Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Web application development with ReactJS

Theoretische
Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Maximilian Zauner

Begutachtet von Dipl. Ing. (FH) Johann Heinzelreiter

Hagenberg, Mai 2017

Abstract

The goal of this thesis is to provide an overview to web application development with JavaScript by using Facebook's front end framework ReactJS. It is directed at developers that would like to start developing web application software in general or at intermediate web developers who want to get started with ReactJS.

However, the emphasis is on raising attention that application architecture and a well structured code base is one of the most important aspects of any web application software project. Currently many software projects exist in the web that are hardly maintainable and have a bad or no application architecture at all which obviously leads to problems when the project has to be maintained over a longer time period.

The research section of this thesis is about the comparison between the application architecture paradigm called Flux which was invented by Facebook and the traditional Model View Controller (MVC) pattern. After research it is shown that the Flux pattern is very similar to the MVC pattern but still viable to be used in web application software projects.

On top of a broad introduction to the general JavaSript ecosystem and all of its tooling that is necessary to get started, this thesis also gives a specific introduction to the ReactJS library itself. It thus comprises a concise summary of all requirements and best practices of developing a web application with ReactJS.

As part of this work a prototype for a web application with ReactJS was developed and is introduced to the user. The research that is presented in this thesis is further corroborated by studying a real world example.

Kurzfassung

Diese Arbeit gibt einen Überblick über die Enwicklung von Webanwendungen mit dem JavaScript Framework ReactJS, welches von Facebook erfunden wurde. Die Arbeit richtet sich einerseits an Entwickler und Programmierer, die noch wenig Erfahrung im Bereich der Web-Entwicklung haben und sich einen Überblick über die Implementierung von Single-Page-Webanwendungen verschaffen möchten. Sie richtet sich aber ebenso an fortgeschrittene Web-Entwickler, die sich für verschiedene Web-Anwendungsarchitekturen interessieren und das ReactJS Framework kennenlernen wollen.

Die Arbeit soll den Leser dafür sensibilisieren, dass eine gute Architektur einer der wichtigsten Aspekte eines Softwareprojekts im Web-Umfeld ist. Webanwendungen leiden oft unter schlechter Wartbarkeit, da es vielen Entwicklern an der nötigen Erfahrung fehlt, eine ordentliche Architektur in einem Web-Projekt zu realisieren. Diese Arbeit zeigt, dass man von guter Anwendungsarchitektur auch auf lange Hinsicht nur profitieren kann.

Der wissenschaftliche Teil dieser Arbeit beschäftigt sich mit dem Vergleich zweier Softwaremuster. Das von Facebook entwickelte Paradigma Flux wird mit dem traditionellen Model-View-Controller (MVC) Modell verglichen, welches seit den 70er-Jahren existiert. Es wird gezeigt, dass Flux viele Aspekte von MVC kopiert und die zwei Muster einander relativ ähnlich sind. Dennoch kann Flux eine sinnvolle Alternative zu herkömmlichen Web-Architekturen sein, besonders dann, wenn Flux für die Programmierung von ReactJS-Anwendungen verwendet wird.

Zusätzlich zu einer ausführlichen Einführung in das ReactJS-Framework bietet diese Arbeit auch einen Überblick über alle für die Programmierung einer fortgeschrittenen Webanwendung mit ReactJS erforderlichen Werkzeuge.

Im Rahmen dieser Arbeit wurde ein Prototyp für eine Webanwendung mir ReactJS entwickelt. Anhand dieser Beispielanwendung wird demonstriert, wie die behandelten Konzepte und Methodiken angewendet werden können.

Contents

Chapter 1

Introduction

The first chapter of the bachelor's thesis will provide a summary of what this thesis is and what the reason was to write this scientific paper. After reading this chapter the reader should have a clear overview of the content of this bachelor's thesis. It should also be clear what to expect from this paper and what knowledge can be obtained by reading this thesis. The reader should have a basic understanding of general software development in order to being able to understand all explanations and discussions that are provided in this thesis.

1.1 Motivation and Reason

Nowadays web development has gotten very important because the web is a steadily growing platform which provides multiple types of content and can be used on almost any operating system. Oftentimes it is easier to develop a web application instead of platform specific software in order to reach more users that can actually use the software. Programmers can get started with web development very fast and very easily as in most cases some kind of scripting languages are used to realize web applications. These scripting languages are not very difficult to learn. This is the very reason why there is a large amount of web developers who do not have any experience in software development nor any knowledge about software architecture that develop all kinds of web applications. Due to this fact countless software projects are being realized that are hardly maintainable or that have to be refactored if not redeveloped from scratch at a later point in time. It can cost companies lots of money and time having to invest into refactoring and redeveloping tasks. The motivation of this paper is that the reader can learn something about web development and to raise attention that application architecture is exceptionally important especially in web application projects.

1.2 Goals and Overview

The thesis is about web application development with the framework named "ReactJS". These days the scripting language JavaScript really has taken over software development as one of the most popular scripting languages to implement all kinds of applications. The language is not only used for web development but it is also heavily utilized to program all other kinds of software like back end server applications for instance. It can also be used to realize applications for all kinds of platforms like Android or iOS for example. The aim of this thesis is to provide a general overview of what it takes to develop web applications using the whole JavaScript ecosystem in conjunction with Facebook's JavaScript front end framework ReactJS.

The paper however will also focus on a very important aspect of software development being application architecture. It is clear that application architecture is a very if not the most important part of realizing any kind of software project. The term "any kind of software" of course also includes web applications. To emphasize the importance of good application architecture even in web development, one of the goals of this thesis is to demonstrate how advantageous it is to implement a web application with a well thought through architecture.

In 2014 Facebook introduced a supposedly new application architecture named Flux which, according to them, solves all problems that current web applications with traditional architectures like MVC suffer from. The goal of the second chapter is to neutrally compare the MVC pattern to the Flux pattern which will enable the reader of this paper to choose which technology to use to develop a web application. The major research aspect is the usability and the practicality of the Flux architecture in conjunction with a JavaScript framework, ReactJS in this case, alongside with a library called Redux that implements the Flux pattern and handles the state of the application. The paper will explain why choosing a Flux-Redux solution is a good idea to handle the state of a web application instead of implementing a traditional MVC solution.

The third chapter will introduce the Reader to the ReactJS framework. It contains every information that is necessary to improve the understanding of what the framework is about and how to get started more quickly. The chapter will also cover a library named Redux which is an implementation of the Flux pattern. Having a closer look at ReactJS in conjunction with Redux can demonstrate, how application state can be handled in client side web applications.

Last but not least, the thesis will quickly introduce the reader to the prototype that was developed as a part of this work. The goal of the prototype was to develop a big-data visualizing web application that is implemented by using the latest cutting edge web technologies. Particular emphasis was placed on producing a code base that is easily maintainable and easily understandable by developers that have to later maintain the project. By explaining the prototype, it is easier to show how best practices that will be described in this thesis are used in real world projects.

Chapter 2

The better web application architecture, Flux or MVC?

Variants of the programming paradigm, Model-View-Controller (MVC), are the most common architectural choices to program user interface applications in the web. Facebook introduced a supposedly new programming paradigm named Flux. This chapter will not only introduce the reader to Flux, but also critically compare it to the MVC pattern. The comparison will reveal advantages and disadvantages of programming a web application with either of the previously mentioned paradigms.

2.1 Introduction to Flux

Flux was introduced by Facebook in mid 2014 and on the official website [Fac14b] the paradigm is described as an application architecture that is used by Facebook to build client-side, single-paged applications. It is clearly stated, that Flux is a programming pattern and not a framework. The source also claims that Flux can easily be implemented in conjunction with Facebook's ReactJS framework because of React's composable view components. In fact, Flux was designed to work well with Facebook's framework ReactJS to increase productivity and the scalability of the application using the framework with the Flux architecture. There will be an in-depth explanation of ReactJS in the Chapter 3 of this bachelor's thesis.

2.1.1 Structure and Dataflow

The most important aspect of Flux is dataflow. Facebook's statement to the mindset of a programmer using Flux [Fac14b, structure-and-data-flow] is that the unidirectional dataflow is central to the Flux pattern and should be the primary mental model for the programmer.

The Flux application architecture consists of 3 main components: The Dispatcher, the Store and the View as it can be observed in the Figure 2.1. Unidirectional dataflow is achieved by only accessing application state via the Dispatcher. The Dispatcher can only manipulate the Store's data by dispatching so-called Actions that reduce the application state to the desired state. Reducing the state means that it is manipulated from the current state to the a new

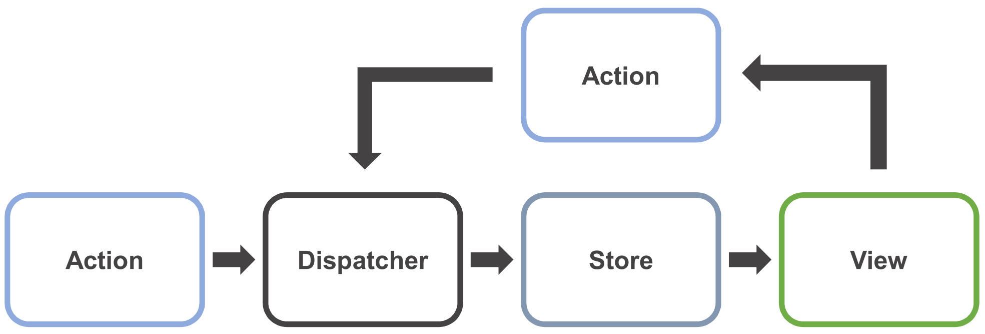


Figure 2.1: Flux application architecture [as in Fac14b, structure-and-data-flow]

state. This so-called reducer functions are pure functions that can be tested very well. The Store then automatically and implicitly updates the View.

After understanding the fundamental concept of Flux, it should also be of interest for the programmer to fully understand the function of each component of the paradigm. It is important to know about the whole concept and all of its components to further being able to understand implementations of Flux or Flux frameworks.

2.1.2 Dispatcher

In the documentation [Fac14b, a-single-dispatcher], a Dispatcher is specified as a "central hub that manages all dataflow in a Flux application". A very important fact stated in [Fac14b, actions] is, that a Dispatcher exposes a method, which can be passed an Action. If provided an Action via that method, the Dispatcher then dispatches that Action to all Stores, triggering an update only on Stores that can react to the dispatched Action. One of the biggest advantages is that many Stores can react to the same Action, which makes the application highly scalable and extensible.

The dispatcher can also explicitly handle more than one Store by handling their registered callbacks in a specified order, if that behavior is desired by the web application programmer.

When the application grows, there can be more than one Action set. The programmer should logically divide all available actions into different files to keep the application organized. This does not change the fact that all actions are dispatched with the same Dispatcher to all registered Stores.

2.1.3 Store

As it can be read in [Fac14b, structure-and-data-flow, Stores], a Store contains an application-specific state and data, as well as data manipulation logic. Every Store has to be registered to the Dispatcher by registering a callback.

Registered callbacks can be used to handle asynchronous data change events in the Store. The Dispatcher uses the Store's callbacks to determine when a data transaction or manipulation has finished in order to being able to dispatch a new action. Registered callbacks are also used to dispatch actions in a specific order to more than one Store.

As mentioned above, Stores also contain data manipulation logic. The way Stores work in conjunction with Dispatchers is that the Dispatcher dispatches one Action to all registered Stores. Each Store can react to that dispatched Action with its own internal data reducing functions.

2.1.4 View

The View component is responsible for correctly displaying any state of provided data as described in [Fac14b, views-and-controller-views]. To achieve that, so called Controller-Views can handle data from any Store and also pass it down to any nested component in the view hierarchy.

Controller-Views react to Store update events and can update whole child component trees by updating their own internal state. By reacting to any data changing event, the Controller-View triggers an update for its child component tree. Each child component can then process the passed data from the parent component, only processing component relevant data. The programmer could even decide to pass a whole Store's data to a Controller-View.

Every important section of a web application should be considered as a Controller-View. In ReactJS it is very easy to determine what components are Controller-Views. As explained in Chapter 3, ReactJS Containers are the so called “smart” components and could be considered as Controller-Views in the Flux pattern. Again, the Flux pattern works exceptionally well with ReactJS, as ReactJS components automatically trigger a render refresh if component properties change. With Flux the programmer only has to introduce a root component bound to the Stores data in order to automatically re-render the application if the Store's data has changed.

2.1.5 Action

The creation of Actions is handled by action creator methods worked out in [Fac14b, structure-and-data-flow, actions]. An Action can be created by a user interaction through an event handler of the corresponding view, through an initialization of a component or even through the server sending an error code for example. As explained before, the Action is then dispatched to the Stores to be handled and to manipulate the Stores to the desired state.

If the application needs to communicate with some kind of API or execute any asynchronous operation it is usually handled inside the action-creator methods. These asynchronous operations are performed and data that might result from those API-calls is then bundled into the Action and dispatched via the Dispatcher.

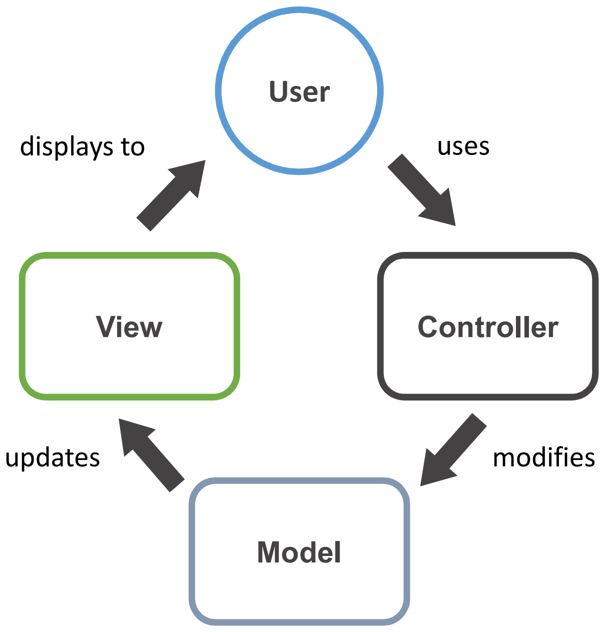


Figure 2.2: traditional MVC [interpreted from ste14, 1.3 Traditional MVC]

2.2 Overview of MVC

As mentioned before, MVC is the most common application architecture in the web. To understand the differences between Flux and MVC, this section provides all necessary information about MVC. That information is intended to help the reader to understand the arguments of the comparison between the two paradigms in this bachelor's thesis. This paper, however, assumes that the reader has a basic understanding of the MVC architecture. Due to that assumption, the overview of each aspect of the MVC pattern is rather short and only shows how this paper is referring to each component of the architecture.

2.2.1 Introduction and history

Traditional MVC as it was initially invented and intended actually also lets data flow in one direction as shown in the Figure 2.2. The programming pattern was invented around the year 1970 to 1980 when the concept of a user interface was not very common [Mar06, #ModelViewController]. The main goal was to make code of user interface applications reusable and to find a way to make UI applications scalable.

The new MVC pattern was also invented to avoid the so called "Smart-UI" anti-pattern [Mat13, S.11,S.13]. In that paradigm there is no separation of concerns meaning that every part of the application logic is located in the interface code. It should be clear for every programmer though that such conditions should be avoided to keep the application scalable and to keep every component of the application interchangeable.

The fact that many people are using the MVC pattern in their applications is also the

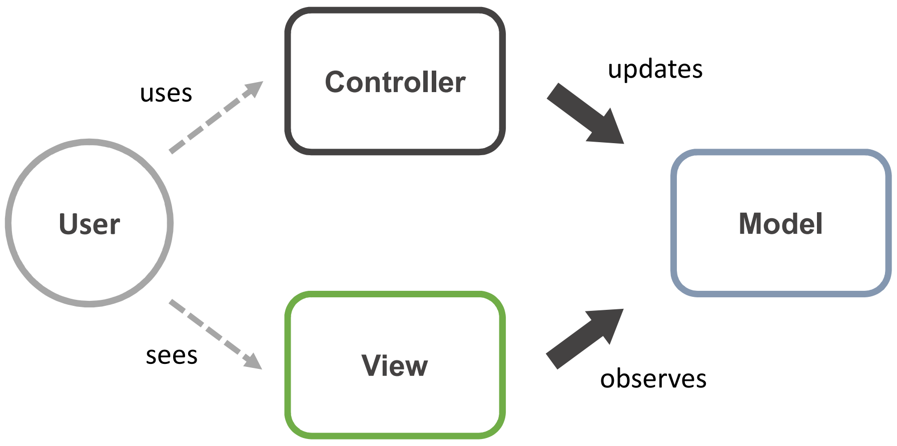


Figure 2.3: another interpretation of MVC [seen by Mar06, #ModelViewController]

reason why the pattern has many variations. There are also many different ways on how it is possible to implement an application using the programming paradigm. There are many interpretations of various aspects of the application architecture that can lead to a false understanding of MVC. This chapter will cover some of the possible misinterpretations of MVC and show how the pattern has evolved over time. It is very important get an overview to the MVC pattern in order to being able to compare it to other competing application architectures.

2.2.2 Model

In this paper the term "Model" is not used to represent the business logic part of the application but rather the representation of the actual application state or "real data" that is used to represent a certain state of the View (Presentation-Model). But what exactly is the Model then? In [ste14, 1.4.1 Model] the Model is described as the component that either contains stored data or functionality to compute view-relevant data.

The Figure 2.2 shows that the model is responsible for updating the View. Every change in the state of the model will trigger an update in the View component but not the other way around.

According to Martin Fowler, in the Figure 2.3, the Model is the passive component. Fowler's MVC diagram (2.3) shows the interaction arrow between Model and View in the opposite direction than the traditional MVC diagram (2.2). Nevertheless, this does not imply that data is flowing in the different direction—quite the opposite—the Model is still storing the data and the View observes the Model. The View is only updated when the state of the Model changes. As described in [Mar06, #ModelViewController] the Model is completely ignorant of the View component and only Stores data that is used to represent the visual state of an application.

2.2.3 View

The most important part, that can't be stressed enough, is described in [ste14, 1.4.2 View]. The View is the component that handles interaction of the user but in **read-only** mode. This implies that the component is incapable of modifying the state of the application. A View should therefore only contain logic that exactly tells the component what to represent or display at a certain state. When following the MVC pattern a View component must not call any API endpoints, nor query any database. According to [ste14, 1.4.2 View] the View observes the Model to know when to update and what to display to the user.

Figures 2.2 and 2.3 show the relation between the View and the Model differently but the flow of the data is the same. Data always comes from the Model and is transferred into the View. How data is transferred is dependent on the actual implementation. In most cases the View observes the Model, updates itself if the Model changes, and is then seen by the user.

2.2.4 Controller

As mentioned before, the View aspect of the programming pattern MVC enables the user to interact with the application. When there is a user interaction though, the View should delegate this action to the so called Controller that can then react to that user input and handle it accordingly as described in [Mat13, S.14]. [ste14, 1.4.3. Controller] describes the Controller as a component that interacts with the Model in read-write mode. It is the business logic of the View, so to speak.

Furthermore, there is always a Controller-View pair for every part of the application that has its own visual representation ([Mar06, #ModelViewController]). An example would be an application that has a login screen and one or more screens to interact with provided services. The application would at least have to have 2 or more View-Controller pairs.

The MVC pattern is often referred as triad but it is important to not forget about the most important part about MVC, the user. Figures 2.2 and 2.3 show, the user is added to the triad to improve the visualization of data-flow of the application.

2.2.5 Traditional or client-side MVC versus server-side MVC

Because the development of MVC started in the 1970s, the software pattern has changed and evolved much until now. It is very important to clearly understand the difference between the aspects of the programming paradigm as MVC in some cases does not equal MVC. In this paper, the most important distinction to know about is traditional and the server-side variant of the software pattern.

When reading this section, it is important to point out that this paper refers to "traditional MVC" as is was developed in the 1970s by the smalltalk community. The article [Chr15], on the other hand, uses the term "traditional MVC" for the version of MVC many people nowadays refer to, the server-side variant of MVC. Another term for "traditional MVC" could be "client-side MVC" as this makes clear that the state only exists on the client.

Traditional MVC was initially developed for applications that keep state and data on the same machine. Nowadays, applications are split into more layers. Modern software is usually

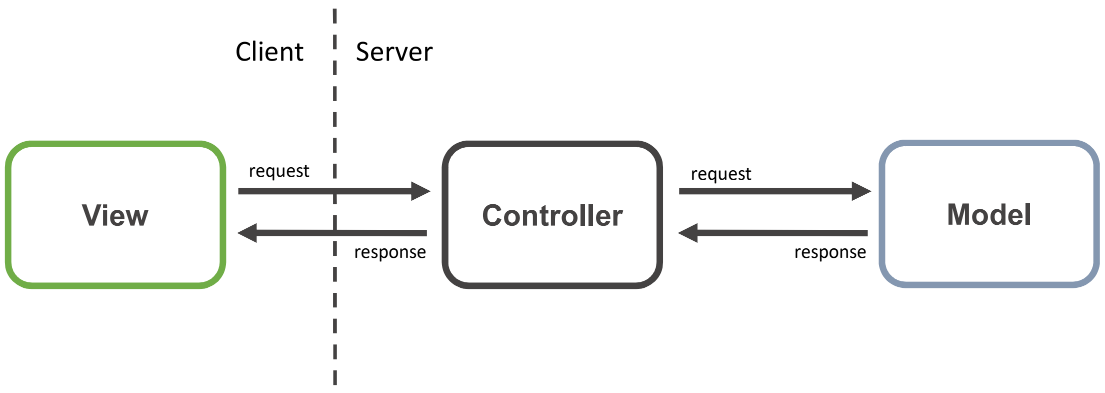


Figure 2.4: MVC in a server-client environment [as in Chr15]

deployed at least on an application and a database server. Especially in the web, applications additionally have to be served over the HTTP-protocol adding yet another layer.

The article [Chr15] states that the biggest problem with server-side MVC is the fact, that the View cannot observe the Model as it was initially designed by the smalltalk community in the 1970s. The fact, that there is a HTTP layer between the View and the Model (visualized in the Figure 2.4) makes it impossible for the View to automatically update when the application state is changed. It always has to be the View initializing the update process. The problem is, that HTTP is a stateless protocol constraining the View to only being able to change the application state during one HTTP-request ([Pau13]). This also implies, that the only component that can persist state is the View due to the stateless protocol violating the original concept of MVC.

In the web many frameworks that are written in PHP for example heavily rely on the server-side version of MVC because they send fully generated and fully rendered views to the client and handle all logic on the server. In that case, it is the Controller that is responsible for sending the View to the client (Figure 2.4). In the last years, so called "single-paged applications" became popular among the web community. These single-paged applications are downloaded and executed only on the client. This makes it possible to use traditional or client-side MVC again as there can be application state, that is able to persist during the whole execution time of the software.

2.3 Comparison of the software patterns

In this section, the Flux and the MVC pattern will be compared. Now that the reader has a basic understanding of what the Flux pattern is about and what it will be compared to it is easier to develop an opinion based on the provided facts. When new software patterns are introduced, it is essential to carefully and critically compare them to existing ones and evaluate their usefulness in the modern software environment, where they can possibly be used.

2.3.1 General facts and misconceptions

When casually searching the Internet for the Model View Controller pattern, many visual representations of the software architecture can be found. In many cases the diagrams show the MVC-triad with arrows in facing different directions, unidirectional and bidirectional, leading to confusion and raising the question in what direction data flows exactly.

Diagrams that show the Model update the View and the View in return update the Model stimulated by user input are quite simply wrong. This is a common misconception. Change of the application state should only be triggered over the Controller. Everything else would imply a violation of the MVC pattern.

In this bachelor's thesis there are two representations of MVC (2.2 and 2.3). Both representations show the same components but different relations. It must be understood though, that data-flow still goes in one direction. Different names for relations may switch the directions of some arrows in a diagram but do not change the semantic meaning of the MVC pattern.

In the presentation where Facebook introduces the supposedly new software architecture Flux ([Fac14a]), the speaker presents the audience a visualization of MVC that can be seen in the Figure 2.5. Facebook talks about how complicated the debugging process was because of countless bidirectional relations between the View and the Model components. To the attentive reader it should be clear, that a bidirectional relation between the View and Model component regarding to data-flow implies a violation of the MVC pattern. The question arises if Facebook's engineers had understood the programming paradigm MVC before developing an allegedly new programming paradigm?

2.3.2 Similarities of Flux and MVC

Looking at the illustrations of Flux (2.1) and MVC (2.2) it can be observed that there are similarities. Flux achieves its unidirectional data-flow by only changing the Store's data by dispatching an Action containing relevant user input data over the Dispatcher. Comparing this behavior to the MVC application architecture, it should be clear that the operation of a change in the Model triggered by a user stimulus handled by the Controller is a quite similar behavior.

One of Flux's biggest advantages, according to Facebook, is the possibility of having global Stores. These global Stores can be used for an application state which has to be used in many different parts of the application e.g. a notification counter that shows the user a count of unread messages. Traditional MVC according to Fowler ([Mar06, #ModelViewContoller]) needs to have a Controller-View pair for every screen in the user interface but also one global Model that stores the application's state. In most cases though, it is recommended to also divide the Models into the logic sections of the application to make the code more transparent and easier to read and debug. This is very similar to how Flux handles global data. Like in Flux in MVC it is also allowed for more Views to observe the same model.

The MVC triad consists of three components: the View, the Model, and the Controller (2.2, 2.3). One could add the User as the fourth optional component that makes the architecture easier to understand. It is the user's stimulus that triggers every action or state change in the application. When looking at the Flux pattern (Figure 2.1) there are also three components

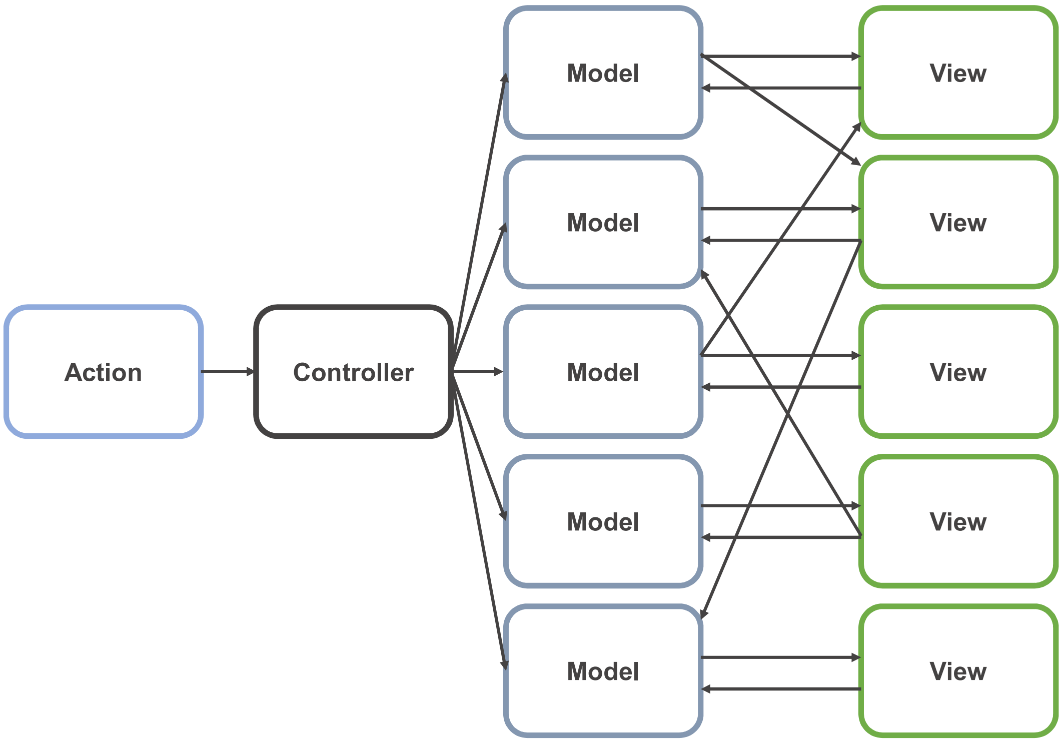


Figure 2.5: MVC as Facebook describes it in [Fac14a, 11:10]

that have the same function as the corresponding components in the MVC diagram. Flux's Dispatcher, for instance, could be seen as MVC's controller and the Store is similar if not identical to MVC's Model.

2.3.3 Differences of Flux and MVC

One of the biggest differences between Flux and MVC is the way the View component reacts to a change in the application state. In the Flux pattern the View is updated by injecting the new application state into the View. The View component is a strict representation of what has to be displayed at a certain state. Updates therefore happen implicitly. MVC on the other hand handles updates in the application state by binding the View to certain values and thus observing the Model. When a Controller modifies the Model every View that has bound its view-components on data in the Model can automatically update. The MVC View updates itself automatically whereas Flux's View component is updated by the new state that is injected into it.

Another big difference is the way application state is modified. As mentioned before, Flux uses a system where Actions that contain data are dispatched to the Stores which then accordingly react to the dispatched Action and update their data via reducing functions. MVC on the other hand should not have any logic in its Model. Because the Model should only be a representation of Data the Controller must handle API-calls and directly access the Model to change its data.

2.3.4 Advantages and Disadvantages of Flux and MVC

A few years ago web developers could not think about another way to engineer a web application other than sending a fully rendered application View to the client and handling user input logic and state on the server. Programmers therefore can only update the View by triggering a change in the state by executing logic in the corresponding Controller. The biggest problem with this approach is that it violates the constraints of traditional MVC as it was designed initially. The Controller and Model instances only live during an HTTP-request, making it impossible to follow MVC's constraint of persisting a Model state during the lifetime of the application. Another big disadvantage is the fact that the View cannot be automatically updated, nor can it observe the model state because of the additional stateless HTTP-layer.

A disadvantage of both patterns is of course programming overhead. Applying a highly complicated application architecture to software always implies increased programming time. What really matters are the long term advantages tough. By using either the Flux or the MVC pattern when developing web applications it makes components interchangeable and increases the application's scalability. Even web applications written in JavaScript for example can heavily benefit from using either one of the mentioned programming paradigms.

2.4 Conclusion

Even though the Flux application architecture seems to be extremely similar to traditional MVC, Flux is still an excellent choice for a web application architecture. On its own, MVC has proven itself over the course of the time of its existence. There has been a big shift in how web applications are build until now. Nowadays, many big web applications are built as single-paged applications and the number is steadily growing. The biggest advantage of having an autonomous application that is executed only on the client opens up the possibility to keep application state on the client without having to reload the web page every single time when the application state changes.

The use of single-page applications do not only enable Flux application architecture to be used, the MVC programming pattern can be applied as well of course. Other frameworks like AngularJS for instance use a special variation of MVC called Model-View-ViewModel (MVVM). It is safe to say that using a programming paradigm like Flux or MVC is of utter importance nowadays when building web applications. Not only does it add structure to the code base, but it also makes code much more readable and easier to understand for developers who have not implemented the application but later have to maintain it.

Initially Facebook presented Flux as a big way forward but upon further review it appears as if Flux was just another variation of MVC. There are many similarities that make the two patterns seem to be very alike. It is honestly disappointing to discover that a much praised and hyped application architecture proposal, which had promised many improvements on how to improve web application development, is just a shallow copy of a present paradigm (MVC) that has existed for many years until now.

It must be emphasized though that Flux is not a bad application architecture approach at all. All the similarities to MVC make Flux a solid pattern that has its merits on its own. After much research it is clear that both patterns can yield enormous advantages if they are used to design web applications. Having no application architecture at all is of course the worst decision a programmer could make. Also, MVC and Flux seem to be very popular among the web developer community.

Upon further reflection it seems as if Facebook did not research properly and implemented MVC wrong. That was the reason why the company came up with a proposal of an allegedly new paradigm. Proof for this theory is the diagram (Figure 2.5) in the video presentation [Fac14a] which clearly shows that the gist of the MVC paradigm was not understood correctly.

After having used Flux as a paradigm for a web application it is clear that the architecture seems to be tailored for one specific framework which is ReactJS. The next chapter will show exactly how Flux perfectly fits into the mindset of ReactJS and how it works really well with the framework making it a very satisfying experience to build an application with a good architecture and seeing the benefits of using such a paradigm.

In conclusion it is very important for web application development to use some form of sophisticated application architecture. The number of applications that rely on some form of web technology is increasing heavily nowadays and thus it is even more important to emphasize the necessity to use a well thought through application architecture.

Chapter 3

ReactJS

In the previous chapter the application architecture Flux was introduced and discussed. Even tough aspects of this programming paradigm seem to be copied from other programming patterns, it is certain that Flux is not a bad application architecture after all. This section will demonstrate how Flux works best with Facebook's front end framework ReactJS ([Fac13]). The goal is to show the reader what is necessary to develop a web application by using cutting edge web technologies. This chapter will also provide an overview over some best practices and help the user to make application architectural decisions.

A few questions will come to mind: "What tools are needed to develop a ReactJS application?" or "Why should a web application programmer use this framework?". Nowadays many large companies use JavaScript frameworks for the development of their web applications. The hard part in starting any web application project is to decide, which framework or technology is will be used. After reading this chapter all mentioned questions will be answered and it should be easier to make the right decision. This paper is *not* a comparison between competing web technologies but it should give a clear overview of the features and benefits when using ReactJS.

3.1 Why ReactJS?

As described in [ZH16, S.7-8] there are several reasons why ReactJS is a good choice for a JavaScript front end Framework as described in the following paragraphs:

First of all, the framework is rather easy to learn and understand to get going and to simply start coding. There are some easy code examples [Fac13, Docs: Hello World] to get started very easily. The difficulty is not very high even for programmers that do not have much experience at all. There is also always a link to a live coding website where basic ReactJS code examples can be tried out.

React's API is not extensive and well documented. The architectural decisions of the framework and the code of an application make it more easy to understand. Furthermore the ReactJS API hardly ever alters in a way that breaks already existing code. Facebook claims on many articles in their blog [Fac13, Blog] that almost no time is wasted in refactoring deprecated ReactJS code from any outdated application. Experience has shown that indeed there had not been any situation where any front end specific code of ReactJS broke by updating to a newer version of ReactJS. If some component is about to be deprecated, it is clearly communicated on the release notes on the blog or by runtime development deprecation warnings that link to migration guides.

ReactJS is a view-only framework. As stated on the project's starting page [Fac13], Facebook does not make any assumptions about the development stack of the programmer. It is completely up to the software engineer to choose what technologies to use in conjunction with ReactJS. There are very few requirements from the framework itself. For instance it is totally up to the developer how to handle routing, application state and the communication to an API. That is also the reason why this framework appears to be very professional, as it is the programmer's responsibility to develop an application architecture and to choose the suiting development tools to tailor the application to the client's needs.

The ReactJS framework exists since the year 2013. During the time of its existence many technologies have been developed that work really well with this framework. Being highly component oriented it is possible to easily integrate third party ReactJS components from other developers. Custom ReactJS components are open source most of the time and can be integrated into any ReactJS project by using some form of a package manager. For example there are high-performance list or table views with lazy loading that optimize handling large amounts of data. Integrating those components can save a lot of time and improve the application's performance without the developer having to implement complicated components by him- or herself.

Another advantage in developing ReactJS applications is that this framework has been in use long enough so there are many excellent developing tools available at the programmer's disposal. Everything from performance benchmarking tools to ReactJS component explorers can be integrated in any ReactJS project. One of the most important pieces of developing tools would be the possibility to automatically recompile and reload the application when files change without loosing the application state.

ReactJS is often compared to other frameworks in terms of performance and scalability. There is no definite answer which framework is the best amongst them of course. The fact that not only Facebook itself uses the framework but also other well known large well known companies like AirBnB[1] or Plex[2] as well provides good evidence that the framework can be well performing and scalable if implemented correctly.

3.2 Introduction to the tooling for ReactJS

When developing a ReactJS application, it is of utmost importance to understand all the tooling that is necessary to develop a ReactJS application. The technology which is necessary to transform JavaScript code into a code bundle that can be interpreted by the browser should not be a "black box" to the developer. When talking about a ReactJS application it often implies that the term "single paged application" is clear to the programmer as well. This section will explain all those terms and all necessary tools to create such a single paged application by using ReactJS.

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

[2]https://plex.tv/

3.2.1 ECMAScript and Babel

ECMAScript[3] or ES for short is a specification for scripting languages and also specifies the scripting language JavaScript. ECMAScript is developed by ECMA International. Because JavaScript is a scripting language it needs to be interpreted by a JavaScript engine. The problem is, that ECMAScript is only a specification and different engines implement different versions of ECMAScript. Oftentimes clients demand application compatibility for older browsers like the Internet Explorer 11 for instance. The browser only supports only 11% of the ES6 standard as this compatibility table[4] shows.

The following question arises: "What can be done if the web application that is coded in ES6 also needs to run on IE11?" Babel[5] is a JavaScript compiler, that transpiles JavaScript to JavaScript. This is especially useful in situations where the developer uses a version of the ECMAScript standard, that not all target runtime environments yet support. Currently almost all JavaScript projects use the Babel compiler to get the greatest compatibility across all runtime environments. The developer simply needs to select the desired ECMAScript version and Babel transpiles all JavaScript code to that specified version. It must be mentioned that not all features of modern ES specifications can be transformed to older ES versions.

The Babel transpiler not only transforms JavaScript to JavaScript but can also polyfill JavaScript features. A polyfill is a piece of code that implements a language feature that is considered native by the developer. For instance, the ES6 standard defines a function `Array.of()` that creates an Array object of a variable number of arguments. Internet Explorer 11 does not support this function, thus it needs to be polyfilled by babel so the browser can interpret the `Array.of()` function. If the function wasn't polyfilled, an error would occur and the application would crash.

ReactJS works very well with modern ECMAScript features and proposals that are not yet implemented in many browsers. That is the reason why Babel is a very important developer dependency in almost any web project that uses modern JavaScript. As the development of JavaScript engines progresses, some time it will be possible to use a more recent ECMAScript version natively in the browser. With Babel the programmer just has to alter one line in the Babel configuration file to switch the transpiling target from ES5 to ES6 for example.

Especially for ReactJS, Babel is very important as it also transforms React's template language JSX into vanilla JavaScript. React can be used without JSX of course but it is not advisable as JSX is one of the most useful features of ReactJS highly increasing productivity and also adding a very useful abstraction layer to the framework. There will be an in-depth description of JSX in Chapter 3.3.1.

3.2.2 NodeJS and NPM

One of the biggest benefits of developing a JavaScript web application is the fact, that developers can use the node package manager npm[6]. This package manager allows the programmer

[3]https://www.ecma-international.org
[4]https://kangax.github.io/compat-table
[5]https://babeljs.io/
[6]https://www.npmjs.com/

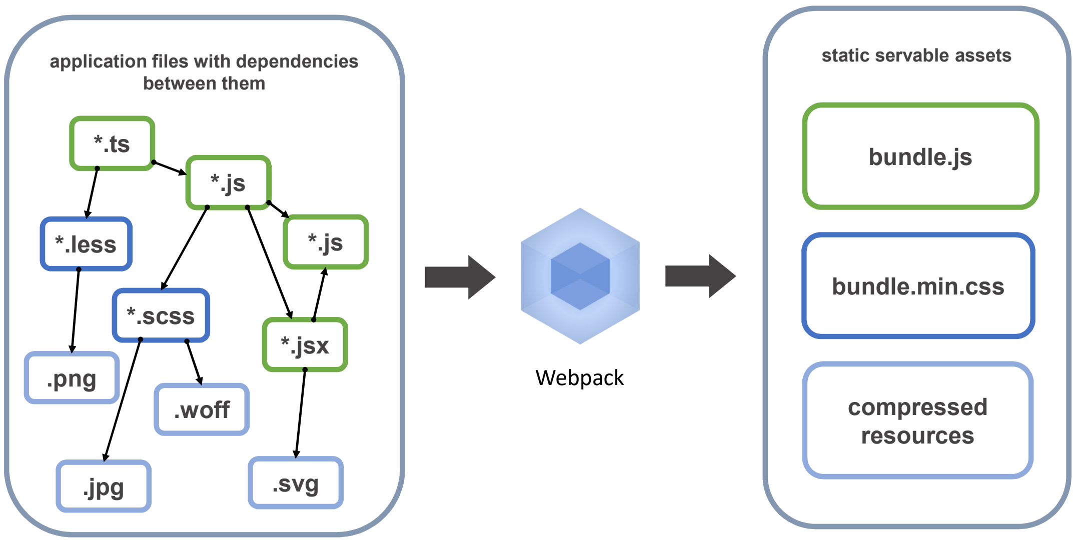


Figure 3.1: Webpack transformation visualization

can easily integrate any package or module into the web application. With a simple command ReactJS, Babel, Webpack and all the other necessary tools can be easily downloaded and installed without worrying about any dependencies. There are other package managers as well but npm is the most widly used JavaScript package manager. When installing NodeJS on a machine, npm is bundled in the installation.

NodeJS[7] is not absolutely necessary for developing a React application, but Webpack offers an implementation of a development NodeJS server, that can automatically hot-reload React applications. This makes the developing process for web applications much faster and easier. In fact, developing a React application without NodeJS would be very inconvenient but possible after all. A bundled React application can be hosted on any static content serving web server. If the routing is configured correctly it is even possible to start the application directly from the file system as well.

3.2.3 Webpack

Webpack is one of the most comprehensive developer tools that can be used for developing a web application using all kinds of files and language flavors. It is a module bundling mechanism that combines all the code base of a project into an application that can be interpreted by a browser. It does not matter if Less, Sass or plain CSS is used, neither does it matter if the project is developed in CoffeeScript, TypeScript or plain JavaScript. Webpack's base functionality is to transform modules with dependencies into static assets as described on the technology's website[8] and as the example Figure 3.1 shows.

Webpack is especially useful in combination with the Node Package Manager. Every module

[7] https://nodejs.org/
[8] https://webpack.github.io/

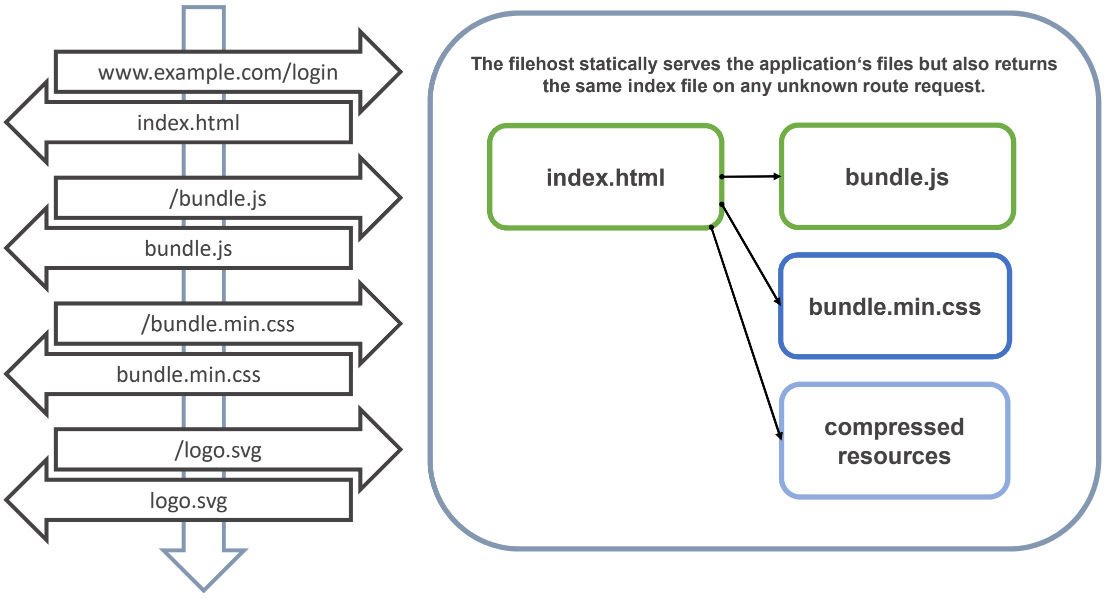


Figure 3.2: Single paged application example

that was downloaded by npm can be integrated into the web application with a simple import statement and Webpack does all the work to integrate the module's code into the final bundle. Webpack will handle any dependency in the code and transform it into a code bundle, that includes not only the application's code but also third party module code, that was imported by the developer.

Another outstanding feature is the code splitting and the on-demand-loading of code. Not only is the code bundled and transformed into static assets but also split into code modules again in an intelligent way. Modules that are more heavily used in the application will be loaded first and other less important module parts can be loaded on demand. To avoid client side caching it is possible to include a generated build hash into the modules' name so the browser interprets each module as a new asset that is not yet cached. Deploying updates to an application is therefore no problem.

Many features can be accomplished with Webpack itself, but on top the module bundling mechanism is highly customizable as well. Webpack does support a plugin and loader system that can accomplish even more customization and variety in code transformation. Loaders can load, transform and then return any resource of the code of a web application. An example would be the loading of Sass files and the transformation to CSS. Another use for loaders would be the integration of image files. In the process of loading the images they can also be compressed and are output as static resources. Plugins on the other hand add useful features. One of the most important Plugins for development and production is the DefinePlugin. This piece of code allows the substitution of constants in the application's code according to the environment. With the help of this plugin the app can be run in development or production mode. There are several loaders and plugins provided by the Webpack community that can be downloaded and integrated by npm.

One of the most important features of Webpack for developing React apps is it's own development server. As stated before, NodeJS allows the developer to start a development server, that hot-reloads code parts so the developer does not have to recompile the whole code again in order to see the updated application after a change in the application's code. Not only NodeJS, but especially Webpack enables this feature. The code bundling mechanism enables watching file changes in order to compile changes on the fly and deploy them while the server that serves the front end code is still running. Again, this feature works with any asset of the app, whether it is a CSS file that has changed or a JavaScript file, the hot-loading mechanism is triggered.

When transforming a web application project with Webpack, the outcome is called a single paged application. The term describes a certain kind of web application that is started from one single index file. Most of the work is done by Webpack itself as it already transforms the web application's code into static assets. The server then only has to statically serve all application relevant data. If the requested route does not reference a static asset it means that the client must be requesting a certain page of the web application. In that case the server always has to serve the same index file. By doing that, the requesting client is guaranteed to always get the special index file that bootstraps the whole web application. The application can then react to the current route and display the correct page. The Figure 3.2 shows how a request of the login page in a single paged application could look like.

3.3 Introduction and overview of the Framework

Now that it is clear what is needed to set up a ReactJS project and there is a better understanding how the tooling works it is now time to dive deeper into the framework itself. This section will give an introduction to ReactJS and its history. Also basic application architecture will be discussed as well as conventions how to program a ReactJS application. Some core components of the framework will be described to achieve more knowledge on how the library works.

To begin with, the JavaScript library ReactJS was developed by Facebook. The framework is open source since 2013 but it is still managed by technical engineers of Facebook and is available under a BSD license. The framework itself alows highly component orientated application architecture, which means code parts of a ReactJS application can be used in a very modular way. Once all the boilerplate code is built up and all standard components are implemented, big progress can be achieved in a very short amount of time. The most time consuming part of starting a React web application is to set up all the tooling and all the boilerplate code. To get a better understanding of the framework, some of its components will be explained to then show how a ReactJS application architecture could look like.

Where ReactJS really excels is when it comes to new web technology. The fact that the framework is a descriptive UI defining top-level API makes it easy to use ReactJS for other platforms and technologies as well. When developing a React application top-level highly abstracted simple components can be used that finally get broken down to library function calls that actually create UI elements. ReactJS could be described as a general user interface component abstraction library. Because ReactJS was engineered that way it is easily possible to use React to not only develop web applications but also to use React to develop mobile applications (React Native[9]) or build virtual reality environments (React VR[10]) without much effort and without having to learn any new frameworks or programming languages.

3.3.1 JSX

As it can be read in [ZH16, S.59], ReactJS is a front end framework that does not need a template language to create user interfaces. Instead, the library provides functions to create or manipulate the actual DOM elements of a React component. To make the process of creating elements easier, ReactJS provides a JavaScript language extension called JSX. Code that is written in JSX can be transformed into JavaScript code by Babel and Webpack.

Using the JSX language extension allows HTML-elemnts to be directly written into the JavaScript application code as this simple example shows. Note that it is necessary to use the JSX notation `className` instead of `class` as this is not vanilla HTML. All provided code examples can be tried out in any online live JavaScript editor[11].

```
const HelloElement = (
  <div className='hello-world'>
    <span>I am a span</span>
    Hello World
  </div>
)
```

The provided code snippet could not be interpreted by any browser and needs to be transpiled into vanilla JavaScript. The best part about JSX is, that components that are transformed this way are infinitely nestable. Components are easily and infinitely composable that way. ReactJS is a highly component oriented front end framework as mentioned before, making it easy to compose any ReactJS components. After the transformation the code looks like the following code snippet:

```
var HelloElement = React.createElement(
  'div',
  { className: 'hello-world' },
  React.createElement(
    'span',
    null,
    'I am a span'
  ),
  'Hello World'
);
```

The example shows, that the pseudo JSX DOM-elements are transformed into functions from the ReactJS API. The reason why components are infinitely nestable is that they get transformed into plain JavaScript functions that are composed as it can be observed in

[9]https://facebook.github.io/react-native/
[10]https://facebook.github.io/react-vr/
[11]http://codepen.io/gaearon/pen/rrpgNB?editors=0010

the example above. Using the functional programming paradigm composition makes it easy to compose any components. The return type is not an actual DOM-element, it is just a lightweight ReactJS JavaScript object but this fact is described in the Section 3.3.6 in greater detail. JSX can be tested with any online JSX transpiler like the babel transpiler for example[12]

JSX is one of the most important features of ReactJS because it adds a very important abstraction layer to the view. The developer can use ReactJS components as normal DOM components without knowing the exact implementation. This is also the reason why ReactJS cannot only be used in the web but also for mobile platforms by simply switching the HTML components for the according components of the target platform. An example would be React Native which produces native apps for Android and iOS by using components like `<View>` instead of the well known HTML element `<div>` for example.

3.3.2 Components and properties

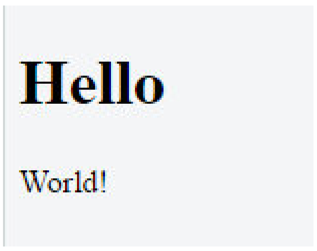
JSX is not only used to create common HTML elements, it can also be used to make use of components that were programmed by the developer. These so called “React components” can be JavaScript classes that extend the React `Component` class or pure functions as described in Section 3.3.4. There was also the approach to use the React class factory function `React.createClass()` to use OOP style components in JavaScript. As the ECMAScript introduced the class syntax the class factory function was deprecated as it is better to use native classes than functions that get used as classes with the help of their closure. In the most cases components are functions or classes that can be used as a JSX element as shown in this example:

```
class Hello1 extends React.Component {
  render() {
    return <div>Hello, World!</div>
  }
}

const Hello2 = () => <div>Hello, World!</div>
```

Note, that the anonymus function is declared as a ES6 “fat-arrow function”. Basically the `Hello1` and `Hello2` variables act identically. When rendered, they show the exact same output but have no functionality attached to them. The following example shows, how the newly created React components can be used with JSX:

```
ReactDOM.render(
  <Hello1/>,
  document.getElementById('container1')
);

ReactDOM.render(
  <Hello2/>,
  document.getElementById('container1')
);
```

[12]https://babeljs.io/repl/


Hello

World!

Figure 3.3: Output of rendering HelloComponent

The name of the component object defines the JSX element name. The two ReactJS API functions (`ReactDOM.render()`) will search for elements in the HTML-DOM that are called "container1" and "container2" and render the two React components instead of them. Usually, the `ReactDOM.render()` function is only used once in a ReactJS web application as a bootstrapping mechanism to set one element as the application root so ReactJS can build its actual DOM tree onto the specified element. As mentioned before, ReactJS components are infinitely nestable so there only has to be one actual root element in the real HTML-DOM to bootstrap onto.

To add functionality to a component it can be accomplished by passing it so called *props* and processing them with JavaScript component logic. Passing a prop is really easy in JSX:

```
const HelloComponent = (props) => {
  props.ownFunction('this will get logged')
  return (
    <div>
      <h1>{props.title}</h1>
      {props.text}
    </div>
  )
}

ReactDOM.render(
  <HelloComponent title='Hello' text='World!' ownFunction={console.log} />,
  document.getElementById('root')
);
```

The output of this snippet of code can be seen in the Figure 3.3. A very useful feature of JSX is that also functions can be passed to a React component. This is very useful when nested child components have to trigger logic in their parent component.

3.3.3 Stateful components in ReactJS

The component logic is very important in React. Functionality can be added by adding member functions to ReactJS class components to manipulate received props and providing a view to the provided data. Like discussed before, there are React components that are functions but there are also React components that extend the class `React.Component`. These classes do have a lifecycle and their own component state which can be manipulated as well. These class components produce more overhead as React treats these class components as smart components with state and lifecycle that has to be handed accordingly. Class components should only be used where it is absolutely necessary to reduce overhead and improve performance.

Lifecycle methods are very important if the component has to handle any asynchronous task or if incomming props have to be mutated. All the available methods are documented in the official documentation [Fac13, Docs: React.Component]. For example there are methods that get executed before the component is about to mount (`componentWillMount()`), when it receives its props (`componentWillReceiveProps()`) or when it is about to unmount (`componentWillUnmount()`). The most important lifecycle method is the `render()` method though. This method gets called when the component has received its props and state and is about to render the view to the user. According to the documentation, API calls should be handled in the `ComponentDidMount()` method. The reason for that is to improve user experience. All components should render with an initial state and update themselves accordingly when asynchronously fetched data arrives. Another very important lifecycle method is the `shouldComponentUpdate()` method. The purpose of this method is to being able to exactly define the data that has to change in order to trigger the `render()` function. By overriding this method performance can be increased greatly because unnecessary rerendering cycles of the component that would otherwise slow down the whole application can be prevented.

Class components also have their own state that can be mutated by the member method `setState()`. As mentioned before, Facebook claims ReactJS does not make any assumptions about the development stack. This does also include the fact that ReactJS does not assume that the developer uses any form of application state container by him- or herself. This is the reason why ReactJS class components can have their own state which is tied to all the lifecycle methods. When either the component's state or props change the component will go through the according lifecycle methods including the `render()` method and is therefore rerendered. Class components are also often called "stateful components".

The following code example shows how a stateful component could look like. Note that it is still possible to pass props to the stateful component. React makes use of the ECMAScript class syntax and extends the `React.Component` base class for creating stateful React components.

```
class StatefulComponent extends React.Component {

  constructor(props) {
    super(props)
    // setting the initial state in the constructor
    this.state = {
      value: 'initial'
    }
  }

  mutateState(value) {
    // mutate state with React Component method to trigger lifecycle methods
    // this method is asynchronous
    // any asynchronous action can be performed like calling an api
    this.setState({ value })
  }

  componentDidUpdate() {
    console.log('i was updated')
  }

  render() {
    return (
      <div>
        <button onClick={this.mutateState.bind(this, 'mutated!!!')}>Mutate Me</button>
        <p>{ this.state.value }</p>
      </div>
    )
  }
}
```

When executed, a button and a text field (paragraph tag) are shown. Initially the text renders as “initial”. Once the button is pressed, the internal state of the component changes and it renders “mutated!!!”. Every time any props or state changes, the React component reiterates all lifecycle methods that are implemented. In this example the `ComponentDidUpdate()` method gets executed after each time the button was pressed because the state of the component changed by executing the `setState()` method.

3.3.4 Pure functional components

The other type of components ReactJS has to offer are so called “functional components” or “stateless components”. This type of component is often used to render parts of the application that do not have any component logic attached to them. Functional components are only a declaration of how the component has to look at a certain state. These components receive the application state via props and can also trigger logic operations by invoking functions that were passed as props as well. Functional components are often called “dumb components” because their only reason of existence is to provide a view to a certain application state.

When seen in the perspective of functional programming one could also call these components pure functions. The fact that the functions are pure without any side effects make the components easily testable. Functional components are pure functions that will always produce the same view for the same constellation of props. React does *not* treat these dumb components as a stateful component. That is the reason why there is no additional overhead as the JSX component gets transformed into a simple `React.createElement()` call as it can be seen in the examples above. Neither lifecycle nor component state has to be considered by React when using pure functional components which greatly improves performance and reduces script execution time overhead.

The following code snippet is considered a functional component. Components like buttons or text input fields are perfect for being a functional component as they will get used very often throughout the whole application.

```
const FunctionalComponent = (props) => (
  <div>
    with the same props i will always render the same...
    {props.text}
  </div>
)
```

The only constraint of functional components to be used in conjunction with JSX is, that they have a capital letter at the beginning so JSX recognizes the function as a React component. The component can then be used very easily in JSX as the following snippet shows:

```
<div>
  <FunctionalComponent text='I am so pure!' />
<div>
```

3.3.5 Property Types

JavaScript is a dynamic typeless scripting language. Because of this very reason ReactJS provides an API that lets developers specify property types for each component. When React is executed in the development mode each component will check the provided property types to ensure that the component is used correctly. The following example shows how the so-called `PropTypes` can be used:

```
const Component = (props) => {

  // if the prop title is not set it gets set to a fallback value
  // the prop is not guaranteed to be set so the programmer has to implement a fallback
  const title = props.title || 'I am the standard title'

  // the callback property is guaranteed as it is set to isRequired
  // if the text prop is not set the defaultProps object will set it
  return (
    <div>
      <button onClick={props.callback}>Button</button>
      <p>{props.title}</p>
      <p>{props.text}</p>
    </div>
  )
}

Component.propTpyes = {
  title: PropTypes.string,
  callback: PropTypes.func.isRequired,
  text: PropTypes.string,
}

Component.defaultProps = {
  text: 'i am the default text'
}
```

The `propTypes` and the `defaultProps` objects are static and can be assigned to statful *and* stateless components. The `propTypes` can be used to specify which property has to be of what type and to specify which component is required. To read about the full potential of `PropTypes` it is advised to have a look at React's documentation ([Fac13, Typechecking With PropTypes]). The `defaultProps` object can be used to set props to a default value if they are not set by the user of the component. By doing this the developer of the component does not have to implement fallback solutions if the property is undefined.

It is important to note that the property check is disabled when ReactJS is bundled in production mode as the type checks are only viable during the development of the application and introduce scripting overhead. The default properties on the other hand get set even when the production mode is enabled.

3.3.6 Virtual DOM

The most expensive operations when it comes to rendering anything in the browser are DOM manipulations by JavaScript. React solves this problem by using a concept named “Virtual DOM” or “shadow DOM”. As stated in [Fac13, Docs: Reconciliation] ReactJS is a declarative framework that allows the user not having to think about the changes that have to applied to the current built up DOM elements. When using frameworks like jQuery, DOM elements have to be referenced directly in order to manipulate them. ReactJS on the other hand builds up its own Virtual DOM which is a plain old JavaScript object, calculates any changes with its built in diffing algorithm and applies possible changes to the DOM making the framework very fast. Instead of having to tell the browser what elements to update it is possible to tell React how the application should look like at any given state (declarative) as it can be heard in [Fac17, 1:50]. This mechanism is called the “reconciliation algorithm” and is a part of the core algorithm of React.

Internally using a Virtual DOM is not only useful in the web where there is an actual DOM but also in other hardware environments as stated in [Fac17, 2:10]. Facebook decided to divide the diffing algorithm and the part of the code that actually handles pending changes in the DOM which led to the distinction between the “renderer” and the “reconciler”. The renderer was transformed into a pluggable part of ReactJS enabling the framework to be available on other platforms as well. React Native for example is a ReactJS framework that makes it possible to produce mobile applications that can be compiled for Android and iOS by simply switching out the renderer part of the ReactJS core algorithm.

3.4 Redux

In Chapter 2 the programming pattern Flux was introduced. In the early days of ReactJS, Flux had to be implemented by the developers themselves because there was no framework that could handle application state management by using the Flux pattern at the time. To encourage developers to use the Flux pattern the Redux framework was born in May 2015 which can be researched in [Abr15]. The library is basically a state container that is designed with the Flux application architecture in mind. Redux can be used for any application but it suits very well to be used in conjunction with ReactJS. If the reader decides to utilize the Flux architecture to program a web application, it is definitely a good idea to use Redux. The library makes it exceptionally easy to develop an application with the new paradigm in mind and takes the overhead of having to design a Flux architecture by the programmer away.

3.4.1 Introduction and overview

With Redux it is very easy to implement a global state on any front end application. Like the Flux design, Redux also consists of components like the Action and the Store component with a Dispatcher handling incoming actions. Redux is a little bit different than Flux though. Like explained in the Section 2.1.3, Flux can have multiple stores that can be registered via a callback. Redux only has one store and achieves this by composing all stores into one store. The goal is, that actions can be dispatched onto the store resulting into a mutated state that can then be set and passed to any connected ReactJS component.

Like described in Section 2.1.5, an Action can be created by any user interaction and has to be dispatched to the store. As it can be read in [Abr15, Actions] an action has to return an Object containing all necessary information for the reducer which will be introduced in the next section. An example of an Action could be the following code snippet:

```
const mutateStoreData = (data) => {
  return {
    type: MUTATE_DATA,
    data,
  }
}
```

As per default, it is rather complicated to perform asynchronous operations in an Action. The documentation [Abr15, Async Actions] suggests to use a middleware (redux-thunk[13]) that allows actions to have a callback to perform asynchronous actions. An example could be the following code snippet:

```
const asyncMutateStoreData = (params) => (dispatch) => {
  doSmthAsync(params, (error, data) => {
    // this is the callback function context of doSmthAsync
    // it gets called once the async operation is complete
    // this could be a file system operation or an API call
    if (error) throw new Error('error')
    // the dispatch callback can be used to resolve asynchronously
    // data from the action can be dispatched to the store at any point in time
    dispatch({
      type: MUTATE_DATA,
      data,
    })
  })
}
```

The so called “Reducer” is a very important aspect of Redux as it can be read in [Abr15, Reducers]. The reducer is responsible for taking the store’s data, changing it by using a pure function that takes the current state and an action and returns the newly calculated state. It is of utter importance to understand, that the state is **not** mutated. The newly evaluated state is then set as the new current global state. This makes applications very easy to debug because there are no side effects in the reducer as it is a pure function. There is even a tool[14] which can restore any previously set states of the store, creating the possibility to “time travel” through all different versions of the application state. It is exceptionally easy for the developer to restore any previously set state which increases the debugability of the application greatly.

The following code example shows how a reducer function could look like:

```
const reducer = (state, action) => {
  switch(action.type) {

    case MUTATE_DATA:
      state.data = action.data
      return state

    default:
```

[13]https://github.com/gaearon/redux-thunk
[14]https://github.com/gaearon/redux-devtools

```
        return state
    }
}
```

3.4.2 Connection to ReactJS

Redux is very often used in conjunction with ReactJS. According to the statistics of Sacha Greif [Sac16, Front-End Frameworks] about 60% of developers that use ReactJS also use Redux for state management. To use Redux with React a package named `react-redux`[15] has to be used in order to connect React components to the Redux store. The react-redux package provides a `connect()` function that can be used to compose a React component that is intended to be connected to the store.

It was promised before to show why Flux-Redux works well with React. The reason why Redux is perfect for React is, that the application state is passed to a component as a prop. As the reader should know by now, React components only update when either props or the state change. Through the react-redux package certain parts of the state can be mapped to a component only causing rerenders when the corresponding prop that was mapped from the state changed.

The following code example shows how easy it is to map the stores state to a component. The `connect()` function takes a state mapping function as a parameter. The `connect()` function is composable and therefore has to be invoked a second time by passing in the component that has to be connected to the store. The first invocation of `connect()` returns a function that will take either a stateful or stateless component. The code example also shows how to dispatch an action to the store. When composing a component with the `connect()` function, the `dispatch()` function automatically gets added to the props of the connected component.

```
const mapStateToProps = (state) => {
  const { dataToExtract } = state
  return { dataToExtract }
}

const MyComponent = (props) => {
  // this component can access the store's current state
  // every time the mapped state changes, this component will rerender
  return (
    <div>
      <button onClick={() => this.props.dispatch(mutateStoreData('mutated!!!')}></button>
      {props.dataToExtract}
    </div>
  )
}

// this is where the magic happens and where the component is connected
export default connect(mapStateToProps)(MyComponent)
```

[15] https://github.com/reactjs/react-redux

3.4.3 Immutability of the store's data

Probably the most important aspect to consider when using Redux with ReactJS is to use a library that makes the Redux's store data immutable. Immutability is a very complex topic when programming applications with JavaScript. As a developer you can never be sure what operations mutate the objects and what functions clone the object and return the mutated object as the function's result. There are several packages that can be used to make the store's data of Redux immutable, like the "seamless-immutable" package. These immutability libraries ensure the programmer that a JavaScript object created by any chosen immutability library cannot be mutated directly. It has to be done by using the libraries API or by creating a new object.

The question that arises is: "Why is immutability so important when using Redux?" The answer to that question is very simple: Once React applications get very big, performance can be a serious problem if too many rendering cycles happen at the same time. ReactJS provides an API to check if a component should be updated or not. This is usually handled by a shallow comparison of the objects or can be implemented by the developer by overriding the lifecycle method `shouldComponentUpdate()`. When mapping a whole component tree to the component, a shallow compare will not be sufficient to note any changes in the JavaScript state object if it was changed deep down in the object tree. The solution to this problem would be a deep compare but having to iterate the whole state object tree can be a very expensive operation in addition to a complete rerender of the component.

If many state changes occur in a very short amount of time the comparing function has to be executed on every prop or state change. That is the very reason why the comparing function should be as performance efficient as possible. With an immutable object the comparison function can just directly compare the objects which is an extremely cheap JavaScript triple equals comparison. The object cannot be mutated by a simple JavaScript operation and once the state has to be mutated it has to be done by the immutability library or by creating a new object. Either one of these cases cause the JavaScript triple equals comparison to not return true therefore causing a rerendering cycle. If the previous and the current state object are equal the component knows that the state has not mutated and will not rerender.

In the Section 3.4.1 a code example of a possible reducer was shown. In that example the object is mutated directly by accessing a property of the store object. In JavaScript mutating an object by accessing a property does not mutate the whole object but only that property. To notice any changes the object cannot be compared by JavaScript's triple equals operator but one would have to write a deep comparison function comparing all properties of the object. When using immutable objects the object itself knows when it was mutated even if the mutation happened deep down in the object tree making the comparison with the triple equals operator possible again.

As the time of writing this, ReactJS is in the version 15.5.4. This version not only provides a `React.Component` base class to extend from, but also a `React.PureComponent` class that automatically implements the `shouldComponentUpdate()` method. This special component achieves this by automatically shallow comparing all current props with every new incomming prop. When using an immutable store this component type gets highly useful as the programmer does not have to implement the `shouldComponentUpdate()` method.

3.4.4 Advantages and disadvantages of using Flux-Redux

Probably the biggest advantage of using Flux-Redux in any React application is the possibility to store a global application state that lives as long as the user is actually using the web application. This can improve user experience greatly. A good example would be the possibility to being able to trigger a function that displays an error modal from any part in the application. The modal's state could be `{ visible: true }` which tells it to be displayed no matter what state the rest of the application is in and what screen component is displayed at the time of triggering the function. Another good example would be user input. The user inputs some form data but changes the view causing different components to be displayed. Once the user comes back to the form component the state is still saved in the Redux store.

A big disadvantage is the amount of boilerplate code that is necessary to create actions and reducing functions. Once the programmer decides to divide the application into more than one store all of these stores have to be somehow composed before they can be used by the Redux library. The framework has an API that lets programmers compose stores and middlewares but it is still boilerplate code that costs time.

Once there is a part of the app that needs to store state in any store several steps have to happen that cost time and programming overhead as well. First an Action has to be implemented that possibly handles API calls and prepares some data for the store. An action constant has to be created as well in order to being able to dispatch the action to the stores. The action constant is the action type that gets parsed in the store's reducer function to decide how to reduce the data. Finally, the corresponding reducing function in the store has to be implemented. Once all of these components are finished they somehow have to be connected to the ReactJS component that will utilize the newly implemented logic. As mentioned before, the package `react-redux` provides an API to compose any React component to be connected to the store. In order to being able to do this mapping functions have to be written that not only map the store's state to the components props but also maps the corresponding actions to the props of the component so that they can be used inside the component. The programming overhead can be considered as big disadvantage but it guarantees unidirectional dataflow, debugability, scaleability, and testability of the application.

3.5 Application Architecture

Now that the reader is a more familiar with React's concept of stateful and stateless components and knows more about how to store application state efficiently it is time to talk about best practices to build well performing and scaling ReactJS web applications as promised at the beginning of this chapter. Of course there are no definite best practices, this paper is based on personal research and personal experience. The reader is invited to evaluate every suggestion him- or herself and form his or her own opinion.

In most cases it is the best practice to think about the fundamental application architecture before starting the project. The application should strictly be divided into smart and dumb components. Currently a community standard is arising that divides web applications into pages (smart components) that handle all the application logic that is relevant for the page and components (dumb components) that can be reused throughout the whole application. Good candidates for stateless components would be a button with a certain style attached to

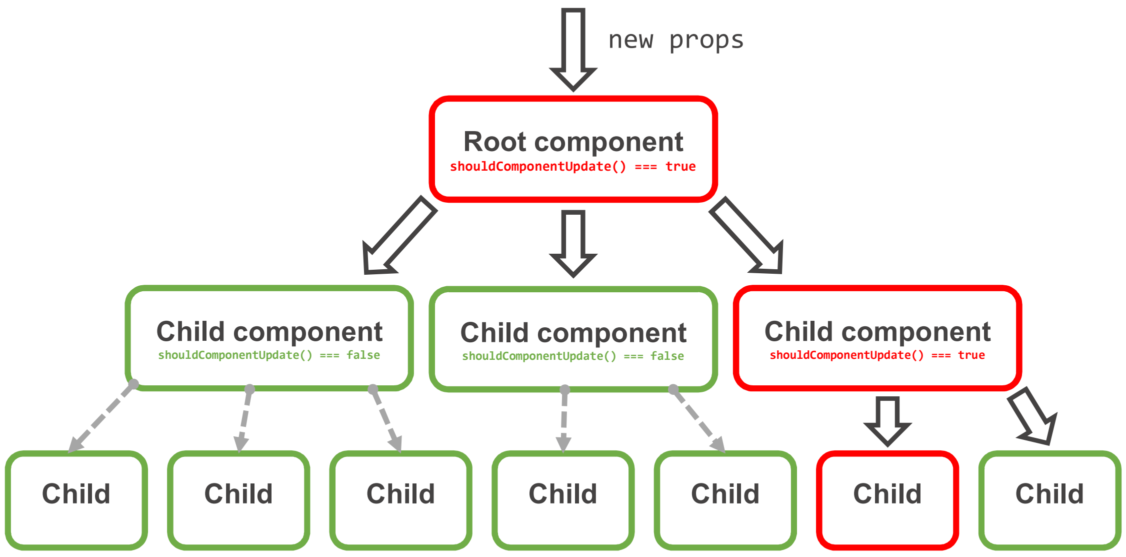

new props
Root component
shouldComponentUpdate() === true
Child component
shouldComponentUpdate() === false
Child component
shouldComponentUpdate() === false
Child component
shouldComponentUpdate() === true
Child
Child
Child
Child
Child
Child
Child

Figure 3.4: React render cycle chain example

it or a component that can handle page layout. Instead of using native DOM elements every application should reach a point in time where the development of stateless components allows to construct a page by using only own components from the existing component directory.

If there is the need to use a stateful component use the `PureComponent` base class from the ReactJS API or the `shouldComponentUpdate()` lifecycle method. It is always a good idea to carefully consider what prop and state changes should cause the component to rerender. Nevertheless, using stateful components should be avoided where ever possible. The best practice would be to only have one parent component that implements all the component logic via member methods and passes them down to all children. The application is much easier debugable because there is only one component where logic is executed and where the data comes from.

Never map child components to the Redux store. The best practice is to only connect page components to the store and pass all relevant data and functions as props to the child components as mentioned before. By strictly following this best practice the structure of the application is much easier understandable and readable. The code base gets more maintainable as new contributors exactly know where all the logic operations happen and where data comes from. When also connecting child components to the store the code base can get very confusing and it is harder for the developer to prevent unnecessary component rerendering cycles.

Another advice is to split the application into logical component sections. In ReactJS child components only rerender if their parent components went through an update cycle thus changing the child component's props. To prevent massive rerendering chain reactions it is only necessary to prevent a parent component from rerendering (either by using React's base class `PureComponent` or by implenting the `shouldComponentUpdate()` method) and a side effect of this is that none of its child components will rerender neither as it can be observed in the Figure 3.4.

Use the new fat arrow syntax of ES6 as it makes the code look more clean and also resolves the problem of binding the correct context to utility functions that have to be passed to child components. The following code example will show the difference between the different approaches and how the code can be kept clean by using fat arrow functions:

```
const ChildComponent = (props) => {
  return (
    <div>
      <button onClick={props.action}>will cause error</button>
      <button onClick={props.correctlyBoundAction}>will mutate state</button>
      <button onClick={props.oldWayOfBindingAction}>will also mutate state</button>
    </div>
  )
}

class ParentComponent extends React.PureComponent {

  constructor(props) {
    super(props)
    // this is ugly boilerplate code
    // it was necessary before fat arrow functions were introduced with ES6
    this.oldWayOfBindingAction.bind(this)
  }

  action() {
    // when executed in the child component this is undefinded -> error
    this.setState({ value: 'mutated' })
  }

  correctlyBoundAction = () => {
    // through the arrow function the this context is correctly bound to the function
    this.setState({ value: 'mutated' })
  }

  oldWayOfBindingAction() {
    // this context is bound in the constructor
    this.setState({ value: 'mutated' })
  }

  render() {
    return (
      <div>
        <ChildComponent
          action={this.action}
          correctlyBoundAction={this.correctlyBoundAction}
          oldWayOfBindingAction={this.oldWayOfBindingAction}
        />
      </div>
    )
  }
}
```

Experience shows, that sometimes it makes sense to swap out functional components with classes that extend the `PureComponent` base class. It makes the component a stateful component that has to go through all lifecycle methods but in some cases where a big amount of data has to be passed to a component or the component is expensive to render it sometimes makes more sense to use a `PureComponent` than a pure function to prevent unnecessary rerendering cycles.

It is strongly advised to implement the `propTypes` object of every component. By doing so the application is easier to understand for developers that are new to the code base. Developers who want to use or debug any specific component can easily check what properties exist for the component and what properties are mandatory to being set. This increases the process of understanding each component's logic even if the code is rather complicated.

Chapter 4

Prototype "Julep Search"

In this final chapter of this thesis the reader will be introduced to the prototype that was developed during the research time of this paper. The understanding of all topics that were presented in this thesis should be greatly improved by going over an actual practical example of a real world ReactJS web application. Many best practice and application architecture suggestions are based on the practical experience that was collected during the development of the prototype.

4.1 Introduction to the project

First there will be an introduction to what Julep Search actually is about and what the tool is trying to achieve. This prototype partly was developed while working in an internship at a small web agency in Vienna which is called "Agentur Mint"[1]. One branch of this company called "Mint Square"[2] is supporting companies with programmatic advertising know-how and and consultancy. The prototype which is presented is a tool which was built for this branch of the Mint agency to alleviate the tedious process of searching large databases for client specific programmatic advertisement data.

4.1.1 Motivation

Mint Square actually has a database for many different advertising platforms and at one point in time there was the need for a tool that could sort and visualize large amounts of raw advertisement information data. That data can be used to optimize the placement of advertisements and help companies to specifically put advertisements on spots that are most relevant to be noticed and realized by the users. This process is called "programmatic advertisement".

The motivation is to easily process programmatic advertisement data that can then further processed by the collaborators of the Mint-Square team. The most important requirement is to make all data searchable by embedding data into an interface that is easily operable.

[1]https://www.agentur-mint.com/
[2]https://www.mint-square.com/

4.1.2 Usecase

Once a company wants to start using programmatic advertising for their web appearance it is necessary to get some information what kind of advertisement would work best in what kind of situation. The target audience has to be evaluated and also what advertisements would work best for that audience. This tool queries a large database for all kinds of key information. For example there are queries for keywords, categories, or inventories. The results then show the corresponding advertisement providers, their request counter and additional information that can be used by the employees of Mint Square to evaluate whether the advertisement provider is valid or not.

4.1.3 Used technologies

The application itself is a staticly served single page application built with ReactJS. It is hosted on an Amazon S3 bucket, which is a simple static file hosting server that is extremely cheap to operate. As it was explained before in this paper, bundling a web application by technologies like Webpack makes it easy to host the application on any platform as only static files have to be served to the client.

The database that contains all relevant advertisement data is an Amazon Elastic Search container[3]. Elastic search is a database that makes querying data with special context very easy. It has a built in matching percentage algorithm that lets the programmer query the most relevant data and also sort the data after its relevancy.

Julep Search does not need a back end server other than its database. For this project a serverless[4] approach was chosen. Serverless is a technology that makes heavy use of Amazon's lambda functions completely omitting the necessity to operate a server for any API calls. Lambda functions won't be explained in this paper as it would go beyond the scope of this paper. Information about Amazon's lambda functions can be read on Amazons official documentation[5].

4.2 The prototype

When designing the prototype the most important goal was to implement a ReactJS web application that conforms to all best practices at the time (summer 2016). The emphasis was to have a code base that is easily readable and maintainable once the project is not actively maintained by the initial developer anymore. This section will give a quick overview over the most important aspects of the prototype and explain all thought and architectural decisions that were made.

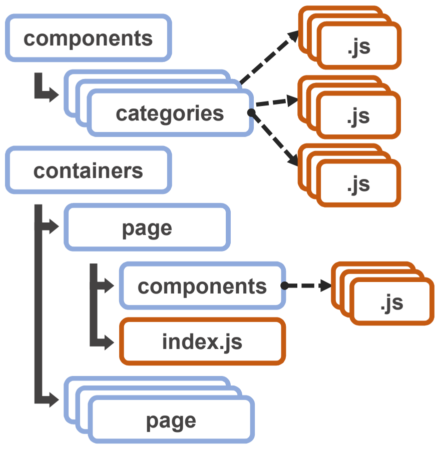


Figure 4.1: Basic application architecture

4.2.1 Architecture

The prototype was implemented keeping all best practices in mind that were explained in the Section 3.5. In real world applications it is very complicated to entirely realize all best practices. The architecture of the prototype is also a little bit out of date as it was developed in summer 2016 and newer best practices and newer community standards have arisen until the time of this writing (May 2017).

The basic architecture of the application splits all parts into containers and components. Like mentioned before, the used application architecture is slightly out of date but it also more or less divides components into smart and dumb components. The Figure 4.1 shows the prototype's current architecture. Every screen of the application has its own container section. The fact that all pages are divided into their own containers makes it easy for Webpack to codesplit the application bundle to prevent the user from having to download the complete application code if only the login screen is shown.

The biggest caveat of the current architecture is that every container still has its own specific components that are only used by that very container as the Figure 4.1 tries to visualize. Application logic is scattered across multiple components which makes the whole application more difficult to understand. Oftentimes it is unclear what component of the container is connected to the store and what component handles which part of the container's logic. The solution to that problem is explained later in the Section 4.2.5.

The components section of the application contains all UI elements that can be used to render simple components like a button or a text input component. As mentioned in the Chapter 3.5 the whole application should be expendable by only using existing UI components from the component library.

[3]https://www.elastic.co/
[4]https://serverless.com/
[5]http://docs.aws.amazon.com/lambda/latest/dg/welcome.html

4.2.2 Component

As mentioned before it is very important to split the application into logical components that can be reused. A good example for a simple component would be the button component. The component wraps additional functionality around the native button DOM element that can be used throughout the whole application more comfortable and do avoid code duplication. The following example shows the basic button component of the prototype:

```
/* ========================= LOAD modules =========================== */

import React, { PropTypes } from 'react'
import classNames from 'classnames'

/* ============================================================== */

const Button = (props) => {

  const className = classNames('button', props.className, {
    'button-primary': props.primary,
    'button-secondary': props.secondary,
    'button-disabled': props.disabled,
    'button-success': props.success,
    'button-failure': props.failure,
  })

  return (
    <button
      className={className}
      onClick={props.onClick}
    >
      {props.children}
    </button>
  )
}

Button.propTypes = {
  onClick: PropTypes.func.isRequired,
  className: PropTypes.string,
  primary: PropTypes.bool,
  secondary: PropTypes.bool,
  disabled: PropTypes.bool,
  success: PropTypes.bool,
  failure: PropTypes.bool,
}

export default Button
```

In the prototype buttons can have multiple appearance variances. For instance, a button can be disabled or have a primary or secondary appearance. This can easily be controlled via “props” in ReactJS. In the following example the button can have many different properties that control how the button looks like. The `Button.propTypes` object shows, what properties can be set in order to change the appearance of the component. One example would be the `primary` property that transforms the button into a primary look. The `Button.propTypes` object also shows, that none of the properties is required so the button has a standard fallback appearance if no property is set.

The property `onClick` has to be a callback function that gets executed when the button is pressed. The static `propTypes` object of the Button component has an `onClick` property that is set to `PropTypes.func.isRequired` which requires the property to be a function that must not be undefined when the component is used. Every time the button component gets clicked, React will handle the event and execute the passed callback function which has to be implemented by the user of the Button component in order to trigger some application logic.

Last but not least there is an implicit property that can be set. If a React component has children, they automatically get attached to the properties of the parent element. The children can be accessed in the parent component via `props.children`. This enables the component to process its children before embedding or rendering them. React provides a top-level API which can modify or rearrange children as it can be read in React's documentation [Fac13, React]

4.2.3 Container

The following example shows how a standard stateful component looks like in ReactJS:

```
/* ========================= LOAD modules =========================== */

import React, { Component } from 'react'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'
import {
  initialLoad,
  updateFilterCriteria,
  updateData,
  exportData
} from 'store/actions/data'

/* ======================= LOAD components ========================== */

import FlexContainer from 'components/common/flexContainer'
import DataTableView from './dataTableView'
import FilterCriteria from './filterCriteria'

/* ================================================================== */

const mapStateToProps = (state) => {
  return ({
    data: state.data,
    user: state.user,
  })
}

const mapDispatchToProps = (dispatch) => {
  return (
    bindActionCreators(
      {
        initialLoad,
        updateFilterCriteria,
        updateData,
        exportData,
      },
      dispatch,
    )
  )
}

class SearchDomainsPage extends Component {

  componentDidMount() {
    this.props.initialLoad()
  }

  componentWillReceiveProps(nextProps) {

    const filter = this.props.data.get('filter')
    const nextFilter = nextProps.data.get('filter')

    if (!filter.equals(nextProps.data.get('filter'))) {
      this.props.updateData(nextFilter)
    }
  }

  render() {

    // the properties are "destructured" from the this object
    // destructuring is an ES6 feature
    // the data, updateFilterCriteria and the exportData variables are extracted
    const { props: { data, updateFilterCriteria, exportData } } = this

    return (
      <div className="domain-search-page">
        <FlexContainer>
          <FilterCriteria
            updateFilterCriteria={updateFilterCriteria}
            data={data}
          />
          <DataTableView
            exportData={exportData}
            data={data}
          />
        </FlexContainer>
      </div>
    )
  }
}

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(SearchDomainsPage)
```

Note how the component is connected to the Redux store of the application at the end where the component gets exported. In this advanced example the full potential of Redux is used as also the store's actions are bound to the component's properties. This feature was not described in this thesis as it would have gone beyond the scope. To learn about `react-redux`'s full potential the documentation can be read [6].

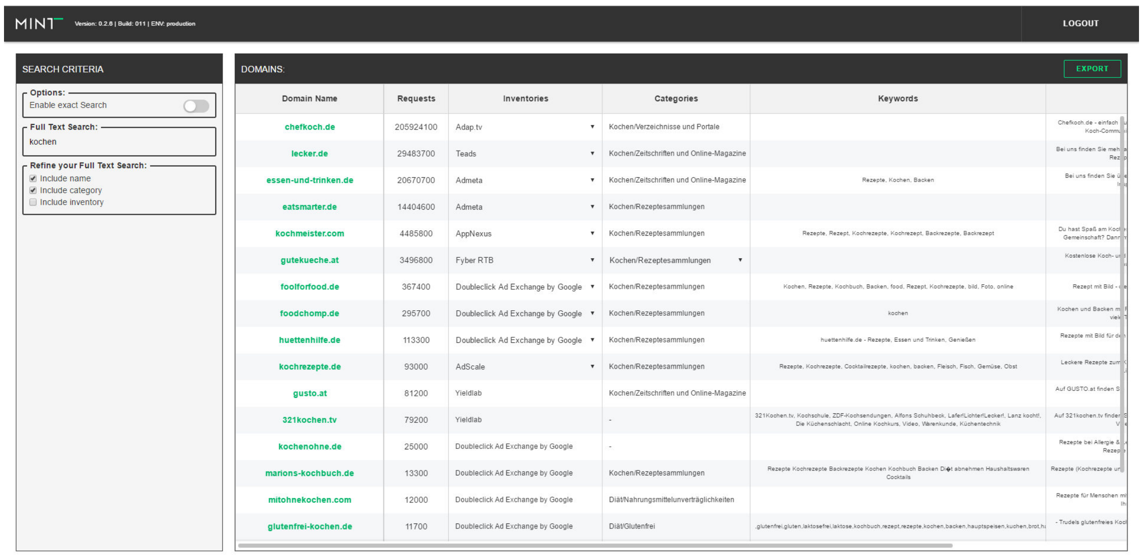
[6] https://github.com/reactjs/react-redux

The `componentWillReceiveProps` method is implemented which can react to the passed props of this component. If there was an update in the advertisement filter that queries the elastic search cluster the component will update itself via the provided callback function `updateData()` that has to be passed as prop to the component.

Because this component is a stateful component it provides the data from the store to all of its child components that are mostly dumb or stateless components in this case.

4.2.4 User interface explanation and demo

The basic screen in Figure (4.2) shows the basic user interface. Some search criteria has already been applied and a list of domains and its corresponding relevant data is displayed. The list is sorted by the number of advertisement requests each domain gets. The panel "search criteria" can be used to refine the search parameters to enable more granular searches of the database. All results can also be exported as CSV files by clicking the "EXPORT" button in the top right corner of the results panel.



Options:
Enable exact Search

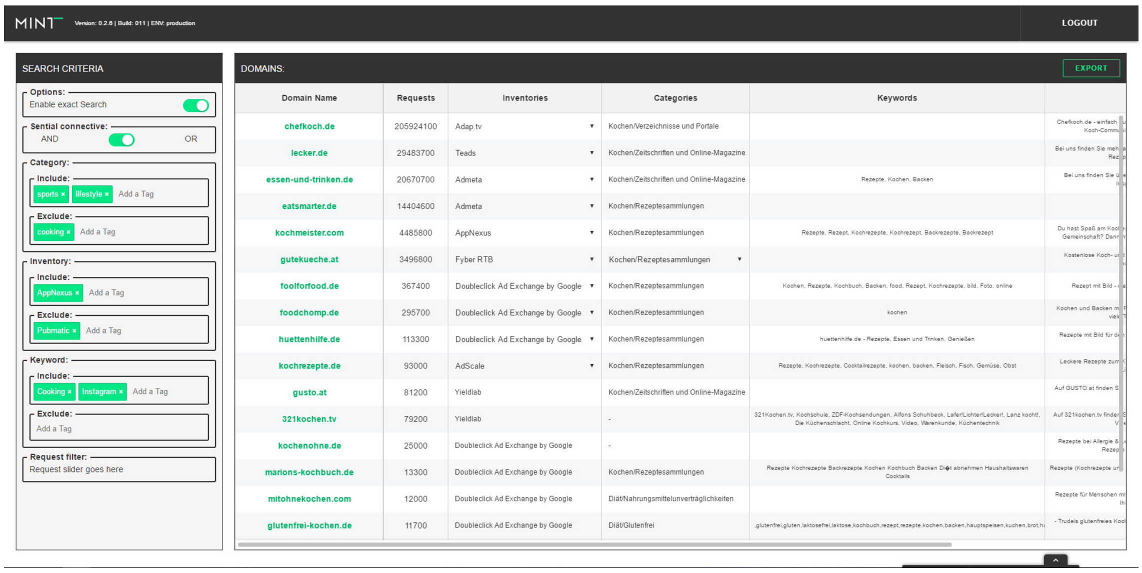
Full Text Search:
kochen

Refine your Full Text Search:
[✓] Include name
[✓] Include category
[ ] Include inventory

Domain Name	Requests	Inventories	Categories	Keywords	
chefkoch.de	205924100	Adap.tv ▾	Kochen/Verzeichnisse und Portale		Chefkoch.de - einfach Koch-Commu
lecker.de	29483700	Teads ▾	Kochen/Zeitschriften und Online-Magazine		Bei uns finden Sie meh Rez
essen-und-trinken.de	20670700	Admeta ▾	Kochen/Zeitschriften und Online-Magazine	Rezepte, Kochen, Backen	Bei uns finden Sie ü
eatsmarter.de	14404600	Admeta ▾	Kochen/Rezeptesammlungen		
kochmeister.com	4485800	AppNexus ▾	Kochen/Rezeptesammlungen	Rezepte, Rezept, Kochrezepte, Kochrezept, Backrezepte, Backrezept	Du hast Spaß am Koc Gemeinschaft? Dan
gutekueche.at	3496800	Fyber RTB ▾	Kochen/Rezeptesammlungen ▾		Kostenlose Koch- u
foolforfood.de	367400	Doubleclick Ad Exchange by Google ▾	Kochen/Rezeptesammlungen	Kochen, Rezepte, Kochbuch, Backen, food, Rezept, Kochrezepte, bild, Foto, online	Rezept mit Bild -
foodchomp.de	295700	Doubleclick Ad Exchange by Google ▾	Kochen/Rezeptesammlungen	kochen	Kochen und Backen m viel
huettenhilfe.de	113300	Doubleclick Ad Exchange by Google ▾	Kochen/Rezeptesammlungen	huettenhilfe.de - Rezepte, Essen und Trinken, Genießen	Rezepte mit Bild für d
kochrezepte.de	93000	AdScale ▾	Kochen/Rezeptesammlungen	Rezepte, Kochrezepte, Cocktailrezepte, kochen, backen, Fleisch, Fisch, Gemüse, Obst	Leckere Rezepte zum
gusto.at	81200	Yieldlab	Kochen/Zeitschriften und Online-Magazine		Auf GUSTO.at finden S
321kochen.tv	79200	Yieldlab	-	321Kochen.tv, Kochschule, ZDF-Kochsendungen, Alfons Schuhbeck, Lafer!Lichter!Lecker!, Lanz kocht!, Die Küchenschlacht, Online Kochkurs, Video, Warenkunde, Küchentechnik	Auf 321kochen.tv finde V
kochenohne.de	25000	Doubleclick Ad Exchange by Google	-		Rezepte bei Allergie & Rezep
marions-kochbuch.de	13300	Doubleclick Ad Exchange by Google	Kochen/Rezeptesammlungen	Rezepte Kochrezepte Backrezepte Kochen Kochbuch Backen Di�t abnehmen Haushaltswaren Cocktails	Rezepte (Kochrezepte un
mitohnekochen.com	12000	Doubleclick Ad Exchange by Google	Diät/Nahrungsmittelunverträglichkeiten		Rezepte für Menschen mi Ih
glutenfrei-kochen.de	11700	Doubleclick Ad Exchange by Google	Diät/Glutenfrei	,glutenfrei,gluten,laktosefrei,laktose,kochbuch,rezept,rezepte,kochen,backen,hauptspeisen,kuchen,brot,h	- Trudels glutenfreies Koc

Figure 4.2: Applied search criteria in normal search

The following screen in Figure 4.3 shows a mockup of how the exact search could look like. As this is only a prototype the functionality was not fully implemented yet as there was no immediate demand for the feature.



SEARCH CRITERIA

Options:
Enable exact Search

Sential connective:
AND OR

Category:
Include: sports × lifestyle × Add a Tag
Exclude: cooking × Add a Tag

Inventory:
Include: AppNexus × Add a Tag
Exclude: Pubmatic × Add a Tag

Keyword:
Include: Cooking × Instagram × Add a Tag
Exclude: Add a Tag

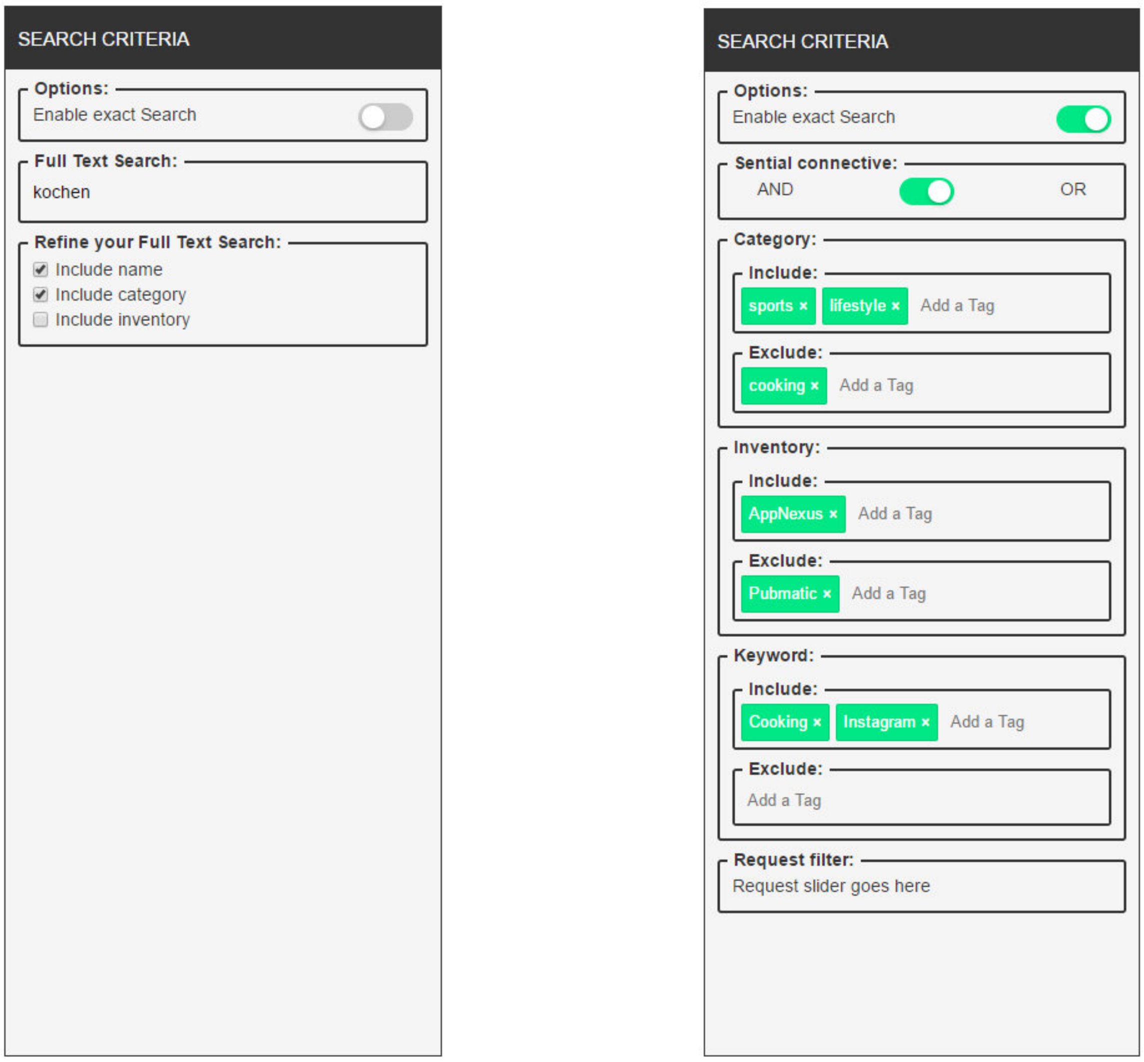
Request filter:
Request slider goes here

DOMAINS:

Domain Name	Requests	Inventories	Categories	Keywords	
chefkoch.de	205924100	Adap.tv	Kochen/Verzeichnisse und Portale		Chefkoch.de - einfach Koch-Commu
lecker.de	29483700	Teads	Kochen/Zeitschriften und Online-Magazine		Bei uns finden Sie meh Rez
essen-und-trinken.de	20670700	Admeta	Kochen/Zeitschriften und Online-Magazine	Rezepte, Kochen, Backen	Bei uns finden Sie ü
eatsmarter.de	14404600	Admeta	Kochen/Rezeptesammlungen		
kochmeister.com	4485800	AppNexus	Kochen/Rezeptesammlungen	Rezepte, Rezept, Kochrezepte, Kochrezept, Backrezepte, Backrezept	Du hast Spaß am Koc Gemeinschaft? Dann
gutekueche.at	3496800	Fyber RTB	Kochen/Rezeptesammlungen		Kostenlose Koch- u
foolforfood.de	367400	Doubleclick Ad Exchange by Google	Kochen/Rezeptesammlungen	Kochen, Rezepte, Kochbuch, Backen, food, Rezept, Kochrezepte, bild, Foto, online	Rezept mit Bild -
foodchomp.de	295700	Doubleclick Ad Exchange by Google	Kochen/Rezeptesammlungen	kochen	Kochen und Backen m viel
huettenhilfe.de	113300	Doubleclick Ad Exchange by Google	Kochen/Rezeptesammlungen	huettenhilfe.de - Rezepte, Essen und Trinken, Genießen	Rezepte mit Bild für d
kochrezepte.de	93000	AdScale	Kochen/Rezeptesammlungen	Rezepte, Kochrezepte, Cocktailrezepte, kochen, backen, Fleisch, Fisch, Gemüse, Obst	Leckere Rezepte zum
gusto.at	81200	Yieldlab	Kochen/Zeitschriften und Online-Magazine		Auf GUSTO.at finden S
321kochen.tv	79200	Yieldlab	-	321Kochen.tv, Kochschule, ZDF-Kochsendungen, Alfons Schuhbeck, Lafer!Lichter!Lecker!, Lanz kocht!, Die Küchenschlacht, Online Kochkurs, Video, Warenkunde, Küchentechnik	Auf 321kochen.tv finden V
kochenohne.de	25000	Doubleclick Ad Exchange by Google	-		Rezepte bei Allergie & Rezep
marions-kochbuch.de	13300	Doubleclick Ad Exchange by Google	Kochen/Rezeptesammlungen	Rezepte Kochrezepte Backrezepte Kochen Kochbuch Backen Di�t abnehmen Haushaltswaren Cocktails	Rezepte (Kochrezepte un
mitohnekochen.com	12000	Doubleclick Ad Exchange by Google	Diät/Nahrungsmittelunverträglichkeiten		Rezepte für Menschen mi Ih
glutenfrei-kochen.de	11700	Doubleclick Ad Exchange by Google	Diät/Glutenfrei	,glutenfrei,gluten,laktosefrei,laktose,kochbuch,rezept,rezepte,kochen,backen,hauptspeisen,kuchen,brot,h	- Trudels glutenfreies Koc

Figure 4.3: How the exact search could look like, only UI is implemented

To get a better overview over the search criteria panel, the following Figure 4.4 provides a better overview of the two different panels. On the left there is the basic search and on the right there is the basic implementation of the exact search panel, that does not have any logic attached to it yet.


SEARCH CRITERIA

Options:
Enable exact Search

Full Text Search:
kochen

Refine your Full Text Search:
- [x] Include name
- [x] Include category
- [ ] Include inventory

SEARCH CRITERIA

Options:
Enable exact Search

Sential connective:
AND OR

Category:
Include: sports × lifestyle × Add a Tag
Exclude: cooking × Add a Tag

Inventory:
Include: AppNexus × Add a Tag
Exclude: Pubmatic × Add a Tag

Keyword:
Include: Cooking × Instagram × Add a Tag
Exclude: Add a Tag

Request filter:
Request slider goes here

(a) normal search (b) exact search

Figure 4.4: Normal and exact search view

4.2.5 What could have been done better

In 2017 a new community standard arose that describes how to structure ReactJS applications as the Figure 4.5 shows. The whole application is divided into pages that can only use components from the component library. This should encourage developers even more to handle state and state-related logic only in one component (being the page component in this case) and pass all necessary logic down to the child components. This ensures that application-relevant logic is only handled in page components making the application much more easy to understand and to maintain.

As stated in the Section 4.2.1, the prototype uses the slightly outdated application architecture that can be seen in the Figure 4.1 that makes it noticeably more difficult to quickly understand the whole application logic. As the project is not maintained anymore the appli-

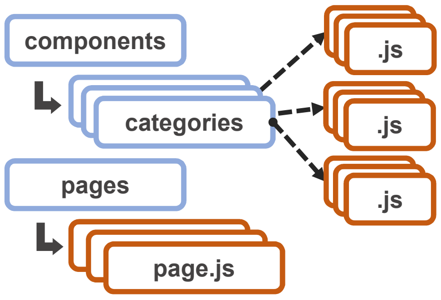


Figure 4.5: Better application architecture

cation was not refactored. The finding that the newer version of the application architecture can alleviate maintenance problems has to be kept in mind during the development of future projects.

Chapter 5

Conclusion

This chapter is a recapitulation of all topics and discussions that were presented in this thesis. It is a conclusion of all findings that were discovered during the development of the prototype and the research phase of this paper.

5.1 Results

The thesis yields very interesting results that can be taken into consideration when having to decide which technology to utilize to start a web application software project. The comparison between Flux and MVC shows that flux is rather similar to MVC but also demonstrates that Flux is still a viable web application architecture. The thesis also gives an overview to web application development with ReactJS in general. It shows what effort it takes to start developing a web application with ReactJS and how the framework can be used to implement real world web applications. Finally, the paper concludes all findings by presenting a prototype that tries to incorporate all results and best practices that are presented in this thesis.

5.1.1 Flux and MVC

In conclusion Chapter 2 shows that Flux and MVC are very similar. Facebook introduced the Flux pattern as a brand new programming pattern but it turns out to be a copy of a slightly changed MVC pattern. Though it must be emphasized that Flux is indeed different to MVC as data is injected actively into the components, whereas MVC depends on data bindings to react to updated data.

There are many versions and mutations of the MVC pattern. In this thesis Flux is compared to the traditional version of MVC as it was developed in 1970 by the lisp community. It must be noted that there are many subversions of MVC that are even more similar to the Flux application architecture. The research of every possible MVC sub-pattern would have gone beyond the scope of this paper.

Even though Flux is very similar to MVC the pattern is still highly usable for developing web applications. The new pattern seems to be specially tailored to be used in conjunction with the ReactJS framework which indeed works very well. Using Flux and ReactJS results in well structured web applications that can be debugged and scaled rather easily as the thesis shows.

5.1.2 Web application development with ReactJS

As discussed in the Section 3.3.1, ReactJS is a very promising framework as it puts an abstraction layer to the view and lets programmers build user interfaces without having to know how the user interface elements are actually created in the background. This feature enables the framework to be used across all platforms. By mastering the declarative syntax of ReactJS the programmer is enabled to reuse components across all platforms including native mobile platforms.

The thesis also shows that ReactJS can be used to develop highly performance optimized web applications. By using all provided best practices it should ease the effort of the reader to develop a well structured, well scaling web application with ReactJS.

There thesis puts a strong emphasis on the importance of having a well thought-through application architecture. In fact, application architecture is important for all kinds of software projects but the thesis should have shown that it is also a very important aspect of web applications. As stated many times throughout this thesis, the application architecture aspect oftentimes gets omitted in web projects which is a big problem the web community currently has to deal with. This thesis tries to alleviate the problem by showing the reader that application architecture is an utterly important aspect of web development.

5.1.3 Prototype

The Chapter 4 shows how ReactJS can be used in a real world scenario. The basic architecture is explained to provide a better overview to the prototype. Its motivation and the goals show that ReactJS can be used for all kinds of application scenarios. As stated in the Section 3.1 and 4.1.3 the prototype also shows that Facebook's framework does not make many assumptions about the development stack and can be used in conjunction with many different web technologies.

As time progresses the whole ReactJS ecosystem is evolving. Since the time of developing the prototype certain best practices were omitted and others were added. In addition to a constantly changing ecosystem the lack of experience of how to develop a well structured ReactJS application complicated the development of the prototype. The last Section 4.2.5 shows what could have been done better.

References

Literature

[Abr15] Dan Abramov. *Redux: Redux is a predictable state container for JavaScript apps.* Ed. by gitbooks. GitHub, 2015. URL: http://redux.js.org/ (cit. on pp. 34, 35).

[Chr15] Christian Alfoni. *Why we are doing MVC and FLUX wrong.* Ed. by Christian Alfoni. 2015. URL: http://www.christianalfoni.com/articles/2015_08_02_Why-we-are-doing-MVC-and-FLUX-wrong (cit. on pp. 16, 17).

[Fac13] Facebook Inc. *React: a JavaScript library for building user interfaces.* Ed. by Facebook Inc. 2013. URL: https://facebook.github.io/react (cit. on pp. 22, 23, 31, 33, 34, 46).

[Fac14b] Facebook Inc. *Flux, an application architecture for building user interfaces.* Ed. by Facebook Inc. 2014. URL: https://facebook.github.io/flux/docs/in-depth-overview.html (cit. on pp. 11–13).

[Mar06] Martin Fowler. *GUI Architectures: ModelViewController.* Ed. by Martin Fowler. 2006. URL: https://martinfowler.com/eaaDev/uiArchs.html#ModelViewController (cit. on pp. 14–16, 18).

[Mat13] Matti Bragge. "Model-View-Controller architectural pattern and its evolution in graphical user interface frameworks". Bachelor's thesis. Helsinki: University of Helsinki, 2013. URL: http://urn.fi/URN:NBN:fi-fe201309024736 (cit. on pp. 14, 16).

[Pau13] Paul Cowan. *Client Side MVC is not server side MVC, erase your brain.* Ed. by Paul Cowan. 2013. URL: http://www.thesoftwaresimpleton.com/blog/2013/03/23/client-side-mvc/ (cit. on p. 17).

[Sac16] Sacha Greif. *The State Of JavaScript: Sacha Greif.* 2016. URL: http://stateofjs.com/2016/frontend/ (cit. on p. 36).

[ste14] stefanoborini. *Understanding Model-View-Controller.* Ed. by gitbooks. 2014. URL: https://stefanoborini.gitbooks.io/modelviewcontroller/ (cit. on pp. 14–16).

[ZH16] Oliver Zeigermann and Nils Hartmann. *React: Die praktische Einführung in React, React Router und Redux.* 1. Auflage. Heidelberg: dpunkt.verlag, 2016 (cit. on pp. 22, 28).

Films and audio-visual media

[Fac14a] Facebook. *Hacker Way: Rethinking Web App Development at Facebook*. Youtube. Facebook talking about ReactJS and their new application architecture. May 2014. URL: https://www.youtube.com/watch?v=nYkdrAPrdcw (cit. on pp. 18, 19, 21).

[Fac17] Facebook. *Lin Clark: A Cartoon Intro to Fiber - React Conf 2017*. Youtube. Facebook talking about the new reconciliation algorithm fiber. Mar. 2017. URL: https://www.youtube.com/watch?v=ZCuYPiUIONs (cit. on p. 34).