TAMPEREEN TEKNILLINEN YLIOPISTO

RIKU MARTTILA

HANDLING UNIDIRECTIONAL DATA FLOW IN A REACT.JS APPLICATION

Master of Science thesis

Examiner: Professor Tommi Mikkonen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 4th May 2016

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
MARTTILA, RIKU: Handling unidirectional data flow in a React.js application
Master of Science Thesis, 44 pages
June 2016
Major: Pervasive Systems
Examiner: Professor Tommi Mikkonen
Keywords: React.js, Single-page application, Flux, Redux, JavaScript

The scale and complexity of web applications is growing as applications move more and more to the users' browser instead of the server. New types of web applications called single-page applications are gaining popularity because they provide a good user experience. In a single-page application the application code is loaded to the user's browser on initial start up. After this navigation and other operations are executed in the users browser. Only additional needed data is fetched from the server in the background.

As applications get more complex, frameworks and libraries are needed to aid building them. Frameworks are a base that applications can be built on top of. They provide generic functionality that helps speed up the development process. When building applications a particular question arises, how to handle the data flow from the server to the user, and how to handle updates if the data changes.

In this thesis the target was to examine how to handle the data flow in a user interface library called React.js. Since React.js is a UI library, it needs application architecture to handle the data flow. In this thesis an example application is created using application architecture called Flux. The same application is also created using an improved version of Flux called Redux. Since the exactly same example is implemented with both alternatives, they can be compared with each other.

Based on the examined applications it can be clearly said that both, Flux and Redux, implement the same architecture at a general level. The example applications were similar, but it was visible that Redux is an improved version of the Flux architecture. This could be seen especially in the use of helper methods in parts that are verbose and error-prone in the Flux version. As a conclusion it was noted that instead of the original Flux architecture, Redux could have been compared with another improved implementation of Flux.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
MARTTILA, RIKU: Data flown hallinta React.js-sovelluksessa
Diplomityö, 44 sivua
Kesäkuu 2016
Pääaine: Pervasive Systems
Tarkastaja: Professori Tommi Mikkonen
Avainsanat: React.js, Single-page application, Flux, Redux, JavaScript

Web-sovellusten koko ja monimutkaisuus kasvaa ohjelmistojen siirtyessä yhä enemmän palvelimilta käyttäjien selaimiin. Uudenlaisissa web-sovelluksissa suosioon on noussut yhden sivun sovellukset, joissa tarvittava sovelluskoodi ladataan selaimeen, kun käyttäjä avaa sivun. Tämän jälkeen siirtymät ja muutokset sovelluksessa suoritetaan käyttäjän selaimessa, vaikka tietoa haetaan taustalla näytettäväksi käyttäjälle.

Kun sovellukset monimutkaistuvat, niiden rakentamiseen tarvitaan erinäisiä sovelluskehyksiä. Sovelluskehys on pohja jonka ympärille sovellus voidaan rakentaa. Näiden kehysten ja kirjastojen avulla monimutkaistenkin sovellusten tekeminen helpoittuu. Ongelmaksi usein varsinkin suuremmissa sovelluksissa muodostuu, kuinka palvelimelta saatu tieto saadaan käyttäjille näytettyä, ja kuinka hoitaa päivitykset mikäli tieto muuttuu esimerkiksi käyttäjän tekemän interaktion seurauksena.

Tässä työssä selvitetään kuinka tietovirtaa voidaan hallita React.js-nimisessä käyttöliittymäkirjastossa. Koska React.js on käyttöliittymäkirjasto, se tarvitsee tuekseen arkkitehtuurin tiedon hakemiseen ja päivittämiseen näkymissä. Työssä esimerkeiksi on otettu Flux-arkkitehtuuri sekä Fluxin alkuperäistä ideaa jalostava Redux. Näitä vaihtoehtoja käyttäen toteutetaan samanlainen esimerkkisovellus. Näin sovelluksen ominaispiirteitä voidaan vertailla keskenään.

Toteutettujen sovellusten pohjalta havaittiin selkeästi, että molemmat kirjastot toteuttavat pääpiirteittäin samaa arkkitehtuuria. Esimerkkisovellukset olivat monin osin toistensa kaltaisia, joskin Redux oli selkeästi edeltäjästään kehittyneempi versio. Tämä havainttiin eritoten siinä, että Fluxin työläät ja virhealttiit alustukset oli hoidettu käyttäen kirjaston tarjoamia apufunktioita. Loppuhavaintona työssä ilmeni, että alkuperäisen Flux-kirjaston ja parannellun Reduxin vertaamisen sijaan olisi voinut verrata Reduxia muihin paranneltuihin Flux-toteutuksiin.

PREFACE

I would like to thank professor Tommi Mikkonen for his excellent guidance throughout the writing of this thesis, without your help this process would've been a lot harder. I would also like to thank everyone else who has helped me through this 6 year endeavor, you all know who you are.

Helsinki, 24.5.2016

Riku Marttila

TABLE OF CONTENTS

1. INTRODUCTION

In the world of single-page applications, frameworks and libraries are important. They provide efficiency and convenience, and increase the overall developer experience. In this thesis a popular user interface library React.js (from now on referred to as React) will be combined with other libraries to create a full-scale web application. React uses a unidirectional data flow, which can be handled with varying architectures and patterns. The two most popular architectures are chosen, and the same application is implemented using both of them.

The term unidirectional data flow means that data flows from other part of the application (also known as the state) to the view, where it is then shown. The state is a global data storage, which is common for the whole application. Now when the data state changes, new data is passed to the view and the shown data is updated. This brings the need for a convention regarding how to change the application state if the user interacts with the view, now that the data is only flowing downwards to the view. This is where different patterns arise. Different patterns have their own way, but at this point it can be generalized that normally a user interaction triggers some sort of an event, which then manipulates the application state, which then flows to view.

The opposite of unidirectional data flow is bidirectional flow, which is commonly utilized in popular frameworks e.g. Angular.js[1] (from now on referred to as Angular) and Backbone.js (from now on referred to as Backbone). In a bidirectional flow the view can also manipulate the state, and views can even manipulate the DOM directly. While in some cases this provides simplicity, it makes applications far more complex since there is not a universal application state.

React is chosen for this thesis because of two main reasons. The first reason is its growing popularity. Figure 1.1 is visualizing hiring trends from hiring discussions on a popular website Hacker News[2]. In a relatively short period of time, React has managed to catch up with Backbone and Angular in the amount of mentions

[1]When Angular is mentioned or compared in this thesis all references are done to the 1.x version. Angular 2.x has multiple changes, but is still in alpha stage at the time of writing this. [1]

[2]https://news.ycombinator.com/news

per month. Even though React is a library, where as Angular and Backbone are frameworks, the comparing of these three makes sense since choosing one of them closes out the others.

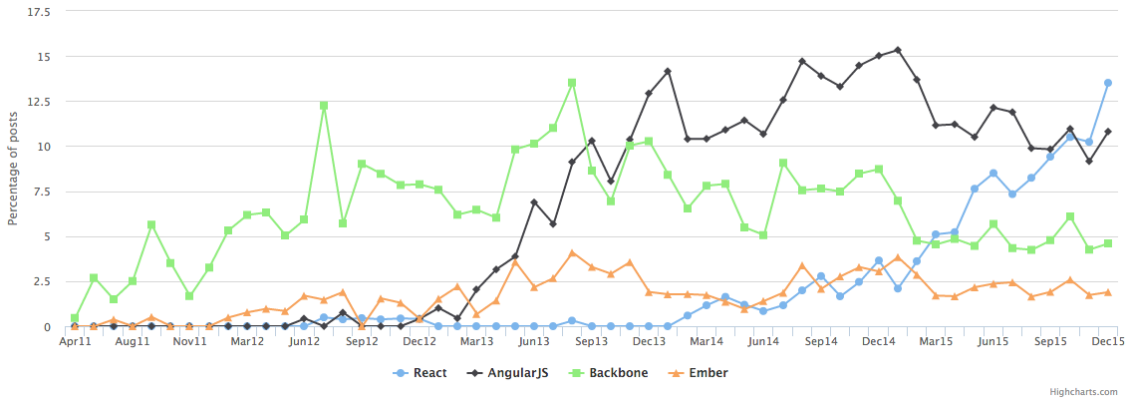

Figure 1.1. *Mentions in Hacker News hiring discussions.* [2]

Secondly, Facebook has created an additional framework, React Native[3], which is used for building native applications with React. The framework is also open-source, and has a support for both iOS and Android development. It enables creating native applications using React components, while doing slight modifications to get the different looks and feels various platforms have. This means that a single React application cannot be run on multiple platforms (web, iOS, Android) but the basics stay the same. Facebook calls this approach "learn once, write anywhere".[3] Even though React Native is not discussed further in this thesis, this is a significant push in favor of React when comparing it to other options.

In chapter 2, the reader is given an overall insight in web applications. Chapter 3 introduces React and its different parts. Chapter 4 introduces the two most popular Flux implementations and their basic ideas. In Chapter 5 a sample application is implemented using both of these architectures. Chapter 6 is the conclusions.

[3]https://facebook.github.io/react-native/

2. BACKGROUND

Even though the main focus of this thesis is handling the data flow with a certain user interface library, further overall knowledge of modern web applications is important for the reader. In the following sections the most vital concepts are explained, including web applications, single-page applications, JSON, and an insight in JavaScript and its role in modern web development.

These sections are chosen because they are essential to the main topic of this thesis. When categorized, React applications are web applications, and more specifically single-page applications. Due to the nature of single-page applications, data exchange is done by transferring JSON from and to the server, also known as the backend. As with other modern web applications, React applications also rely highly on new JavaScript standards and tooling eg. transpilers, so an insight to this is also valuable to the reader.

2.1 Web applications

Web applications are applications that are executed in the user's browser. They consist of HTML and CSS, combined with some programming language to add logic and functionality. Because of its wide adoption and browser support this language is usually JavaScript [4]. Web applications are an alternative for traditional native software applications which require installation. As the application is targeted to function in a browser environment, platform-specific differences are abstracted away. Even though browsers also have minor differences, the development pace can be considered to be more rapid.

The difference between a web application and a website is indeterminate and subjective. One definition would be that a web application is defined by its user interactions, when a website is defined by its content. This is a difference of creation versus consumption. This definition has also some corner cases, e.g. web applications which users use just to consume content, but it can be considered as a good generalization. An example of a web application is Facebook[1], where users can inter-

[1]https://www.facebook.com/

act with the application in multiple ways; register, login, write messages to others, etc. A news site, e.g. CNN[2], is an example of a website where nearly all content is static. [5]

2.2 Single-page applications

Modern web applications usually consist of a single page, which is loaded at startup. The initially needed content, HTML[3], CSS[4] and JavaScript, is loaded at the beginning. After the first load, data is then loaded asynchronously in the background when a user interacts with the interface. After the data is fetched, it is dynamically added to the user interface without reloading the page. Because of this the whole page does not need to be reloaded just to update a piece of data. This enables a more fluent user experience when the user interacts with the application. Even though the page is not reloaded at a different URL after a user interaction, routing can be handled in the application by JavaScript. This enables linking to specific content in a singe-page application, without the need of reloading. Dynamic data fetching is commonly done by a group of technologies named AJAX[5] (asynchronous JavaScript and XML). With AJAX data can be fetched from the server in the background using a API called XMLHttpRequest[6] (XHR). Despite both of the acronyms including XML[7], data transmitting is not bound to XML only. Most commonly the data is loaded in one of two formats, XML and JSON. Even though XML was initially intended to be used with AJAX, JSON has become increasingly popular in data interchange. Even though statistics about JSON versus XML usage does not widely exist, the claim about JSON's increasing popularity is backed up by large web services dropping XML support in favor of JSON [6] and research stating JSON's performance over XML [7].

Since the browser page is not refreshed when new data is received, the user interface is updated using a process called DOM manipulation. The DOM[8] (Document Object Model) is a convention for presenting objects in a HTML document. All the elements, commonly called nodes, in a HTML document are organized into a DOM tree structure in the browser, where these nodes can then be accessed and manipulated when needed. [8]

[2]http://edition.cnn.com/
[3]https://www.w3.org/html/
[4]https://www.w3.org/Style/CSS/
[5]http://searchwindevelopment.techtarget.com/definition/Ajax
[6]https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[7]https://www.w3.org/XML/
[8]https://www.w3.org/DOM/

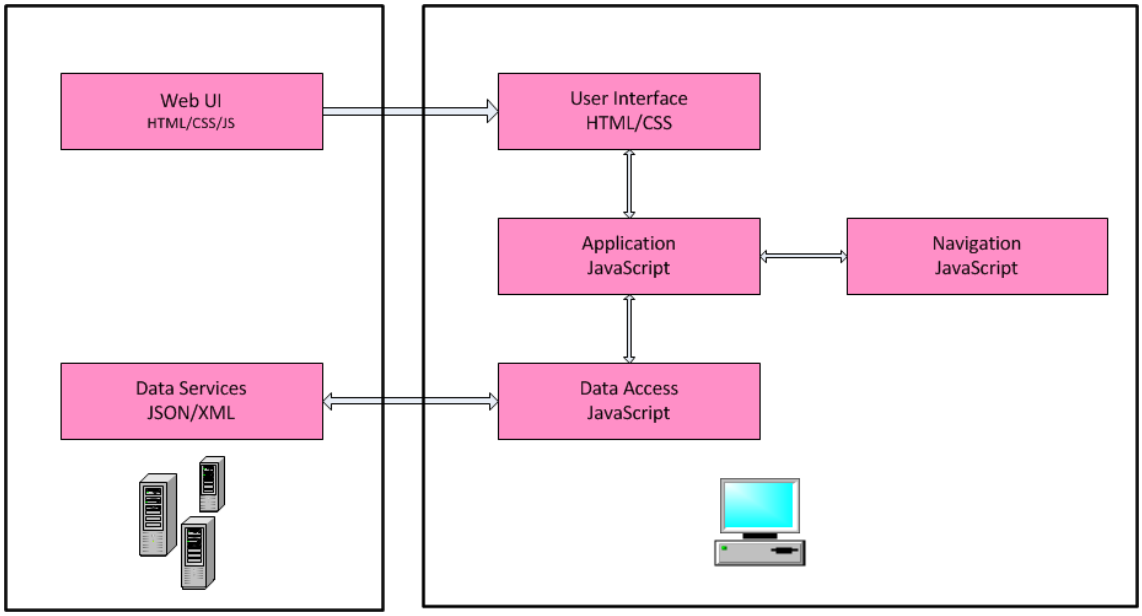

Figure 2.1. *Single-page application architecture.*

In Figure 2.1 above the overall architecture of a single-page application is visualized. Operations can be simply split into two, the server-side on the left and the client-side on the right. After the server serves the initial user interface parts (HTML, CSS and JavaScript), all computation is done in the user's browser, except for additional data fetches to the server. Since the interface manipulation and routing is done on the client-side, less computation power is required from the server compared to a traditional web application.

2.3 JSON

JSON[9] (JavaScript Object Notation) is a popular format used to transmit data objects as attribute-value pairs. JSON is currently described in two standards, RFC 7159 and ECMA-404. While describing the same format, these standards diverse slightly. The differences of the two standards go out of the scope of this thesis, but it can be generally said that the ECMA-404 standard only describes the allowed grammar syntax, while the RFC standard has more in-depth views about other aspects, e.g. security [9]. JSON consists of two stuctures; collections of name/value pairs and ordered lists of values. For values JSON has a support for multiple well-known primitive data types,

- `Number`
- `String`

[9]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

- `Boolean`
- `Array`
- `Object`
- `NULL`

The JSON standards allow an arbitrary amount of whitespace characters which will be ignored outside of Strings, this enables the possibility of minifying JSON files. [10]

2.4 JavaScript and ECMAScript specifications

JavaScript is an implementation of the ECMAScript scripting language specification [11]. These two names are interchangeable, but the use of both is necessary, since Oracle owns the trademark for JavaScript [12].

2.4.1 JavaScript

JavaScript is a dynamic, untyped and interpreted programming language which is standardized in the ECMAScript language specification. JavaScript is the most essential programming language in creating content for the World Wide Web (WWW), hence most of todays websites and web applications use it. This is mostly because of the fact that all modern browsers support it without the need of plugins or other add-ons. [13]

JavaScript engines have improved over the years, which has increased the use of JavaScript in server-side applications also. Most notably the JavaScript engine powering Google Chrome and Chromium, V8, has been integrated to an open-source, cross-platform runtime environment called Node.js [14]. Node.js uses the Google V8 and contains built-in libraries for networking and other I/O operations, which enables it to work as a web server and compete with traditional software like Apache[10], Nginx[11] and IIS[12]. Node.js is a asynchronous event driven framework which allows multiple connections to be handled concurrently. Because of this the programmer can develop scalable systems without dealing with normal concurrency problems such as dead-locking. Popularity among especially high traffic sites is visible in Figure 2.2 below showing the market positions of the most popular web servers.

[10]http://www.apache.org/
[11]http://nginx.org/
[12]https://www.iis.net/

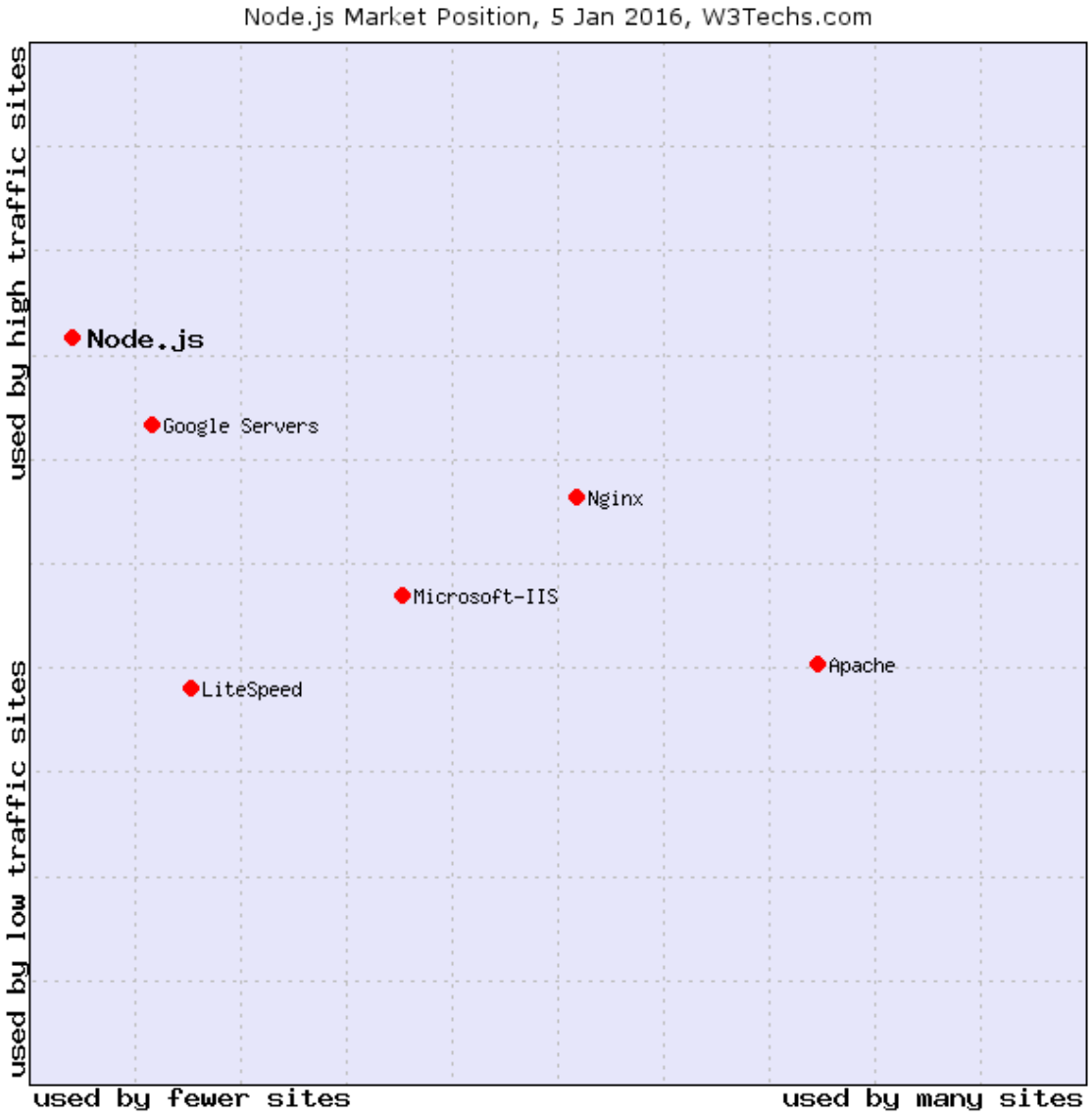

Figure 2.2. *Node.js market position.*

Although the number of sites adopting Node.js is small compared to its competitors, its popularity can be clearly seen among sites with high traffic. Examples of large corporations using Node.js are LinkedIn, Paypal and eBay [15].

The rise of Node.js' popularity has brought the web development to a situation where one language can be used both on the server-side, and the client-side. This brings a synergy benefit to development, compared to having to use different languages for each side. Now a developer can only learn one language, and use the same tools for creating full web applications.

2.4.2 ECMAScript standards

ECMAScript standards are handled by a technical committee called TC39[13] (Ecma Technical Committee 39). They are a committee with the goal to evolve and improve JavaScript. It consists of companies which send their representatives to the regularly held meetings.

Most modern browsers support ECMAScript 5 (ES5) which is a standard finalized in December 2009 [16]. In June 2015 a new standard was approved by the name of ECMAScript 6 (ES6), also known as ECMAScript 2015. ES6 brings multiple desirable improvements to the language which developers have long been waiting for. ES6 is a superset of ES5, which means switching to use ES6 will not break existing code. This in other words means, that no one is forced to use the new features introduced in ES6, if they do not wish to do so. To further enhance the development of the ECMAScript specification and Javascript language, the technical committee has decided to start using time-box releases starting with ES7 (ECMAScript 2016), which means a new version will be released every year with whatever features are ready at that point.

In this thesis ES6 is used because of all the new features it introduces. Since the browser adoption of ES6 is still in mid-way [17], a transpiler called Babel is used to convert written ES6 code to ES5. Transpilers are commonly used to get the best out of both worlds, to be able to write according to the newest standard, and at the same time maintain a wider browser compatibility. Transpilers are compilers which the written code is processed through, which then outputs the same functionality using an older version of the language. [18]

[13] http://www.ecma-international.org/memento/TC39.htm

3. REACT.JS

React.js[1] is a JavaScript library for creating user interfaces (also referred to as UI). The library is created and open-sourced by Facebook and is currently in version 0.14 at the writing of this thesis. It has been announced [19] that as of the next version, React will change its versioning scheme to use major revisions, starting with the next in order minor number as a major. This means that the next released version of React will be 15.0.0. It is acknowledged that this does not follow the traditional semver[2] numbering, but is done for future clarity.

React can be considered as the View in a traditional Model/View/Controller[3] architecture, and the UI consists of so called components. React embraces the single responsibility principle strongly, all components should handle one single thing, and the components should be contained into larger components containing subcomponents.

In this chapter the user is given an overall idea about what components are, and how they should be used and organized to create a user interface. The basic semantics of a component are introduced, what kind of methods the component must have, what kind of lifecycle methods are available for convenience and how data should be passed from a component to another. Lastly the Virtual DOM and approximations used by React are introduced as a reasoning on why the ideology of re-rendering parts of the DOM is a powerful and efficient way of creating a web application.

3.1 Components and JSX

The main idea of React is to break down the user interface into small reusable and composable components which present data. A component holds markup and the corresponding view logic tied together, compared to the traditional, separation of templates and display logic. A good example of an alternative to React's components is Angular, which generally embraces the traditional way using HTML markup and

[1] https://facebook.github.io/react/
[2] http://semver.org/
[3] http://whatis.techtarget.com/definition/model-view-controller-MVC

separate controllers for the logic [20]. That said, it is also possible to create so-called components (directives in Angular terms) in Angular. These directives are a combination of display logic and a string template, but they are not generally used for structuring applications to the same extent as in React. As component-based web applications grow more popular, this ideology has been also suggested to be used with Angular [21; 22].

In a post reasoning on why Facebook created React [23], three points are mentioned to support the use of components over templates:

- JavaScript is a flexible, powerful programming language with the ability to build abstractions. This is incredibly important in large applications.
- By unifying your markup with its corresponding view logic, React can actually make views easier to extend and maintain.
- By baking an understanding of markup and content into JavaScript, there's no manual string concatenation and therefore less surface area for XSS vulnerabilities.

The view can be represented either by using raw JavaScript or an extension called JSX which is explained in detail in a subsection 3.1.4.

A React component should always have a `render()` method, which returns a lightweight representation of the view. This lightweight representation of the component can be used to determine how the component possibly changed compared to a previous render, and only the required parts need to be updated. This of course does not apply if the component is rendered for the first time. This comparison is done using a technology called Virtual DOM and a collection of approximations that improve the performance.

3.1.1 States and Props

React has two types of model data, `state` and `props`. These two provide different functionalities, props are the attributes passed from a parent to a child component, whereas state is for storing data that changes over time when the component is interacted with.

`Props` are an abbreviation of properties. They are attributes that are passed to React components, and they can consist of values and callback functions. As seen especially in subsection 3.1.3 `props` are a vital part of organizing a React application which utilizes the division of components into container components and presentational components. The importance of `props` seems even greater with

the introduction of functional components which are discussed in subsection 3.1.2. Despite the dynamic nature of JavaScript, components have a way of determining types for properties with the `propTypes` object. `React.PropTypes` exports a collection of validators which can be used to validate the property data passed from the parent component. If a property which does not match the mentioned type is found, a warning is printed into the browser console. The list of available validators include e.g. checking if the property is a JavaScript primitive, a renderable node, an array of certain type or a function. By default all the declared validations are checked only if the specific property is present. If required, the validator can be chained with a `isRequired` property to give a warning if the property is missing. Additionally custom validators can be created, and they should return an `Error` if the validation fails. [24]

`state` is the internal state of the component, which can be initialized with the `getInitialState()` function or set by calling the `setState()` method. It is important that state is never directly mutated, but the `setState()` -method is always used. The `setState()` method can take only a partial state as a parameter, which is then merged into the full state object. After the state has been changed, it triggers a series of lifecycle methods to be invoked, and eventually a re-render is possibly executed. The lifecycle methods and optionality of the render is discussed further in section 3.2. As discussed later on, the trend in structuring a React application is to prevent the use of state in components where it is not necessary, and try to keep components stateless. This is done foremost to prevent redundancy, but also to keep the application more readable and easier to reason with. [25] The division of what should be put in to `state` is subjective, but Facebook offers a good baseline for this, "State should contain data that a component's event handlers may change to trigger a UI update." As for what should not to put in the state the list is longer. The points mentioned by Facebook are listed below,

Computed data

Data that can be computed from `state` should not be stored in `state`. An example of this would be storing both a list of items and its length in the `state`. This is unnecessary duplication and the amount should be calculated based on the array in the `render()` method.

React components

Components should not be stored in state, they should be build in the `render()` method based on the components `props` and `state`.

Duplicated data from props

Props should be considered the source of truth when possible so they should

not be duplicated in to the `state`. One valid use case for this is when the previous values of a certain property need to be stored.

To conclude this, the state of a component should contain the minimal amount of data needed to present a components UI's state. [25]

3.1.2 Functional Components

Components in React can be declared either as classes or functions. Functional components lack some features that are present for class components, most notably lifecycle hooks. They are generally meant to be used for simpler components which do not require an internal state. These components ideally comprise a large portion of applications. Components declared as functions have not been available from the initial release of React, but were introduced in version 0.14. In the release notes it is stated that "This pattern is designed to encourage the creation of these simple components that should comprise large portions of your apps. In the future, we'll also be able to make performance optimizations specific to these components by avoiding unnecessary checks and memory allocations." [26] This indicates that functional components will be a vital part of the React ecosystem in the future. In the snippet below we can see a simple example of a component declared as a function:

```
const Header = (props) => {
  const { greeting } = props;
  return (
    <div>
      <Greeting>{greeting}</Greeting>
      <Logout />
    </div>
  );
};
```

As seen above, functional components take a single parameter, `props`, which are the properties passed to the component. With the given props the component returns a JSX expression. Functional components are pure functions which are also idempotent. This makes testing a component easy since the same input always produces the same output, since the component has no internal state present, and the output relies solely on the input.

3.1.3 Composition

As web applications normally consist of complex user interfaces, the UI needs to be split into components that are used as building blocks for larger components. When deciding how to split the user interface into components it is important to keep the single responsibility principle [27] and separation of concerns [28] in mind. When components are small, have a certain responsibility and are encapsulated, they make code reusable and are more easily testable.

Below in Figure 3.1 we have an example application, where the user can input a keyword, and search for images related to it. Under the search bar results are listed in a grid formation.

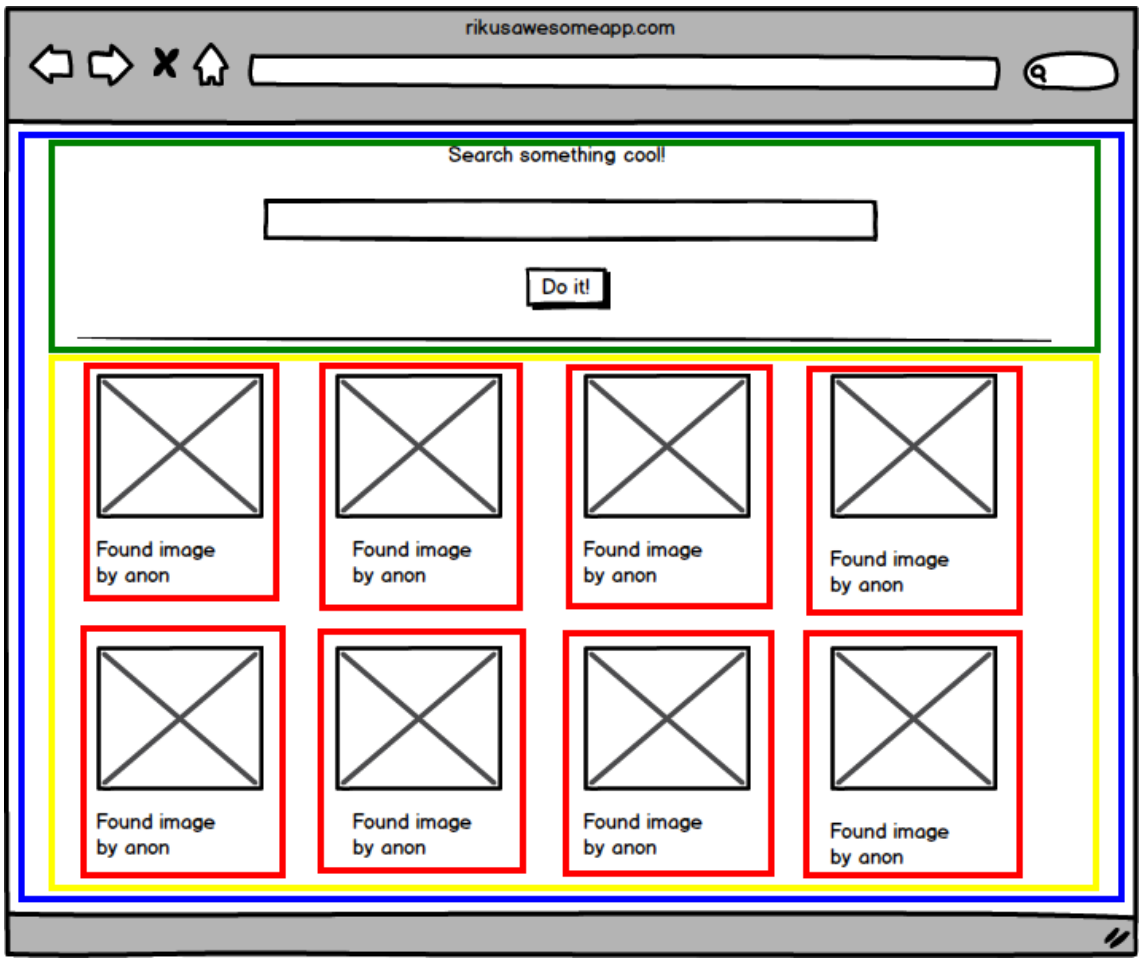

Figure 3.1. *Example user interface split into components.*

The UI can be divided into four different components,

- ImageSearch (blue): The main component which holds all the subcomponents shown in the UI.

- SearchBar (green): The search component which receives the users input and initiates a new search.
- ImageGrid (yellow): A component which contains the result items.
- ResultItem (red): A component for a single result item. This component is repeated for all the results.

Now the UI is divided into smaller components, and the `ResultItem` component can even be reused for all the received result items. At the same time as we split the UI into components, we achieved a hierarchy for them. The `ImageSearch` component has two children, `SearchBar` and `ImageGrid`. `ImageGrid` has an arbitrary amount of `ResultItem` children depending on the search result.

As the user interface is divided into components in the aforementioned way, a so-called pattern can be distinguished. These components can be categorized into container components and presentational components. This is not a pattern originally emphasized by Facebook's React team, but has risen popular among the community. It can be generalized that this method refers to the same idea as Fat and Skinny, Stateful and Pure or Smart and Dumb components. The pattern can be simplified to mean that containers fetch data, which is then rendered in the container's subcomponents. As Dan Abramov stated in his article about the pattern [29], presentational components are concerned with how things look, when container components are concerned with how things work. [29; 30]

In Figure 3.1, we can see that the `ImageSearch` component is a container, whereas the others are representational components. Now that the data fetching, fetching images related to the keyword in this case, is handled by the container, the `ImageGrid` can be reused in any parts of the application that might require displaying of search results. This is opposed to the option that `ImageGrid` handles data fetching, which would mean the component is tied down to the specific data source. An easy reusability example of this is using the component in both displaying the search results, and items the user possibly added as favorites. If the surrounding container handles fetching the data and passes them to `ImageGrid`, the component becomes totally independent of the source of data, as long as the `ResultItem` format is the same.

There are multiple benefits in using this pattern in a React application, which are stated in the list below,

- Firstly, this embraces the aforementioned separation of concerns principle since the data fetching and presentation are split into separate components. This improves the readability of the UI code.

- Secondly, this improves the reusability of your components as explained above. The same component can be used to present data from different data sources.
- Thirdly, as the presentation is separated from the actual application logic, people with knowledge about basic HTML markup and styling (e.g. designers) can contribute to the UI without in-depth understanding of React.

In conclusion, dividing components into container components and presentational components makes the application more readable and greatly improves reusability of code.

3.1.4 JSX

JSX is a JavaScript syntax extension that can be used with React to define tree structures with attributes. It has a familiar XML-like syntax, which makes it easy to adopt. The familiarity of JSX is one of the points Facebook makes when reasoning on why to use JSX [31]: "It's more familiar for casual developers such as designers." Another reasoning is the benefit of balanced opening and closing tags, which increases readability compared to function calls or object literals. As JSX is just an extension, and is transformed into JavaScript, it's use is not required. When JSX is used, a transpiler must be a part of the build flow as the code needs to be compiled to JavaScript in order to be executed in the browser.

A valid JSX expression always evaluates as a `createElement` function call which is part of the React core API [32]. The function takes three types of parameters. First a HTML tag name as a string, or a `ReactClass`. This infers the type of the element that is to be created. Second argument is an object containing all the properties that are passed to the newly created element, or `null` if none are present. After the arguments mentioned previously, the function can have an arbitrary number of `createElement` function calls as arguments, representing the elements children. In the snippet below a simple JSX transformation to JavaScript is shown. For improved readability only the returning of the JSX expression and the `createElement` returning is shown.

```
return (
 <div className="container">
   <Head style={{color: "blue"}}>Im a title for a list</Head>
   <ul>
     <li>{item.getFirstName()}</li>
     <li>{item.getSecondName()}</li>
   </ul>
```

```
    </div>
  );
```

In the snippet above there are three JSX specific features to take notice of. First, the surrounding `<div>` element has an attribute `className`, instead of the traditional `class` attribute used in HTML. This is a convention used since JSX is actually JavaScript, and `class` is a reserved word in JavaScript. Secondly the style attribute is passed as an object instead of a string which is normally used in inline styles. The other curly braces present a JavaScript expression inside of the JSX expression. The same curly braces are used in the unordered list. `item.getFirstName()` simply calls the `getFirstName()` function of `item`, and the return value is shown as the child of the list item. In the snippet below the same code is shown in JavaScript:

```
return React.createElement(
  "div",
  { className: "container" },
  React.createElement(
    Head,
    { style: { color: "blue" } },
    "Im a title for a list"
  ),
  React.createElement(
    "ul",
    null,
    React.createElement(
      "li",
      null,
      item.getFirstName()
    ),
    React.createElement(
      "li",
      null,
      item.getSecondName()
    )
  )
);
```

As visible in the code above, the JSX expression is declared in a single JavaScript function. As stated before, the `createElement` functions align in the same hierarchy as the HTML elements did in the JSX example.

Now when comparing the JSX and native JavaScript versions, it is visible that the JSX expression is more clearly readable and less verbose than the same presented in native JavaScript functions. The main advantage gained by writing React components as native JavaScript functions is the fact that then the code is in fact already JavaScript, and an extra build step (JSX to JavaScript compilation) is not needed. Because of its less verbose syntax JSX is used as the primary way of expressing React components in this thesis, even though the community has a lot of differing opinions about the matter. [33]

3.2 Component creation and lifecycle

In this section React component specification and lifecycle methods are explained. This includes general component methods and objects that are part of the React component specification, and special methods that are called in certain parts of the components lifecycle.

3.2.1 Component specification

Component creation is possible in three different ways. These ways are; invoking `React.createClass()` method, as a plain JavaScript class that extends `React.Component` or as a stateless function. For the two first ones the only required method to be implemented is `render()`. [34] The method should utilize `props` or `state` and return a single child element. The returned element can either be another React component, or a virtual representation of a DOM element. As an exception to returning a child element, the method may also return `null` or `false`. This is done to imply that nothing should be rendered. The return value of a stateless function has the same aforementioned requirements, with the exception that stateless functions only receive `props` as a parameter and no `state` is present.

As like a stateless function, the `render()` method should also be pure, meaning the function should not modify component state or have other side-effects. Other methods that are a part of the component specification are listed in Table 3.1 below,

Table 3.1. React component specification.

Method name	Description
getInitialState()	This method is invoked once before the component has been mounted. The method should return the initial value of the components `state`. This method is not available when creating a component using the JavaScript `class` syntax. In those cases the initial state for the component should be set in the classes constructor.
getDefaultProps()	This method maps default values for `props` that are missing when the component is created. If the property value is specified by the parent component, that value takes precedence.
propTypes	This object is used to validate props that are being passed to the component when the application is executed in development mode.
mixins	`Mixins` is an array, which contains common functionality among components, mainly if multiple components require similar functionality in multiple lifecycle methods. This property is only available when using the `React.createClass` method, and is not going to be implemented for the `class` syntax. A new compositional API is being designed to make common operations more simpler to implement. [35]
statics	This object allows the defining of static methods which can be called without creating an instance of the component. This also means that these methods do not have access to `props` or `state` because of their static nature.
displayName	`displayName` is a string value which is displayed in debugging messages. If the value is `undefined`, React uses the name of the component in debug messages.

It should be noted that only `propTypes` of the above is present when creating a component as a stateless function. Stateless function components also have a `defaultProps` property for settings default property values.

3.2.2 Lifecycle methods

React components have methods that are executed at certain points of a components lifecycle. The lifecycle methods can be grouped into three different categories; mounting, updating and unmounting. The available methods and their category can be found in Table 3.2 below.

These methods above should be used to handle actions that are tied to a certain point of a components lifecycle and to increase performance when `props` are changed without the actual need of a re-render. It should be noted that a component created with a stateless function does not have any of the aforementioned methods. [34]

3.3 Virtual DOM

DOM manipulation is a slow operation, which can ruin the performance of an application. This is why React resorts to a technology called virtual DOM. The idea of virtual DOM is to have an in-memory representation of the DOM in plain JavaScript. Now, when a change needs to be done to the DOM, using efficient difference algorithms the smallest change can be calculated and all the information on the screen does not necessarily need to be re-rendered. Virtual DOM is not created by Facebook, and it can be used in other libraries and frameworks also. [36]

Calculating the least number of modifications between two trees is an $O(n^3)$ problem [37], but with React's approximations Facebook claims this can be reduced to a non-optimal $O(n)$ operation. This is based on two assumptions, which Facebook lists in the React documentation:

- Two components of the same class will generate similar trees and two components of different classes will generate different trees.
- It is possible to provide a unique key for elements that is stable across different renders.

These two cases can be split into a pair-wise difference, where two nodes are compared to each other, and a list-wise difference where a change in a list of nodes is examined. [38]

3.3.1 Pair-wise difference

In the pair-wise difference two nodes are compared, and if their node types are different, an assumption is made that they have different sub-trees. Because of this

Table 3.2. React component lifecycle methods.

Method name and category	Description
componentWillMount() (mounting)	This function is invoked one single time, immediately before the initial render is done. If `state` is manipulated in this method, the component is still rendered only once.
componentDidMount() (mounting)	The function is invoked once, directly after the initial render. A child components `componentDidMount()` function is invoked before its parents, so a component's children are available when this function is invoked. Interacting with the DOM or sending AJAX requests to the server should be done in this method.
componentWillReceiveProps(nextProps) (updating)	This function is invoked when the component receives new `props` from its parent, except on the initial render. At this point both the old and the new properties are available so possible calculations needing both of these can be done in this method. If the `state` is changed in this method it will not cause an additional render.
shouldComponentUpdate(nextProps, nextState) (updating)	Invoked when new `props` or `state` is being received. This is invoked before the component is rendered, and acts as an opportunity to prevent the component from re-rendering. Since this method is used to optionally prevent a re-render, this is not invoked on the initial render.
componentWillUpdate(nextProps, nextState) (updating)	Invoked before re-rendering after receiving new `props` or `state`. This method is intended to execute possible preparations before the update. It should be noted that `state` cannot be changed in this method.
componentDidUpdate(previousProps, previousState) (updating)	This method is invoked after the component changes are updated. This method is not called on initial render.
componentWillUnmount() (unmounting)	Invoked before a component is removed from the DOM. This method is intended to be used to perform any possible cleanup that is required when the component is removed.

assumption, no further calculation is required, the old node can be removed and the new one inserted. This assumption is applied to all kinds of nodes, basic DOM nodes and custom components.

This assumption is easy to reason with. Assuming we have two custom components `<Header />` and `<Menu />`, and the first one is being replaced by the later one. It is unlikely that the sub-tree of `<Menu />` will look the same as the sub-tree of `<Header />` so further calculation should be avoided. On the other hand, if a `<Header />` is replaced with `<Header />`, the structure is worth examining because similarities are likely to be found.

If the node types match, there are two possible actions depending on whether the nodes are basic DOM nodes or custom components. If matching basic DOM nodes are found, their attributes are compared and the old ones are replaced with the new ones. In case of a custom component attributes cannot be simply replaced with new ones since the components might store a state. In this case the attributes of the new component are taken, and passed to the old component by a lifecycle method called `componentWillReceiveProps()`. Now the changes have been applied to the component and the difference algorithm can continue on.

3.3.2 List-wise difference

When comparing two lists, React iterates over both of the lists at the same time and creates a mutation when a difference is found. Difference calculations are shown in the snippets below,

```
1. : <ul><li>first<li/></ul>
2. : <ul><li>first<li/><li>second<li/></ul>
=> [insertNode <li>second<li/>]
```

On the first line is the original list, the second line has the new list and the third line has the mutation operations that are required. This works, but problems occur if children are moved to a different place in the the list, or new items are placed in the beginning of the list.

```
1. : <ul><li>first<li/><li>second<li/></ul>
2. : <ul><li>second<li/><li>first<li/></ul>
=> [replaceAttribute textContent 'second'],
=> [insertNode <li>first</li>]
```

As shown in the snippet above, the insertion of a new item requires two mutations, even though clearly it could be handled with just one operation. To solve this React

has introduced a new attribute that can be added to all of the items in a list. A unique key is given to all of the items and it is used to do the matching of existing items. Facebook claims that insertion, deletion, substitution and moves are now possible as an O(n) operation using a hash table.

```
1. : <ul><li key="abc">1.<li/><li key="def">2.<li/></ul>
2. : <ul><li key="def">2.<li/><li key="abc">1.<li/></ul>
=> [insertNode <li>2.</li>]
```

Now the operation is done in one mutation because of the use of keys. A key can be something as simple as an id that is already assigned to the element on the server, or a hash of its content. Even though it was previously said that keys have to be unique, it should be noted that they only have to be unique amongst their siblings, not globally. With these approximations and virtual DOM, doing repetitive re-renders can be used as an efficient way of creating a web application. [38]

4. DATA FLOW HANDLING

As applications grow, the data flow must be handled in some way. The problem occurs when separate parts of the application rely on the same source data. If the data is changed in one part, how should the other part be notified. Facebook introduced an architecture called Flux[1] to solve this problem. The overall idea was well received, but multiple implementations occurred. In this section the original Flux idea is introduced, along with a popular library called Redux which evolves the ideas of Flux.

4.1 Original Flux idea

There are three main parts in a Flux application; the dispatcher, stores and the views. In this division the views are React components. Even though there is a slight resemblance, this does not directly map to the traditional Model-View-Controller pattern. Flux has so-called "controller-views", which are often the container components mentioned in the previous chapter. Even though the Flux documentation speaks of "controller-views", in this thesis they are referred to as container components in order to be consistent. These components fetch data from the stores and pass it on to their child components. The state of the application is stored in the `stores`, from where the data then flows to the `views`. This means various parts of the application are highly decoupled, as all data originated from the same source. Below, in Figure 4.1, the basic data flow in a Flux application is shown.

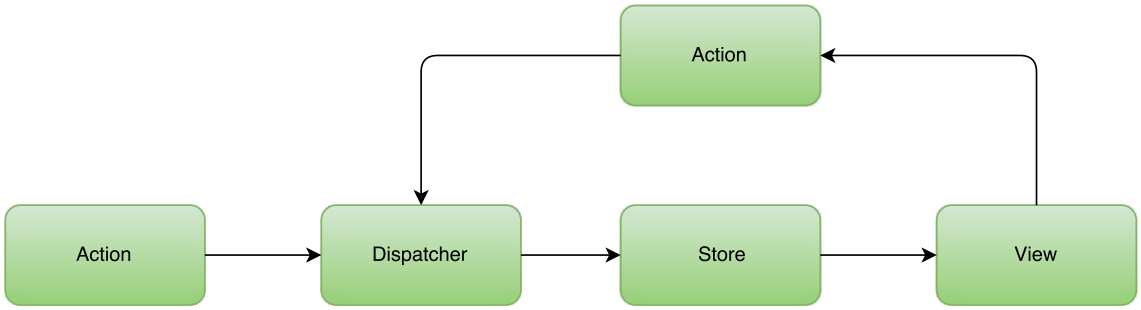

Figure 4.1. *The Flux architecture.*

[1]https://facebook.github.io/flux/

In the Flux documentation [39] this is told to be "the primary mental model for the Flux programmer". In this model the `dispatcher`, `stores` and `views` are clearly separate independent modules, which have clear inputs and outputs visualized by the arrows. `Actions` are objects which hold the new data, and a `type` informing how the action will change the application state. As the figure states, `actions` can be emitted from the `views` as a result of an user interaction, or from certain points of the applications lifecycle, eg. after initial start up. In the following subsections the different parts of the Flux architecture are explained more in-depth.

4.1.1 Dispatcher

There is a single `dispatcher` present in a Flux application. All data in the application passes through the `dispatcher` and the `dispatcher` can be thought of as a collection of callbacks to various `stores`. These callbacks are called when the `dispatcher` is invoked in an `action creator` method. In larger applications `stores` might have dependencies to other `stores`, which the `dispatcher` then handles. The `dispatcher` can invoke callbacks in a certain order, and `stores` can declaratively wait for another `store` to finish updating before executing its own callback. This makes it easier for `stores` to depend on each other.

4.1.2 Stores

The application state is located in `stores`. An application can have multiple `stores`, which all hold the state of a particular domain in the application. Clearly separate parts of the application should be split into their own `stores` for clarification. The Flux documentation [39] contains a good real life example of `store` division. It refers to Facebook's Lookback Video Editor[2] where a user could see a video collage of images they had posted on Facebook in the past. In this small Flux application there was a `TimeStore`, which kept track of the playback position and the state of the playback. In addition to this, there was a `ImageStore` which had the responsibility of keeping track of an collection of images to be shown in the video. It is not required for an application to have multiple `stores` if the data domain is narrow.

As stated in the previous section, a `store` registers itself with the `dispatcher`, providing it with a callback. The callback gets an `action` as a parameter along with a possible data payload. The `store` then acts according to the type and the payload, changing its internal state. After a `store's` state has changed, it

[2]https://www.facebook.com/lookback/

broadcasts an event notifying all listeners that the state has changed. After receiving the notification, `views` know to ask for the new state and update accordingly.

4.1.3 Views

As previously stated, container components (or controller-views as stated in the Flux documentation) are a certain type of top-level `view`, which have child components. They differ from normal `views` with their task of data interaction. These container components listen to `stores` for updates, fetch the new data after it is available and then pass it to their children as `props`. It is also these components which initiate an `action` to be sent to the `store`, e.g. after a user interacts with the UI.

Even though these components usually exist on the top-level of the component hierarchy, sometimes it can be necessary to have container components in the lower parts of the component hierarchy. This is often done in order to wrap a certain domain of the application on its own. This introduces a new entry point of data in to the component tree, which makes the application harder to reason with in general since it deviates from the traditional application structure.

4.1.4 Actions

To trigger a dispatch to a store, a method in the `dispatcher` must be invoked with an `action` as payload. The `action` must contain a `type` attribute telling the type of the `action`, along with an arbitrary amount of data to be processed with the `action`. The following snippet displays a simple `action` object,

```
{
  type: 'ADD_TODO_ITEM',
  text: 'Buy some milk',
  priority: 5,
}
```

The type of the `action` is usually a string literal, and the purpose of it is to help the `store` determine what to do with the accompanied data. These `actions` are mainly created by `views` when the user interacts with the UI, but can be also created as a result of other outside interactions eg. a message from the server. Lifecycle methods in React components also often create `actions`, to initialize data for example.

When discussing `actions`, the term `action creator` often arises. `Action creators` are helper methods, which are used to create payload for the `dispatcher` and trigger the `dispatcher`. The example below implements an `action creator` for the `ADD_TODO_ITEM` action,

```
function createTodoItem(text, priority) {
  const action ={
    type: 'ADD_TODO_ITEM',
    text,
    priority,
  };
  Dispatcher.dispatch(action);
}
```

The function creates an object containing the data received as a parameter, along with the type of the action. After creating the object the function provides the `dispatcher` with the `action`.

4.2 Redux

Redux is a Flux implementation created by Dan Abramov while working on a React Europe conference talk called "Hot Reloading with Time Travel". The goal was "to create a state management library with minimal API but completely predictable behavior, so it is possible to implement logging, hot reloading, time travel, universal apps, record and replay, without any buy-in from the developer". The library has been very widely adopted in use, and even Facebook used it when creating a conference application for their 2016 F8 developer conference [40]. Because of its popularity, and adoption amongst large companies including Facebook, it can be said that Redux is the de facto Flux implementation at the writing of this thesis. It should be explicitly pointed out that Redux is a Flux implementation, and it evolves the ideas of Flux rather than re-inventing them. This means, that variations are often small, but have a larger impact on day-to-day usability of the library. [41]

Below in Figure 4.2 the overall architecture of Redux is visualized in the same way as Flux was in Figure 4.1

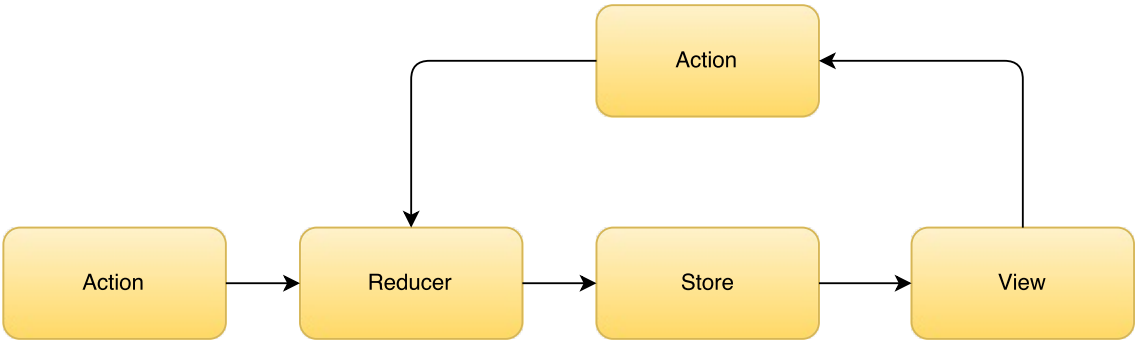

Figure 4.2. *The Redux architecture.*

As seen from the figure, Redux and Flux look a lot alike. The most notable deviation between these two architectures is that Redux is missing a `Dispatcher`, but instead has a `Reducer` section in the flow chart. `Reducers` are pure functions which describe how actions effect the application state. This will be explained in more detail later on in the section. In this section the basics and principles of Redux will be introduced thoroughly, and compared to their counterparts in Flux when possible.

4.2.1 Three principles of Redux

Redux has three basic principles it can be described with [41] :

Single source of truth

The state of the application is stored as a object tree. The object tree is located within a single `store`.

State is read-only

The state is read-only, which means that the only way to change it is to initiate an `action`. All dispatched `actions` are collected in to a centralized place and the state is mutated in a predictable order. This prevents subtle race conditions and other potentially bugs that are hard to find. This complements the original motivation behind the library, as this makes it possible to implement logging, storing and replaying `actions` for debugging or test purposes.

Changes are made with pure function

Special functions called `reducers` are written to describe how an `action` alters the application state. The functions must be pure functions which take the previous state and an `action` as a parameter, and return the next state. As stated in the previous principle, it is important to remember to return a new state object, not mutate the previous state.

4.2.2 Actions in Redux

`Actions` in Redux are plain JavaScript objects, same as in traditional Flux. They are the only type of information sent to the `store`. Like in Flux, Redux actions require a `type` attribute declaring the type of the `action`, and informing how the action should mutate the store. Additionally to the type, actions can also have an arbitrary amount of data which will be used to alter the application state.

`Action creators` are also used in Redux, but with a slightly varying meaning compared to traditional Flux. In Redux `action creators` do not directly initiate a dispatch, they just create a JavaScript object containing the type of the action and additional payload information. In Redux this object is then given to a `store.dispatch()` function which then dispatches the `action` to the `store`. It should be noted that `bound action creators` are possible in Redux, these are special action creators which directly initiate the dispatch.

4.2.3 Reducers

When actions describe that the something happened in the application, reducers describe how that affects the state of the Redux application. This is done by a pure function which takes in the previous state and an `action`, and returns the next state. The fact that the reducer must be a pure function is strongly emphasized in the documentation, which also lists the following things which should never be done inside a `reducer`,

- The functions arguments should never be mutated
- Side effect should never be performed, e.g. API calls
- Non-pure functions, e.g. `Date.now()`, should never be called

As the `reducer` should handle more that one action, a `switch-case` statement is often used to execute different functionality for each `action`. The snippet below defines a basic `reducer` for the `ADD_TODO_ITEM` action we introduced in the previous section.

```
function exampleApp(state = initialState, action) {
  switch(action.type) {
    case ADD_TODO_ITEM:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
```

```
            {
              text: action.text,
              priority: action.priority
            },
          ],
        });
      default:
        return state;
    }
  }
```

There are a few points in the snippet above which should be noted. Firstly the function utilizes ES6's default parameter feature. This means if the state parameter is missing, or `undefined`, the state is set to an initial value. In this example the `initialState` object could be as follows,

```
const initialState = {
  todos: [
    {
      text: 'A starting item!',
      priority: 5,
    },
  ],
  priorityFilter: 0,
}
```

This works as an initial state when the application is run for the first time, or if the state is missing because of an error situation. Another noteworthy feature to prevent error situations is the `default` clause in the `switch-case` statement. This way if the `action` type does not match any known types, or is missing, the given state is returned without alteration.

The object that is returned from the function is created with the `Object.assign()` function. In the snippet a new object is created, the previous state is copied to the new object, and the `todos` attribute is then over-written with new data. The new state has a list of `todos`, which contains all the todo-items from the previous state, appended with the new item got as payload. Once again, the previous set of items is not mutated, but they are returned in to a new array using another ES6 feature, the spread operator.

As Redux only has a single store, different parts of the application state can be assigned their own reducer. This is called `reducer composition` and it is

a fundamental pattern in Redux applications. This provides the possibility to encapsulate various parts of the state, and make the application structure more clear. If inspecting our example state object above, an example of a division would be to create separate `reducers` for the `todos` and the `priorityFilter` attributes. After the reducers have been created, Redux offers a utility function called `combineReducers()` which seemingly combines the separate `reducers` in to single `reducer` for easier use.

4.2.4 Store in Redux

As stated previously, unlike in traditional Flux, in a Redux application there is only one `Store`. The separate data domains can be split into their own modules by using `reducer composition` which was introduced in the previous section. As listed in the Redux documentations, a `store` has the following responsibilities:

- Holding application state
- Allow access to the state by exposing a `getState()` function
- Makes state mutations possible with `dispatch(action)`
- Registers new listeners with `subscribe(listener)`
- Makes possible to unregister listeners by calling the function returned by the `subscribe()` function

A store is created by passing the `reducer` to a helper function called `createStore()`. After the store is created, actions can be passed to the `store.dispatch(action)` method, which then passes the `action` to the appropriate `reducer`.

5. SAMPLE APPLICATION

In this chapter a sample application is created using an open-source project called TodoMVC [42]. TodoMVC is a project that helps developers choose the correct framework for their uses. It offers the same Todo application implemented in multiple JavaScript frameworks. The project has gained major traction in the JavaScript community because of the underlying framework fatigue [43]. The idea of the application is that the functionality and outlook would stay the same even if the framework changes underneath. Figure 5.1 displays a screenshot of the application.

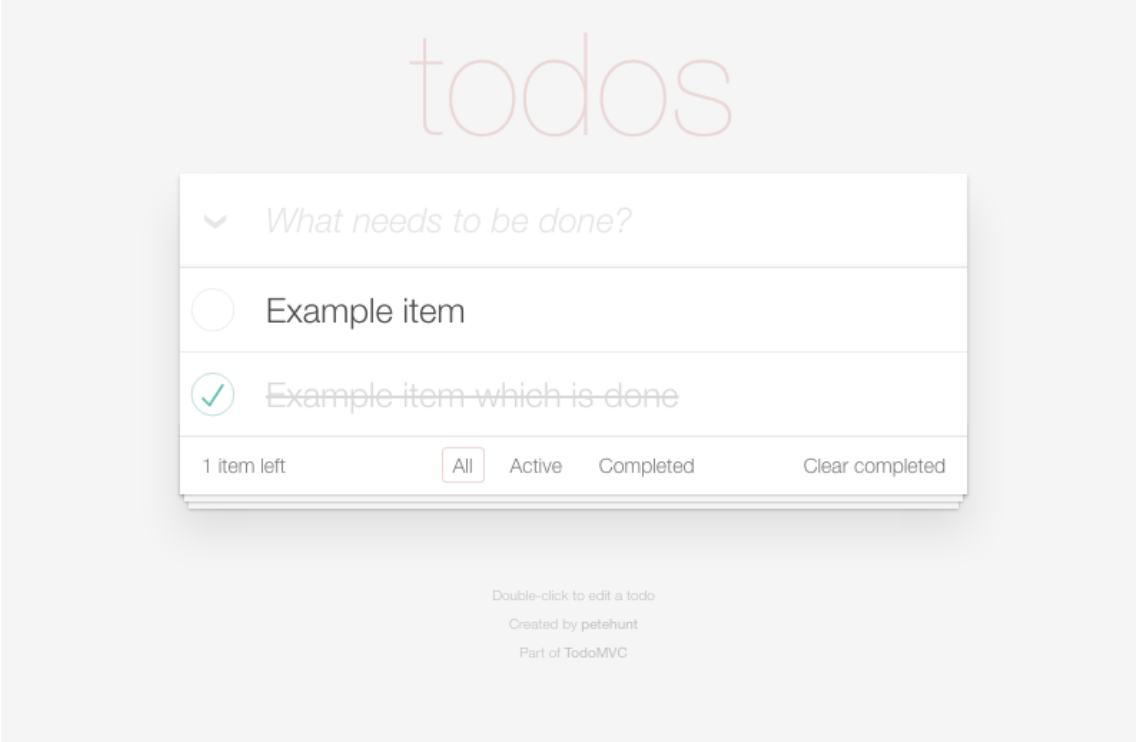

Figure 5.1. *Screenshot of the TodoMVC application.*

Even though the application is simple, and has a rather minimalistic user interface, it offers a comprehensive list of functionalities that are required in real life applications. These features consist of the following functions:

- Adding new items

- Editing items
- Deleting items
- Renaming items
- Toggling item state between complete and active
- Deletion of items that are completed
- A filter to show either all items, completed items or active items

In this chapter both the Flux and the Redux implementations are introduced part by part. Parts with the same type of functionality are compared together.

5.1 Overall structures

The application structures and key parts are displayed for both the Flux and the Redux implementation in the following subsections.

5.1.1 Flux

The Flux version of the TodoMVC application resides in the original Flux repository in Github[1] which is owned by Facebook [44]. The application folder structure has a clear resemblance to figure 4.1, and is showed in the snippet below,

```
./
  index.html
  js/
    actions/
      TodoActions.js
    app.js
    dispatcher/
      AppDispatcher.js
    components/
      Footer.react.js
      Header.react.js
      MainSection.react.js
      TodoApp.react.js
      TodoItem.react.js
      TodoTextInput.react.js
```

[1] https://github.com/

```
  stores/
    TodoStore.js
```

As in the flowchart in figure 4.1, the structure has 4 clear sections, `actions`, `dispatcher`, `stores` and `components` which are the `views` mentioned in the flowchart.

5.1.2 Redux

The Redux TodoMVC application is found in the Redux Github repository. The repository is under an organization called React Community, which also has a collection of other React libraries, e.g. a popular routing library called React Router. The application structure of the TodoMVC application is shown below,

```
./
  actions/
    index.js
  reducers/
    index.js
    todos.js
  components/
    Footer.js
    Header.js
    MainSection.js
    TodoApp.js
    TodoItem.js
    TodoTextInput.js
  store/
    configureStore.js
  index.html
  index.js
```

This also resembles the Redux flowchart in figure 4.2. Both implementations have clear similarities; `actions`, `components` and `store(s)`. Even though these segments might have different implementation details, they can be easily compared together. The largest differentiation comes from `dispatcher` and `reducers`. As both of these work closely with `store(s)` concept, these parts can be combined and compared together.

5.2 Dispatcher and Stores vs Store and Reducer

First, the Flux version is implemented. The `Dispatcher` creation is usually done in a different file, and then exported as a singleton object for other parts to use. The functionality of a `Dispatcher`, for example the `register()` function, is included in the Flux library provided by Facebook and it can be used directly. As the application state is simple, and consists of only one data domain, the application has only one Flux `store`.

```
const AppDispatcher = require('flux').Dispatcher;

let _todos = {};

const TodoStore = Object.assign({},
  EventEmitter.prototype,
  {
    areAllComplete: function() {
      for (var id in _todos) {
        if (!_todos[id].complete) {
          return false;
        }
      }
      return true;
    },
    getAll: function() {
      return _todos;
    },
    emitChange: function() {
      this.emit(CHANGE_EVENT);
    },
    addChangeListener: function(callback) {
      this.on(CHANGE_EVENT, callback);
    },
    removeChangeListener: function(callback) {
      this.removeListener(CHANGE_EVENT, callback);
    }
  });
```

Using the `Object.assign()` function we create a new object, which receives functionality from the `EventEmitter` library, and a group of functions for registering

to the store and listening to changes. The `EventEmitter` is an event library which provides the `store` with the capability of creating and listening to events. Because of the simple nature of the application, the `store` also acts as an in-memory storage. In a larger real-life application data persistency would normally be handled on the server.

After the `store` has been created, we can register it to the dispatcher created previously.

```
AppDispatcher.register(function(action) {
  let text;

  switch(action.actionType) {
    case TodoConstants.TODO_CREATE:
      text = action.text.trim();
      if (text !== '') {
        create(text);
        TodoStore.emitChange();
      }
      break;
    case TodoConstants.TODO_UNDO_COMPLETE:
      update(action.id, {complete: false});
      TodoStore.emitChange();
      break;
    case TodoConstants.TODO_COMPLETE:
      update(action.id, {complete: true});
      TodoStore.emitChange();
      break;
    case TodoConstants.TODO_UPDATE_TEXT:
      text = action.text.trim();
      if (text !== '') {
        update(action.id, {text: text});
        TodoStore.emitChange();
      }
      break;
    case TodoConstants.TODO_DESTROY:
      destroy(action.id);
      TodoStore.emitChange();
      break;
    case TodoConstants.TODO_DESTROY_COMPLETED:
```

```
        destroyCompleted();
        TodoStore.emitChange();
        break;
      default:
        // no action if the case is not met
    }
});
```

The `switch-case` statements use `TodoConstants` which is not declared in the snippet for brevity, but it is a collection of constant string literals defining `action` types. Constants are used to prevent spelling mistakes in `action` types. The register function uses a set of helper functions, `create()`, `update()`, etc, which are used to change the `store` content. The implementation of these is also not shown for brevity, but their task is to alter the in-memory storage `_todos`. After each change `TodoStore.emitChange()` is called to notify all listeners that a change has occurred.

In the following code, the Redux `store` is created, along with with `reducers`.

```
const store = createStore(todoReducer, initialState);

function nextId(state) {
  return state.reduce((maxId, todo) => {
    return Math.max(todo.id, maxId);
    }, -1) + 1;
}

function todoReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        {
          id: nextId(state),
          completed: false,
          text: action.text
        },
        ...state
      ];
    case DELETE_TODO:
      return state.filter(todo =>
        todo.id !== action.id
```

```
        );
      case EDIT_TODO:
        return state.map(todo =>
          todo.id === action.id ?
            Object.assign({}, todo,
            { text: action.text }) :
            todo
        );
      case COMPLETE_TODO:
        return state.map(todo =>
          todo.id === action.id ?
            Object.assign({},
            todo,
            { completed: !todo.completed }) :
            todo
        );
      case COMPLETE_ALL:
        const areAllMarked = state.every(todo => {
          return todo.completed
        });
        return state.map(todo => {
          return Object.assign({}, todo, {
            completed: !areAllMarked
          }));
      case CLEAR_COMPLETED:
        return state.filter(todo => {
          return todo.completed === false;
        });
      default:
        return state;
  }
```

The `store` is created by using a helper function provided by the Redux library. It takes two parameters, the `reducer` function that is going to be used for handling `actions`, and an initial state for the application. The initial state defines the state of the application at startup. The same initial state is also passed to the `reducer` as a default parameter to handle the case where the state is missing when the `reducer` is called. The `reducer` function has a similar `switch-case` statement that was previously implemented in the Flux `dispatchers` register function. Unlike the Flux version of the `switch-case`, the Redux version takes heavy use of JavaScript

methods that can be used to return a new state, instead of mutating the previous state.

As a conclusion, these two parts share a fairly similar implementation, with small differences. The most significant difference is that in Redux the `reducer` must be a pure function and return the next state. In Flux the functions registered to the `dispatcher` change the state of the `store` in a pure or impure way, and then emit a change event to inform listeners.

5.3 Actions

After the `store(s)`, `dispatcher` and `reducer` have been implemented, the application needs `actions` for the components to call. In the snippet below `actions` are implemented for the Flux application,

```
const TodoActions = {
  create: function(text) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_CREATE,
      text: text
    });
  },
  updateText: function(id, text) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_UPDATE_TEXT,
      id: id,
      text: text
    });
  },
  toggleComplete: function(todo) {
    const id = todo.id;
    const actionType = todo.complete ?
        TodoConstants.TODO_UNDO_COMPLETE :
        TodoConstants.TODO_COMPLETE;

    AppDispatcher.dispatch({
      actionType: actionType,
      id: id
    });
  },
```

```
  destroy: function(id) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_DESTROY,
      id: id
    });
  },
  destroyCompleted: function() {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_DESTROY_COMPLETED
    });
  }
};
```

A group of `actions` or `action creators` are defined for the needed functionality. Once again `TodoConstants` is used to prevent typing errors in `action` types. The set of functions basically dispatch an `action` with the correct type and possible payload got as a parameter.

The same is implemented for the Redux application below,

```
export function addTodo(text) {
  return { type: types.ADD_TODO, text }
}

export function deleteTodo(id) {
  return { type: types.DELETE_TODO, id }
}

export function editTodo(id, text) {
  return { type: types.EDIT_TODO, id, text }
}

export function completeTodo(id) {
  return { type: types.COMPLETE_TODO, id }
}

export function completeAll() {
  return { type: types.COMPLETE_ALL }
}

export function clearCompleted() {
```

```
    return { type: types.CLEAR_COMPLETED }
  }
```

In a Redux application `actions` do not dispatch anything to the `store`, they simply return a JavaScript object with the correct type and possible payload. In the snippet constants are just for `action` types to prevent typing errors.

For `actions` the functionality differs slightly. While Flux `actions` already dispatch the `action` to the `dispatcher`, in Redux `actions` return plain JavaScript objects with a type and a payload. These `actions` are then passed to the `store.dispatch(action)` function and handled with the correct `reducer`.

5.4 Component

When all other parts of the architecture are implemented, the components can be created to rely on `store` items. In the snippet below the Flux application is set to listen to changes in the `TodoStore`.

```
function getTodoState() {
  return {
    allTodos: TodoStore.getAll(),
    areAllComplete: TodoStore.areAllComplete()
  };
}

const TodoApp = React.createClass({
  getInitialState: function() {
    return getTodoState();
  },
  componentDidMount: function() {
    TodoStore.addChangeListener(this._onChange);
  },
  componentWillUnmount: function() {
    TodoStore.removeChangeListener(this._onChange);
  },
  render: function() {
    return (
      <div>
        <Header />
```

```
          <MainSection
            allTodos={this.state.allTodos}
            areAllComplete={this.state.areAllComplete}
          />
          <Footer allTodos={this.state.allTodos} />
        </div>
      );
    },
    _onChange: function() {
      this.setState(getTodoState());
    }
  });
```

The content of the `store` is kept in the `state` of the React component, and it is updated when ever the `_onChange()` listener gets called. This way the data passed down to the `<MainSection />` component is always in sync with the `store`.

The application has a component named `<TodoTextInput />` which can be used as an example how the component might call an action. The component is simplified in the snippet below,

```
<TodoTextInput
  id="new-todo"
  placeholder="What needs to be done?"
  onSave={this._onSave}
/>

_onSave: function(text) {
  if (text.trim()){
    TodoActions.create(text);
  }
}
```

The component receives the item name in an input field, and then creates a `create` action for the item name when the `onSave` method is invoked.

The corresponding Redux version uses a high-order function called `connect` which is provided by a library called `react-redux`. The application structure is implemented in the snippet below,

```
class App extends Component {
```

```
    render() {
      const { todos, actions } = this.props;
      return (
        <div>
          <Header addTodo={actions.addTodo} />
          <MainSection todos={todos} actions={actions} />
        </div>
      );
    }
  }

  export default connect(
    mapStateToProps,
    mapDispatchToProps
  )(App);
```

As stated previously, the created application is passed to a high-order function, with two functions as a parameter. `mapStateToProps` maps the state of the `store` to the `props` of the component, so they can be easily accessed. `mapDispatchToProps` binds `actions` and the `store.dispatch` function together and places them in the components `props`. The `action` can now be called directly from the `props` and the `action` is dispatched to the `store`. The content of the `store` and the possibility to create a new item is then passed on to the child components as `todos` and `actions.addTodo`.

Below the functionality of the Flux application is adapted for the Redux example,

```
  <TodoTextInput
    onSave={this.handleSave}
    placeholder="What needs to be done?" />

  handleSave(text) {
    if (text.length !== 0) {
      this.props.addTodo(text)
    }
  }
```

When the `onSave` property is triggered, the handler function calls the addTodo action which was passed as a `prop`. The dispatched action is then handled by the

corresponding `reducer` function.

Listening to the application state is fairly similar between the two implementations. In Redux the dispatching of `actions` and reading state through `props` is the main difference between these two. Also implementation details vary slightly, as in the Flux application an `action creator` is called which then calls the `dispatcher`. In Redux the chosen `action` is directly sent to the listening `store`.

5.5 Evaluation

Overall the research can be thought of as a success. The differences between the two architectures were not significant, but it can be seen that Redux is a newer improvement of the Flux architecture.

Despite the similarities, there were also fundamental differences, as an example the fact that the state in a Redux application has to be immutable. Also in parts where the Flux application needed to emit events, in Redux a helper function was used to achieve similar functionality. Using these kinds of helpers help prevent coding errors. The fact that Redux does not require creating a `Dispatcher` and registering `stores` to it makes the code less verbose, which is important when creating complex systems. The choice of functional composition in Redux compared to registering callbacks in Flux also helps reduce complexity in the codebase.

The use of pure functions and immutable state in Redux also gives it an advantage concerning the overall developer experience. There are multiple convenient developer tools to use with Redux, which make re-playing `actions` and following the changing of state trivial. A development feature called Hot Reload makes it possible to change code in the application without losing application state. It can be clearly said that Redux is the favorable option when choosing a Flux implementation for a project.

6. CONCLUSION

The idea of this thesis was to implement an example application using two architectures that handle the data flow in a React application. The starting point was to take the original architectural idea from Facebook, Flux, and compare it to an enhanced version of it called Redux. Even though it was clear in the beginning that these two are different implementations of the same thing, the fact that how similar they were came as a surprise mid-process. On the other hand the observations made in the thesis support exactly this. Redux is an improvement, which shows in the use of helper functions in verbose and error-prone parts of the application.

The success of Redux is not only code-related. Redux has a huge and growing ecosystem around it which supports its status as the de facto Flux implementation at the time of writing this. A lot of this is thanks to the helpful creator of Redux, Dan Abramov, who has an endless list of forum posts helping people in their struggles and articles explaining the in-depths of Redux. Redux also has a large amount of complementary libraries which integrate well with it and the idea of immutable application state.

Now that the basic idea of Flux is familiar, a good continuation for the research would be to compare Redux to other improved versions of Flux which have managed to gain at least some usage. Choosing the alternatives would be hard because no other Flux implementation has managed to gain nearly as much traction as Redux.

LÄHTEET

[1] *Angular 2: Documentation* . 2015. [Cited 02.12.2015] Available: https://angular.io/docs/ts/latest/quickstart.html.

[2] Williams, R. *Hacker News Hiring Trends.* 2016. [Cited 05.01.2016] Available: http://www.ryan-williams.net/hacker-news-hiring-trends/2015/december.html?compare1=AngularJS&compare2=Backbone&compare3=Ember&compare4=React.

[3] Occhino, T. *React Native: Bringing modern web techniques to mobile.* 2015. [Cited 05.01.2016] Available: https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/.

[4] *Comparison of the usage of JavaScript vs. Flash vs. Silverlight for websites.* 2016. [Cited 07.01.2016] Available: http://w3techs.com/technologies/comparison/cp-flash,cp-javascript,cp-silverlight.

[5] Borodescu, C. *Web sites vs. web apps: What the Experts Think.* 2013. [Cited 23.01.2016] Available: http://www.visionmobile.com/blog/2013/07/web-sites-vs-web-apps-what-the-experts-think/.

[6] DuVander A. *Twitter Goes JSON-Only With One API, More to Come?* 2010. [Cited 05.01.2016] Available: http://www.programmableweb.com/news/twitter-goes-json-only-one-api-more-to-come/2010/11/10.

[7] Nurseitov, N., Paulson, M., Reynolds, R. & Izurieta, C. *Comparison of JSON and XML Data Interchange Formats: A Case Study.* [Cited 05.01.2016] Available: http://www.cs.montana.edu/izurieta/pubs/caine2009.pdf.

[8] *W3C: Document Object Model (DOM).* [Cited 08.01.2016] Available: http://www.w3.org/TR/WD-DOM/introduction.html.

[9] Nilanchala. *Introduction to JSON Basics.* 2016. [Cited 09.01.2016] Available: http://javatechig.com/json/introduction-to-json-basics.

[10] *Introducing JSON.* [Cited 05.01.2016] Available: http://www.json.org/.

[11] *Mozilla Developer Network: JavaScript.* [Cited 13.05.2016] Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[12] United States Patent and Trademark Office. *Trademark Status & Document Retrieval: JAVASCRIPT.* [Cited 05.01.2016] Available:

http://tsdr.uspto.gov/#caseNumber=75026640&caseType=SERIAL_NO&searchType=statusSearch.

[13] Flanagan, D. *JavaScript: The Definitive Guide*. 2010. 1078 pages.

[14] *About Node.js*. [Cited 05.01.2016] Available: https://nodejs.org/en/about/.

[15] *Projects, Applications, and Companies Using Node*. [Cited 09.01.2016] Available: https://github.com/nodejs/node-v0.x-archive/wiki/Projects,-Applications,-and-Companies-Using-Node.

[16] kangax. *ECMAScript 5 compability table*. 2016. [Cited 05.01.2016] Available: https://kangax.github.io/compat-table/es5/.

[17] kangax. *ECMAScript 6 compability table*. 2016. [Cited 05.01.2016] Available: https://kangax.github.io/compat-table/es6/.

[18] Compilers.net. *Compiler*. [Cited 05.01.2016] Available: http://www.compilers.net/paedia/compiler/index.htm.

[19] Markbåge, S. *New Versioning Scheme*. 2016. [Cited 21.02.2016] Available: https://facebook.github.io/react/blog/2016/02/19/new-versioning-scheme.html.

[20] *Angular 1: Documentation*. 2016. [Cited 24.01.2016] Available: https://angularjs.org/.

[21] Gerritsen, N. *Why you should ditch Angular controllers for directives*. 2015. [Cited 24.01.2016] Available: http://wecodetheweb.com/2015/07/18/why-you-should-ditch-angular-controllers-for-directives/.

[22] Parviainen, T. *How I've Improved My Angular Apps by Banning ng-controller*. 2014. [Cited 24.01.2016] Available: http://teropa.info/blog/2014/10/24/how-ive-improved-my-angular-apps-by-banning-ng-controller.html.

[23] Hunt, P. *Why did we build React?* 2013. [Cited 24.01.2016] Available: https://facebook.github.io/react/blog/2013/06/05/why-react.html.

[24] *React documentation: Reusable Components*. [Cited 21.02.2016] Available: https://facebook.github.io/react/docs/reusable-components.html.

[25] *React documentation: Interactivity and Dynamic UIs*. [Cited 21.02.2016] Available: https://facebook.github.io/react/docs/interactivity-and-dynamic-uis.html.

[26] Alpert, B. *React v0.14 Release Notes.* 2015. [Cited 04.02.2016] Available: https://facebook.github.io/react/blog/2015/10/07/react-v0.14.html.

[27] *Single Responsibility Principle.* [Cited 24.01.2016] Available: http://deviq.com/single-responsibility-principle/.

[28] *Separation of Concerns.* [Cited 24.01.2016] Available: http://deviq.com/separation-of-concerns/.

[29] Abramov, D. *Presentational and Container Components.* 2015. [Cited 03.02.2016] Available: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.s0lmjvoi1.

[30] Learn React with chantastic. *Container Components.* 2015. [Cited 03.02.2016] Available: https://medium.com/@learnreact/container-components-c0e67432e005#.omsw6nsv9.

[31] *React documentation: JSX in Depth.* [Cited 09.02.2016] Available: https://facebook.github.io/react/docs/jsx-in-depth.html.

[32] *React documentation: Top-level API.* [Cited 09.02.2016] Available: https://facebook.github.io/react/docs/top-level-api.html.

[33] *Github jsforum Issue: To JSX or not to JSX.* [Cited 09.02.2016] Available: https://github.com/jsforum/jsforum/issues/1.

[34] *React documentation: Component Specs and Lifecycle.* [Cited 17.02.2016] Available: https://facebook.github.io/react/docs/component-specs.html.

[35] Markbåge, S. *React v0.13.0 Beta 1 Release Notes.* 2015. [Cited 17.02.2016] Available: https://facebook.github.io/react/blog/2015/01/27/react-v0.13.0-beta-1.html#mixins.

[36] *virtual-dom.* [Cited 09.01.2016] Available: https://github.com/Matt-Esch/virtual-dom.

[37] Bille, P. *A Survey on Tree Edit Distance and Related Problems.* [Cited 09.01.2016] Available: http://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf.

[38] *React Documentation: Reconciliation.* [Cited 09.01.2016] Available: https://facebook.github.io/react/docs/reconciliation.html.

[39] *Flux documentation: Structure and Data Flow.* [Cited 26.03.2016] Available: https://facebook.github.io/flux/docs/overview.html#content.

[40] *Building the F8 2016 App.* [Cited 17.04.2016] Available: http://makeitopen.com/.

[41] *Redux documentation.* [Cited 17.04.2016] Available: http://redux.js.org/.

[42] *TodoMVC.* [Cited 20.04.2016] Available: http://todomvc.com/.

[43] *A JS framework on every table.* 2015. [Cited 20.04.2016] Available: http://todomvc.com/.

[44] *Flux TodoMVC Example.* [Cited 20.04.2016] Available: https://github.com/facebook/flux/tree/master/examples/flux-todomvc.