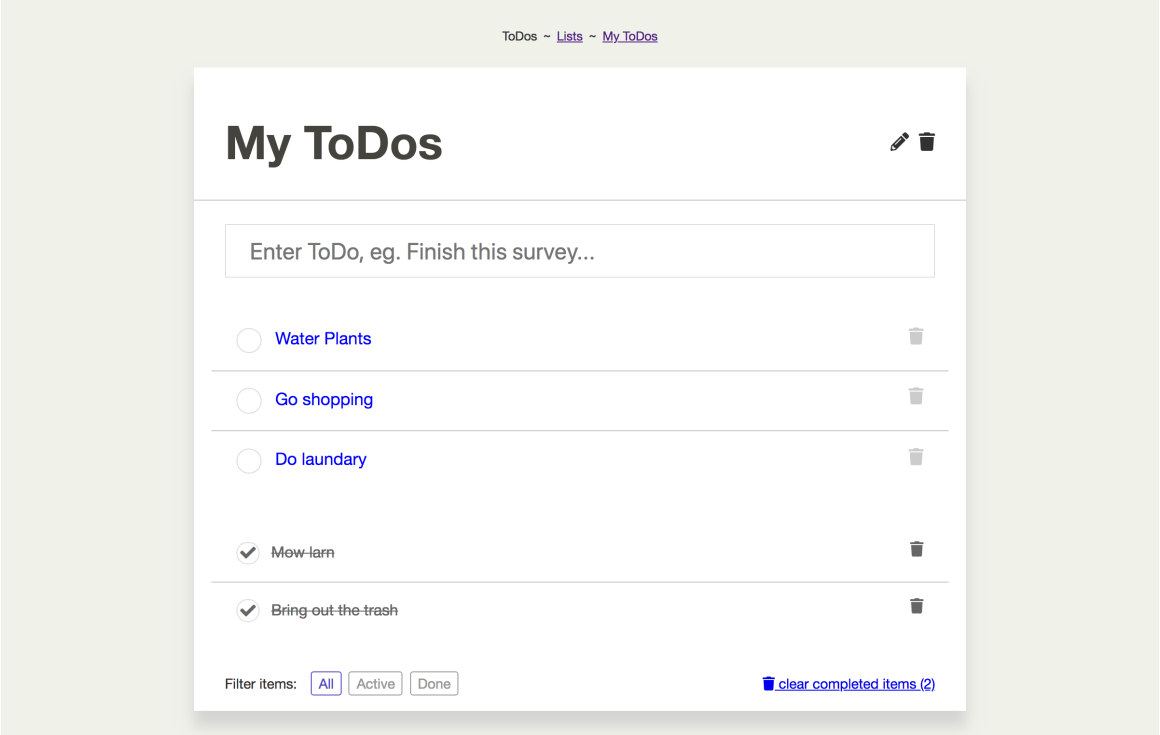

Figure 5.3: The detail view of a todo list.

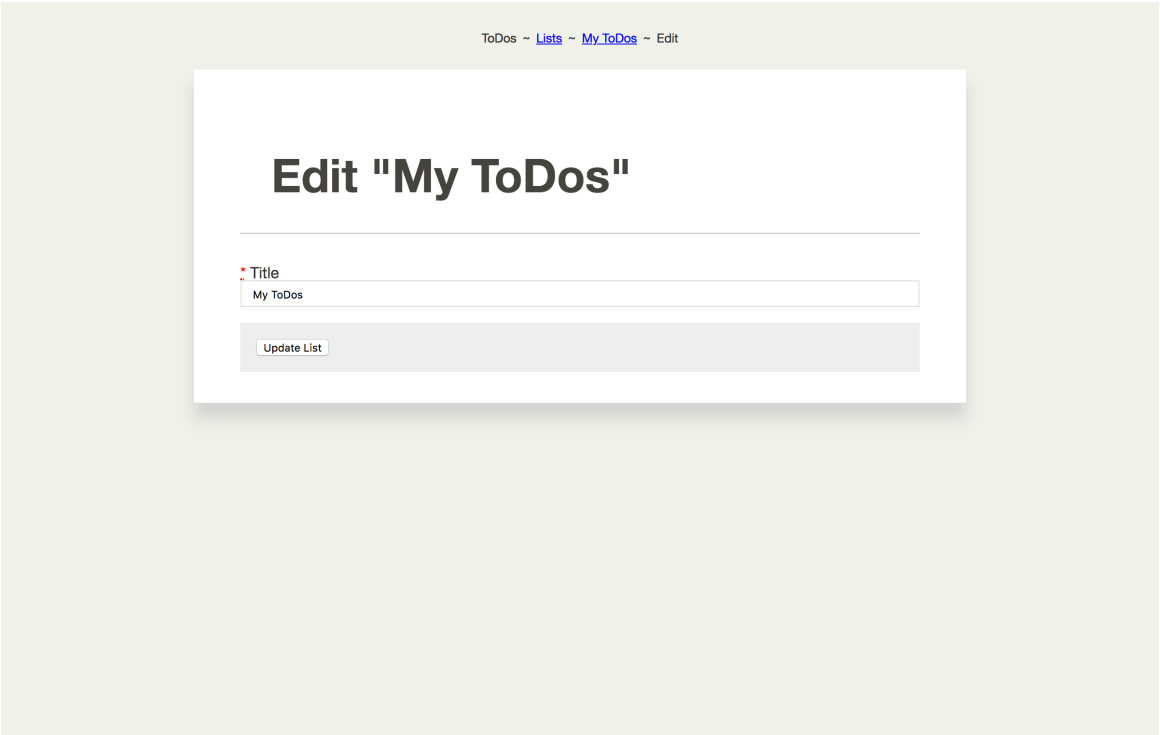

Figure 5.4: The form used to edit a todo list title.

removes all completed items from the list. Similar to the index page, the *Detail View* provides two buttons to the right of the title: a trash can icon to delete the entire list and a pencil icon to edit the name of the list. The list is no longer present in the application once the user clicks on the remove link. Therefore, the browser is redirected to the index page once a list is removed.

Clicking the pencil icon of the *Detail View* navigates to the *Edit List View*, the fourth view of the ToDo List Application, shown in Figure 5.4. This view looks very similar to the *New List View*, but there are important differences. The title of the todo list is rendered in the title of the page. The breadcrumbs section indicates that the *Edit List View* is nested semantically under the *Detail View*. Clicking the *Update List* button submits the form and the user is redirected back to the *Detail View* of the list.

5.2 The Rationale for Building the Todo List Application

The rationale behind building the ToDo List Application was to test the frameworks under real-world conditions. Performing a comprehensive study is not possible by looking at the documentation alone. In order to test the capabilities of each framework, the following challenges had to be dealt with:

- *Basic Components*: The GUIs has to be built entirely from components implemented with the help of the framework in question. Putting the compositional capabilities of each framework to the test reveals valuable insights into the internal workings of each framework. Most noticeable is how much code is needed to create individual components and how the Separation of Concerns principle is implemented.

- *Container Components*: The ToDo List Application consists of two main layouts: the *Table Top* layout, used for the index view, and the *Paper* layout, used by the rest of the application. Advanced component features are tested by using reusable container components to implement these two layouts.

- *Conditional Rendering*: There are several different places where the application needs show or hide different parts of a template. The most prominent one is in the *Detail View*. Depending on whether the todo list contains items, the empty notice needs to be rendered instead of todo items.

- *Data Binding*: The todo application requires dynamic content to be processed by the templates in several different places. The title section of the *Detail View* needs to reflect the title of the current list. Rendering the breadcrumbs section requires data to be passed to the breadcrumb subcomponent. The filters section requires dynamically updating the classes on the individual items to indicate the currently active filter and the list rendering capabilities are tested by rendering lists of todo lists in the *Index View* and individual todo list items in the *Detail View*.

- *Event Handling and Message Passing*: Click events and passing messages to parent components needs to be handled in several different locations of the application. For example, clicking on a filter should not only result in the filter item being highlighted, but the list of items needs to be updated accordingly.

- *Form Handling*: The application contains three different forms, each of which needs to be handled differently. The *New Todo* form needs to handle pressing the Enter key. The *New List* form requires the handling of new data and the *Edit List* form requires updating the state of an already existing list. Each form involves a specific set of challenges, requiring the implementation of two-way data bindings and handling of form submissions.

- *SPA and Routing*: In order to navigate between the independent views, routing capabilities need to be implemented. It is required to implement a Universal Resource Locator (URL) fragment-based approach, in order to simulate the todo list GUI being integrated as part of a larger application.

- *External API Calls*: To keep the application state consistent across different implementations, the GUI has to deal with asynchronous Application Programming Interfaces (APIs) calls to fetch and manipulate lists and items on demand.
- *Vendor Styles*: The Todo List Application uses FontAwesome to add icons to the application. A specific type of element along with a specified CSS class is needed to add an icon to the interface.
- *Post-Processing of Fetched Data*: The API of the ToDo List Application is deliberately written in such a way that post-processing is necessary by the client-side GUI. These operations include filtering and sorting of todo items.

Chapter 6

Framework Comparison

The ToDo List Application described in Chapter 5 was rebuilt by the author of this survey using each of the three JS GUI frameworks in the following order: Vue, React, and Angular. The author had no prior experience with any of the three frameworks, but did have extensive knowledge of JS.

This chapter provides an in-depth comparison of common web-design tasks and how each framework handles them. Each section contains a brief introduction to the problem. For implementation-based comparisons, each framework is examined by presenting small snippets of code alongside the explanation. These code snippets are not always extracted from the ToDo List Application. Every section starts with a tabular overview of the most important points.

6.1 Basic Components

	Angular	React	Vue
Component-Based	yes	yes	yes
Nested Components	yes	yes	yes
Dynamic Naming	no	yes	yes

Table 6.1: Basic component functionality in each framework.

Every website can be constructed using components, as outlined in Chapter 2. All three frameworks incorporate the component-based web design philosophy, although their implementations vary. As shown in Table 6.1, Angular, React and Vue support basic components as well as nested component structures. The code examples in this section show a simple user greeter component, which uses a subcomponent to add a quote of the day.

The implementations of React (Listing 6.1) and Vue (Listing 6.2) are very similar in terms of basic structure and templating. Both frameworks load nested components through Module imports into the parent component. The subcomponent is inserted into the template using a HTML tag-like structure. Developers are able to customise the component's tag-name at this stage and are able to change it to their liking.

Angular is different in this regard, as can be seen in Listings 6.3, 6.4, and 6.5. First, Angular requires developers to define a `selector` at the time a component is defined. This selector behaves similar to the other frameworks in that for each match of the selector, a component is bound to the corresponding HTML element. Since the `selector` has to be defined during component definition, it is harder for developers to change it later on. This behaviour implies that developers need to think of a unique component name right from the beginning.

```
// my_component.jsx
import React from 'react';
import ReactDOM from 'react-dom';

import Quote from './quote_of_the_day'

class MyComponent extends React.Component {
  render() {
    return (
      <div className='my-component'>
        <h1>Hello, {this.props.user}!</h1>
        <Quote />
      </div>
    )
  }
}

export default MyComponent;
```

Listing 6.1: A greeter component written in React. This example demonstrates the basic structure of a component and demonstrates using a nested `Quote` component. Developers are free to use whatever component name they prefer in the template.

Another implication is that component names are defined globally in terms of the modules in which they are used. As a result, there is a risk of component names colliding later on in development.

```
<!-- my-component.vue -->
<template>
  <div class='my-component'>
    <h1>Hello, {{ user }}!</h1>
    <quote />
  </div>
</template>

<script>

import Quote from "./quote_of_the_day"

export default {
  props: ['user'],
  components: { Quote }
}

</script>
```

Listing 6.2: A greeter component written in Vue. This example demonstrates the basic structure of a Vue component and shows how a nested `Quote` component is used. Similar to React, developers are free in their choice of names.

```
// my-module.module.ts
import { NgModule } from '@angular/core';

import { MyComponent } from './my_component.component'
import { QuoteOfTheDayComponent } from './quote_of_the_day.component'

@NgModule({
  declarations: [
    MyComponent,
    QuoteOfTheDayComponent
  ],
  exports:[
    MyComponent
  ]
})

export class MyModule {}
```

Listing 6.3: A greeter module written in Angular. The module is responsible for loading all dependent components and modules.

```
<!-- my-component.component.html -->
<h1 class='my-component'>Hello, {{ this.user }}!</h1>
<quote-of-the-day></quote-of-the-day>
```

Listing 6.4: The template used by the Angular greeter component.

```
// my-component.component.ts
import { Component, Input } from '@angular/core';

import template from './my-component.component.html'

@Component({
  selector: 'my-component',
  template: template
})

export class MyComponent {
  @Input() user: string
}
```

Listing 6.5: A greeter component written in Angular. This example demonstrates the basic structure of an Angular component and shows how a nested `Quote` component is used. The name of the subcomponent is related to the `selector` option defined in the subcomponent definition.

6.2 Resulting DOM Structure

	Angular	React	Vue
Root Element Required	implicit	yes	yes
Renders Additional Elements	yes	no	no
Virtual Root Elements	no	yes	yes
Scoped CSS	no	no	yes

Table 6.2: The rendering techniques applied by each framework and its implications for the resulting DOM structure.

There are differences in the structure of a component and how this structure affects the final DOM of the browser. Table 6.2 gives an overview of the most important features. Vue requires every component to define a single root element under which the sub HTML tags and components are nested. However, the root element is not required to be valid DOM element. As a result, developers are able to use another virtual `<template>`

element as the root element of their component. When the component is rendered to the browser's DOM, this `<template>` element is replaced with its content.

React has a similar restriction, as it also requires single root elements inside loop bodies and functions. Similar to Vue, React provides the means of using a virtual root element. In React, these are called "Fragments" and behave similar to Vue `<template>` tags.

Angular does not place any restrictions on developers in this regard. In contrast to both other frameworks, Angular does not replace the original element tag. Instead, it relies on "custom elements" defined in HTML 5. This practice needs to be taken into account when developing an application. As a side-note, this practice was the reason the author of this survey was required to slightly change the CSS of the demo application described in Chapter 5.

6.3 Container Components

	Angular	React	Vue
Basic Containers	yes	yes	yes
Multiple Outlets	yes	no	yes

Table 6.3: Support for container components.

Container components are an important aspect of component-based web design. Containers are used to provide layouts which are dynamically filled with subcomponents. All three major frameworks are capable of providing this functionality, as shown in Table 6.3. The code examples presented in this section illustrate the use of container components to provide reusable page layouts.

Angular (Listings 6.6 and 6.7) uses the `<ng-content>` tag to define a placeholder for the injected content. Additionally, developers can specify a `select` attribute. The CSS selector in this attribute instructs Angular to extract the matching elements and place them in the specified placeholder. Every element which does not match a selector is rendered in the default `<ng-content>` placeholder, which must not specify a `select` attribute.

In Vue, a similar approach is followed. The `<slot>` tag provides the same placeholding behaviour as the `<ng-content>` tag of Angular. Naming slots is also possible by specifying the `name` attribute of a slot. Listings 6.8 and 6.9 show how this behaviour is used in Vue.

React follows a different approach, as shown in Listings 6.10 and 6.11. Instead of using a pre-defined tag, React places the children of a component into a pre-defined property. To inject child elements into a container component, the `props.children` needs to be specified.

```
<!-- post.component.html -->
<page-layout>
  <headline>
    Angular Slots! <small>Demonstrating named slots</small>
  </headline>
  <p>This could be the text of a blog post.</p>
</page-layout>
```

Listing 6.6: The template of a regular Angular component. It uses the layout provided in Listing 6.7.

```
<!-- page-layout.component.html -->
<main class='page-layout'>
  <h1>
    <ng-content select='headline'></ng-content>
  </h1>
  <ng-content></ng-content>
</main>
```

Listing 6.7: The template of an Angular container component demonstrating the definition of named slots.

```
<!-- post.vue -->
<template>
  <page-layout>
    <template slot='headline'>
      Named slots are cool! <small>Demonstrating Vue Slots</small>
    </template>

    <p>This could be the text of a blog post.</p>
  </page-layout>
</template>
```

Listing 6.8: Using a Vue container component. It fills the default slot with a piece of text and the named *headline* slot of the layout with a headline.

```
<!-- page_layout.vue -->
<template>
  <main>
    <h1><slot name='headline'></slot></h1>
    <slot></slot>
  </main>
</template>

<style scoped>
main { [...] }
</style>
```

Listing 6.9: A simple layout component using Vue. It provides a default slot and a named slot called *headline*. The style is restricted in scope to the component and does not affect other components of the application.

```
// post.jsx
render() {
  return (
    <page-layout>
      <h1>React Children <small>Demonstrating React's Container System</small
         ></h1>
      <p>This could be the contents of a blog post.</p>
    </page-layout>
  )
}
```

Listing 6.10: The `render` function of a regular React component. It demonstrates the usage of a container component to provide a reusable layout around a post component.

```
// page_layout.jsx
render() {
  return (
    <main className='page-layout'>
      { this.props.children }
    </main>
  )
}
```

Listing 6.11: The `render` function of a React container component. It uses `this.props.children` to inject the contents of subcomponents into the container.

```
<!-- list.vue -->
<template>
  <ul>
    <li v-for='item in items'>
      {{ item }}
    </li>
  </ul>
</template>
```

Listing 6.12: The template of a Vue component demonstrating iteration over a list of strings using the `v-for` directive.

6.4 Directives

	Angular	React	Vue
Supports Directives	yes	no	yes
Custom Directives	yes	no	yes

Table 6.4: Support of template directives.

Directives are used to attach small pieces of code to DOM elements through the template of a component. This feature is supported by both Angular and Vue. React relies on plain JS functionality to achieve similar behaviour, as shown in Table 6.4. The following code snippets provide an example of how to iterate over a list of strings.

Vue provides a set of basic directives. Every directive is prefixed with `v-`. The most common ones are `v-for`, for iterating over a collection of items, and conditional directives, presented in Section 6.5. Listing 6.12 gives an example of how the `v-for` directive is used to render a list of items.

In Angular, directives are usually prefixed with an asterix. The asterisk is syntactical sugar, which instructs Angular to wrap the element in a virtual `<ng-template>` tag. Angular provides many different directives. The most common ones are `*ngFor`, to iterate over a collection of items, as well as several conditional directives. Listing 6.13 provides a simple example of how to iterate over a set of todo items using directives.

Many use cases are covered by the directives provided by the frameworks. However, developers might want to define specialised directives for their components. Therefore, both Angular and Vue support the definition of custom directives.

```
<!-- list.component.html -->
<ul>
  <li *ngFor='item in items'>
    {{ item }}
  </li>
</ul>
```

Listing 6.13: The template of an Angular component demonstrating iteration over a list of strings using the `*ngFor` directive.

6.5 Conditional Rendering

	Angular	React	Vue
Conditional Rendering	yes	yes	yes
Technique	Directives	pure JS	Directives

Table 6.5: Conditional rendering in each framework.

Hiding or showing parts of a template is an essential technique of component-based web design. In GUI frameworks, this technique is called *conditional rendering*. As shown in Table 6.5, all three major frameworks support conditional rendering. The examples in this section illustrate how conditional rendering is applied to render different parts of a template, depending of the loading state of a blog post.

Angular provides conditional rendering by using directives. There are two main ways of using conditionals in Angular templates: First, there is `*ngIf`, as demonstrated in Listing 6.14. The `*ngIf` directive checks a condition, in its basic form, and renders the tag and its subcomponents accordingly. Additionally, developers can specify a fallback by adding an `else` followed by a layout reference to the directive. Second, there is the `*ngSwitch` directive. It allows developers to check the value of a variable and render parts of the layout accordingly.

React does not support directives and thus handles conditional rendering differently. The philosophy "it's just JS" is at the core of JSX. Therefore, developers can rely on the conditional expressions of JS itself, as the example in Listing 6.15 shows.

Similar to Angular, Vue implements conditional rendering through directives. As shown in Listing 6.16, Vue approaches conditionals slightly differently. The directives are named after typical programming keywords: `v-if`, `v-else-if`, and `v-else`. Vue does not support layout references. As a result, the conditional parts of the template need to be adjacent.

```
<!-- post.component.html -->
<section *ngIf='post; else loading'>
  <h1>{{ post.title }}</h1>
  <p>{{ post.body }}</p>
</section>

<section #loading>
  <h1>Loading Post</h1>
  <p>Please stand by.</p>
</section>
```

Listing 6.14: The template of an Angular component demonstrating the use of the `*ngIf` directive. The first section is rendered if a post is present. If not, Angular falls back to a loading screen.

```
// post.jsx
render() {
  const { post } = this.props;

  return (
    {post ? (
      <section>
        <h1>{ post.title }</h1>
        <p>{ post.body }</p>
      </section>
    ) : (
      <section>
        <h1>Loading Post</h1>
        <p>Please stand by.</p>
      </section>
    )}
  )
}
```

Listing 6.15: An example `render` function demonstrating how conditional rendering is implemented in React. The example highlights the fact that JSX is closely releted to JS.

```
<!-- post.vue -->
<template>
  <section v-if='post'>
    <h1>{{ post.title }}</h1>
    <p>{{ post.body }}</p>
  </section>

  <section v-else>
    <h1>Loading Post</h1>
    <p>Please stand by.</p>
  </section>
</template>
```

Listing 6.16: An example vue template demonstrating the use of conditional directives `v-if` and `v-else` to provide a loading screen, if a post is not present.

6.6 Data Binding

	Angular	React	Vue
Attribute Bindings	yes	yes	yes
Text Binding	yes	yes	yes

Table 6.6: The data binding capabilities of each framework.

Another important aspect of GUI frameworks is to dynamically update the DOM whenever the underlying data changes. This concept is called *data binding*. There are two kinds of data bindings: *attribute bindings*, which conditionally add, remove or update a HTML attribute, and *text bindings*, which dynamically update the text content of a DOM node. All three major frameworks support both kinds of bindings, as illustrated in Table 6.6. The listings of this section demonstrate how binding of both attributes and text content are handled by each framework. The example in this section uses a `<p>` tag, which states the favourite colour of a user and assigns the equivalent class to the tag.

In Angular, bound attributes are wrapped in square brackets, as shown in Listing 6.17. Angular interprets the value of bound attributes as a JS expression, which is evaluated whenever the underlying data changes. Bound text is annotated with double curly braces. If a bound attribute belongs to a subcomponent, Angular passes the bound value to the correspondingly named `@Input` variable of the subcomponent.

React does not distinguish between bound attributes and bound text content. JSX allows developers to specify JS expressions anywhere in the template, which are denoted with single curly braces, as demonstrated in Listing 6.18. These expressions are evaluated whenever the underlying `state` or `props` object changes. Where a value is bound to an attribute of a subcomponent, the bound value is passed in the `props` object.

Vue implements data binding similar to Angular. Text content is bound to the template using double curly braces, as shown in Listing 6.19. Attributes are bound using the `v-bind` directive. Since the `v-bind` directive is used frequently, it is usually abbreviated as a single colon character. Vue allows passing values to subcomponents by specifying HTML attributes. However, developers need to additionally specify which attributes are

```
<!-- color_picker.component.html -->
<p [class]='color'>
  Your favourite color is: {{ color }}
</p>
```

Listing 6.17: This listing demonstrates binding data to the `class` attribute and text content in an Angular template.

```
// color_picker.jsx
render() {
  return (
    <p className={this.props.color}>
      Your favourite color is: {this.props.color}
    </p>
  )
}
```

Listing 6.18: This listing shows how data is bound to the `class` attribute and text content in a React `render` function.

allowed to be passed in the `props` section of the subcomponent definition.

```
<!-- color_picker.vue -->
<template>
  <p :class='color'>
    Your favourite color is: {{ color }}
  </p>
</template>
```

Listing 6.19: This example shows how data is bound to a class attribute and inline text in a Vue component.

6.7 Event Handling

	Angular	React	Vue
Basic Event Handing	yes	yes	yes
Event Modifiers	no	no	yes

Table 6.7: Event handling capabilities.

In order to provide a reactive GUI, it is vital to respond to user-generated events. As indicated in Table 6.7, all three frameworks are able to react to DOM events. However, there are differences between the individual frameworks. In this section, the code examples implement a simple counter component, which allows users to click a button to increment a counter variable.

Event handling in Angular is based on directives, as shown in Listings 6.20 and 6.21. Similar to data bindings, event handlers are bound to the DOM using a directive and are wrapped in parentheses. The expression specified as the value of the HTML attribute is evaluated as soon as an event occurs.

In React, event handlers are bound to the DOM similar to data bindings using the same curly braces syntax. However, developers need to take the particularities of JS into account. In JS, the `this` variable depends on the context from where a method is called. Therefore, it is important to `bind` the `this` variable of an event handler to the current component, as illustrated in Listing 6.22, in order to achieve consistent behaviour.

Event handlers in Vue are bound using the `v-on` directive. The framework provides great flexibility when it comes to defining event handlers. Developers can decide whether an event is handled inline, using an inline JS expression, or if it should be handled by a dedicated event handling method. The latter approach is demonstrated in Listing 6.23. Similar to the `v-bind` directive, the `v-on` directive can be abbreviated as an `@` sign. In contrast to the other frameworks, Vue allows the definition of additional event modifiers and filters. As a result, it is possible to prevent the propagation or default action of an event in the same place the handler is attached to the DOM.

```
<!-- counter.component.html -->
<button (click)="handleClick($event)">Click Me!</button>
<p>Current count: {{ count }}</p>
```

Listing 6.20: An Angular template using an event handler to react to clicks on a button. The click event is delegated to the `handleClick` function of the corresponding component, shown in Listing 6.21.

```
// counter.component.ts
import { Component } from '@angular/core';

import template from './counter.component.html'

@Component({
  selector: 'counter',
  template: template
})

export class Counter {
  private count: number = 0;

  handleClick() {
    this.count += 1;
  }
}
```

Listing 6.21: A simple counter component written in Angular. It provides the event handling function `handleClick` used in the template in Listing 6.20.

```
1  // counter.jsx
2  import React from 'react';
3  import ReactDOM from 'react-dom';
4
5  class Counter extends React.Component {
6    constructor(props) {
7      super(props);
8
9      this.state = {
10       count: 0
11     };
12   }
13
14   increment() {
15     this.setState({
16       count: this.state.count + 1
17     });
18   }
19
20   render() {
21     return (
22       <div class='counter'>
23         <button onClick={ this.increment.bind(this) }>Click Me!</button>
24         <p>Current count: { this.state.count }</p>
25       </div>
26     );
27   }
28 }
29
30 export default Counter;
```

Listing 6.22: A simple counter component implemented with React. The event handling method `increment` on line 14 is passed to the `onClick` event listener on line 23.

```
1  <!-- counter.vue -->
2  <template>
3    <div class='counter'>
4      <button @click='increment'>Click Me!</button>
5      <p>Current count: {{ count }}</p>
6    </div>
7  </template>
8
9  <script>
10 export default {
11   data() {
12     return {
13       count: 0
14     }
15   },
16   methods: {
17     increment() {
18       this.count += 1;
19     }
20   }
21 }
22 </script>
```

Listing 6.23: A basic counter component built with Vue. Line 4 illustrates binding the event handling method `increment` to the `click` event using the `@`-prefix notation.

6.8 Message Passing

	Angular	React	Vue
Message Passing	yes	yes	yes
Approach	bottom-up	top-down	bottom-up
Technique	@Output + EventEmitter	Callback Function	Event emitting

Table 6.8: Message passing techniques.

Usually, components do not live on their own, but are part of a greater ecosystem. Therefore, passing messages between components is an important part of component-based web design. Parents communicate with their subcomponents using data bindings. However, data bindings alone are not capable of providing the means for communication of a child component with its parent, as they are intended as one-way data streams. Another technique called "Message Passing" is used to support the reverse stream of data, usually in the form of events. As Table 6.8 shows, all three major frameworks support this functionality. The code examples of this section are based on a parent `Counter` component, which uses a dedicated `IncrementButton` subcomponent to handle user input.

Angular follows a rather verbose approach, as shown in Listings 6.24, 6.25, 6.26, and 6.27. It requires several steps to establish the communication from the child to the parent component: First, the parent component is required to define an event handler, which handles the incoming events from the child. This event handler is passed to a property marked with the `@Output` annotation of the child. The child needs to initialise this property using an instance of the `EventEmitter` class. By calling the `emit` method on the event emitter, a child is able to send events to its parent. By convention, output properties are usually prefixed with `on`, to signal their event-like behaviour. Angular therefore uses a bottom-up approach, since children explicitly define output properties.

React approaches this problem differently using a top-down approach. Instead of defining dedicated annotations in the child component, it relies on its powerful `props` feature, as demonstrated in Listings 6.28 and 6.29. By utilising the `props`, parent components are able to pass a callback function to a child. This approach is identical to the way traditional DOM events are handled.

Vue implements message passing by relying on the event system, similar to Angular. In contrast to Angular, Vue does not require any additional setup, other than setting up an event listener. The child component is able to trigger an event using the `$emit` method, provided by Vue, as illustrated in Listings 6.30 and 6.31.

```
<!-- counter.component.html -->
<div class='counter'>
  <increment-button (onIncrement)='handleIncrement()'></increment-button>
  <p>The current count is: {{ count }}</p>
</div>
```

Listing 6.24: The template of the Angular version of the `Counter` component. It renders the current count, includes a child component called `<increment-button>`, and binds the `handleIncrement` function to the `onIncrement` event. The corresponding implementation is illustrated in in Listing 6.25. The child component is shown in Listings 6.26 and 6.27.

```
// counter.component.ts
import { Component } from '@angular/core';

import template from './counter.component.html'

@Component({
  selector: 'counter',
  template: template
})

export class Counter {
  private count: number = 0

  handleIncrement() {
    this.count += 1;
  }
}
```

Listing 6.25: The implementation of the `Counter` component in Angular. It defines the `handleIncrement` event handler.

```
<!-- increment_button.component.html -->
<button (click)='handleClick($event)'></button>
```

Listing 6.26: The template of the Angular `IncrementButton` component. It delegates the `click` event to the `handleClick` function of the component implementation, outlined in Listing 6.27.

```
1  // increment_button.component.ts
2  import { Component, Output, EventEmitter } from '@angular/core';
3
4  import template from './increment_button.component.html'
5
6  @Component({
7    selector: 'increment-button',
8    template: template
9  })
10
11 export class IncrementButton {
12   @Output() onIncrement = new EventEmitter<any>()
13
14   handleClick(e) {
15     this.onIncrement.emit(null)
16   }
17 }
```

Listing 6.27: The implementation of the Angular `IncrementButton` component. It defines an `@Output` event emitter called `onIncrement`. The function `handleClick` delegates the event to said event emitter.

```
// counter.jsx
import React from 'react';
import ReactDOM from 'react-dom';

import IncrementButton from './increment_button';

class Counter extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      count: 0
    }
  }

  increment() {
    this.setState({
      count: this.count + 1
    })
  }

  render() {
    return (
      <div className='counter'>
        <IncrementButton increment={this.increment.bind(this)} />
        <p>The current count is: {{ count }}</p>
      </div>
    )
  }
}

export default Counter;
```

Listing 6.28: The `Counter` component implemented in React. It passes its own `increment` function as a `prop` to the `IncrementButton` component, shown in Listing 6.29.

```
1  // increment_button.jsx
2  import React from 'react';
3  import ReactDOM from 'react-dom';
4
5  class IncrementButton extends React.Component {
6    render() {
7      return (
8        <button onClick={this.props.increment}>Increment!</button>
9      )
10   }
11 }
12
13 export default IncrementButton;
```

Listing 6.29: The React implementation of the `IncrementButton` component. The event handling function passed through the `props` of the component is directly assigned to the `onClick` handler.

```
<!-- counter.vue -->
<template>
  <div class='counter'>
    <increment-button @increment='handleIncrement' />
    <p>The current count is: {{ count }}</p>
  </div>
</template>

<script>

import IncrementButton from "./increment_button"

export default {
  data() {
    return {
      count: 0
    }
  },
  components: {
    IncrementButton
  },
  methods: {
    handleIncrement() {
      this.count += 1
    }
  }
}

</script>
```

Listing 6.30: The Vue version of the `Counter` component. It its binds the `handleIncrement` method to the `@increment` event of the `increment-button` shown in Listing 6.31.

```
<!-- increment_button.vue -->
<template>
  <button @click='$emit("increment")'>Increment!</button>
</template>

<script>

export default {}

</script>
```

Listing 6.31: The Vue implementation of the `IncrementButton` component. It delegates the `@click` event to its parent. The inline expression of the `@click` event handler is responsible for triggering the `increment` event by calling the `$emit` function.

```
<!-- greeter_form.component.html -->
<form>
  <input [(ngModel)]='userName' placeholder='Enter your name' />
  <p>Hello, {{ userName }}!</p>
</form>
```

Listing 6.32: The template of a simple Angular form. The `ngModel` directive is used to establish a two-way data binding between the input element and the `userName` property. The implementation of this component is shown in Listing 6.33.

6.9 Form Handling

	Angular	React	Vue
Supports Forms	yes	yes	yes
Technique	Two-way binding	One-way Binding + Event Handling	Two-way binding

Table 6.9: Handling user input through form elements.

Forms are a central part of any web application, since they are the basis for providing interactivity in a web application. GUI frameworks treat their internal data structures as the "source of truth" of how the page has to be rendered. As a result, it is important to sync the internal data structures with the input elements. All three frameworks provide the means to handle forms, as shown in Table 6.9. The listings of this section illustrate how each framework handles input elements in forms. The examples are based on a simple form used to greet a user.

Applications built with Angular rely on the `*ngModel` directive to sync internal state with an `<input>` element, as illustrated in Listings 6.32 and 6.33. The `*ngModel` directive installs a two-way binding between the internal data structures and the DOM. Changes to the input element are automatically performed on the JS data structures and changes to the data structures are automatically reflected in the input element.

Vue applications follow a similar two-way binding model for handling form input elements. The `v-model` directive is used to establish a two-way binding between an input field and an internal data field, as shown in Listing 6.34. If either one is changed, the other is automatically updated as well.

React does not provide the means to automatically sync data structures. Instead, developers need to manually implement change handlers, which reflect changes to an input field to the internal data structures. Changes to internal data structures, however, are handled using standard data binding techniques. Listing 6.35 shows how forms are handled in React.

```
// greeter_form.component.ts
import { Component } from '@angular/core';

import template from './greeter_form.component.html'

@Component({
  selector: 'greeter',
  template: template
})

export class Greeter {
  private userName: string
}
```

Listing 6.33: The implementation of a form-handling component in Angular. It defines a `userName` property, which is bound in the template.

```
<!-- greeter_form.vue -->
<template>
  <form>
    <input v-model='userName' placeholder='Enter your name' />
    <p>Hello, {{ userName }}!</p>
  </form>
</template>

<script>

export default {
  data() {
    return {
      userName: ""
    }
  }
}

</script>
```

Listing 6.34: The Vue implementation of a form component. The `v-model` directive establishes a two-way binding between the value of the `<input>` element and the data property `userName`. The text binding `{{ userName }}` is updated automatically as the value changes.

```
// greeter_form.jsx
import React from 'react';
import ReactDOM from 'react-dom';

class GreeterForm extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      userName: ""
    }
  }

  changeHandler(event) {
    this.setState({
      userName: e.target.value
    })
  }

  render() {
    return (
      <form>
        <input
          value={this.state.userName}
          onChange={this.changeHandler.bind(this)}
          placeholder='Enter your name' />
        <p>Hello, {this.state.userName}</p>
      </form>
    )
  }
}


export default GreeterForm;
```

Listing 6.35: This example demonstrates a form component implemented with React. It establishes two-way data binding by binding the `state` property `userName` to the `value` attribute of the `<input>` element. In addition, the `changeHandler` method is passed to the input element. The event handler is called whenever the value of the input changes through user input and is responsible for updating the component's `state` and the `userName` placeholder.

6.10 Server-Side APIs

	Angular	React	Vue
AJAX Library provided	yes	no	no

Table 6.10: Asynchronous JavaScript and XML (AJAX) support.

Many web applications need to talk to a web server at some point. This might be to gather additional data or to perform operations in the database. Arguably, there is no need for additional framework support. With the introduction of the `fetch` API in ES 5, a simple yet powerful standard has been established for accessing application APIs via Asynchronous JavaScript and XML (AJAX), which might be the reason only Angular provides a dedicated HTTP client.

Dependency injection is a core concept of Angular. Whenever there is the need for non-component-based functionality, Angular urges developers to use service objects. The HTTP client provides the basis for a service object to communicate with a web-based application API.

6.11 Routing

	Angular	React	Vue
Routing Module	yes	third party	yes
Hash-Based	yes	yes	yes
History pushState	yes	yes	yes
Dynamic Path Segments	yes	yes	yes
Nested Routes	yes	yes	yes

Table 6.11: The routing capabilities of each framework.

The GUI of a web application usually consists of multiple views, which can accessed and exchanged independently from other parts of the application. Since the GUI has to keep track of different states of an application, traditional links combined with a page reload present a problem. Therefore, the concept of Single Page Applications (SPAs) was invented. In essence, an SPA is a single HTML page providing a JS-based multipage GUI. An SPA cannot rely on the web server to provide the correct page to the user. Therefore, the frontend application needs to implement some sort of routing mechanism to map URLs to the corresponding application pages. Routing modules are provided by all three major GUI frameworks, as indicated in Table 6.11. Another important aspect of using routes is the user's capability to bookmark and share a link to the application. In order to provide the sharing behaviour, the URL in the browser needs to be changed. There are two main strategies for doing so.

First, there is the URL fragment approach. The fragment part of a URL was traditionally used to link to specific parts of long web pages that would not fit onto a screen as a whole. By providing links using URL fragments, developers were able to automatically scroll the browser to a specific part of the page. Linking to fragments on the same URL simply scrolls the browser to a specific location without triggering a page reload. Fragment-based routers rely on the this behaviour. By navigating to a predefined URL fragment, it is possible

to change a browser's URL via JS without loosing state information for the GUI. However, this simple approach also has its drawbacks. When a page is loaded, the server in not notified about the fragment attached to the URL. As a result, the server has to reply with a generic response and is not capable of performing loading speed optimisations.

Second, there is a more modern JS-based solution. With the introduction of HTML 5, the `History` API [W3C, 2014] was introduced to browsers. It allows JS applications to access the history of a browser and push additional paths onto it. As a result, the URL can be changed via JS as if the user clicked on a traditional link, but without loosing state information. However, additional effort needs to be taken for when a user actually navigates to a given path through, for example, an external link. Since the path does not actually exist on the server, additional techniques such as server side rendering discussed in Section 6.12 need to be applied.

Both strategies are well supported by all three frameworks. Each strategy has its own drawbacks and no generally valid recommendation can be given. When deciding on a strategy, developers need to consider the drawbacks and decide which is more acceptable for their specific use case.

An important aspect of routing in general is the use of dynamic path segments, which are usually integer-based IDs embedded in the URL. They are used to link to dynamic resources such as a blog post or a product in an online shop. As shown in Table 6.11, all three frameworks support this feature.

All three frameworks take the traditional idea of routing on the client side even further. They allow the use of independent child routes for nested components. Using this technique, it is possible to use one set of routes for general application navigation and another independent set of routes for a subcomponent.

6.12 Server-Side Rendering

	Angular	React	Vue
Server-Side Rendering	yes	yes	yes
Rendered by	Node.js	Node.js	Node.js

Table 6.12: Support for server-side rendering.

It is vital to provide a visual page to the user as quickly as possible. SPAs have a major drawback in this regard. Browsers needs to download the application code as well as make additional API calls in order to being able to completely render a page. This procedure requires several round-trips to the server, which is inherently slow. In order to optimise the initial page load, all three frameworks provide the means of rendering the requested page on the server side using Node.js, as shown in Table 6.12.

Server-side rendering is usually paired with the use of the `history`-based routing technique mentioned in Section 6.11. Since the `History`-based technique aims to generate path-based URLs, the requested path is passed to the server as part of the request. The server interprets the requested path and renders the appropriate view state. The resulting HTML code is served to the user, which mitigates the initial setup time the SPA would otherwise require.

6.13 Tooling

	Angular	React	Vue
Command-Line Tools	yes	yes	yes
Project Setup	yes	yes	yes
Development Server	yes	yes	yes
Generators	yes	no	yes
Tests	yes	yes	yes
Production Build	yes	yes	yes

Table 6.13: Tooling provided by each framework.

The ease of setting up new projects is key to the development cycle of a web application. Command-line tools provide a simple yet powerful way of automating setup tasks. Each framework provides a set of tools aimed at supporting developers during the implementation process, as Table 6.13 shows.

Angular provides the `ng` command as an optional development dependency. It provides the means for setting up new projects, starting development web servers, testing and linting source code, generating basic component and module files, and building optimised versions for production environments. The `ng` command is designed to be used throughout the development cycle.

The React framework provides a similar command-line tool, called `create-react-app`. In contrast to the `ng` tool, the `create-react-app` command itself only sets up a new React project skeleton. Additionally, the `react-scripts` package is installed. The package extends the `npm` command provided by the node package manager [NPM, 2018]. The provided functionality includes starting a development web server, running a test daemon, and build procedures for production environments. Similar to Angular, the React tooling is intended to be used throughout the development cycle, even though less functionality is provided.

The structure and behaviour of the Vue command-line toolset is comparable to the React toolset, but is split into two related packages. The `vue` command is used as a global command for creating new projects. Its main task is to create project skeletons and to install the `vue-cli-service` package inside new projects. The services package is used to provide additional functionality, such as starting web servers, generating component skeletons, running tests, and building for production environments.

6.14 Documentation

	Angular	React	Vue
Getting Started Guide	yes	yes	yes
User Guide	extensive	basics	basics
API Documentation	yes	yes	yes
User Experience	needs improvement	great	great

Table 6.14: Documentations provided by each of the frameworks.

The documentation is a key part of every framework. At some point, every developer needs to have a look at it. There are usually three important parts of documentation: *Getting Started Guides*, aimed at new users and

covering the basics, *User Guides*, covering advanced topics in more detail, and the *API Documentation*, giving detailed information on methods provided by the framework.

All three frameworks provide a *Getting Started Guide* for new users. All of them are easy to understand and follow. The focus is on the core concepts of each framework and is presented with small snippets of code to allow new users to follow along easily.

Both React and Vue provide a moderate number of *User Guides*, covering common topics and basic use cases. Angular, however, not only covers these basic topics, but also goes into great detail when it comes to more advanced topics. This proves to be a double-edged sword. On the one hand, the guides describe in great detail the thinking behind each step, which seem to be useful for beginners. On the other hand, the sheer amount of text mixed with unnecessarily long examples and repetitions make it nearly impossible for intermediate developers to quickly find an answer to a specific question. The authors of the Angular documentation seem to have noticed this problem as well, as they have additionally provided a cheat sheet summarising the basic concepts.

The *API Documentation* compares similarly. React and Vue both provide concise *API Documentation* stripped down to the essentials with a clear structure. In contrast, the *API Documentation* of Angular tries to present all concepts at once. Even though a basic structure is recognisable, developers can become overwhelmed by the choice of ordering of the entries. Instead of ordering the presented items by type, they are sorted by name, resulting in a mess of functions, classes, interfaces, and directives.

6.15 Community

	Angular	React	Vue
Open Issues	2061	389	149
Closed Issues	13,317	5741	6649
Proportion of Issues Closed	86.60%	93.65%	97.81%
Open Pull Requests	346	95	101
Closed Pull Requests	9077	6872	1117
Proportion of Pull Requests Closed	96.33%	98.64%	91.71%
Contributors	674	1198	189
Stack Overflow Postings	121,551	91,850	19,863
NPM Packages	23,147	59,335	13,885

Table 6.15: Community activity for the various frameworks. The number of open and closed issues, open and closed pull requests, and contributors was gathered from the respective official GitHub repositories ([Google, 2018c; Facebook, 2018a; Vue, 2018a]). The number of Stack Overflow [Stack Overflow, 2018] postings was determined using the tags feature. The number of npm [NPM, 2018] packages was determined through a keyword-based search. The data was collected on 16 Jul 2018.

An active community is an important part of a framework. There are several metrics indicating how active a community is engaged with a framework. First of all, there is the open source project itself. The number of open and closed issues and pull requests, and the number of contributors suggest how actively a framework is developed and how many people are involved in this process. As shown in Table 6.15, Angular is by far the most active repository in terms of open and closed issues and pull requests. However, React is leading in terms of the number of contributors. Vue, despite having far fewer contributors, is comparable to React in terms of open and closed issues.

While absolute numbers show the total volume of issues and pull requests, it is also important to look at the proportion of issues and pull requests which have been closed. Vue has the highest proportion of issues closed, followed by React and Angular. In terms of proportion of pull requests closed, the order is reversed with Angular in the lead, followed by React and Vue.

Regarding posts on Stack Overflow [Stack Overflow, 2018], Angular is in the lead, followed by React. Vue on the other hand seems to raise far less questions, which may indicate a smaller user base compared to the other two frameworks or that it is simply easier to use. The metrics for each project were gathered from Stack Overflow using the results returned for corresponding framework tag.

The number of related npm packages indicates how many plugins are available for each given framework. React has by far the most related packages. Angular is in second place, followed by Vue. The results were gathered using keyword-based search on npm.org [NPM, 2018].

To summarise, Angular and React are both widely used with a large user base. Vue, in comparison, cannot quite keep up with the other two frameworks in terms of community. These results were to be expected, since both Angular and React are actively developed by two large internet corporations, and Vue is an independent project.

6.16 Target Audience and Intended Use Cases

	Angular	React	Vue
Target Audience	Medium-Large Teams	Small-Medium Teams	Small-Medium Teams
Intended Use Case	SPAs, (Small Components)	SPAs, Small Components	SPAs, Small Components

Table 6.16: Target audience and intended use cases for each framework.

The structure, underlying concepts, documentation and community reflect to some extent the intended target audience and use cases of each framework. While small teams usually prefer integrated and concise GUI frameworks, large teams tend to use more modular approaches. Changes to one part of the application are unlikely to break other parts of the application in a modular system. However, it is impractical to use heavy-weight GUI frameworks for small parts of an application, due to the introduced complexity.

The highly modularised structure of Angular suggests that its target audience are medium to large corporate teams. Compared to other frameworks, Angular introduces increased complexity due to its modularity. By relying on TS, a strongly typed language, it appeals to programmers comfortable in such languages. Strongly typed languages are usually found in corporate environments in the form of Java or C#, which further supports this hypothesis. Another consequence of the complexity of Angular is that it is aimed at building large SPAs rather than small independent components only used in small parts of a GUI.

React's strength lies in its simplicity. GUIs written in React only rely on the features of JS without any additional dependencies. As a result, React components are easy to understand while still maintaining high reusability. React components can either be used to build complete web applications or to build standalone components. This combination is ideal for small to medium-sized teams comfortable with JS.

Vue was the last of the three frameworks to be released to the public. Vue combines many concepts introduced by the other two frameworks and expands on them. By maintaining the simplicity of React and utilising Angular's directive-based approach, Vue combines the best parts of both frameworks. Vue is capable of enhancing small parts of the GUI of an already existing application by reusing inlined templates specified in the DOM element it is bound to. The framework provides both the means to partially enhance web applications as well as to building complete SPAs. The target audience of Vue seems to be small to medium-sized teams.

6.17 Overall Comparison

	Angular	React	Vue
Initial Startup Time (in hours)	6	3	2
Total Time Until Finished (in hours)	17	12	10
Lines of Code	1353	1078	785
Number of Files	78	32	28
Documentation	needs improvement	good	great
Beginner Experience	poor	ok	great

Table 6.17: Overall comparison of the development process for the ToDo List Application.

Overall, the process of building the ToDo List Application interface in each of the frameworks was reasonably straightforward. The Vue interface was created first, followed by the React implementation, and finally the Angular implementation.

Vue and React proved to be the least challenging for beginners. With two to three hours of studying the documentation and tutorials, it was possible to build the first components. By the end of the day, both the Vue and the React implementations were finished.

Angular, proved to be more difficult, as many more concepts had to be learned and understood. While the use of TS might be pleasant for experienced developers, its use proved to be more challenging than using plain JS to the author of this survey. Due to the modularity of Angular, developers need to know exactly what to look for in the documentation and what to import from the provided core libraries. It took around six hours to understand the framework, about twice as long as Vue and React. The overall development time was about 10 hours for the Vue implementation, 12 hours for React, and 17 hours for Angular, as shown in Table 6.17.

The Vue implementation consists of 785 lines of code in 28 files and its structure is shown in Figure 6.1. The React implementation of the ToDo List Application GUI compares similarly, with 1078 lines of code in 32 files. The structure of the React implementation is shown in Figure 6.2. The Angular implementation is the largest one consisting of 1353 lines of code in 78 files. The Angular implementation requires more than double the number of files. Angular encourages the separation of the components into `.component.ts` and `.component.html` files, and the use of `.module.ts` files to bundle similar components into modules. Due to the noticeable increase in the number files, only a simplified version of the directory tree is shown in Figure 6.3.

Regarding documentation, Vue provides the best for beginners. Many concepts are explained with the help of code examples and concise explanations in the form of small guides. The documentation of the React framework is also very good. The framework surface of React is rather small, so only a few simple concepts need to be understood. In comparison to the other two frameworks, Angular provides rather poor documentation. Its documentation style is similar to the other two frameworks, however many guides try to provide a very detailed view of the framework, which negatively impacts the time spent reading the documentation to find relevant pieces of information.

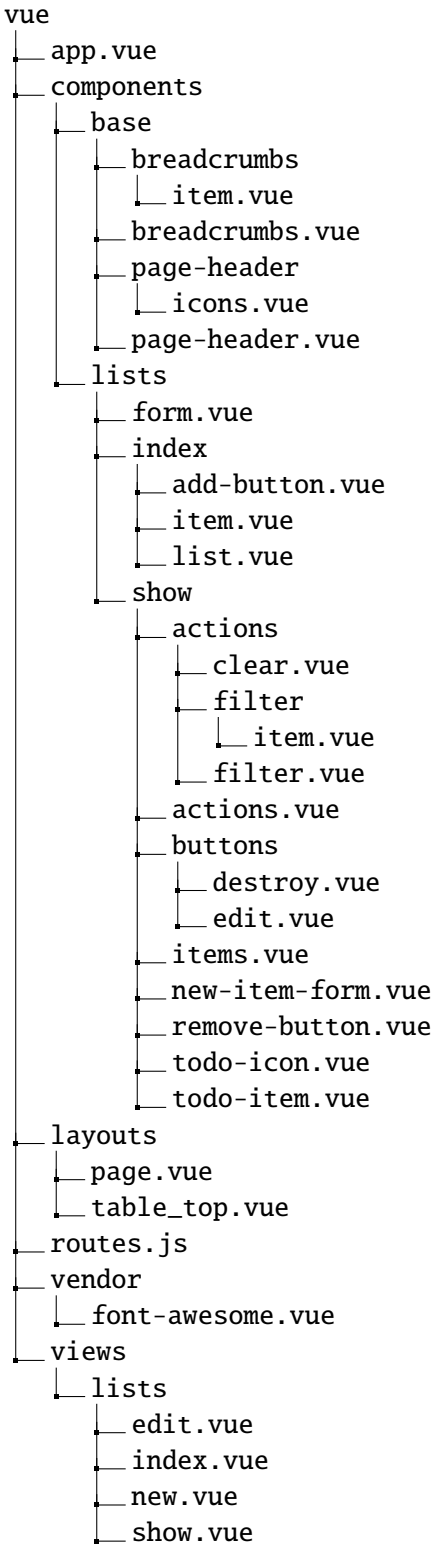

Figure 6.1: The directory tree of the Vue ToDo List Application.

```
react
├── components
│   ├── common
│   │   ├── breadcrumbs
│   │   │   └── item.jsx
│   │   ├── breadcrumbs.jsx
│   │   ├── error.jsx
│   │   ├── loading.jsx
│   │   ├── page_header
│   │   │   └── icons.jsx
│   │   └── page_header.jsx
│   ├── list
│   │   ├── actions
│   │   │   ├── clear.jsx
│   │   │   ├── filter
│   │   │   │   └── item.js
│   │   │   ├── filter.jsx
│   │   │   └── group.jsx
│   │   ├── actions.jsx
│   │   ├── buttons
│   │   │   ├── destroy.jsx
│   │   │   └── edit.jsx
│   │   ├── content.jsx
│   │   ├── empty_notice.jsx
│   │   ├── list_items
│   │   │   ├── check_icon.jsx
│   │   │   ├── item.jsx
│   │   │   └── remove_button.jsx
│   │   ├── list_items.jsx
│   │   └── new_item_form.jsx
│   ├── list_form
│   │   └── form.jsx
│   └── todos
│       ├── add_list_button.jsx
│       ├── lists
│       │   └── item.jsx
│       └── lists.jsx
├── containers
│   └── todo_lists
│       ├── edit.jsx
│       ├── index.jsx
│       ├── new.jsx
│       └── show.jsx
├── layouts
│   ├── page.jsx
│   └── table_top.jsx
├── router.jsx
└── vendor
    └── font-awesome.jsx
```

Figure 6.2: The directory tree of the React ToDo List Application.

```
angular
├── app
│   ├── api.service.ts
│   ├── app-routing.module.ts
│   ├── app.component.html
│   ├── app.component.ts
│   ├── app.module.ts
│   ├── breadcrumbs
│   │   ├── breadcrumb.component.html
│   │   ├── breadcrumb.component.ts
│   │   ├── breadcrumbs.component.html
│   │   ├── breadcrumbs.component.ts
│   │   ├── breadcrumbs.module.ts
│   │   └── breadcrumbs.service.ts
│   ├── [...]
│   ├── layouts
│   │   ├── layouts.module.ts
│   │   └── [...]
│   ├── list-common
│   │   ├── list-common.module.ts
│   │   ├── list-form.component.html
│   │   └── list-form.component.ts
│   ├── lists.service.ts
│   ├── [...]
│   ├── todo-list
│   │   ├── add-item-form.component.html
│   │   ├── add-item-form.component.ts
│   │   ├── [...]
│   │   ├── header-icons
│   │   │   └── [...]
│   │   ├── items
│   │   │   ├── check-icon.component.html
│   │   │   ├── check-icon.component.ts
│   │   │   ├── destroy-item-button.component.html
│   │   │   ├── destroy-item-button.component.ts
│   │   │   ├── item.component.html
│   │   │   ├── item.component.ts
│   │   │   ├── items.component.html
│   │   │   ├── items.component.ts
│   │   │   └── items.module.ts
│   │   ├── list-actions
│   │   │   └── [...]
│   │   ├── todo-list.component.html
│   │   ├── todo-list.component.ts
│   │   └── todo-list.module.ts
│   ├── todo-lists
│   │   └── [...]
│   └── [...]
└── [...]
```

Figure 6.3: Simplified directory tree of the Angular ToDo List Application GUI.

Chapter 7

Concluding Remarks

In this survey, the three popular GUI frameworks Angular, React, and Vue were compared. All three frameworks provide a large set of features and are capable of building rich interactive interfaces. The underlying concepts are very similar. However, the implementations are rather different with particular optimisations in mind.

Vue seems to be the most friendly for beginners. It provides concise documentation and an easy-to-understand single-component format. Data is handled on a per-component basis and it is possible to change it at every level of component nesting.

The community around React seems to be the most active. There are many plugins and tools available to improve the development experience. React provides a very JS-like experience, which is especially visible in the template syntax. Another important aspect of React interfaces is that data only resides in carefully-chosen top-level components and is propagated via immutable props to the subcomponents. In order to change the underlying database, callback functions of the top-level component must be called.

Angular seems to be the most corporate-friendly framework. TS provides a familiar environment for developers familiar with other strongly-typed languages. Angular endorses the development of modular components. Individual modules are capable of isolating the development process to specific parts of an application. Large-scale applications benefit from this, since many different teams are able to work on the application independently.

In conclusion, there is no clear winner amongst the three frameworks. Choosing the right framework for a project depends on many different factors. The experience of developers, the use case of the application, and the organisational structure of the development team all play a role in the choice of framework.

Acronyms

AI Artificial Intelligence. 7

AJAX Asynchronous JavaScript and XML. 46

API Application Programming Interface. 19, 46, 47, 49

CSS Cascading Style Sheets. 1, 3, 10, 11, 13, 14, 19, 24, 25

DOM Document Object Model. 1, 7, 9–11, 24, 25, 28, 31, 33, 37, 43, 50

ECMA European Computer Manufacturers Association. 7

ES ECMAScript. 7, 46

GUI Graphical User Interface. 1, 7, 8, 11, 13–15, 18, 19, 21, 29, 31, 33, 43, 46, 47, 50, 51, 54, 55

HTML Hypertext Markup Language. 1, 3, 8–10, 21, 24, 25, 31, 33, 46, 47

HTTP Hypertext Transfer Protocol. 13, 46

JS JavaScript. 1, 3, 7–10, 13–15, 21, 28–31, 33, 43, 46, 47, 50, 51, 55

MIT Massachusetts Institute of Technology. 7

OOP Object-Oriented Programming. 7

SPA Single Page Application. 10, 14, 18, 46, 47, 50

TS TypeScript. 8, 9, 14, 50, 51, 55

URL Universal Resource Locator. 18, 46, 47

XML Extensible Markup Language. 10

Bibliography

Adobe [2018]. *Deliver Breakthrough Web Experiences Across Platforms and Devices*. 13th Apr 2018. https://adobe.com/products/flashplayer.html (cited on page 7).

Cromwell, Vivian [2016]. *Between the Wires Interview | Evan You*. 3rd Nov 2016. https://web.archive.org/web/20171022024046/https://betweenthewires.org/2016/11/03/evan-you/ (cited on page 10).

ECMA [2015]. *ECMAScript 2015 Language Specification*. 18th Jun 2015. https://ecma-international.org/ecma-262/6.0/ECMA-262.pdf (cited on page 14).

Facebook [2017]. *Virtual DOM and Internals*. 20th Nov 2017. https://reactjs.org/docs/faq-internals.html (cited on page 11).

Facebook [2018a]. *A Declarative, Efficient, and Flexible JavaScript Library for Building User Interfaces*. 23rd Apr 2018. https://github.com/facebook/react (cited on page 49).

Facebook [2018b]. *React - A JavaScript Library for Building User Interfaces*. 13th Apr 2018. https://reactjs.org/ (cited on pages 1, 7).

Fluin, Stephen [2018]. *Version 6 of Angular Now Available*. 7th May 2018. https://blog.angular.io/version-6-of-angular-now-available-cc56b0efa7a4 (cited on page 8).

Fonticons [2018]. *Font Awesome*. 16th Jun 2018. https://fontawesome.com/ (cited on page 15).

Frost, Brad [2016]. *Atomic Design*. 28th Nov 2016. http://atomicdesign.bradfrost.com/ (cited on page 3).

Google [2018a]. *Angular JS - One Framework. Mobile & Desktop*. 13th Apr 2018. https://angular.io/ (cited on pages 1, 7–8).

Google [2018b]. *Angular JS - Superheroic JavaScript MVW Framework*. 15th Apr 2018. https://angularjs.org/ (cited on page 8).

Google [2018c]. *One Framework. Mobile & Desktop*. 23rd Apr 2018. https://github.com/angular/angular (cited on page 49).

Green, Brad and Shaym Seshadri [2013]. *Angular JS*. O'Reilly, 11th Sep 2013. ISBN 1449344852 (cited on page 8).

Hannah, John [2018]. *The Best JavaScript Frameworks You're Not Using*. 2nd Jan 2018. https://javascriptreport.com/best-javascript-frameworks-youre-not-using/ (cited on page 1).

Inferno [2018]. *Inferno*. 16th Jun 2018. https://infernojs.org/ (cited on page 1).

Jobs, Steve [2010]. *Thoughts on Flash*. 15th Apr 2010. https://apple.com/hotnews/thoughts-on-flash/ (cited on page 7).

Katz, Yehuda [2018]. *Handlebars: Minimal Templating on Steroids*. 31st Mar 2018. https://handlebarsjs.com/ (cited on page 10).

Koppers, Tobias [2012]. *Code Splitting*. 23rd Feb 2012. https://github.com/medikoo/modules-webmake/issues/7 (cited on page 14).

Koppers, Tobias [2018]. *Foreword*. 19th Apr 2018. https://survivejs.com/webpack/foreword/ (cited on page 14).

NPM [2018]. *NPM - Build amazing things*. 29th Apr 2018. https://npmjs.com/ (cited on pages 48–50).

Oracle [2018]. *Java Plug-in Technology*. 13th Apr 2018. http://oracle.com/technetwork/java/index-jsp-141438.html (cited on page 7).

Papp, Andrea [2018]. *The History of React.js on a Timeline*. 4th Apr 2018. https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/ (cited on page 9).

Rails [2016]. *rails/rails: Ruby on Rails*. 26th Mar 2016. https://github.com/rails/rails (cited on page 15).

Resig, John [2005]. *Selectors in Javascript*. 22nd Aug 2005. https://johnresig.com/blog/selectors-in-javascript/ (cited on page 7).

Resig, John [2006]. *BarCampNYC Wrap-up*. 16th Jan 2006. https://johnresig.com/blog/barcampnyc-wrap-up/ (cited on page 7).

Saha, Debasis [2018]. *React, Angular, and Vue: Which One Is Best and Why*. 12th Jun 2018. https://dzone.com/articles/angular-framework-advantages-compare-to-react-amp (cited on page 1).

Smiechowski, Maciej [2018]. *How does Angular's rendering differ from a Virtual DOM approach such as React?* 7th May 2018. https://quora.com/How-does-Angulars-rendering-differ-from-a-Virtual-DOM-approach-such-as-React (cited on page 11).

Stack Overflow [2018]. *Stack Overflow - Where Developers Learn, Share, & Build Careers*. 16th Jul 2018. https://stackoverflow.com (cited on pages 49–50).

Svelte [2018]. *Svelte - The Magical Disappearing UI Framework*. 16th Jun 2018. https://svelte.technology/ (cited on page 1).

Terlson, Brian [2018]. *ECMAScript® 2019 Language Specification*. 13th Apr 2018. https://tc39.github.io/ecma262/ (cited on page 7).

Vue [2016]. *Announcing Vue.js 2.0*. 27th Apr 2016. https://vuejs.org/2016/04/27/announcing-2.0/ (cited on page 11).

Vue [2018a]. *A Progressive, Incrementally-Adoptable JavaScript Framework for Building UI on the Web*. 23rd Apr 2018. https://github.com/vuejs/vue (cited on page 49).

Vue [2018b]. *Vue Router - Introduction*. 16th Apr 2018. https://router.vuejs.org/ (cited on page 10).

Vue [2018c]. *Vue.js - The Progressive JavaScript Framework*. 13th Apr 2018. https://vuejs.org/ (cited on pages 1, 7).

Vue [2018d]. *What is Vuex?* 16th Apr 2018. https://vuex.vuejs.org/ (cited on page 10).

W3C [2014]. *HTML5 - A Vocabulary and Associated APIs for HTML and XHTML*. 28th Oct 2014. https://w3.org/TR/html50/browsers (cited on page 47).

Webpack [2018a]. *Concepts*. 19th Apr 2018. https://webpack.js.org/concepts/ (cited on page 14).

Webpack [2018b]. *Webpack - Bundle Your Assets*. 16th Apr 2018. https://webpack.js.org/ (cited on page 14).

Wikipedia [2018a]. *Object-Oriented Programming*. 13th Apr 2018. https://en.wikipedia.org/wiki/Object-oriented_programming (cited on page 7).

Wikipedia [2018b]. *React (JavaScript Library)*. 15th Apr 2018. https://en.wikipedia.org/wiki/React_(JavaScript_library) (cited on page 9).

You, Evan [2015]. *Vue.js: A (Re)Introduction*. 25th Oct 2015. http://blog.evanyou.me/2015/10/25/vuejs-re-introduction/ (cited on page 10).