MASTERARBEIT

VERGLEICH DES REDUX PARADIGMA MIT ZUSTANDSMANAGEMENT IN BISHERIGEN JAVASCRIPT MV*-ARCHITEKTUREN

Welche Vorteile hat das Redux Paradigma gegenüber dem Zustandsmanagement in bisherigen JavaScript MV*-Architekturen?

ausgeführt an der

am Studiengang
Software Engineering Leadership

Von: Paul Vincent Beigang
Personenkennzeichen: 1420030013

Münnerstadt, am 21.04.2017 .....
Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....

Unterschrift

DANKSAGUNG

Zunächst möchte ich mich bei allen Personen bedanken, die mich bei der Erstellung der Masterarbeit unterstützt haben und diese Arbeit möglich gemacht haben.

Ein besonderer Dank gilt meinem Betreuer Manfred Steyer. Durch seine Unterstützung von der Themenfindung über die Gliederung der Arbeit bis hin zu inhaltlichen Ratschlägen wurde meine Arbeit sehr bereichert. Seine detaillierten Rückmeldungen, die Bereitschaft, Zeit in die Betreuung zu investieren und das offene Ohr, haben die positive Zusammenarbeit bestärkt. Nicht zuletzt hat sein Feedback zur fachlichen sowie wissenschaftlichen Verbesserung der Arbeit beigetragen.

Daneben bedanke ich mich beim Festo Bildungsfond für den Bildungskredit, welcher mir das Masterstudium möglich machte.

Des Weiteren bedanke ich mich bei meiner Familie, allen Kollegen und Freunden, die durch Diskussionen und ihre Meinung zur Masterarbeit beigetragen haben.

Mein besonderer Dank gilt meiner Frau Katja, die mich durch die Wertschätzung der Masterarbeit und die notwendigen zeitlichen Freiräume bei der Erstellung der Masterarbeit unterstützt hat.

KURZFASSUNG

JavaScript ist Sprache des Webs geworden und die Entwicklung von komplexen JavaScript Web Applikationen nimmt stetig zu. Bei der Entwicklung von modernen JavaScript Web Applikationen ist das Zustandsmanagement ein zentraler und erfolgskritischer Themenkomplex. Das Hauptziel der Arbeit ist es, die möglichen Vorteile eines Redux basierten Zustandsmanagement für JavaScript Web Applikationen gegenüber herkömmlichen JavaScript MV*-Architekturen zu erarbeiten.

Bisherige JavaScript-MV* Architekturen boten bislang Kompromiss behaftete und speziell in sehr umfangreichen Web Applikationen keine Lösungen, um das Zustandsmanagement möglichst vorsehbar zu machen. Viele Komponenten, Zustandsinteraktionen und Veränderungen aus verschiedensten Quellen machten es besonders schwer, den aktuellen Applikationszustand transparent darzustellen.

Das Redux Paradigma formuliert und verbreitetet eine explizite Sprache und definierte Regeln, um den Zustand in komplexen Web Applikationen vorhersehbar verwalten zu können. Besonders in Bezug auf die Nachvollziehbarkeit und die Wartbarkeit des Programmcodes haben die Untersuchungen der Masterarbeit ergeben, dass die Implementierung des Redux Paradigmas große Vorteile gegenüber klassischen Zustandsmanagement Konzepten bieten kann. Es existieren eine Vielzahl von Entwicklerwerkzeugen, um die tägliche Arbeit zu unterstützen und zu vereinfachen.

Beim Codeumfang konnten nach den Erkenntnissen der vorliegenden Masterarbeit keine Vorteile des Redux Paradigmas gegenüber einer klassischen Implementierung eines Daten Service für das Zustandsmanagement erkannt werden.

Bezüglich der Render Performance ist eine Abhängigkeit zwischen Datenmenge und durchschnittlicher Renderzeit erkennbar. Bei geringer Daten Menge ist die Render Perfomance gegenüber der Daten Service Implementierung schneller, bei mittlerer Datenmenge sind beide Implementierung beinahe gleich schnell und bei einer hohen Datenmenge weist die Daten Service Implementierung eine bessere Render Performance gegenüber der Redux Implementierung auf.

Die zunehmende Verbreitung und Popularität innerhalb der Entwicklergemeinde zeigt, dass das Redux Paradigma den Puls der Zeit trifft und eine angemessene Lösung für ein weit verbreitetes Problem bieten kann.

Die Ergebnisse der vorliegenden Masterarbeit sollen mehr Klarheit in die Bewertung von Zustandsmanagement Lösungen in JavaScript Web Applikationen bringen und Entscheidern helfen, eine fundierte Auswahl bezüglich eines Konzepts zur Verwaltung des Zustands für ihre individuelle Herrausforderung treffen zu können.

ABSTRACT

JavaScript has become the language of the Web and the development of complex JavaScript Web applications are steadily increasing. In the development of modern JavaScript Web applications, state management is a central and critical issues. The main objective of the work is to develop the potential benefits of a Redux-based state management for Web applications compared to conventional JavaScript MV*architectures.
Previous JavaScript MV* architectures offered so far, prone to compromise, and especially in very large Web applications with no solutions to make the state management predictable. Many of components that condition the interactions and changes from a variety of sources made it particularly difficult for the current application state in a transparent way.

The Redux paradigm formulated and maintained an explicit language and defined rules, to the state in a complex Web applications, predictable to manage. Particularly in relation to the traceability and the maintainability of the program code, the examinations of the master's thesis have shown that the implementation of the Redux paradigm can offer great advantages compared to classic state management concepts. There are a variety of developer tools to support the daily work and simplify.

In the areas examined, Code size of did not show any benefits of the Redux paradigm compared to a classic implementation for state management. The Render Performance measurements are showing fast rendering for less data, similar render times for average data and slower render times for a lot of data in the Redux implementation.

The increasing prevalence and popularity within the developer community shows that the Redux paradigm hits the pulse of time and an appropriate solution to a widespread problem.

The results of the present master's thesis are to bring more clarity in the evaluation of the available state management solutions for JavaScript Web applications, and to help decision-makers to make an informed selection of appropriate concept or paradigm to manage state in JavaScript web applications for each individual context.

GLEICHHEITSGRUNDSATZ

Aus Gründen der Lesbarkeit wurde in dieser Arbeit darauf verzichtet, geschlechtsspezifische Formulierungen zu verwenden. Jedoch möchte ich ausdrücklich festhalten, dass die bei Personen verwendeten maskulinen Formen für beide Geschlechter zu verstehen sind.

INHALTSVERZEICHNIS

1 EINLEITUNG

Die Verwendung der Skriptsprache JavaScript hat sich in den letzten zehn Jahren professionalisiert und JavaScript ist mittlerweile oftmals ein elementarer Bestandteil in moderneren Web Applikationen. Web Applikationen können heutzutage oftmals einen Funktionsumfang und eine Benutzerfreundlichkeit aufweisen, die ähnlich wie bei nativen Desktop Applikationen ist. Um der kontinuierlich steigenden Komplexität, insbesondere bezüglich des erfolgskritischen Themas des Zustandsmanagements in JavaScript Web Applikationen, begegnen zu können, gilt es bisherige Architekturmuster und Konzepte entsprechend den aktuellen Anforderungen anzupassen und weiter zu entwickeln.

1.1 Motivation

JavaScript wird von (Fink & Flatow, Pro Single Page Application Development, 2014, S. 15) als die Sprache des Webs betitelt.

(Osmani A. , JavaScript Application Design - Vorwort, 2015, S. 16) beschreibt, dass der Softwaredesign Prozess von robusten JavaScript Web Applikationen in den letzten Jahren elementare Veränderungen erfahren hat. JavaScript wird verwendet, um immer größere, komplexere und Web Applikationen und Oberflächen zu entwickeln, die von mehr und mehr Benutze- und Intersysteminteraktionen gekennzeichnet sind. Nach Meinung des Autors sei die Zeit gekommen, sich die aus dem klassischen Software Engineering bekannten Architektur- und Designfragen auch im Kontext von JavaScript Web Applikationen zu stellen.

Auch (Freeman, 2012, S. 1) kommt zu dem Schluss, dass die Komplexität von clientseitigen Web Applikationen an einem Wendepunkt angekommen ist. Die Themen Skalier- und Wartbarkeit von Programmcode sind ausschlaggebend und erfolgskritisch geworden. Freeman ist der Meinung, dass die Zeiten in denen schnell zusammengebastelte Lösungen ausreichend waren nun vorbei sind und Lösungen mit einer umfangreich geplanten und komfortabel erweiterbaren Architektur die Basis der Zukunft sind.

In der Literatur findet sich auch die Bezeichnung des „Zeitalters der Single Pape Application“, kurz SPA.

Im „Zeitalter der Single Page Applications“ nutzen und interagieren die Anwender mit den größten Firmen der Welt, Regierungsinstitutionen, NGOs und dem klassischen Online Handel über JavaScript getriebenen Programmcode, der im Web Browser ausgeführt wird.

Auch (Shapiro, Web Component Architecture & Development with AngularJS, 2015, S. 4) stellt fest, dass viele Organisationen bemerken, dass die Größe, die Performanz und die Komplexität des clientseitigen Programmcodes im Laufe der Zeit so schlecht geworden ist, dass eine flüssige Ausführung nicht mehr gegeben ist oder eine Web Applikation mehrere Sekunden zum Laden

braucht. Seiner Meinung nach sind die eben genannten Organisationen gezwungen, den bisherigen Programmcode zu verwerfen und von Grund auf neu entwerfen und anschließend entwickeln. Shapiro nennt unstrukturierten und nicht performanten Code den Begriff „jQuery Spaghetti“ und bezeichnet diesen als nicht mehr zeitgemäß.

In dem genannten Übergang vom „jQuery Spaghetti“ Programmcode zu einer verbesserten Client-Architektur für JavaScript Web Applikationen ist eine der größeren Herausforderungen des Thema Zustand Management. Klassischerweise wurden die Daten während einer Seitenanfrage aus der Datenbank gelesen und das als Antwort ausgegeben. Für hoch interaktive JavaScript Web Applikationen ist das Thema des Zustandsmanagements ein komplett anderer Prozess. Das klassische „Anfrage – Antwort – Modell“ funktioniert nicht mehr. Stattdessen werden die Daten am entfernten Server abgerufen (API) und temporär clientseitig vorgehalten (MacCaw, JavaScript Web Applications, 2011, S. 31).

Des Weiteren stellt der Autor fest, dass das Vorhalten des Zustands auf dem Client große Herausforderungen mit sich bringt. Er beschäftigt sich mit den Fragen, wo der temporäre Zustand gespeichert werden soll. Als Möglichkeiten nennt der Autor die Vorhaltung als Variable oder im Document Object Model (DOM). Zustandsmanagement ist einer der kritischsten Punkte bei der Entwicklung einer JavaScript Web Applikation, an dem viele Entwickler falsche Entscheidungen treffen. Gerade weil der Bereich des Zustandsmanagements so essentiell für die korrekte Funktion und die Wartbarkeit einer JavaScript Web Applikation ist, ist es sehr wichtig, auf diesen Punkt bei der Entwicklung ein großes Augenmerk zu legen (MacCaw, JavaScript Web Applications, 2011, S. 49). Auch (Abramov, Getting Started with Redux, 2016) beschreibt Zustandsmanagement als einen zentralen Punkt einer JavaScript Web Applikation, der in der Praxis oft ohne richtigen Plan implementiert wird.

Die wachsende Notwendigkeit zum strukturierten Zustandsmanagement folgt den immer komplizierteren Anforderungen, die für JavaScript Web Applikationen definiert werden. Auch (Elliot, 10 Tips for Better Redux Architecture, 2016) schreibt, dass die Anforderungen von JavaScript Web Applikationen immer mehr werden. Heutzutage wird eine Vielzahl an Informationen im clientseitigen Zustand festgehalten und Zustandsmanagement gilt generell als keine simple Aufgabe. Insbesondere wenn in wie bisherigen MV*-Applikationen ein Model ein anderes Model aktualisieren und auch der View ein Model aktualisieren kann, welches wiederrum ein weiteres Model verändert und dieses wiederrum einen View aktualisiert wird es unübersichtlich. Ab einem gewissen Punkt ist nicht mehr nachzuvollziehen, was in der selbst entwickelten Web Applikationen passiert und die Kontrolle über den aktuellen Zustand geht verloren. Innerhalb von nicht-deterministischen Systemen ist es schwer, Fehler zu reproduzieren oder neue Funktionalitäten hinzufügen.

Neben diesen genannten Herausforderungen warten noch viele weitere Herausforderungen auf aktuelle JavaScript Web Applikationen. Von den Entwicklern wird erwartet, dass sehr aktuelle Konzepte wie optimistische Datenaktualisierung, serverseitiges Rendering und das Vorladen von Informationen umgesetzt werden können (Abramov & Contributors, Redux Motivation, 2015).

Einer der schwierigsten Aufgaben für Web Entwickler ist es, Programmcode zu schreiben, der flexibel mit der jeweiligen Projektgröße wachsen kann. Je größer und je komplizierter ein Projekt wird, desto schwieriger wird es für den Entwickler, den Programmcode entsprechend zu verwalten. Ohnehin ist es eine große Herrausforderung, Programmcode so zu strukturieren und in einer Form zu schreiben, anhand der die kontinuierliche Wartung und Verbesserung einfach zu realisieren ist. Das gilt natürlich auch für JavaScript Web Applikationen (Scott, SPA Design and Architecture, 2016, S. 22).

> *„Models are the heart of any application because they are the data structures that hold the application data. Without data there wouldn't be an application." (Fink & Flatow, Pro Single Page Application Development, 2014, S. 76)*

Zusammenfassend kann festgestellt werden, dass das Thema Zustandsmanagement bei der Entwicklung von JavaScript Web Applikationen ein kritischer Erfolgsfaktor und gleichzeitig keine triviale Aufgabe ist. Vor dem Hintergrund der stetig zunehmenden Anforderungen und der wachsenden Komplexität in Hinblick auf Funktionalität und Komfort stellt sich die Frage, welche Ansätze, Architekturen und Lösungsmöglichkeiten für die Herrausforderung des Zustandsmanagements in modernen JavaScript Web Applikationen optimaler Weise Verwendung finden.

1.2 Ziel der Arbeit

Das primäre Ziel der vorliegenden Masterarbeit ist es, die möglichen Vorteile des Redux Paradigma gegenüber den bisherigen Ansätzen bezüglich des Thema Zustandsmanagement in herkömmlichen JavaScript MV*-Architekturen zu erarbeiten.

Um den anstehenden Vergleich sinnvoll und logisch vollziehen zu können, ist es ein weiteres Ziel der Arbeit, die Notwendigkeit für ein professionelles Zustandsmanagement in modernen JavaScript Web Applikationen zu erläutern und die Leser für diese Notwendigkeit zu sensibilisieren.

1.3 Vorgehensweise und Gliederung der Masterarbeit

Um das in 1.2 genannte Ziel erreichen zu können, wird zu Beginn der Kontext der vorliegenden Masterarbeit definiert und erklärt, warum Zustandsmanagement ein zentrales Thema ist, das für JavaScript Web Applikationen immer relevanter und wichtiger ist und wird. Des Weiteren werden die zum Verständnis notwendigen Fachbegriffe erläutert, um die Inhalte und den betrachteten Kontext nachvollziehen zu können.

Wie bereits in Kapitel 1.1 dargestellt, entsteht eine immer größer werdende Notwendigkeit, das Thema Zustandsmanagement in JavaScript Web Applikationen umfassend und professionell zu

lösen. Um die Vorteile des Redux Paradigma gegenüber Zustandsmanagement Lösungen in bisherigen JavaScript MV*-Architekturen zu erarbeiten, werden im ersten Teil des Kapitels 2 Zustandsmanagement Lösungen in bisherigen JavaScript MV*-Architekturen vorgestellt und erklärt. Das Erlangen eines Grundverständnisses für das Zustandsmanagement in bisherigen MV*-Architekturen ist ein weiteres Ziel der vorliegenden Masterarbeit. Aufbauend auf dieses Verständnis werden anschließend die Wurzeln des Redux Paradigmas, d.h. das Flux Konzept und daran anschließend das eigentliche Redux Paradigma detailliert erklärt.

Anhand von zwei konkreten Programmcode Implementierungen in Kapitel 3. und weiteren Evaluationsansätzen werden die herkömmlichen Zustandsmanagement Lösungen in MV*-Architekturen mit dem Redux Paradigma verglichen. Die Methodik zur Evaluation der verschiedenen Zustandsmanagement Konzepte verfolgt einen Vergleich nach den folgenden quantitativen Merkmalen

- Programmcode Zeilen
- Performance
- Geschätzte Dauer Neuentwicklung von Funktonalität mit Auswirkung auf den Applikationszustand
- Geschätzte Dauer Änderung von Funktonalität mit Auswirkung auf den Applikationszustand
- Geschätzte Dauer Entwicklung eines Unit Tests für den Applikationszustand
- Befragung nach dem Prinzip der geringsten Überraschung

sowie dem qualitativen Merkmal

- Der Evaluierung der zeitlichen Entwicklung der Verbreitung und Adaption des Redux Paradigmas
- „Thinking Aloud“ bei der Konfrontation von Dritten mit beiden Ansätzen

Nach der Evaluation werden die möglichen Vorteile des Redux Paradigmas beschrieben und die Masterarbeit in Kapitel „4 Schlussbetrachtung“ zusammengefasst. Der wissenschaftliche Anspruch der Arbeit stützt sich auf die umfassende und vergleichende Darstellung der Themen relevanten Literatur sowie auf die quantitative und qualitative Evaluation der betrachteten Zustandsmanagement Konzepte.

Die vorliegende Masterarbeit verzichtet bewusst auf die Einführung und Erläuterung der grundlegenden Web Technologien und die in diesem Kontext relevanten Konzepte. Auch auf eine Einführung in die Skriptsprache JavaScript wird in dieser Masterarbeit verzichtet. Die vorliegende Masterarbeit setzt Grundwissen bezüglich der JavaScript Syntax und Erfahrung mit der Sprache voraus, um den Beispiel Programmcode lesen und nachvollziehen zu können. Es wird Programmcode in ECMAScript 2015 (ES6) und TypeScript zu finden sein.

2 ZUSTANDSMANAGEMENT THEORIE IN JAVASCRIPT WEB APPLIKATIONEN

"Managing state in an application is critical, and is often done haphazardly."
Dan Abramov

Im zweiten Kapitel der vorliegenden Masterarbeit werden die theoretischen Grundlagen erarbeitet, um einen Vergleich und anschließend die möglichen Vorteile des Redux Paradigmas gegenüber bisherigen Zustandsmanagement in MV*-Architekturen durchführen und herausarbeiten zu können.

In den nächsten beiden Unterkapiteln werden jeweils die Begriffe JavaScript Applikation und Zustandsmanagement definiert und erklärt. Anschließend wird das Thema Zustandsmanagement in bisherigen JavaScript MV*-Architekturen dargestellt. Weiterhin wird die Entstehung und Funktionsweise des Redux Paradigmas betrachtet.

Das 2. Kapitel wird abgeschlossen mit einem theoretischen Vergleich zwischen dem Zustandsmanagement in bisherigen MV*-Architekturen und dem Redux Paradigma.

2.1 JavaScript Web Applikationen

In der vorliegenden Masterarbeit wird der Begriff „JavaScript Web Applikation“ synonym zum Begriff „Single Page Application“, kurz SPA, verwendet.

Um das anhaltende Wachstum in der Popularität von JavaScript Web Applikationen verstehen zu können, folgt der Vergleich zwischen traditioneller, d.h. serverseitiger Web Applikation und einer SPA.

Serverseitige Web Applikationen rendern Websites nachdem diese eine HTTP Anfrage vom Web Browser des Clients erhalten haben. Die gerenderten Webseiten werden als HTTP Antwort an den anfragenden Web Browser zurückgegeben und lösen ein Neu Laden der Website auf Seite des Clients aus. Während des Neuladens durch den Web Browser wird die komplette alte Website durch die neue Website ersetzt. Abbildung 1 zeigt den Lebenszyklus einer traditionellen Website.

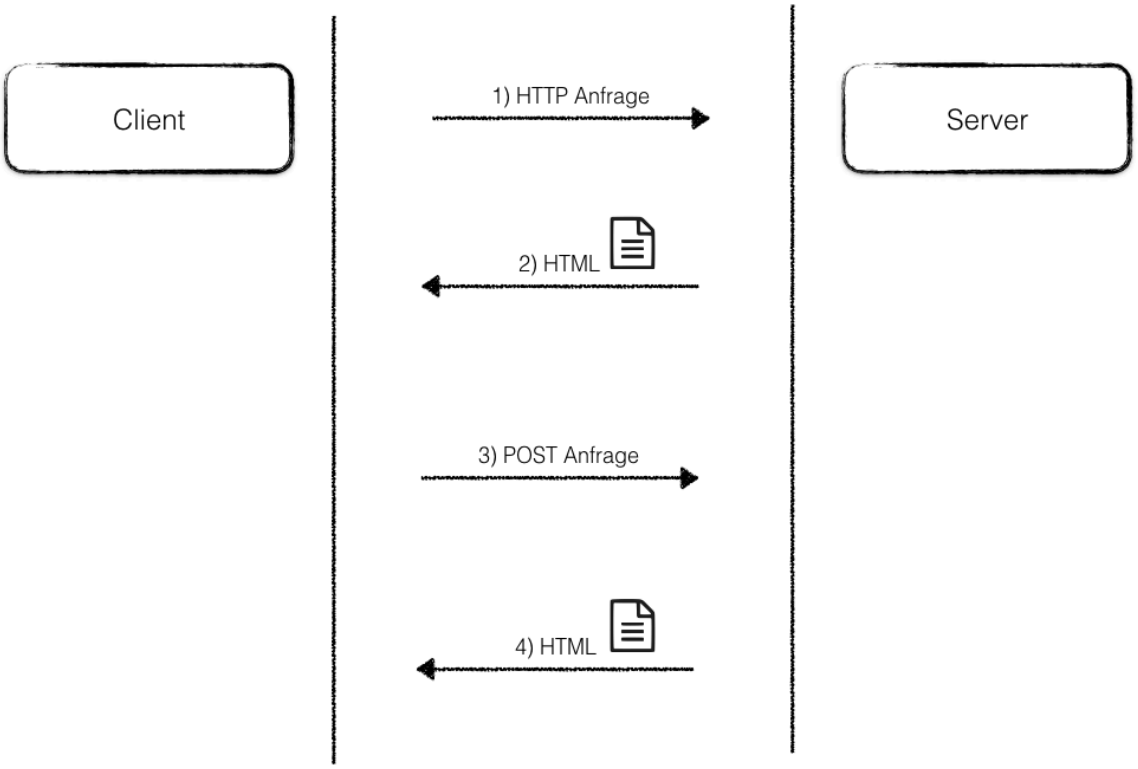


Abbildung 1 Traditioneller Website Lebenszyklus

Wie aus der Abbildung ersichtlich ist, wird eine Anfrage an den Server gesendet und die Serverantwort wird als HTML Dokument an den Client gesendet. Im oben dargestellten Beispiel stellt der Client eine zweite Anfrage an den Server, zum Beispiel schickt der Anwender ein Formular ab und sendet somit eine POST Anfrage an den Server. Das zurückgelieferte Ergebnis ist erneut ein HTML Dokument. Wenn der Client dieses HTML Dokument vollständig empfangen hat, wird die Website erneut komplett neu geladen.

Als Ergebnis des serverseitigen Ansatzes ist die Aktion bzw. das Gewicht auf der Seite des Clients gering und JavaScript wird hauptsächlich für kleine Änderungen in der Benutzeroberfläche oder Animationen verwendet.

Um einen kompletten Eindruck der sehr leichtgewichtigen Clientseite einer traditionellen Web Applikation zu gewinnen zeigt Abbildung 2 eine Gesamtübersicht über Traditionelle Web.

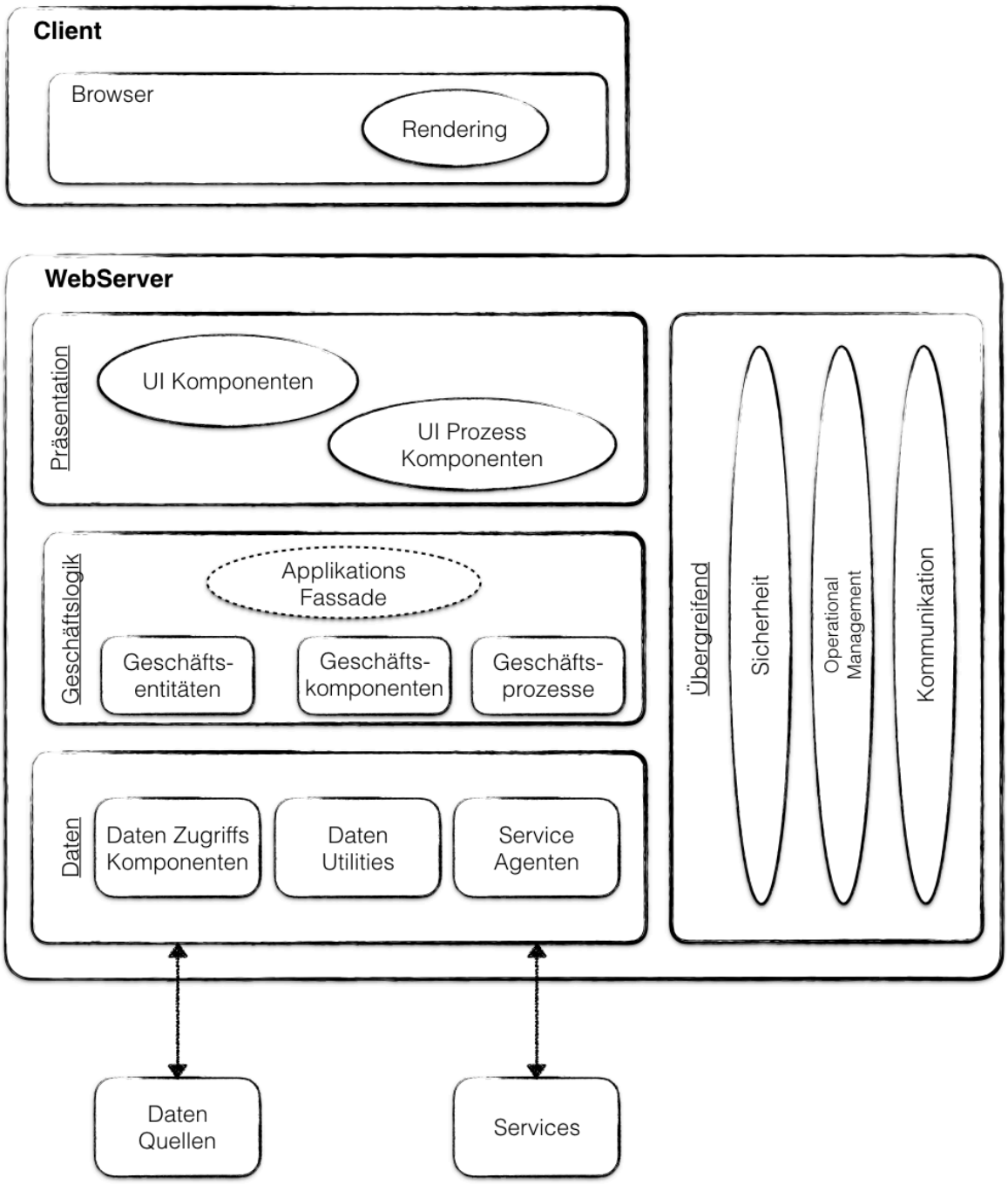


Abbildung 2 Traditionelle Web Applikationsarchitektur nach „Microsoft Application Architecture Guide, 2009“

Wie man auf der Abbildung erkennen kann, läuft ein Großteil der Applikation auf der Serverseite und der Client wird lediglich zum Darstellen der Website verwendet. In traditionellen Web Applikationen werden Anwender Interaktionen vom Server verarbeitet. Zum Beispiel ein Klick auf einen Button löst eine HTTP Anfrage aus. Diese Anfrage wird klassischerweise von einem Server verarbeitet. Anschließend lädt der Web Browser auf Anwenderseite die Website neu und das Ergebnis des Button Klicks wird dargestellt. Die serverseitige Verarbeitung von Ereignissen kann sehr lange dauern, so dass der Anwender warten muss und ein nicht immer optimales Anwendungserlebnis hat. Außerdem kann eine serverseitige Web Applikation ohne aktive

Internetverbindung aufgrund des traditionellen Website Lebenszyklus bestehend aus Anfrage und Antwort nicht benutzt werden.

„By design, the building blocks of the Web are stateless.“ (Cravens & Brady, S. 2)

Ein weiterer Aspekt von traditionellen Web Applikationen ist die Art und Weise des Daten- und Zustandsmanagements. Der Anwender- und Applikationszustand wird hauptsächlich auf der Serverseite mithilfe von Sessions oder Applikationszuständen gespeichert. Daraus ergibt sich ein weiterer Nachteil, nämlich, dass jedes Mal Daten oder Zustände vom Server angefragt werden müssen, wenn diese auf der Anwenderseite gebraucht werden. Jede dieser Anfragen führt zu einem Neu Laden der Website im Web Browser des Anwenders. Des Weiteren müssen lokale Zustands- oder Datenänderungen auf den Server synchronisiert werden, was wiederrum Zeit kostet, so dass der Anwender auf die Antwort des Servers warten muss. Das ist Grund dafür, dass die Reaktionsfähigkeit von traditionellen Web Applikationen nicht optimal ist (Fink & Flatow, Pro Single Page Application Development, 2014, S. 7-8).

Der Hauptunterschied zwischen einer traditionellen serverseitigen Web Applikation und einer modernen JavaScript Web Applikation ist der Website Lebenszyklus. Abbildung 3 zeigt den Lebenszyklus einer JavaScript Web Applikation.

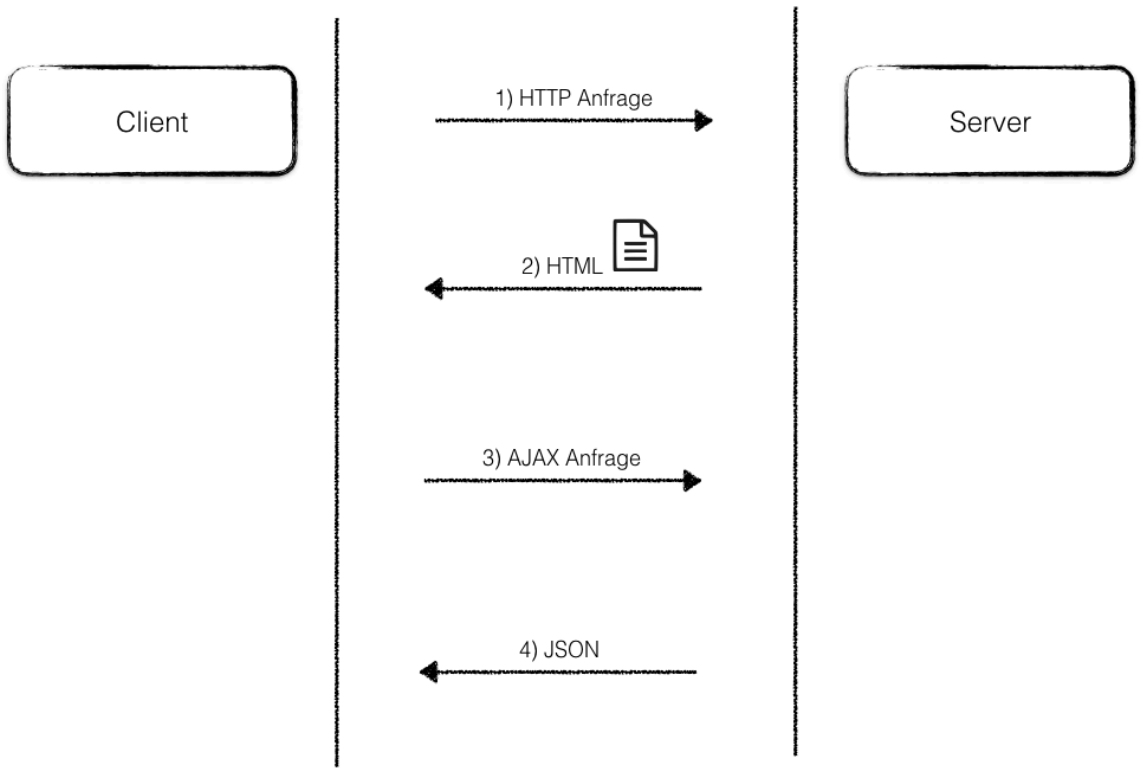


Abbildung 3 JavaScript Web Applikation Lebenszyklus

Vergleicht man den Lebenszyklus einer traditionellen, d.h. serverseitigen Web Applikation mit dem Lebenszyklus einer JavaScript Web Applikation wird schnell ersichtlich, dass der

Hauptunterschied darin liegt, was nach der erstmaligen HTTP Anfrage folgt. In einer JavaScript Web Applikation wird AJAX verwendet, um die auf die erste Anfrage folgenden Anfragen an den Server zu senden. Das Ergebnis wird vom Server oftmals als JSON oder vorgerenderte HTML Fragmente zurückgegeben. Sobald der Web Browser die Antwort erhalten hat, werden nur Teile des HTML Dokuments ausgetauscht, um die Änderungen im Web Browser des Anwenders anzuzeigen. Auch das Navigieren von einer zu einer anderen Seite wird komplett auf Anwenderseite im Browser von der JavaScript Web Applikation bearbeitet, in einer traditionellen Web Applikationsarchitektur würde auch hier wieder pro Wechsel ein kompletter „Anfrage – Antwort – Neu Laden“ Zyklus ablaufen, bevor das Ergebnis den Web Browser des Anwenders erreicht.

Ein weiterer Unterschied zwischen einer JavaScript Web Applikation und dem traditionellen Ansatz ist, wie der Zustand der Applikation gehandhabt wird. In einer JavaScript Web Applikation kann der Zustand im Arbeitsspeicher des Browsers vorgehalten werden, weil die Website nicht neugeladen wird. Das hat zur Folge, dass der Applikationszustand in Anwendungsfällen ohne aktive Internetverbindung oder beim Schließen des Web Browsers gespeichert werden kann, um den letzten Zustand bei wieder verfügbarer Internetverbindung dem Anwender ohne erneute Kommunikation mit dem Server zur Verfügung stellen zu können (Fink & Flatow, Pro Single Page Application Development, 2014, S. 12).

Es folgt ein tabellarischer von serverseitigen Web Applikationen, nativen Applikationen und JavaScript Web Applikationen.

Tabelle 1 Vergleich von traditionellen, nativen und Web Applikationen

Merkmal	**Serverseitige Web Applikation**	**Native Applikation**	**JavaScript Web Applikation**
betriebssystemübergreifend	Ja	Nein	Ja
Clientseitiger Zustand	Nein	Ja	Ja
keine Installation notwendig	Ja	Nein	Ja

Wie aus der Vergleichstabelle ersichtlich wird, sind JavaScript Web Applikationen der Zusammenschnitt aus serverseitigen Web- und nativen Applikationen. Sie vereinen die Vorteile beider Welten (Flatow & Fink, 2014, S. 11).

Entwickelt man eine JavaScript Web Applikation bekommt man die Portabilität und die betriebssystemübergreifende Kompatibilität einer Web Applikation in Kombination mit der Möglichkeit des Clientseitigen Zustandsmanagements und einem Antwortverhalten einer nativen Applikation.

Ein weiterer Vorteil von JavaScript Web Applikationen ist es, dass die Bereitstellung und die Aktualisierung im Hintergrund und ohne zu Tun des Anwenders passieren kann. Dahingegen

müssen native Applikationen im normal Fall immer noch vom Anwender regelmäßig selbst aktualisiert werden.

In den meisten JavaScript Web Applikationen ist die Geschäftslogik auf der Clientseite implementiert und der Server wird per API genutzt, um Authentifizierungs-, Validierungs- und Persistenz Verfahren (Datenbank) zur Anwendung bringen zu können. Aufgrund der Tatsache, dass ein Großteil der Applikationslogik direkt auf dem Client ausgeführt wird, können JavaScript Web Applikationen schnell reagieren und teilweise eine ähnliche Nutzungserfahrung wie native Applikationen für den Anwender ermöglichen. Oftmals muss nicht auf die Ausführung serverseitiger Prozesse gewartet werden und ein erneutes, komplettes Neu Laden der Website wird durch den Einsatz von clientseitigen JavaScript Web Applikationen überflüssig.

Auch wenn die JavaScript Web Applikation mit dem Server interagiert ist das Nutzererlebnis oft verbessert im Vergleich zu traditionellen Web Applikationen, weil die Interaktion zwischen Client und Server über AJAX ausgeführt wird, so dass die Benutzeroberfläche weiterhin Ergebnisse, wie zum Beispiel den Prozessfortschritt anzeigen kann und die Web Applikation dem Anwender nicht den Anschein vermittelt, der Prozessfortschritt würde ins Stocken geraten. Nach der Meinung von (Fink & Flatow, Pro Single Page Application Development, 2014, S. 13) machen die genannten Vorteile und Faktoren eine JavaScript Web Applikation zum perfekten Werkzeug zum Erstellen von zukunftsträchtigen Web Applikationen.

2.2 Zustandsmanagement in JavaScript Applikationen

Das nachfolgende Kapitel definiert und erläutert den Begriff des Zustandsmanagements in JavaScript Applikationen. Es wird eingegangen auf die vielfältigen Herausforderungen, die das Zustandsmanagement in JavaScript Web Applikationen mit sich bringt.

In den Computerwissenschaften ist der Begriff „Zustand“, englisch „State“, definiert als eine eindeutige Zusammenstellung an Informationen in einer Applikation (Mikowski & Powell, Single Page Web Applications, 2014, S. 85). Die Definition von (Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 1) lautet im Kontext von JavaScript Web Applikationen wie folgt: „Daten, die sich verändern und mindestens während der Anwendungszeit der Applikation bestehen bleiben müssen“.

Mit dem Begriff „Zustandsmanagement“ wird in der vorliegenden Arbeit der Zustand auf der Clientseite einer Web Applikation bezeichnet. „Zustand“ wird von (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 82) definiert als die aktuelle Sicht mit spezifischen Daten zu einem fixen Zeitpunkt. Osmani schreibt, dass das Thema Zustandsmanagement oft im Zusammenhang mit JavaScript Web Applikationen genannt wird, wo das herkömmliche Konzept von Zustand seiner Auffassung nach simuliert werden muss.

„All kinds of frameworks and architectures have state.“ (Parviainen, 2015)

(Alfoni, 2015) nähert sich der Definition des Begriffs Zustand über die Funktionsweise von Templates. Templates sind dazu da, etwas anzuzeigen. Das, was angezeigt werden soll, ist der nach Meinung von Alfoni der Zustand.

(Schläpfer, 2014) stellt fest, dass das Zustandsmanagement in fast jeder Programmierumgebung eine große Herausforderung darstellt. Er unterscheidet zwischen „Datenzustand“, welcher sich einfach im Model verwalten ließe. Darüber hinaus spricht der Autor von „Applikationszustand“, der definiert, welche Daten in welcher Art und Weise zu einem gewissen Zeitpunkt angezeigt werden.

Für (Abramov & Contributors, Redux Motivation, 2015) beinhaltet der Zustand zum Beispiel zwischengespeicherte Daten wie eine Serverantwort oder Daten die lokal auf der Clientseite erstellt wurden und noch nicht auf dem Server gespeichert sind. Weiterhin beschreiben die Autoren die zunehmende Komplexität des Zustands der Benutzeroberfläche von Web Applikationen, zum Beispiel beinhaltet eine JavaScript Web Applikation im Normalfall das Routing, kümmert sich um das Anzeigen von Ladeanimation oder stellt dem Anwender Links bereit, mithilfe dessen Inhalte Stück für Stück verarbeitet werden können. (Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 1) nennen als konkrete Beispiele für den Zustand einer JavaScript Web Applikation die folgenden Bereiche: Benutzer-Zustand, Status einer Bestellung und den Sendezustand einer Nachricht.

(Pucella, 1998, S. 48-57) beschreibt den Zustand einer JavaScript Web Applikation als äußert interaktiv, da Ereignisse innerhalb und außerhalb ständig Einfluss auf die Applikation und damit

deren Zustand nehmen. (Cooper & Krishnamurthi, 2006, S. 294-308) kommen zu dem Schluss, dass ein manuelles Verwalten von Zustandsänderungen und Datenabhängigkeiten komplex und damit Fehler anfällig ist, insbesondere sehen sie Zustandsänderungen zur falschen Zeit oder in der falschen Reihenfolge als sehr häufige Erscheinungen von schlecht gelöstem Zustandsmanagement.

In einer Analyse von nativen Adobe Applikationen wurde festgestellt, dass fast die Hälfte aller Fehler aufgrund von fehlerhafter Programmlogik, die die applikationsinternen Ereignisse verwaltet, entstanden sind (Maier, Rompf, & Odersky, 2010, S. 1). (Geary, 2016) von IBM geht sogar noch weiter und behauptet, dass nicht korrektes Zustandsmanagement die Quelle fast aller Softwarefehler ist.

(Bainomugisha, Carreton, Cutsem, Mostinckx, & De Meuter, 2013, S. 53) schlagen vor, dass insbesondere für die Ereignisverwaltungen und das Zustandsmanagement Strukturen und Sprachabstraktionen eingeführt werden, um die Belastung von Programmieren zu verringern. (Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 2) beschreiben die Tatsache, dass eine JavaScript Web Applikation, die nicht mehr wie eine herkömmliche HTML Website mit jeder HTTP Antwort komplett neu geladen wird, mit der neuen Herrausforderung konfrontiert ist, wie der Zustand der Website bzw. der Web Applikation verwaltet werden soll, wenn dieser nicht mehr mit jeder HTTP Antwort zurückgesetzt wird.

In den herkömmlichen Ansätzen gibt es Probleme in der Strukturierung von JavaScript Web Applikationen, insbesondere auch beim Datenaustausch zwischen den einzelnen Komponenten einer Web Applikation. (A M & Sonpatki, 2016, S. 219) schlagen vor, dieses Problem mithilfe eines einheitlichen Zustandsmanagements zu lösen.

Auch (Mulder, Full Stack Web Development with Backbone.js, 2014, S. 8) empfiehlt eine klare Trennung zwischen Zustandsmanagement und den weiteren Teilen einer JavaScript Web Applikation.

Zustandsmanagement wird in anderen Kontexten auch als Daten Architektur (Coury, Lerner, Murray, & Taborda, 2016, S. 153) bezeichnet.

(MacCaw, JavaScript Web Applications, 2011, S. 49) definieren die Möglichkeit, den Applikationszustand im DOM vorzuhalten, als veraltet.

Auch der Begriff des „Databindings“ zwischen Zustand und View gehören zum Thema des Zustandsmanagements. Databinding lässt sich definieren als Prozess, bei dem die Daten eines Models einem Element der Benutzeroberfläche zugeordnet werden (Scott, SPA Design and Architecture, 2016, S. 28).

“Two-Way Data-Binding” bezeichnet den Zustand, in dem Änderungen von Objekteigenschaften direkt Änderungen in der Benutzeroberfläche nach sich ziehen. Two-Way Databinding funktioniert auch genau die gegenläufige Richtung, d.h. Änderungen in der Benutzeroberfläche werden direkt in den Objekteigenschaften des Models wiedergespiegelt (Monteiro, 2014, S. 14).

Neben des One- und Two-Way Databindings gibt es noch das sog. „One Time Databinding“, bei dem das automatische Prüfen auf Änderungen entfällt.

Als Überblick über die verschiedenen Arten des Databindings folgt eine Tabelle.

Tabelle 2 Databinding Übersicht

Bindungstyp	Verhalten
One-Way	Die Daten fließen in eine Richtung, wenn sich die Quelldaten verändern werden auch die Zieldaten aktualisiert
Two-Way	Die Daten fließen in zwei Richtungen, Model und View werden zu jederzeit synchronisiert
One-Time	Die Daten werden nur einmal aktualisiert und fließen vom Model zum View

aus (Scott, SPA Design and Architecture, 2016, S. 37-39)

2.3 Bisheriges Zustandsmanagement in JavaScript MV*-Architekturen

In der Vergangenheit haben die Architekturmuster Model-View-Controller (MVC), Model-View-Presenter (MVP) und Model-View-ViewModel häufig bei der Strukturierung von nativen Applikationen und traditionellen Web Applikationen Anwendung gefunden. Erst in den letzten Jahren finden diese Muster auch in der JavaScript Welt immer mehr Anwendung (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 79). Auch (Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 6) sehen den enormen Erfolg des MVC Musters im Bereich von Desktop Applikationen und serverseitigen Architekturen in den letzten 30 Jahren und bezeichnen MVC in JavaScript Web Applikationsarchitekturen als relativ neues Konzept.

(Shapiro, Web Component Architecture & Development with AngularJS, 2015, S. 5) schreibt, dass ab dem Jahr 2010 immer mehr bekannte Architektur Muster in der JavaScript Welt implementiert wurden. Weiterhin beschreibt der Autor, dass damals auch die Komponenten Architektur eine weitere Herangehensweise war, die verschiedenen Teile einer Web Applikation zu separieren und eine notwendige Entkoppelung realisieren zu können (Shapiro, Web Component Architecture & Development with AngularJS, 2015, S. 6).

Nachfolgend wird die Herkunft des MVC Musters und dessen Anwendung in JavaScript Web MV*-Architekturen betrachtet. Diese Betrachtung ist die Grundlage, die verschiedenen Zustandsmanagement Ansätze in bisherigen MV*-Architekturen erläutern zu können.

2.3.1 MVC Einführung

Die Notwendigkeit von schnell verfügbaren, komplexen und reaktionsfreudigen Web Applikationen führt dazu, dass immer mehr Programmlogik auf der Clientseite vorhanden ist und damit auf dieser Seite verwaltet werden muss. Dass wiederrum führt dazu, dass der Programmcode in JavaScript Web Applikationen stark wächst und immer mehr an Komplexität gewinnt. Dieser Prozess hat MV*-Architekturen auch auf die Clientseite von Web Applikationen gebracht und die Popularität von Singe Page Applications enorm gesteigert. Der Bedarf nach einer echten Architektur auch für Web Applikationen ist auch beeinflusst durch die ständige Wartung und Erweiterung einer Web Applikation im Verlaufe des Web Applikation Lebenszyklus (Osmani A. , 2016). Auch (Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 4) halten fest, dass man jetzt, mit den langlebigen und zustandsbehafteten Websites Planung und Struktur für die Applikationen braucht. Cravens & Brady benennen die Trennung von verschiedenen Angelegenheiten innerhalb einer Web Applikation (Separation of Concerns) und die nötige Wiederverwendbarkeit von Modulen als zentrale Motivationen hinzu einer durchdachten Web Applikationsstruktur.

(Mulder, Full Stack Web Development with Backbone.js, 2014, S. 22) stellt fest, dass der Großteil an Software, die eine Benutzeroberfläche hat und damit mit menschlicher Interaktion in Kontakt kommt, in gewisser Form einer Variation des MVC Musters folgt.

Das MVC Muster wurde Ende der 1970iger Jahre von Trygve Reemskaug entworfen. Macht man sich bewusst, in welchem Jahr das MVC Muster entstanden ist wird klar, dass die Ursprünge schon weit zurückliegen und daher schon vergleichsweise sind. Dennoch findet sich das MVC Muster in vielen Dokumentationen und wird oftmals als grundlegende Architektur, auch noch heute, beschrieben. Damals wurde das MVC Muster entwickelt für Benutzeroberflächen von großen Arbeitscomputern. In den 1990iger Jahren wurde MVC vermehrt populär mit dem Bekanntwerden von Benutzeroberflächen in nativen Applikationen auf herkömmlichen Anwender PCs.

(Monteiro, 2014, S. 11) beschreibt MVC als Software Architektur Muster das während der 1980iger Jahre entstanden ist und dessen Hauptaugenmerk auf der Trennung von visueller Präsentation und der Anwender Interaktion liegt.

Die Motivation, immer wiederkehrenden Design Herausforderungen mit einer Art Musterlösung zu begegnen, folgt verschiedenen Implikationen. (Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 5) beschreiben, dass es nicht darum geht, jedes Projekt von Null auf neu zu entwickeln um immer wieder zu den gleichen Strukturen und Abstraktionen zu gelangen:

> *„[...] arriving at the conclusion that this doctor's office was going to need a large room with lots of seating where patients could wait until the doctor was ready to see them"*
> *(Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 5)*

sondern dass damals von Architekten Design Muster identifiziert wurden, wie in dem o.g. Zitat „das Wartezimmer" als abstraktes Konzept, welches immer dann implementiert wird, wenn es nützlich ist und gebraucht wird.

(Gamma, Helm, Johnson, & Vlissides, 1997, S. 14) definieren ein Design Muster im Allgemeinen durch die folgenden vier Eigenschaften. Ein Design Muster hat einen sprechenden Namen, eine passende Beschreibung des Problems auf das sich das Design Muster anwenden lässt, die jeweilige Problemlösungsskizze und eine Abhandlung der Konsequenzen, wenn das Muster implementiert wird.

(Osmani A. , Learning JavaScript Design Patterns, 2012, S. 88) ist der Meinung, dass die „Gang of Four" MVC nicht als Design Muster betrachtet, sondern eher als Zusammenstellung von Klassen um eine Benutzeroberfläche zu entwickeln. Er beschreibt, dass es aus Sicht der GoF eine Variation von drei klassischen Design Mustern ist. MVC bestünde aus dem Observer Muster, dem Strategy Muster und dem Composite Muster. Je nach Implementierung würde MVC außerdem das Factory und Decorator Muster nutzen.

Die Grundidee von MVC ist eine verbesserte Struktur einer Applikation durch die logische Trennung von verschiedenen Angelegenheitsbereichen. Das Model-View-Controller Muster beschreibt auf einer abstrakten Ebene die Trennung von Geschäftsdaten (Model), der Benutzeroberfläche (View) und der Steuerungslogik (Controller). Konkreter geht es nach (Elliot,

The Best Way to Learn to Code is to Code: Learn App Architecture by Building Apps, 2016) darum, die folgenden Teile in jeweils Einheit vorzuhalten:

- Darstellung (Layout, Style, DOM Anpassungen)
- Ereignismanagement (Anwenderaktionen und Informationen von externen Systemen aufnehmen und in Applikationsinterne Ereignisse umwandeln)
- Routing (Übersetzen von URLs zu Ereignissen)
- Geschäftslogik (Regeln wie Daten innerhalb der Applikation verändert werden können)
- Clientseitiges Zustandsmanagement (clientseitige in-memory Datenstrukturen)
- Langzeit Datenpersistenz (Datenbank Anbindung)

Darüber hinaus halten (Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 7) fest, dass der Mehrwert der Verwendung eines bekannten Musters auch in der Organisation anhand einer bekannten Konvention liegt. Wird einem bestehendem Projekt, welches anhand des MVC Muster strukturiert ist, ein neuer Entwickler zugewiesen, weiß er in bestem Fall direkt, wo bestimmter Programmcode abgelegt ist, weil die Struktur abstrakt genug und damit zwischen den Projekten gleich ist. Auch sehen die Autoren Vorteile bei der Fehleranalyse und während der Wartung einer bestehenden Applikation, wenn diese einer allgemein verständlichen Architektur bzw. Konvention folgt.

Des Weiteren schreibt (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 79-80), dass es wichtig ist zu verstehen, vor welchen Hintergründen das ursprüngliche MVC-Muster entwickelt wurde und welche Veränderungen es seit dem Ursprung, insbesondere in Hinblick auf JavaScript Web Applikationen, erfahren hat. Besonderen Augenmerk legt der Autor auf die folgenden Aspekte der Smalltalk-80iger MVC Architektur:

- Das Domänenobjekt ist bekannt als Model und hat kein Wissen über die Benutzeroberfläche inklusive ihrer Interaktionsmöglichkeiten.
- Die Präsentation von Elementen auf dem Bildschirm ist direkt gekoppelt an einen Controller, so der View und der Controller nicht unabhängig voneinander existieren können und damit keine wirkliche Trennung besteht.
- Die Aufgabe des Controllers ist die Handhabung von Benutzer Eingaben.
- Das Observer Muster wird verwendet, um den Views eines Elements zu aktualisieren, wenn sich das Model ändert.

Das oben erläuterte MVC Muster findet in angepasster Form Verwendung in modernen JavaScript Applikationen, um ohne viel Aufwand Struktur in den Programmcode einer Applikation zu bekommen. (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 80) betont die Notwendigkeit, unbedingt unstrukturierten und damit schlecht lesbaren und damit nur schwer wartbaren Programmcode, sog. „Spaghetti Code“ zu vermeiden. Stattdessen schlägt er vor, die Vorteile einer durchdachten Software Architektur, auch für JavaScript Web Applikationen, wirklich zu durchdringen.

(Mulder, Full Stack Web Development with Backbone.js, 2014, S. 23) hält fest, dass viele JavaScript Bibliotheken bzw. Frameworks einer Variation der MVC-Architektur folgen, jedoch ein großer Unterschied im Vergleich zur serverseitigen MVC Architektur besteht. Nach Mulder sei auf der Clientseite sei das MVC Muster viel näher am klassischem MVC Konzept, wo der jeweilige Benutzeroberflächen Zustand über die Zeit verwaltet wird.

Wie wir feststellen können findet sich die Idee des MVC-Muster auch in Architekturen von JavaScript Web Applikationen wieder, die drei wichtigsten Bestandteile dieses Musters können wie folgt definiert werden:

1. Das Model beschreibt eine Sammlung von domänenspezifischen Daten. Oftmals enthält eine Applikation mehr als ein Model. Als Beispiel können wir uns ein Nutzer Model vorstellen, welches die Attribute wie den Namen, ein Geburtsdatum und Zugriffsrechte enthalten könnte, die den Nutzer beschreiben. Außerdem kann das Model auch Geschäfts- und Datenvalidierungslogik enthalten.
2. Der View wird als Präsentationsschicht der Applikation bezeichnet. Der View zeigt die Model Daten und Funktionen der Applikation, die dem Anwender zur Verfügung stehen, um mit den bereitgestellten Daten zu interagieren. In JavaScript Web Applikationen entspricht der View der grafischen Benutzeroberfläche mit Texten, Bilder, Buttons und Formularelementen.
3. Der Controller beheimatet die Logik der Applikation. Von hier aus kann das Model erreicht und verändert werden. Außerdem sendet der Controller die Daten, welche er vom Model erhalten hat, an den View. Der Controller reagiert auch auf Ereignisse von Anwender Interaktionen mit der Applikation. So werden zum Beispiel durch Texteingabe Daten im Model geändert. Der Controller koordiniert klassischerweise auch welcher View dargestellt wird, wenn mehrere Views in der Applikation vorhanden sind, was normalerweise der Fall ist. Laut (Scott, SPA Design and Architecture, 2016) ist im klassischen MVC Vorschlag der Controller der Einstiegspunkt in die Applikation.

Da es sehr viele Interpretationen und Variationen der MVC Architektur gibt ist die Bezeichnung „MV*“ statt „MVC“ entstanden. Die Bezeichnung mit Asteriskus liest sich als „Model-view-„whatever-you-want-to-call-it““. Diese Bezeichnung ist heutzutage sehr weit verbreitet, weil in JavaScript Web Applikationen der Begriff des Controllers nicht mehr eindeutig zu definieren ist (Minar, 2012).

Letztendlich geht es beim MVC Muster nicht um Magie, sondern darum, den Programmcode so zu organisieren, dass klar abgegrenzte Programmteile um die Konzepte des Models, des Views und eines „Controllers“ entstehen. MVC ist ein Vorgehen, um die verschiedenen Aspekte der Applikation voneinander sauber zu trennen, Funktionalität zu kapseln und Daten in diskreten Objekten zu speichern. (Cravens & Brady, Building Web Apps with Ember.js, 2014, S. 7) sehen hier eindeutige Parallelen zur objektorientierten Programmierung.

(Osmani A. , 2016) fasst zusammen, dass durch die ständige Weiterentwicklung, eine kontinuierliche Anpassung und das dauerhafte Ausprobieren JavaScript Entwickler nach und nach die Gewinne des traditionellen MVC-Musters auf die Entwicklung von Frameworks für

JavaScript Web Applikationen übertragen konnten, so zum Beispiel auf die bekannten Frameworks „Backbone.js“ und „AngularJS“, die in nachfolgenden Abschnitten der vorliegenden Masterarbeit in Bezug auf das Thema Zustandsmanagement im Detail vorgestellt und erklärt werden.

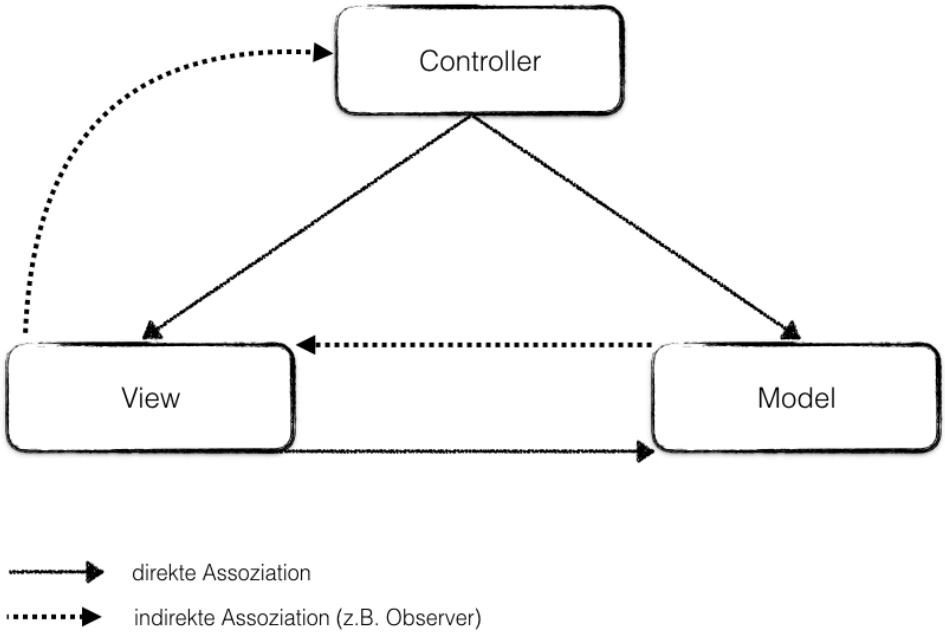


Abbildung 4 Model-View-Controller Muster nach (Elliot, Programming JavaScript Applications, 2013, S. 114)

2.3.2 MVC Variationen

MVP

Das Model-View-Presenter Muster, kurz MVP, wurde vom MVC Muster abgeleitet und ist 1996 von Taligent veröffentlicht wurden (Scott, SPA Design and Architecture, 2016, S. 26). Der Fokus des MVP Musters liegt auf der Verbesserung der Darstellungslogik. Die Hauptunterschiede zwischen MVP und MVC lauten wie folgt.

Der Presenter ist eine Komponente, welche die Benutzeroberflächen Logik für die Darstellung enthält. Aufrufe in der Darstellung werden innerhalb des MVP Musters an den Presenter weitergeleitet, so dass die Darstellung lediglich über ein „Interface“ bzw. in JavaScript über Ereignisse mit der Logik verbunden und nicht mehr direkt mit der Logik gekoppelt ist.

> *„ As we don’t have the interface construct in JavaScript, we’re using more a protocol than an explicit interface here. It’s technically still an API and it’s probably fair for us to refer to it as an interface from that perspective.“ (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 89)*

Im klassischen MVC Muster findet das soeben erklärte Delegieren an den Presenter nicht statt. (Scott, SPA Design and Architecture, 2016, S. 26) bezeichnet innerhalb des MVP Musters den View als Einstiegspunkt des Anwenders in die Applikation.

Die meist genutzte Implementierung des MVP Musters implementiert einen sog. Passiven View. Ein passiver View enthält sehr wenig bis keine Logik und hat kein Wissen über das Model. Der passive View wird vom Controller aktualisiert (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 94). Models im MVP Muster sind den Models im MVC Muster weitestgehend gleich.

Der Presenter fungiert als Vermittler zwischen dem View und dem Model. Model und View sind komplett voneinander getrennt. Es wird beschrieben, dass der Presenter in MVP den Controller aus dem MVC Muster ersetzt und die Verbindung zwischen Model und View sicherstellt. Der Presenter empfängt zum Beispiel Daten, verändert diese und bestimmt, wie die Daten im View angezeigt werden. Ggf. ist es auch die Aufgabe des Presenters, die vom Model ausgesendeten Ereignisse zu beobachten, um anschließend den View zu aktualisieren. In dieser Art von passiver Architektur gibt es keine unmittelbare Datenbindung zwischen Model und View. Der View muss sog. „Setter"-Funktionen bereitstellen, mit denen der Presenter Daten verändern kann.

Die Vorteile von MVP gegenüber MVC ist die saubere Trennung zwischen View und Model. Weiterhin entsteht durch die Entkopplung eine verbesserte Möglichkeit zum Mocking des Views in Unit Tests und insgesamt lässt sich festhalten, dass sich mit der Implementierung des MVP Musters die Testbarkeit einer JavaScript Web Applikation steigern lässt. Außerdem steigt mit der klaren Trennung auch die Wiederverwendbarkeit der Darstellungsschicht was für große Unternehmensanwendung relevant sein kann.

Der Nachteil von MVP ist, dass ohne eine unmittelbare Datenbindung diese fehlende Datenbindung bei Bedarf selbst erstellt werden muss.

(Osmani A. , Learning JavaScript Design Patterns, 2012, S. 89-90) fasst zusammen, dass man mit dem MVP Muster sehr ähnliche Erfahrungen machen wird wie bei der Verwendung des klassischen MVC Musters. Nach Meinung von Osmani ist der größte Unterschied für JavaScript Web Applikationen zwischen dem MVC- und MVP-Muster auf semantischer Ebene festzustellen. Der Autor sieht den entscheidenden Vorteil in der klaren Trennung von Model, View und Controller bzw. Presenter den die Implementierung beider Muster Variationen konzeptionell beschreibt.

Abbildung 5 Model View Presenter Konzept nach (Fowler, Presentation Model , 2004)

MVVM

Das „Model View ViewModel", kurz MVVM, Muster basiert auf MVC und MVP und verfolgt weiterhin das Ziel einer noch klareren Trennung der Entwicklung der Benutzeroberfläche (UI) und der Entwicklung von Geschäftslogik und dem Verhalten einer Applikation. Aus diesem Grund nutzen viele Implementierungen des MVVM Musters das Konzept der deklarativen Datenbindung,

um die Entwicklung der Benutzeroberfläche von anderen Schichten eindeutig trennen zu können (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 92).

Diese Trennung wiederum ermöglicht eine fast parallel stattfindende Entwicklung von Benutzeroberfläche und der eigentlichen Programmlogik. Entwickler, die zuständig sind für die Erstellung der Benutzeroberfläche schreiben die Datenbindung in das ViewModel direkt innerhalb des HTML Quellcodes und parallel kann das Model und das ViewModel von Programmlogik Entwicklern bearbeitet werden (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 92).

Wie auch in MVC und MVP besteht das Model im MVVM Muster konzeptionell ausschließlich aus den Geschäftsdaten. Das Model enthält innerhalb MVVM keine Logik und auch kein Applikationsverhalten.

Die Geschäftslogik bzw. dass Verhalten der Applikation wird in einer neuen Schicht, die mit dem Model, interagiert, gekapselt. Diese neue Schicht heißt in MVVM „ViewModel“.

Der View ist in MVVM wie bei MVC die einzige Schnittstelle mit der der Anwender direkt interagiert. Die Formatierung der Daten wird vom View durchgeführt. Der View ist die interaktive Benutzeroberfläche, die den Zustand des ViewModels visuell repräsentiert. Man nennt in MVVM die Darstellung aktiv gegenüber einem passiven View in MVP. Der aktive View enthält die Datenbindungen, Ereignisse und das Applikationsverhalten, welches wiederrum Wissen über das Model und das ViewModel braucht. Auch wenn das Verhalten mit Eigenschaften verbunden wird, ist der View nach wie vor verantwortlich für die Übergabe von Ereignissen an das ViewModel. Weiterhin ist es wichtig festzuhalten, dass der View nicht für das Zustandsmanagement verantwortlich ist, sondern lediglich den Zustand mit dem ViewModel synchronisiert.

Das ViewModel kann als spezieller Controller, der als Datenkonverter fungiert, bezeichnet werden. Das ViewModel transformiert Modeldaten in Darstellungsinformationen und reicht Ereignisse vom View zum Model weiter. Das ViewModel stellt teilweise auch Funktionen zur Veränderungen des Darstellungszustands bereit, aktualisiert das Model auf Basis von Ereignissen aus dem View und stößt Ereignisse im View an. Zusammenfassend stellt das ViewModel Modeldaten, die in der Darstellung gebraucht werden zur Verfügung und steht als Quelle für die Darstellung von Daten und Ereignissen bereit (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 93-97).

Abbildung 6 Model View ViewModel Muster nach (Weisse, 2016, S. 28)

Vergleich von MVC – MVP – MVVM

MVC ist der Vorgänger von MVP und MVVM. Der Hauptunterschied zwischen den drei Implementierungen ist die Abhängigkeit und der Grad der Kopplung zwischen den einzelnen Schichten innerhalb der jeweiligen Muster.

Zusammenfassend schreibt (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 103), dass sich im MVC Muster der View auf der Oberseite der Architektur, ganz unten das Model und zwischendrin der Controller befindet. Der View hat dabei im MVC Muster direkten Zugriff auf das Model. Das kann zu Sicherheits- und Performanz Problemen führen, deswegen wurde das MVC Muster weiterentwickelt.

Das MVP Muster zeichnet aus, dass der Controller aus dem MVC Muster durch den Presenter ersetzt wird. Der Presenter befindet nicht wie im MVC Muster unterhalb des Views sondern neben, d.h. auf einer Ebene mit dem View. Der Presenter hört auf Ereignisse des Views und vom Model. Dabei ist der Presenter der Vermittler zwischen View und Model. Es gibt keine unmittelbare Bindung zwischen View und ViewModel so wie es im MVVM Muster der Fall ist. Deswegen muss der View eine eigene Schnittstelle zur Interaktion mit dem Presenter bereitstellen.

Das MVVM Muster sieht vor, eine darstellungsspezifische Untermenge des Model zu erstellen, welche Zustandsdaten und Programmlogik enthalten kann. Dabei muss nicht wie im MVC Muster das komplette Model dem View zugänglich gemacht werden. Im Gegensatz zum Presenter im MVP Muster braucht ein ViewModel nicht genau einem View zugeordnet zu sein. Im View können bestimmte Informationen über das ViewModel verfügbar gemacht werden. Das ViewModel wiederrum bezieht diese Informationen aus dem Model. Durch die Abstraktion der Darstellung in MVVM ist weniger Programmcode erforderlich um die Verbindung zwischen Daten im View und im Model herzustellen. Das „MVVM“ Muster zeichnet weiterhin aus, dass es gegenüber dem MVP Muster weiter standardisiert und in Microsoft Silverlight seit Jahren in Benutzung ist (Monteiro, 2014, S. 12)

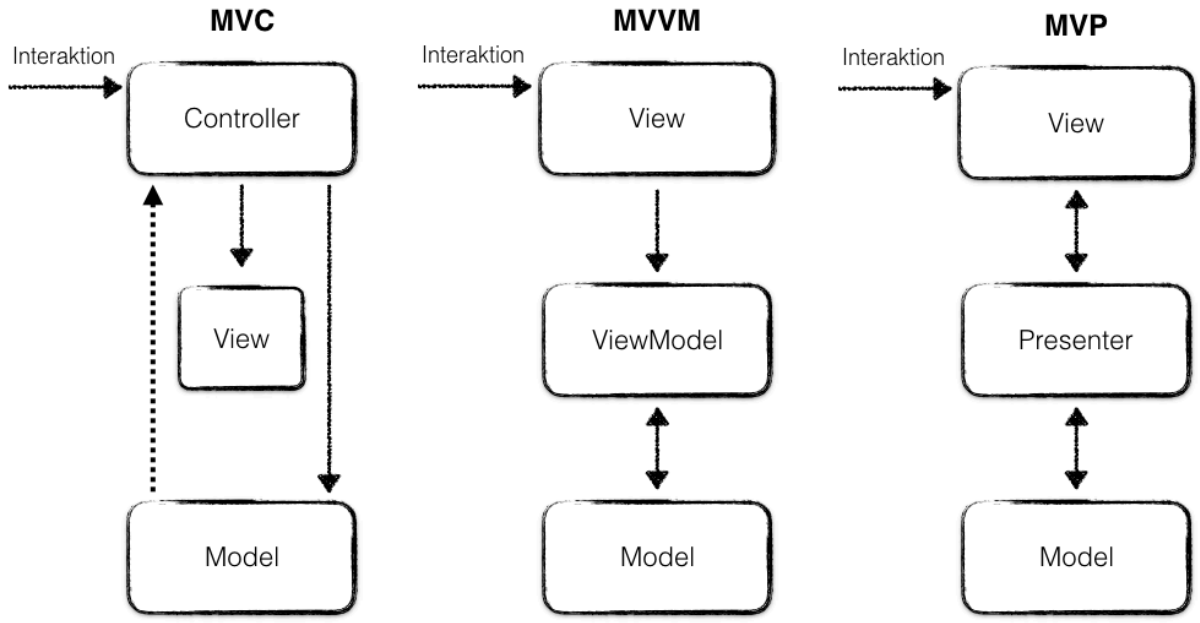


Abbildung 7 Übersicht MVC, MVVM und MVP nach (Monteiro, 2014, S. 15)

2.3.3 Exemplarische JavaScript Web Applikation MV*-Architektur Implementierungen

(Fink & Flatow, Pro Single Page Application Development, 2014, S. 52) beschreiben, dass die Nutzung eines MV*-Frameworks auf Clientseite der gleichen Motivation wie in herkömmlichen Applikationen folgt. Man teilt die vorliegende JavaScript Web Applikation in wiederverwendbare und testbare Teile auf. Die Verwendung eines Architektur- oder Entwurfsmuster schafft Struktur und kann später die Wartung und Weiterentwicklung der Applikation vereinfachen. Ausgehend von diesem Punkt folgt eine kurze Übersicht ausgewählter JavaScript Web Applikation Bibliotheken bzw. Frameworks.

Alle von den vorgestellten JavaScript Frameworks behaupten nicht, dass sie das MVC Muster in der klassischen Form implementieren und alle Autoren sind sich einig, dass die vorgestellten Muster wie MVC und MVP nicht exklusiv zu betrachten und damit auch nicht eins zu eins implementiert sind (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 90).

Auch (Scott, SPA Design and Architecture, 2016, S. 28) stellt fest, dass die endgültige Einordnung einer Bibliothek bzw. eines Frameworks in eines der vorgestellten Muster eine nutzlose Angelegenheit ist.

Aus dieser Erkenntnis, dass die Auseinandersetzung und Einordnung in Architekturen zwar wichtig, jedoch nicht ausschließlich entscheidend ist, entstand der Begriff „Model-View-Whatever“ (Minar, 2012).

Nach Meinung von Minar sollten Design Muster anpassbare Strategien und keineswegs unflexible Befehle sein. Weiterhin beschreibt er, dass Bestandteile der traditionellen Muster in jedem Fall erkennbar und gleichzeitig weniger strikt interpretiert und damit individuell in den jeweiligen Frameworks umgesetzt wurden sind. Auch (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 90-92) stellt fest, dass die meisten JavaScript Frameworks einer eigenen Interpretation der klassischen Muster folgen, sei es aus Zufall oder nach Plan.

Nachfolgend wird exemplarisch für die konkrete Implementierung von Zustandsmanagement in JavaScript MV*-Architekturen Backbone.js und AngularJS vorgestellt und für beide Vertreter der Themenkomplex des Zustandsmanagements theoretisch erläutert. Dies ist die Basis um die Problematik des bisherigen Zustandsmanagements erläutern zu können um dann das Redux Paradigma und dessen Vorteile erarbeiten zu können.

2.3.4 Backbone.js

Backbone.js ist eine JavaScript MV*-Bibliothek, die fünf grundlegende Komponenten beinhaltet. Backbone.js wurde ursprünglich entwickelt von Jeremy Ashkenas. Die Idee von Backbone.js ist es, für den eigenen Programmcode eine Strukturhilfe bereitzustellen und Backbone.js stellt nur eine minimale Sammlung an Funktionen zur Erstellung einer kompletten JavaScript Web Applikation bereit. Backbone.js ist eines der ersten JavaScript MV*-Bibliothek und hat viele jüngere Frameworks entscheidend beeinflusst (Fink & Flatow, Pro Single Page Application Development, 2014, S. 53). (Shapiro, Web Component Architecture & Development with

AngularJS, 2015, S. 6) bezeichnet Backbone.js als das Framework mit den schwächsten Konventionen. Backbone.js wird oftmals zusammen mit jQuery genutzt und stellt das Routing, REST Komponenten, einen View Logikcontainer und gewisse Teile für das Zustandsmanagement bereit. Der Entwickler muss sich bei Backbone.js selbst um die Datenbindung zwischen Model und View kümmern. Aufgrund des leichten Fußabdrucks von Backbone ist (Shapiro, Web Component Architecture & Development with AngularJS, 2015, S. 8) der Meinung, das Backbone.js kein Framework, sondern eher eine Basis für JavaScript Web Applikationen ist und somit die Bezeichnung Bibliothek anstatt Framework treffender ist.

Derick Bailey, der Entwickler von Marionette.js, einer Sammlung nützlicher Backbone Strukturen und Komponenten, hat Backbone.js wie folgt in das Thema MV*-Architekturen eingeordnet:

> *„ Backbone, in my opinion, is not MVC. It's also not MVP, nor is it MVVM (like Knockout.js) or any other specific, well-known name. It takes bits and pieces from different flavors of the MV* family and it creates a very flexible library of tools that we can use to create amazing websites. So, I say we toss MVC/MVP/MVVM out the window and just call it part of the MV* family." (Bailey, 2011)*

Zustandsmanagement in Backbone.js:

Der Zustand einer JavaScript Web Applikation wird innerhalb von Backbone.js typischerweise in den Backbone.Models festgehalten. Deswegen ist in Backbone.js findet das Zustandsmanagement in Backbone.js hauptsächlich unter der Verwendung der zur Verfügung gestellten Objekten statt.

Backbone.js ist Programmcode getrieben, d.h. Models und Views werden programmatisch erstellt, indem Backbone.js Objekte erweitert werden (Scott, SPA Design and Architecture, 2016, S. 30). Das Erweitern der Backbone Objekte bringt eine Reihe von weiterer Funktionalität auf einem Backbone Model mit sich, die sog. Modellogik. Zum Beispiel die „validate" Eigenschaft, anhand dieser Eigenschaft können Werte im Backbone Model anhand von Regeln validiert werden. Deswegen kann der Zustand der Applikation in Backbone.js nicht als simples JavaScript Objekt (POJO) vorgehalten werden. Backbone.js implementiert „Setter" und „Getter" Methoden, um die Attribute, in Form von Key –Value Paaren, eines Models zu setzen und auszulesen.

Backbone.js gruppiert gleiche Models innerhalb von sog. Collections. Das hat den Grund, dass es selten ein Model mit nur einer Entität gibt. Backbone.js Collections helfen eine Model Sammlung einfach zu verwalten und stellt hilfreiche Funktionen zur Arbeit mit einer Vielzahl von Models bereit, unter anderem auch für das Änderungsmanagement, d.h. das Aktualisieren und rendern innerhalb von Collections (Fink & Flatow, Pro Single Page Application Development, 2014, S. 80).

Backbone implementiert einen eher klassischen MVC Ansatz was die Datenbindung zwischen Model und View betrifft. Die Logik um einen View in Backbone.js aktuell zu halten ist folgendermaßen implementiert. Wenn sich der Zustand eines Models ändert, werden typischerweise die Beobachter des Models, z.B. die Backbone Views, über die Änderung im Model informiert, so dass die Beobachter entsprechend reagieren können. Im Falle des Views

würde z.B. ein erneutes Rendering erfolgen um die aktualisierten Daten im View darzustellen (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 81).

Two Way Data-Binding wird von Backbone.js nicht unterstützt (Mulder, Full Stack Web Development with Backbone.js, 2014, S. 21).

Backbone.js hat im herkömmlichen Sinne keinen Controller, diese Funktionalität teilt sich in Backbone.js meist auf den View und den Router, daher wird Backbone.js auch MV*-Bibliothek bezeichnet (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 85).

2.3.5 Angular (Version 2)

AngularJS bezeichnet sich selbst als „superheroic JavaScript MVW Framework“ und wurde so entwickelt, dass möglichst ein Großteil aller gebräuchlichen Funktionalitäten, die gebraucht werden, um eine komplette JavaScript Web Applikation zu entwickeln, im AngularJS Framework enthalten sind. In AngularJS sind Teile der Web Applikation per Programmcode zu beschreiben, andere Teile werden deklarativ über spezifische HTML Attribute deklariert (Scott, SPA Design and Architecture, 2016, S. 30).

AngularJS wird entwickelt und gewartet von Google. Dabei ist wichtig herauszustellen, dass Google im dritten Quartal 2016 eine neue Version von AngularJS veröffentlich hat.

Angular kann in diesem Sinne zwar als Nachfolger von AngularJS1 bezeichnet werden, jedoch haben sich viele grundlegende Konzepte geändert, so dass immer die betrachtete AngularJS Version mitangegeben werden sollte.

„[…] Angular 2 […] is a completely new framework.“ (You, Beziuk , & Fritz, 2016)

Angular fußt auf eine komponentenbasierte Architektur und vertritt starke Konventionen innerhalb des eigenen Frameworks. Deswegen kann die Lernkurve am Anfang etwas steiler sein, um das MV*-Framework Angular nutzen zu können, da ohne das Einhalten der vorgegebenen Konventionen nicht viel funktioniert. Angular stellt sehr viele Funktionalitäten für moderne JavaScript Web Applikationen bereit und wird als „All in one“ Lösung zum Entwickeln einer SPA betrachtet (Fink & Flatow, Pro Single Page Application Development, 2014, S. 53).

Zustandsmanagement in Angular

AngularJS 1 wurde bekannt für das „Two-Way Data-Binding“ und das implementierte MVVM Muster. Der Datenfluss der Komponenten in AngularJS 1 ist zum Einstieg sehr einfach, da das ViewModel die Daten beinhaltet und als $scope Variable bereitstellt, so dass zum Beispiel ein Formularfeld das entsprechende Model live manipuliert und somit zu jeder Zeit der Zugriff auf einen aktuellen Datenstand realisiert ist (Coury, Lerner, Murray, & Taborda, 2016, S. 106). Das für AngularJS 1 typische Zustandsmanagement wurde durch die Einführung des Scopes realisiert, indem die ganze Applikation die gleichen Daten teilt und Änderungen direkt den relevanten Teilen der AngularJS Applikation mitgeteilt werden.

Was in AngularJS1 noch als kreisförmiger Prozess, dem sog. „Digest Cycle“ implementiert war findet sich in Angular als „Change Detection Graph“ in Form eines Baums wieder. (Savkin, 2016) beschreibt die daraus resultierenden Vorteile als gesteigerten Performanz und vor allem einer verbesserten Nachvollziehbarkeit des Datenflusses und damit eine bessere Möglichkeit zur Fehleranalyse innerhalb der Angular Applikation.

In Angular gibt es kein Scope Objekt mehr, stattdessen kann der Funktionskontext eines Views als Ausführungskontext für die eigenen Templates verwendet werden, da eine Komponente der einzige direkte Besitzer eines jeden Views ist (Sotelo, 2014).

(Coury, Lerner, Murray, & Taborda, 2016, S. 75) stellen fest, dass es in Angular nicht eine einzige Lösung für das Zustandsmanagement vorgesehen ist, sondern dass verschiedene Lösungsmöglichkeiten innerhalb von Angular existieren. Angular ist bezogen auf das Thema Zustandsmanagement ziemlich flexibel und implementiert verschiedene Zustandsmanagement Ansätze. Daraus folgt, dass selbst eine Entscheidung getroffen werden muss, wie das Zustandsmanagement in einer Angular Web Applikation gelöst werden soll, diese Entscheidung liegt z.B. beim Entwickler oder Architekten der betrachteten Web Applikation.

Wie auch schon in AngularJS 1 gibt es in Angular eine sehr einfache Variante, Daten zwischen mehreren Komponenten auszutauschen. Die Möglichkeit besteht darin, sog. Services als Klassen in Komponenten zu injizieren. Die bereitzustellenden befinden sich innerhalb der injizierten Klasse. Es gibt globale und lokale Services. Bei einem globalen Service existiert eine Instanz für alle Komponenten innerhalb der Applikation, bei einem lokalen Service entsteht eine neue Instanz der Serviceklasse für jede Komponente (Böhm, 2015).

Der Service (Model) ist dann auch dafür verantwortlich, alle Beobachter über Änderungen im Model zu benachrichtigen. (Coury, Lerner, Murray, & Taborda, 2016, S. 106) sehen die Idee des Service als Model zwar als simpel in der Verwendung an, jedoch nicht zwingend praktikabel für die Praxis, da viele Detailfragen des Zustandsmanagements mithilfe des „Service-Models“ nicht beantwortet werden.

(Coury, Lerner, Murray, & Taborda, 2016, S. 106) konstatieren, dass der empfohlene Weg für das Zustandsmanagement in Angular den Ansatz des „One-Way Data-Binding“ verfolgt. One-Way Data-Binding beschreibt, dass die Daten innerhalb der Applikation nur in eine Richtung fließen, und zwar von oben nach unten.

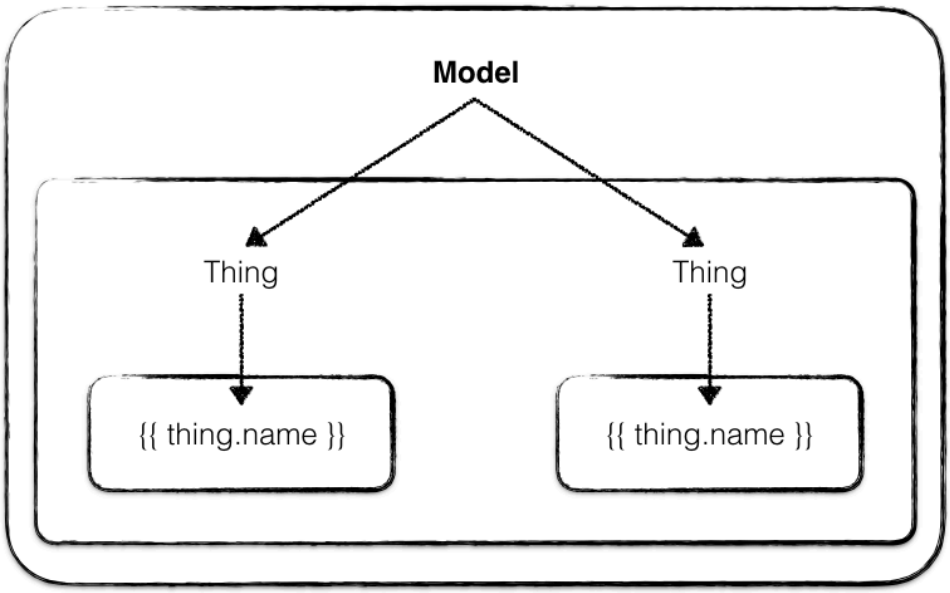


Abbildung 8 AngularTop Down Data-Binding Konzept nach (Savkin, 2016)

Wenn die Daten bzw. das Model verändert werden muss, werden beim One-Way Data-Binding Ereignisse ausgelöst, die die Änderungen an der oberen Spitze vollziehen und diese dann wieder runter zu den einzelnen Komponenten weitergeben.

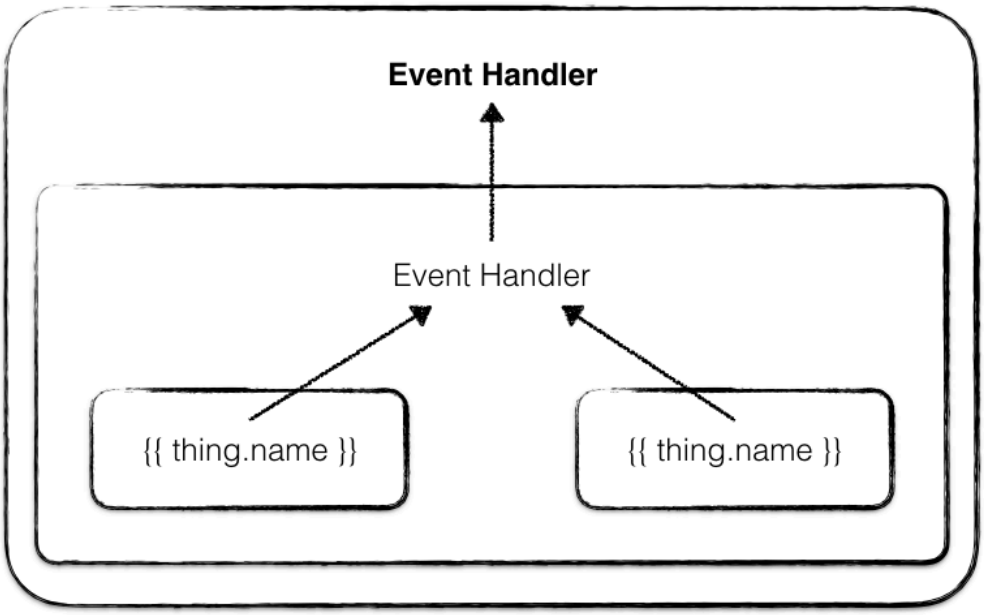


Abbildung 9 Angular Event Model Updates nach (Savkin, 2016)

Die Autoren halten auch fest, dass One-Way Data-Binding auf den ersten Blick kompliziert wirkt und nach unnötigem Mehraufwand aussieht. Die Autoren sind sie sich einig, dass das Konzept des One-WayData-Bindings in Bezug auf die Herausforderung der Model-View Synchronisierung und Nachvollziehbarkeit des Datenflusses in einer Web Applikation enorm hilfreich sein kann.

Es gibt zwei bekannte Ansätze, die in Angular implementiert werden können, anhand dessen das Zustandsmanagement strukturiert und vollzogen werden kann:

- Eine Observable basierte Architektur wie RxJS oder
- eine Flux basierte Architektur.

Außerdem steht in Angular nach wie vor die Direktive „ngModel“ bereit, jedoch ist diese speziell für den Einsatz mit Formularen gedacht, indem die Direkte ein Model an ein Formular bindet. „ngModel“ wird dadurch ausgezeichnet, dass es vordergründig das Verhalten des bekannten Two-Way Data-Bindings implementiert. Das hat den Hintergrund, dass es für Formulare sehr praktisch ist, ein Two-Way Data-Binding zu haben (Coury, Lerner, Murray, & Taborda, 2016, S. 149).

2.4 Die Problematik des Zustandsmanagements in herkömmlichen JavaScript MV*-Architekturen

Es lässt sich sagen, dass MV*-artige Architekturen sehr beliebt und weit verbreitet sind, um JavaScript Web Applikationen zu strukturieren und zu entwickeln. Gleichzeitig haben die bisher vorgestellten klassischen Implementierungen der MVC-Architektur wie MVP und MVVM eine nicht optimal gelöste Herausforderung.

(Paul & Nalwaya, React Native for iOS Development, 2016, S. 75-77) beschreiben, dass je komplexer der Programmcode innerhalb einer MV*-Architektur wird, desto komplexer wird auch das Zustandsmanagement innerhalb dieser Applikation. Bei einfachen Web Applikationen funktionieren laut den Autoren die bisherigen Implementierungen der MVC Architektur gut.

Wenn eine Web Applikation wächst und immer mehr Funktionalität dazukommt, sollte struktureller Platz vorhanden sein für weitere Models and Views.

Abbildung 10 zeigt was innerhalb einer MV*-Architektur passieren kann, wenn die Anzahl an Models und Views immer weiterwächst.

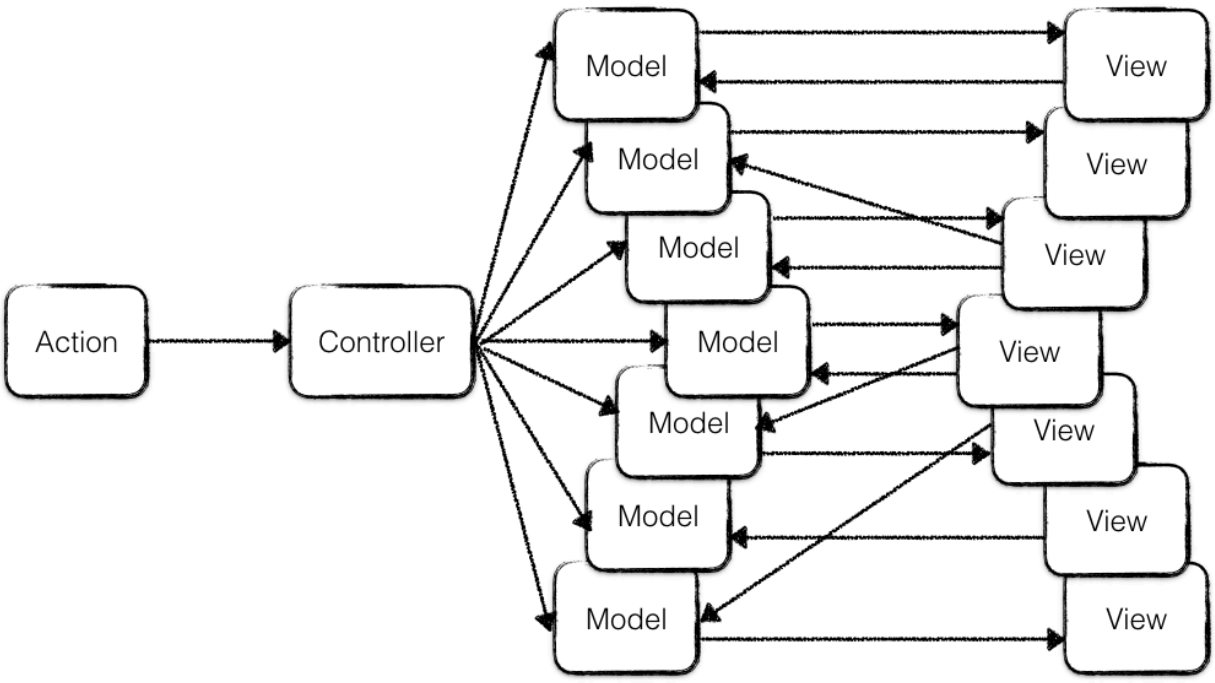


Abbildung 10 Das klassische „MVC-Problem“ in skalierten Web Applikationen

Gibt es innerhalb einer anhand der herkömmlichen MV*-Architektur strukturierten Web Applikation viele Models und viele Views die miteinander interagieren, kann es dazu kommen, dass ein Model eine Änderung in einem View anstößt und dann der View wiederrum ein anderes Model aktualisiert was in einem unendlichen Kreis enden kann. Das wirkliche Problem in solch einer Situation ist jedoch die Nachvollziehbarkeit des Zustands, es ist sehr schwer herauszufinden, wo welche Änderung ihren Ursprung hat und zu Stande kommt. Das kann dazu

führen, dass eine Applikation instabil und fragil wird (Paul & Nalwaya, React Native for iOS Development, 2016, S. 75-77)

Ähnliche Probleme haben sich in Praxis bei Implementierungen des Two-Way Data Bindings gezeigt. Auch hier hat der Entwickler keine direkte Nachvollziehbarkeit oder Transparenz gegenüber der Model-View Aktualisierungskette.

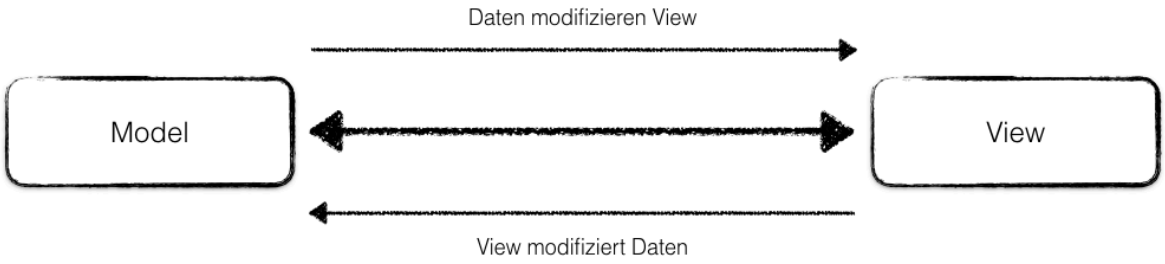


Abbildung 11 Two-Way Data Binding Schema

Das Problem in herkömmlichen MV*-Architekturen kann auftreten, wenn mehrere verschiedene Views Daten aus einem einzigen Model enthalten. Werden immer mehr Models und Views einer JavaScript Web Applikation hinzugefügt, können die Änderungsketten in unübersichtlichen „Spaghetti“ Verbindungen enden, was zu Abhängigkeiten und Auswirkungen führen kann, die in einer Endlos-Aktualisierungsschleife münden.

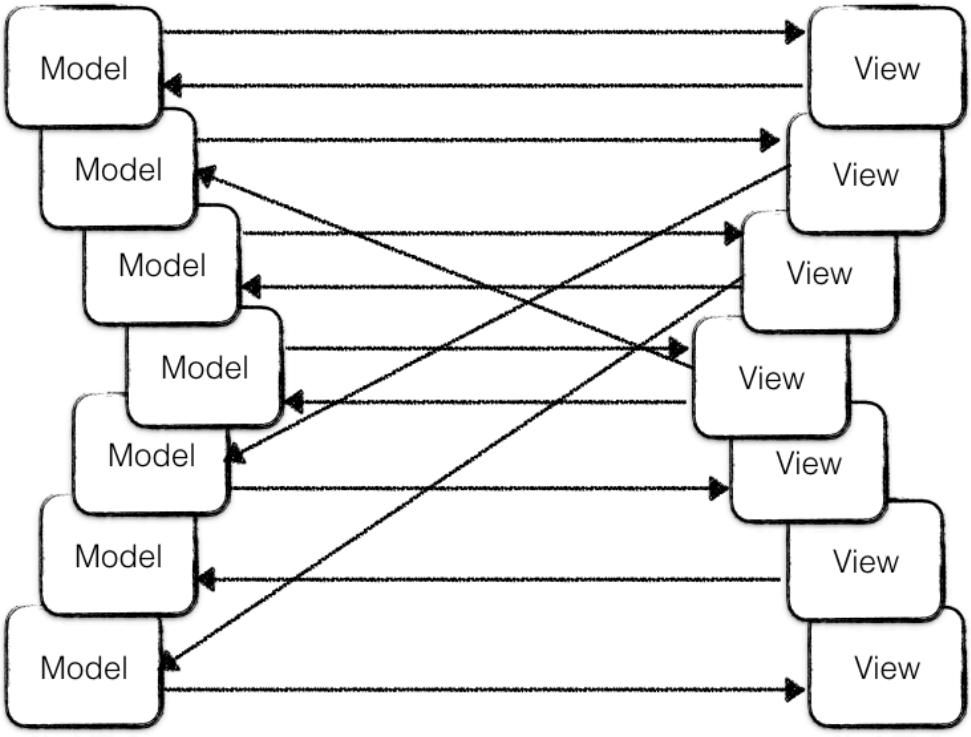


Abbildung 12 Beispiel unübersichtliche „Spaghetti“ Beziehungen zwischen vielen Models and Views

(Paul & Nalwaya, React Native for iOS Development, 2016, S. 4-6) halten auch fest, dass eine solche Architektur dazu führen kann, dass es für neue Entwickler unnötig lange dauern kann, bis sie verstanden haben, welche Models und Views welche Abhängigkeiten haben. Das heißt die möglichen Folgen einer Programmcode Änderungen lassen sich nur schwer voraussehen, was im Endeffekt zu erhöhten Entwicklungskosten führen kann.

Auch (Coury, Lerner, Murray, & Taborda, 2016, S. 149) stellen die möglichen sich kaskadierenden Effekte einer Two-Way Data-Binding Architektur fest und sagen, dass in einer solchen Architektur ab einer gewissen Projektgröße der Datenfluss nur schwer nachvollzogen werden kann. Außerdem fügen die Autoren hinzu, dass in einer Two-Way Data-Binding Architektur der implementierte Datenstrukturbaum oftmals stark an den DOM-Baum angelehnt wird. Beide Strukturen sollten eigentlich nicht voneinander abhängen (Coury, Lerner, Murray, & Taborda, 2016, S. 106).

2.5 Zustandsmanagement anhand des Redux Paradigma

Das Redux Paradigma basiert auf der Flux Architektur, daher wird erst die Flux Architektur erklärt und nachfolgend das aus dem Flux Konzept resultierende Redux Paradigma.

Flux und Redux definieren ein Konzept für ein konkretes Zustands- und Datenfluss-Modells innerhalb einer JavaScript Applikation wohingegen im klassischen MVC Muster viel weniger strikt definiert wurde, wie der Zustand und die Daten innerhalb einer Applikation vorgehalten und verändert werden.

2.5.1 Flux

Flux kann als konzeptioneller Leitfanden bezeichnet werden, der eine Lösung für ein allgemeine Herausforderungen mit JavaScript Web Applikation vorschlägt. Dabei ist Flux keine konkrete Implementierung. Vergleichbar mit den verschiedenen MV*-Implementierungen gibt es eine Reihe an Flux Implementierungen. (Elliot, 10 Tips for Better Redux Architecture, 2016) beschreibt Flux als transaktionales Zustandsmodell, welches der Nachfolger des herkömmlichen Datenflusses in MV*-Architekturen ist.

Das Flux Konzept beschreibt einen unidirektionalen Datenfluss innerhalb einer Applikation und wie die Daten und der Zustand vorgehalten und verändert werden können. Flux versucht anhand der formulierten Regeln die Nachvollziehbarkeit des Applikationszustands zu erhöhen und eine erhöhte Erweiterbarkeit zu erreichen (Fedosejev, 2015, S. 140).

Eine JavaScript Web Applikation macht ohne Benutzeroberfläche keinen Sinn, deswegen findet sich die View Schicht oft in etwaigen Flux Erklärungen wieder. Der View ist aber nicht Bestandteil der bekannten Flux Implementierungen, sondern muss durch eine weitere Bibliothek hinzugefügt werden, z.B. mit React.

Flux besteht aus den folgenden drei Hauptelementen. Dem Dispatcher, den Stores und den Actions. Es folgt eine detaillierte Beschreibung der jeweiligen Flux Bestandteile nach (Gackenheimer, 2015, S. 87-92).

Der Dispatcher

Der Dispatcher ist der Dreh- und Angelpunkt des Datenflusses innerhalb des Flux Konzepts. Der Dispatcher ist ein Singleton und kontrolliert, welche Daten in die jeweiligen Stores gelangen. In Flux erzeugen die Stores Callback-Funktionen, die im Dispatcher registriert werden. Wenn ein Action Creator eine neue Action an einen Dispatcher sendet stellt der Dispatcher sicher, dass die Action von ihm an alle registrierten Stores weitergeben wird, indem jede registrierte Callback-Funktion der Stores mit der Action als Parameter aufgerufen wird.

Besonders für größere Web Applikationen ist der Dispatcher von zentraler Bedeutung, da die Reihenfolge der Ausführung der einzelnen Callback-Funktionen kontrolliert wird. Außerdem kann z.B. ein Store auf die Fertigstellung der Aktualisierung innerhalb eines anderen Stores warten, bevor er sich selbst aktualisiert.

Die Stores

Ein Store in Flux enthält Teile des kompletten Applikationszustands, d.h. Daten und Benutzeroberflächen Zustand. Man kann sagen, dass die Stores in Flux die Besitzer des Applikationszustands sind. Außerdem sendet ein Store ein Ereignis, wenn sich Daten im Store geändert haben. Die in der Web Applikation vorhandenen Views können die Änderungsereignisse eines Stores abonnieren um ggf. die eigene Anzeige zu aktualisieren, wenn sich relevante Daten geändert haben. Alle Actions werden an alle Stores weitergeleitet und der Store entscheidet selbst, ob die jeweilige Action für ihn relevant ist und ggf. eine Änderung des Zustands innerhalb des eigenen Stores ausgeführt wird.

Eine weitere wichtige Eigenschaft der Stores ist, dass sie geschlossene Systeme sind. Die Stores stellen lediglich öffentliche Getter-Funktionen zum Lesezugriff auf die Daten innerhalb des Stores bereit. Niemand kann Daten in die Stores einfügen oder Daten in irgendeiner Weise verändern, außer ein Store selbst.

Es wurde beschrieben, dass niemand die Daten innerhalb des Stores ändern kann, jedoch muss es einen Weg geben, den Zustand der Applikation generell zu ändern. Das funktioniert innerhalb von Flux mithilfe sog. Actions.

Actions

In Flux sind Actions eine Form von Zustandsinformationen, die an die Stores gesendet werden. Actions sind einfach beschrieben „Dinge, die in der Applikation passieren“. Actions werden meistens von Benutzerinteraktionen, z.B. durch einen Buttonklick, erstellt. Die Daten die in einer Action enthalten sind können beispielsweise auch aus einem API Aufruf oder dem Ablauf eines Timers kommen. Jede Action enthält einen Typ und einen optionalen Payload. Eine Funktion, die die ein Action Objekt zurückgibt und damit eine Action erstellt, wird „Action Creator“ genannt.

In der einfachsten Form lässt sich das Flux Konzept wie folgt darstellen:

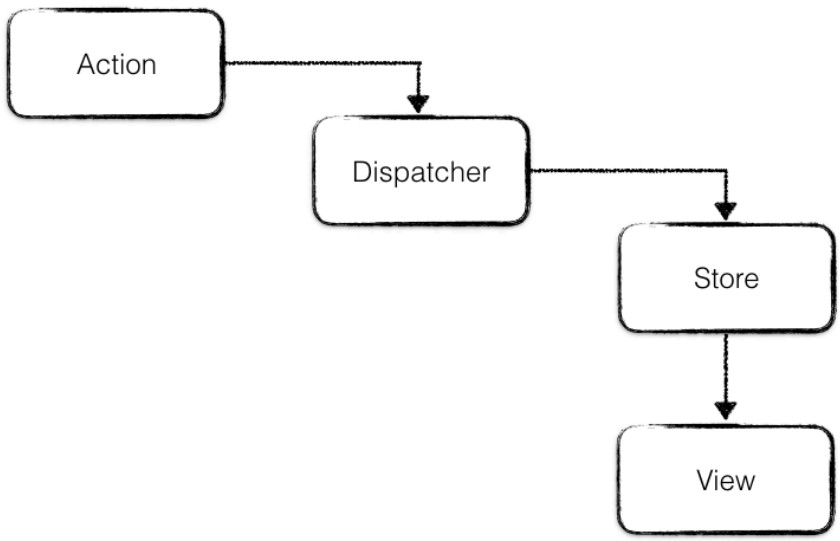


Abbildung 13 Das Flux Konzept

Der typische Flux Datenfluss wird gestartet mit einer Action. Diese Action gelangt zum Dispatcher. In Flux stellt der Dispatcher sicher, dass es innerhalb des Datenflusses der Web Applikation zu keinen sich kaskadierenden Effekten, wie sie in größeren MV*-artigen Konstrukten entstehen können, auftreten. Der Dispatcher ist auch dafür verantwortlich, dass die Actions in der Reihenfolge ausgeführt werden, in der sie an den Dispatcher gelangen, so dass etwaige „Race Conditions“ vermieden werden. Wird dann die Action über die Callback-Funktion an die Stores übergeben, kann zu dieser Zeit keine weitere Action in einen aktiven Store übergeben werden. Anschließend benachrichtigt der Store alle verknüpften Views, dass sich Daten geändert haben und der View kann sich aktualisieren um die veränderten Daten anzuzeigen.

Der View selbst trägt in der Weise zum Datenfluss in Flux bei, dass von ihm aus erneut Actions gesendet werden, welche dann erneut über den Dispatcher in einen Store gelangen, wie die folgende Abbildung zeigt.

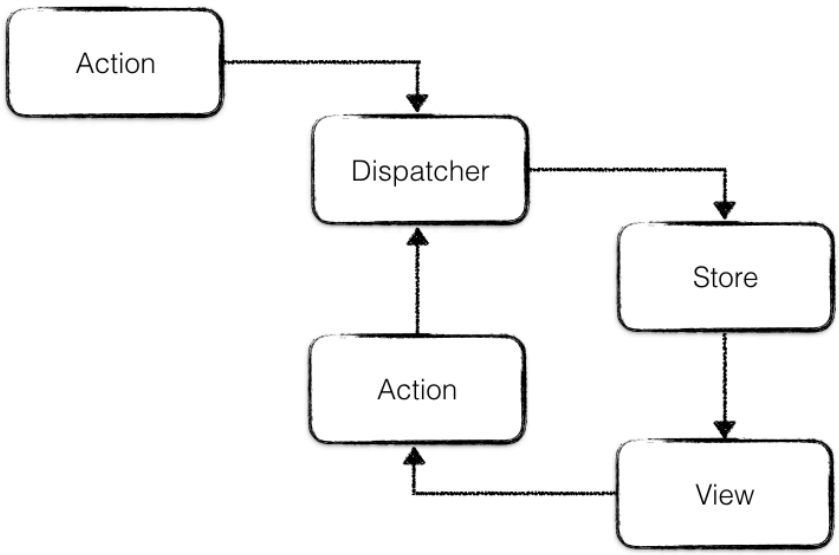


Abbildung 14 Flux Lebenszyklus

Das Flux Konzept führt zu einem deterministischen Applikationszustand und View, indem Seiteneffekte, wie zum Beispiel asynchrone Netzwerk Anfragen, separiert sind.

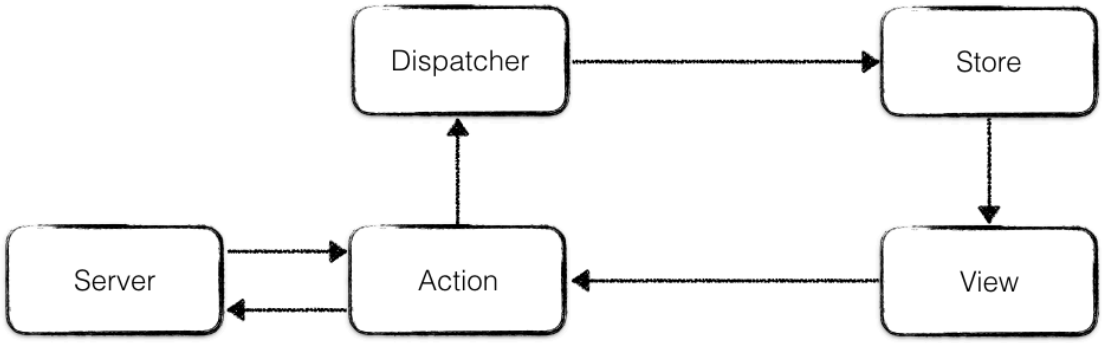


Abbildung 15 Klare Trennung von Seiteneffekten in Flux nach (Elliot, 10 Tips for Better Redux Architecture, 2016)

Zusammenfassend lässt sich sagen, dass Flux ein Konzept für das Zustandsmanagement innerhalb einer Applikation ist, die es vereinfacht, nachzuvollziehen, wie Datenänderungen in einer Applikation fließen und wie der aktuelle Zustand der Daten und der Applikation zu Stande gekommen ist (de Sousa Antonio, Pro React, 2015, S. 207). Eine Gesamtübersicht des Flux Konzepts zeigt Abbildung 16.

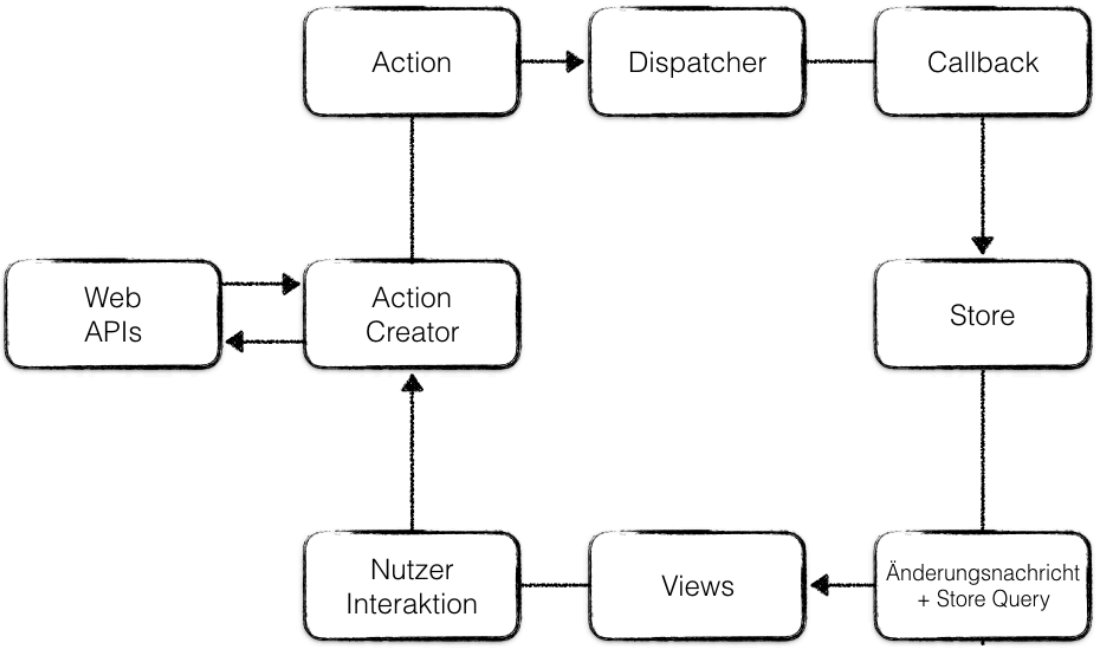


Abbildung 16 Das komplette Flux Konzept

2.5.2 Redux

Redux basiert auf dem im vorigen Kapitel beschriebenen Flux Konzept und wird beschrieben als Zustandscontainer der dabei unterstützen soll, das eine Web Applikation sich konsistent verhält. Redux probiert das Flux Konzept zu vereinfachen, indem Redux sich von Konzepten der Programmiersprache Elm beeinflussen lassen hat (Abramov, Getting Started with Redux, 2016).

Der Begriff Redux als solches bedeutet „zurückgegeben“ und ist gleichzeitig ein Kunstwort aus den beiden Wörtern „Reducer“ und „Flux“ (Geary, 2016).

(Kol, 2016) beschreibt Redux in der Weise, die unsere Denkweise über den Zustand einer Web Applikation in eine explizite Richtung führt und den Fokus darauflegt, den Daten- sowie auch den Applikationszustand während der Erstellung einer JavaScript Web Applikation professionell zu modellieren.

Redux wird auch als ein Paradigma bezeichnet. (Valdecantos, 2016, S. 36) definieren ein Paradigma als eine definierte Denk- und Herangehensweise die gegenüber einem Problem eine durch einen Computer ausführbare Lösung formuliert.

Redux stellt ein konkretes Lösungsmodell für das Thema Zustandsmanagement bereit. Redux folgt den Konzepten von Flux, Command Query Responsibility Segregation (CQRS) und Event Sourcing (Abramov & Contributors, Redux Motivation, 2015).

Das zentrale Ziel von Redux ist den Zustand einer Web Applikation vorhersehbar zu machen. Diese Vorhersehbarkeit soll entstehen durch festgelegte Einschränkungen, wie und wann Aktualisierungen im Zustand durchgeführt werden können.

Nach (Lumpe, Purkhardt, Muller, Cravero, & Chentnik, Developing a Redux Edge, 2016, S. 7-16) basiert Redux auf den folgenden Kernkonzepten. Actions, Action Creators, Reducer Funktionen und einem Store. Diese Kernkonzepte werden im Folgenden erklärt, um zu verstehen, wie daraus das Redux Paradigma entsteht.

Actions

Actions werden genutzt um zwischen einer Web Applikation und Redux zu kommunizieren. Eine Action ist der Weg um Redux mitzuteilen, dass sich etwas geändert hat oder der Anwender mit der vorliegenden Applikation interagiert hat. Actions können auch als Beschreibungsobjekte bezeichnet werden, die Informationen darüber enthalten, was passiert ist.

(Elliot, 10 Tips for Better Redux Architecture, 2016) nennt Actions auch „Transaktionsobjekte”.

Eine Action ist ein einfaches JavaScript Objekt welches minimal die Eigenschaft „type“ hat. Neben der „type“ Eigenschaft kann das Action Objekt enthalten was immer gebraucht wird. Es folgen zwei Redux Action Beispiele. Das erste Beispiel zeigt ein minimales Action Objekt, indem die “type” Eigenschaft ausreichend ist, um die Änderung im Zustand der Web Applikation zu beschreiben.

```
{
  type: 'BACKSPACE'
}
```

Listing 1 Minimales Redux Action Objekt

Und ein Action Objekt mit zusätzlichem Attribut

```
{
  type: 'CHARACTER_TYPED',
  character: 'x'
}
```

Listing 2 Redux Action Objekt mit zusätzlicher Eigenschaft

Innerhalb des Redux Paradigmas wird ein API Aufruf als Seiteneffekt bezeichnet, weil die Verarbeitung des API Aufrufs asynchron abgearbeitet wird. Deswegen wird solch ein Aufruf innerhalb einer Action isoliert (Elliot, How to Redux, 2016).

(Abramov & Contributors, Redux - Three Principles, 2016) beschreiben, dass Actions ein Zustandsänderungsvorhaben formulieren.

Action Creator

Action Creators sind einfache JavaScript Funktionen, die ein Action Objekt zurückgeben. Man könnte das Action Objekt auch zu jeder Verwendung erneut deklarieren, jedoch gilt es dem „Don´t repeat yourself“ (DRY) Prinzip folgen, Action Creator zu verwenden, um Schreibfehler zu vermeiden und die die Testbarkeit der Web Applikation zu erhöhen.

```
const insertCharacterActionCreator = (char) => {
  return {
      type: 'CHARACTER_TYPED',
      character: char
    }
};
```

Listing 3 Redux Action Creator Funktion

Reducer

Ein Reducer verarbeitet bzw. interpretiert Actions. Der Reducer ist implementiert als Funktion die Eingangsdaten entgegennimmt und Ausgangsdaten zurückgibt.

```
1  function (a, b) {
2    // c = Processing result of a and b
3    return c;
4  };
```

Listing 4 Redux Reducer Basis Konzept

Der Beispiel Reducer operiert auf den zwei Argumenten und produziert ein Ergebnis das zurückgegeben wird.

Eines der zentralen Redux Konzepte definiert weiterhin, dass bestehende Zustandsdaten niemals verändert werden dürfen. Stattdessen soll die Reducer Funktion immer auf einem frischen Objekt die Zustandsmutation durchführen und dieses neue Objekt zurückgeben, was mit dem „Spread Operator“ Vorschlag oder einer Bibliothek wie „Immutable“ gut realisiert werden kann (Abramov & Contributors, Redux - Prior Art, 2016). Man nennt diese Art der Datenstruktur auch eine „immutable“ Datenstruktur. Wird der Zustand der Applikation geändert wird ein neuer Zustandsbaum generiert, welcher die neue Änderung enthält, das vorige Zustandsobjekt bleibt nach wie vor bestehen. Beide Zustandsbäume sind komplett unabhängig voneinander, der Übergang von Zustandsbaum A in Zustandsbaum B ist schematisch in Abbildung 17 dargestellt.

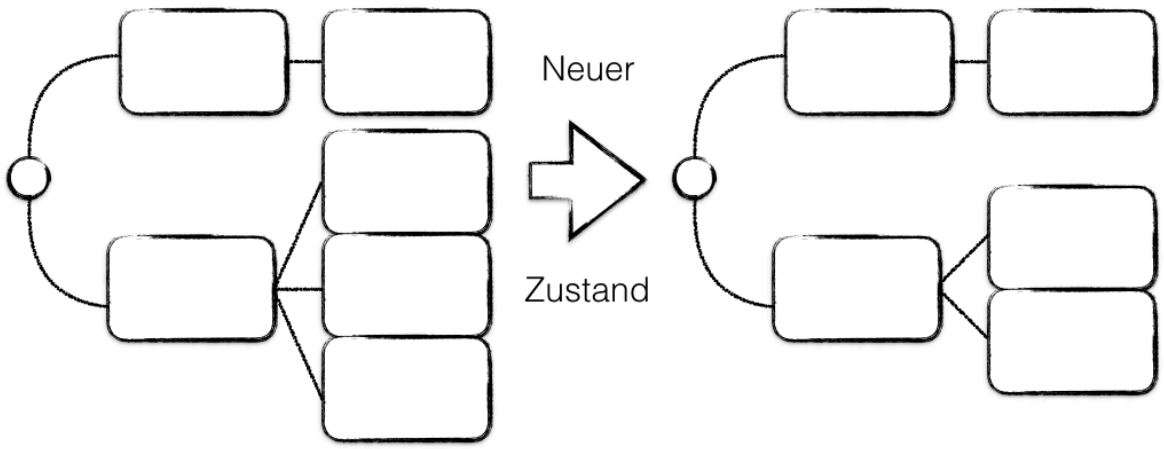


Abbildung 17 Redux Zustandsänderungsschema nach (Parviainen, 2015)

Damit die Ergebnisse der Reducer Funktion deterministisch sind und immer die gleichen sind muss die Reducer Funktion pur sein. Pure Funktionen sind solche Funktionen, die nur auf ihren Eingangsdaten operieren und darauf basierend immer die gleichen Ausgangsdaten produzieren. Eine pure Funktion darf keine veränderbaren Variablen außerhalb ihres eigenen Ausführungskontexts referenzieren. Außerdem darf in einer puren Funktion keine nicht deterministische Funktion wie zum Beispiel „Math.random“ verwendet werden, weil sich der Rückgabewert dieser Funktionen mit jeder Ausführung ändern würde und nicht ausschließlich abhängig von den Funktionseingangsdaten wäre. Alle nicht deterministischen Programmteile müssen außerhalb der Reducer Funktionen geschrieben werden. Die Aspekte der funktionalen

Programmierung zeigen sich am deutlichsten in der Art und Weise der Implementierung der Reducer Funktion innerhalb von Redux (Elliot, 10 Tips for Better Redux Architecture, 2016).

In Redux werden die Reducer Funktionen dazu verwendet, den Zustand einer Applikation aufgrund von Actions zu verändern.

```
function (state, action) {
   // newState = Processing result of state and action
  return newState;
}
```

Listing 5 Redux Reducer Konzept mit Redux Terminologie

Listing 6 zeigt eine Reducer Funktion die zwei Actions aus den vorigen Listings verarbeiten kann.

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

    case 'CHARACTER_TYPED':
      const { character } = action;
      return state + character;

    case 'BACKSPACE':
      return state.substr(0, state.length - 1);

    default:
      return state;
  }
}
```

Listing 6 Vollständiger Redux Reducer

Zu Beginn der Reducer Funktion wird sichergestellt, dass der aktuelle Zustand der Applikation vorhanden ist, falls nicht, wird der Zustand der Applikation auf einen leeren String gesetzt.

Anhand der switch-Anweisung wird der Action “type” ausgewertet und in den entsprechenden Änderungszweig gesprungen.

Außerdem muss in jedem Redux Reducer die “default” Anweisung existieren, die der Reducer immer dann befolgt, wenn der Action “type” unbekannt ist. Das ist wichtig, weil in einer normalen Web Applikation mehrere Reducer und viele Actions existieren. Würde die Reducer Funktion nur etwas zurückgegeben, wenn der Action „type“ bekannt ist, würde eine unbekannte Action den Applikationszustand auf ihren Ursprungszustand zurücksetzen.

Mithilfe der Reducer Komposition kann ein großer Reducer in mehrere, kleinere Reducer Funktionen aufteilt werden. Das hilft dabei, den Applikationszustand in besser überschaubarere Teile aufzuspalten. Für diese Aufgabe stellt Redux die Helferfunktion „combineReducers“ bereit.

Der Store

Innerhalb von Redux wird der komplette Applikationszustand in einem einzigen Objektbaum gespeichert. Dieser Objektbaum befindet sich innerhalb eines Stores (Abramov & Contributors, Redux - Three Principles, 2016). Der Store in Redux ist verantwortlich dafür, dass die Actions zur Reducer Funktion gelangen und über Benachrichtigungen, dass sich etwas im Zustand der Applikation geändert hat. Der Store kapselt den Zustand und stellt ein Interface bereit, mithilfe dessen mit dem Zustand interagiert werden kann. Auf den Store kann nur lesend, nicht schreibend zugegriffen werden. Diese Einschränkung stellt auch sicher, dass weder über den View noch über Netzwerk Callbacks direkt der Zustand der Applikation verändert wird (Abramov & Contributors, Redux - Three Principles, 2016).

Der Store stellt die Funktionen “dispatch(action)”, “getState()” und “subscribe(listener)” bereit und wird folgendermaßen erstellt.

```
1  import { createStore } from 'redux';
2
3  // reducer is the reducer we defined earlier
4  const store = createStore(reducer);
```

Listing 7 Redux Store Erstellung

Mit dem erstellten Store können Actions gesendet werden. Senden heißt im Fall von Redux dass eine Action dem Store übergeben wird, welcher wiederum die Action an die Reducer-Funktion weitergibt, damit diese die Action verarbeiten kann. Das senden einer Action funktioniert wie in Listing 8 gezeigt wird, hier wird bewusst das inline Objekt Literal als Actions verwendet und nicht der sonst empfohlene Action Creator.

```
1  store.dispatch(
2    {
3      type: 'CHARACTER_TYPED',
4      char: 'x'
5    }
6  );
```

Listing 8 Redux Store senden einer Action

Wir können die Funktion von “dispatch()” prüfen indem der Zustand des Applikation bzw. des Stores abgefragt wird.

```
const state = store.getState();
// Logs 'x'
console.log(state);
```

Listing 9 Redux Store getState()

Es fehlt noch die letzte Funktion, anhand der ein Event Listener beim Store registriert werden kann. Dieser Beobachter wird jedes Mal ausgeführt, wenn sich etwas am Zustand der Applikation ändert.

```
const unsubscribe = store.subscribe( () => console.log( store.getState() ) );
```

Listing 10 Redux Store benachrichtigt Subscriber nach Änderung

Wenn anschließend eine Action gesendet wird, wird der Beobachter den neuen Zustand in die Konsole loggen. Das passiert für jede Action, d.h. für jede Änderung im Applikationszustand, unabhängig von wem oder wo die Änderung initiiert wird.

```
// Logs 'x'
store.dispatch(insertCharacter('x'));
// Logs 'xy'
store.dispatch(insertCharacter('y'));

// `removeCharacter` is an action creator wrapping the `BACKSPACE` action
// Logs 'x'
store.dispatch(removeCharacter());
// Logs ''
store.dispatch(removeCharacter());
```

Listing 11 Redux Store Beobachter Demo

Um das Beobachten der Änderungen im Zustand wieder zu beenden wird die Funktion ausgeführt, die von “store.subscribe()” zurückgegeben wurde. Diese entfernt den vorher angelegten Beobachter vom Store.

```
unsubscribe();
// Nothing is logged
store.dispatch(insertCharacter('z'));
```

Listing 12 Redux Store Unsubscribe

Damit haben wir einen vollfunktionsfähigen Redux Store erstellt und erklärt.

Abbildung 18 zeigt den kompletten Redux Lebenszyklus.

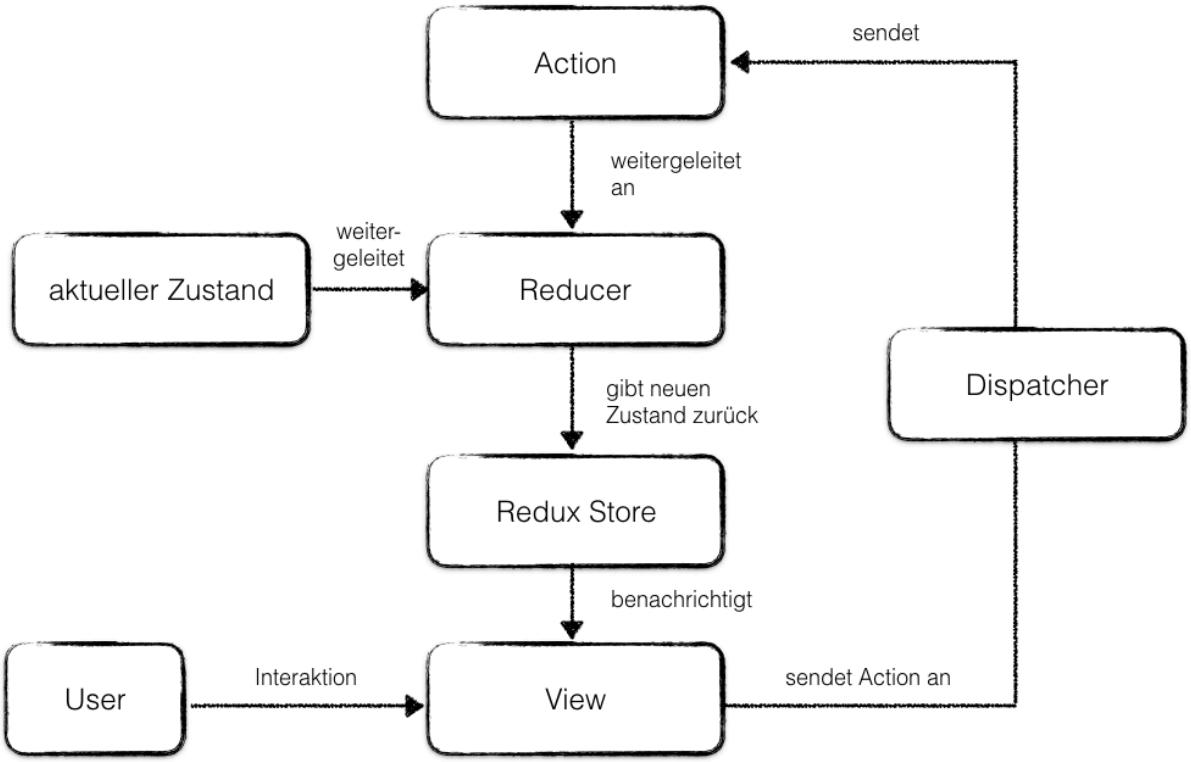


Abbildung 18 Redux Lebenszyklus nach (Geary, 2016)

Zusammenfassend lässt sich sagen, dass Redux dazu führt, dass man explizit über den Zustand einer Web Applikation nachdenkt und sich Zeit nimmt, diesen zu modellieren und in einer expliziten Struktur zu verwalten (Tol, 2016).

2.5.3 Vergleich des klassischen Zustandsmanagements mit dem Redux Paradigma

Einleitend zum abschließenden Kapitel des theoretischen Teils der vorliegenden Masterarbeit möchte ich auf den in Kapitel 2.1 „JavaScript Web Applikationen“ definierten Kontext von modernen JavaScript Web Applikationen verweisen, bei dem es um komplexe Web Applikationen mit hoher Benutzerinteraktivität und einer Vielzahl an Zustandsänderungen geht.

Für die vorliegende Arbeit wurde Redux als zurzeit populärste „Flux Implementierung“ gewählt.

Möchte man nun die Zustandsmanagement Lösungen in bisherigen MV*-Architekturen mit dem Redux Paradigma vergleichen, lässt sich feststellen, dass beide Ansätze auf einer nicht direkt vergleichbaren Ebene agieren. Man kann sagen, dass das Redux Paradigma ein Teil einer MV*-Architektur sein könnte, indem jedoch zur herkömmlichen MV*-Architektur in einer extrem strikten Art und Weise definiert ist, wie der Datenfluss innerhalb der Applikation implementiert ist und wie der Zustand verwaltet wird.

Um auf theoretischer Ebene den Vergleich machen zu können wurden als Vertreter für bisherige MV*-Architekturen die Bibliothek Backbone.js und Angular ausgewählt. In Backbone.js wird der Zustand einer JavaScript Web Applikation in einem Backbone.Model vorgehalten. In AngularJS Version 1 wird der Zustand oft in Factories und Services verwaltet. In Angular gibt es verschiedene Möglichkeiten den Zustand und Datenfluss einer JavaScript Web Applikation zu gestalten. In Flux Implementierungen wird der Applikationszustand in Stores vorgehalten (Parviainen, 2015).

Es lässt sich festhalten, das Backbone.js das klassischste MV*-Architektur Zustandsmodell implementiert. Backbone stellt Model Objekte zur Verfügung deren Zustand mit einer einfachen „Setter-"-Methode verändert werden kann (Abramov & Contributors, Redux - Prior Art, 2016). Das Rendering bzw. die Zuordnung zwischen Änderungsereignis auf dem jeweiligen Model und der Aktualisierung der Darstellung im View muss manuell definiert werden. Weitere Konventionen oder Vorgaben bezüglich des Zustand Managements liefert die Backone.js Bibliothek nicht.

AngularJS hat im Übergang von Version 1 zu Version 2 starke Änderungen, insbesondere auch in Anbetracht der empfohlenen Zustandsmanagement Konzepte erfahren. War AngularJS 1 noch nah an der MVVM Variante der MVC-Architektur anzutreffen, ist das Thema Zustandsmanagement in Angular bereits auf die Konzepte eines unidirektionalen Datenflusses vorbereitet. Weiterhin steht in Angular eine ngModel Direkte zur Verfügung, die eine emulierte Art des „Two-Way Data-Binding" bereitstellt.

In den Zustandsmanagement Praktiken in den bisherigen MV*-Architekturen war der Datenfluss oft problematisch, da dieser nicht hinreichend explizit definiert wurde. Insbesondere in größeren Web Applikationen mit vielen Models und Views führt keine definierte Struktur zur schwer vorhersehbaren Abhängigkeiten. (Schläpfer, 2014) stellt weiterhin fest, dass viele bisherige Ansätze dazu tendieren, den Applikationszustand implizit im DOM zu speichern, was dazu führt, dass der Zustand vergessen und an anderer Stelle unnötigerweise dupliziert wird.

Auch (Nesher, 2016) beschreibt, dass eine MV-Architektur immer komplexer wird, je mehr die betrachtete Web Applikation wächst. Nach Meinung des Autors liegt der Hauptgrund hierfür in den sich kaskadierenden Änderungsabhängigkeiten zwischen verschiedenen Models and Views. Er stellt fest, dass die Nachvollziehbarkeit des Datenflusses in den herkömmlichen MV*-Architekturen nur schwer möglich ist.

Auch wenn „Two-Way Data-Binding" deklarativ ist und es vergleichsweise leichtfällt damit zu arbeiten, kann die Fehleranalyse innerhalb dieser Art von Datenbindung schwieriger sein als bei imperativem Code, zu diesem Schluss kommt (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 98).

Außerdem wird beschrieben, dass die Performanz Einbußen für die Datenbindung in einer MVVM Architektur sehr hoch sein können (Osmani A. , Learning JavaScript Design Patterns, 2012, S. 103).

Einer der zentralen Unterschiede zwischen Zustandsmanagement in bisherigen MV*-Architekturen und dem Redux Paradigma ist, dass bei Redux der Zustand in einer einzigen Baumstruktur gespeichert und nicht auf viele Models, wie in herkömmlichen MV*-Architekturen

oftmals implementiert, verteilt ist (Parviainen, 2015). Darauf ergibt sich laut des Autors, dass über den Applikationszustand erstmals unabhängig vom Verhalten der Applikation nachgedacht werden kann. Der Zustand in Redux besteht aus puren Daten, der Zustand hat keine Methoden und wird genau an einem Ort vorgehalten.

(Geary, 2016) ist der Meinung, dass Flux die Ideen bezüglich des Datenflusses in der MVC-Architektur erstmals richtig implementiert und gleichzeitig die Komplexität innerhalb der Architektur unnötig steigert. Gegen die gesteigerte Komplexität in Flux wirkt Redux, indem es die reaktiven funktionalen Konzepte der Programmiersprache Elm im Kontext des Zustandsmanagements in JavaScript Applikationen nachbildet. Auch (Alfoni, 2015) kommt zu dem Schluss, dass Flux eigentlich genau so funktioniert, wie die traditionelle MVC-Architektur gedacht war. Auch Alfoni stellt fest, dass die Definition des Stores die Daten, d.h. die Models, sowie den aktuellen Applikationszustand beinhaltet.

Vergleicht man Redux mit MVC lässt sich laut (de Sousa Antonio, Pro React, 2015, S. 168) sagen, dass der Store in Redux den Models aus MVC entspricht, wobei der Store lediglich Leseoperationen und keine Schreiboperationen, wie in herkömmlichen MV*-Architekturen, erlaubt.

Die sich kaskadierenden Änderungsschleifen in herkömmlichen MV*-Architekturen können mit Redux durch die explizite Kontrolle der Ausführungsreihenfolge von Actions innerhalb des Stores vermieden werden (Gackenheimer, 2015, S. 8).

Auf den ersten Blick kann das Redux Konzept als Limitierung wahrgenommen werden, besonders, wenn man von einem objektorientierten Hintergrund auf das Paradigma schaut, jedoch stellt (Parviainen, 2015) fest, dass die feste Struktur hilft sich ausschließlich auf den Zustand und die Daten innerhalb der Applikation zu fokussieren was wiederum eine Erleichterung für den Entwickler darstelle.

Letztendlich hält (Elliot, How to Redux, 2016) fest, dass Zustandsmanagement anhand des Redux Paradigma noch mehr die Trennung von verschiedenen Bestandteilen innerhalb einer JavaScript Web Applikation manifestiert, indem der Teil des Zustandsmanagements und des Datenflusses viel genauer und strikter definiert wird. Diese genaue Definition des wichtigsten Themas des Zustandsmanagements führt hinzu einem nachvollziehbaren und deterministischem View Ergebnis, indem Seiteneffekte gekapselt sind (Elliot, 10 Tips for Better Redux Architecture, 2016).

Tabelle 3 JavaScript Zustandsmanagement Lösungen Übersicht

	Typ	**Starke Konventionen**	**Alle SPA Komponenten**	**Enthält Datenmodelling**	**Enthält Datenbindung**
Backbone.js	Bibliothek	nein	nein	ja	nein
Angular	Framework	Ja	Ja	teilweise	ja, One-Way Two-Way
Redux	Zustands-container	Ja	Nein	Ja	teilweise, One-Way

3 IMPLEMENTIERUNG UND VERGLEICH DER BETRACHTETEN ZUSTANDSMANAGEMENT PARADIGMEN

Der Praxisteil der vorliegenden Masterarbeit basiert auf zwei konkreten Programmcode Implementierungen einer Beispielapplikation in Angular. Die erste Implementierung realisiert ein Zustandsmanagement anhand von einem Daten Service in Angular. Die zweite Implementierung realisiert das Zustandsmanagement anhand des Redux Paradigmas.

Ergänzend zur praktischen Umsetzung der zu vergleichenden Zustandsmanagement Paradigmen werden innerhalb des folgenden Praxisteils weitere Maßnahmen ergriffen, um die beiden Ansätze miteinander vergleichen zu können.

Die Methodik zur Evaluation der verschiedenen Zustandsmanagement Paradigmen verfolgt einen Vergleich nach den folgenden quantitativen Merkmalen

- Programmcode Zeilen
- Performance
- Geschätzte Dauer der Neuentwicklung von Funktonalität mit Auswirkung auf den Applikationszustand
- Geschätzte Dauer der Änderung von Funktonalität mit Auswirkung auf den Applikationszustand
- Geschätzte Dauer Entwicklung eines Unit Tests für den Applikationszustand
- Befragung nach dem Prinzip der geringsten Überraschung

Beide Beispielimplementierungen werden zusammen mit einer Umfrage weiteren Web-Entwicklern zur Verfügung gestellt, um neben den persönlichen Erfahrungen im Verlauf der Umsetzung relevante Daten erheben zu können. Die Entwicklerbefragung erhebt eingangs zur besseren Einordnung und Gewichtung die bisherige Entwicklungserfahrung der Teilnehmer und adressiert im Hauptteil der Umfrage die vergleichenden Größen zu den beiden vorliegenden Beispielimplementierungen.

Des Weiteren betrachtet der sich anschließende Vergleich die beiden qualitativen Merkmale der Entwicklung der Verbreitung des Redux Paradigmas und das „Thinking Aloud“ bei der Konfrontation von Dritten mit beiden Ansätzen.

3.1 Beispielhafte Umsetzung Zustandsmanagement

Als eines der zurzeit stark verbreiteten JavaScript MV*-Frameworks werden die zu betrachtenden Beispiel Implementierungen in Angular (zum Zeitpunkt der Bearbeitung ist Version 2 aktuell) umgesetzt. „Angular“ steht als Synonym für Angular ab Version 2 und für weitere Folgeversionen. Der Begriff „AngularJS“ bezeichnet Angular in Version 1.x (Fluin, 2017).

Die Beispiel Applikation greift die vorhandene „example app“ von @ngrx als Ausgangspunkt auf. Die Beispielapplikation ist eine im Browser lauffähige Bücherverwaltung. Per Google Books API kann nach Buchtiteln gesucht werden. Gefundene Buchtitel können in einer lokalen Sammlung gespeichert werden. Auch das Entfernen von vorhandenen Einträgen aus der lokalen Büchersammlung ist möglich.

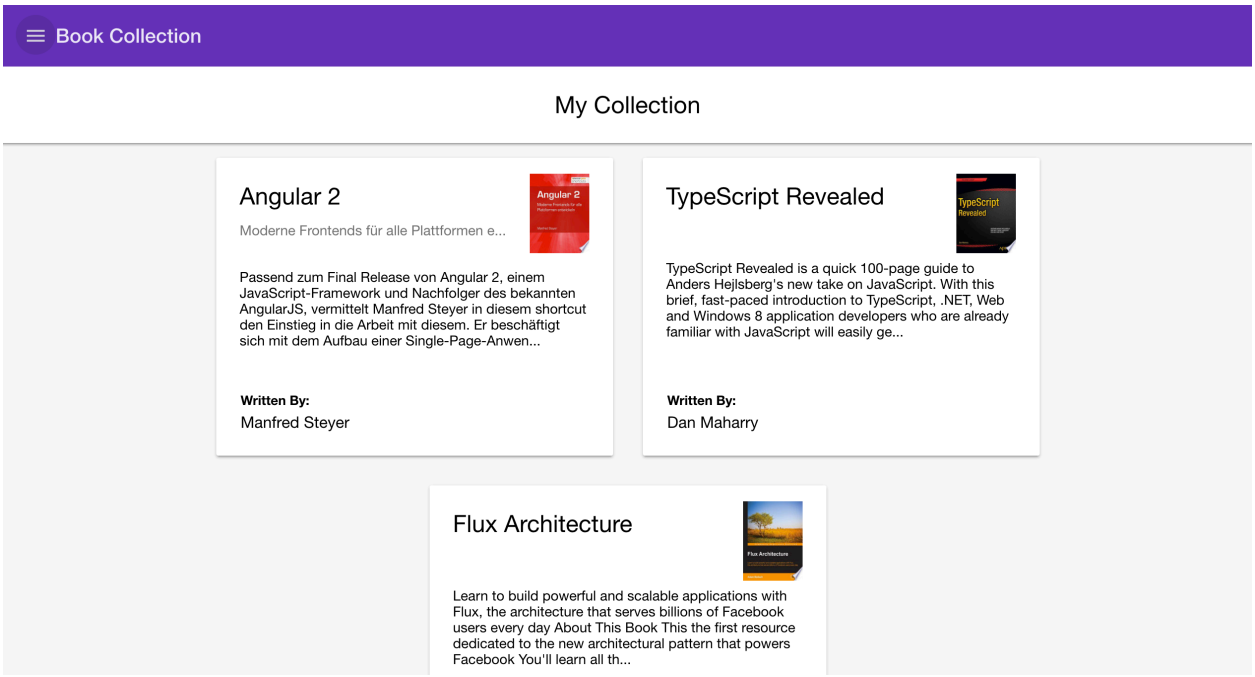


Abbildung 19 Die Beispiel Applikation „Büchersammlung“

Wichtig zu erklären ist, warum im anschließenden Vergleich zwischen den beiden Zustandsmanagement Paradigmen jeweils RxJS eingesetzt wird. Da RxJS einen gewissen Stellenwert als De Facto-Implementierung in Angular hat und von @ngrx verwendet wird, ist es die als zukunftweisende „reaktive Programmierung“ im Praxisteil der vorliegenden Masterarbeit aufgegriffen wurden.

Die reaktive Erweiterung für JavaScript (RxJS) ist eine reaktive Stream Bibliothek, die es ermöglicht, mit asynchronen Daten Streams zu arbeiten.

@ngrx stellt eine professionelle Zusammenstellung von gut getesteten Bibliotheken bereit, um reaktive Programmierung in Angular verwenden zu können (Troncone & Wormald, @ngrx - Reactive Extensions for Angular 2, 2017).

Weiterhin ist festzuhalten, dass das Redux Paradigma keinerlei direkte Abhängigkeit zu RxJS hat. Das Redux Paradigma lässt sich gut mit RxJS kombinieren, jedoch sind keine zwingenden Abhängigkeiten zwischen beiden Projekten gegeben.

Eine live Demo der Beispielapplikation kann unter http://ngrx.github.io/example-app/ aufgerufen werden.

Das Frontend der Beispiel Applikation basiert auf Googles Material UI.

3.2 Implementierung eines Daten Service getriebenen Zustandsmanagement mit Angular

Das Zustandsmanagement der ersten Implementierung der Büchersammlung Applikation wurde anhand eines klassischen Daten Services realisiert. Der Code für diese Implementierung kann im digitalen Anhang der vorliegenden Masterarbeit und zum Schnellzugriff auch im online Repository unter https://github.com/pablopaul/data-service-example-app aufgerufen werden.

In der betrachteten Implementierung wird ein wiederverwendbarer Service erstellt, indem die Datenabfragen durchgeführt und die Daten bereitgestellt werden. Durch die Nutzung eines Service kann die Trennung der verschiedenen Verantwortlichkeiten (Daten versus Komponenten) sichergestellt werden. Außerdem können die Daten nach der Abfrage mehreren Komponenten zur Verfügung gestellt werden.

```
14  @Injectable()
15  export class BooksService {
16
17    bookEntities: BehaviorSubject<any[]>;
18
19    searchTerms = new Subject<string>();
20    searchResultbooks$: Observable<Book[]>;
21
22    idsInCollection: BehaviorSubject<any[]>;
23    idsInCollection$: any;
24
25    selectedBookId: string;
26    isSelectedBookInCollection = new BehaviorSubject(false);
27    isSelectedBookInCollection$ = this.isSelectedBookInCollection.asObservable();
28
29    constructor(private AppService: AppService,
30                private GoogleBooksService: GoogleBooksService,
31                private dbService: DbService) {
32
33      // Book collection
34      this.bookEntities = new BehaviorSubject([]);
35
36      this.idsInCollection = new BehaviorSubject([]);
```

Listing 13 BooksService Klasse dekoriert als Angular „Injectable“

Der Konsument des Daten Service weiß nicht, wie und woher der Service die Daten bekommt. Den Konsumenten der Daten ist es egal, ob der Daten Service beispielsweise auf localStorage oder eine entfernte Schnittstelle zugreift. Das ist einer der klaren Vorteile, wenn man den Datenzugriff aus der Komponente auslagert, da so die Implementierung des Datenzugriffs beliebig geändert werden kann, ohne dass die Komponenten, die die Daten konsumieren, geändert werden müssen.

Um den Daten Service in einer Komponente nutzen zu können, wird der Daten Service per „import“ importiert, um den Service im weiteren Code verwenden zu können. Anschließend wird

der Service im Constructor injiziert, so dass der Daten Service anschließend zur Verfügung steht und genutzt werden kann. Das Injizieren des „BooksService“ passiert im folgenden Beispiel in der „FindBookPageComponent“ in Zeile 21.

```
1  import { Component, ChangeDetectionStrategy } from '@angular/core';
2  import { Observable } from 'rxjs/Observable';
3  import { Subject } from 'rxjs/Subject';
4
5  import { Book } from '../models/book';
6
7  import { AppService } from '../services/app';
8  import { BooksService } from '../services/books';
9
10 @Component({
11   selector: 'bc-find-book-page',
12   template: `
13     <bc-book-search [searching]="loading$ | async" (search)="search($event)"></bc-book-search>
14     <bc-book-preview-list [books]="books$ | async"></bc-book-preview-list>
15   `
16 })
17 export class FindBookPageComponent {
18   books$: Observable<Book[]>;
19   loading$: Observable<boolean>;
20
21   constructor(private BooksService: BooksService,
22               private AppService: AppService) {
```

Listing 14 Injizierung des Data „BooksService“ im Constructor des Konsumenten

Damit Angular den Daten Service „BooksService“ in der Komponente injizieren kann, wird der entsprechende Service in den „providers“ Metadaten registriert, sichtbar in Listing 15 Zeile 51.

```
26  @NgModule({
27    imports: [
28      CommonModule,
29      BrowserModule,
30      MaterialModule,
31      ComponentsModule,
32      RouterModule.forRoot(routes, { useHash: true }),
33
34      /**
35       * `provideDB` sets up @ngrx/db with the provided schema and makes the Database
36       * service available.
37       */
38      DBModule.provideDB(schema),
39    ],
40    declarations: [
41      AppComponent,
42      FindBookPageComponent,
43      SelectedBookPageComponent,
44      ViewBookPageComponent,
45      CollectionPageComponent,
46      NotFoundPageComponent
47    ],
48    providers: [
49      GoogleBooksService,
50      AppService,
51      BooksService,
52      DbService
53    ],
```

Listing 15 Registrierung des Daten Service als Angular Provider

Somit steht der Daten Service in der Komponente bereit und kann dort genutzt werden. Es kann auf Eigenschaften (Zeile 28 Listing 16) und Methoden (Zeile 34 Listing 16) des „BooksService“ Daten Service zugegriffen werden.

```
@Component({
  selector: 'bc-find-book-page',
  template: `
    <bc-book-search [searching]="loading$ | async" (search)="search($event)"></bc-book-search>
    <bc-book-preview-list [books]="books$ | async"></bc-book-preview-list>
  `
})
export class FindBookPageComponent {
  books$: Observable<Book[]>;
  loading$: Observable<boolean>;

  constructor(private BooksService: BooksService,
              private AppService: AppService) {

    // Set loading indicator
    this.loading$ = AppService.loading$;

    // Get books from book service
    this.books$ = BooksService.searchResultbooks$;
  }

  // Act on the search query
  search(query: string) {
    // Kick off the next book search
    this.BooksService.search(query);
  }

}
```

Listing 16 Zugriff auf eine Eigenschaft und Methode des „BooksService“ innerhalb einer Komponente

Während der Implementierung des Daten Service getriebenen Zustandsmanagement ist aufgefallen, dass das es keine definierte Sprache für den Datenzugriff (Selektoren) gibt.

3.3 Implementierung eines Redux getriebenen Zustandsmanagement mit Angular

Die Implementierung der Beispiel Applikation „Büchersammlung“ mit einem Zustandsmanagement anhand des Redux Paradigma nutzt folgende @ngrx Bibliotheken:

- ngrx/store
- ngrx/effects
- ngrx/store-devtools
- ngrx/db

Wie eingangs schon beschrieben wurde, zeigt die Beispiel Applikation eine Redux Implementierung unter Verwendung von RxJS, da die RxJS Erweiterungen bereits fester Bestandteil des Angular Kernprogrammcodes sind.

@ngrx/store ist ein kontrollierter Zustandscontainer, welcher darauf abzielt, performante und konsistente Web Applikationen auf Angular Basis zu ermöglichen. @ngrx/store ist abgeleitet von dem Redux Paradigma und davon ausgehend findet sich der Zustand innerhalb von @ngrx/store als einzelne, nicht veränderbare Datenstruktur wieder. Außerdem beschreiben innerhalb von @ngrx/store die sog. Actions Zustandsänderungen. Auch Reducer spielen eine zentrale Rolle, diese sind wie im Redux Paradigma vorgegeben pure Funktionen, die einen bisherigen Zustand mit einer Action kombinieren um den nächsten Zustand bereitzustellen. Auf den Zustand wird als „Store“ zugegriffen, der in der verwendeten Implementierung als Observable des Zustands und als Observer von Actions fungiert. Die Observable und Observer Funktionalität ist @ngrx/store spezifisch und nicht im Redux Paradigma genannt. Durch die Umsetzung der beschriebenen Prinzipien kann die in Angular vorhandene „OnPush“ Änderungsdetektion eingesetzt werden, was eine sehr intelligente und performante Form der Änderungsdetektion ermöglicht.

@ngrx/effects sind Quellen von Actions und werden für Seiteneffekt behaftete Actions genutzt. @ngrx/effects wird auch als „Seiteneffekt“ Modell für @ngrx/store bezeichnet. Anhand des @Effect() Decorators wird markiert, welche Observables eines Service eine Quellfunktion für Actions sind. Die Bibliothek führt die Action Streams automatisch zusammen, so dass man diese am Store abonnieren („subscriben“) kann. @ngrx/effects exportiert außerdem ein Actions Observable, welches jede in der Anwendung verwendete Action empfängt.

@ngrx/store-devtools ist eine Bibliothek, die wertvolle Entwickler Werkzeuge für das Zustandsmanagement bereithält. @ngrx/store-devtools setzt die installierte „Redux Devtools Extension“ in der jeweiligen Umgebung voraus, im Falle der vorliegenden Masterarbeit wird die „Redux Devtool Extension“ als Chrome Erweiterung verwendet. Mit diesem Entwickler Werkzeug ist es möglich, die Änderungen des Zustands der Beispiel Applikation zu protokollieren und einfach zu verändern.

@ngrx/db ist ein experimenteller Wrapper um IndexedDB in Kombination mit RxJS speziell für Angular Web Applikationen.

(Davis, Ryan, & Salathe, 2016)

Nach (Troncone, Comprehensive Introduction to @ngrx/store, 2016) besteht @ngrx/store aus den zwei Hauptkomponenten Store und Dispatcher. Beide erweitern RxJS Subjects und sind daher Observable und Observer zur gleichen Zeit, das heißt, man kann Sie abonnieren und gleichzeitig kann man am Subject auch eine andere Quelle abonnieren.

Da Subjects auch Observer sind, können per Aufruf der „next()“ Funktionen neue Werte in den Observer gereicht werden. Abonnenten des Subjects werden anschließend über die neuen Werte informiert. Die nachfolgenden Code Beispiele zeigen die in @ngrx/store verwendet Beispiele in konzeptueller Art und Weise, so dass diese einfach nachvollziehbar sein, die eigentliche @ngrx/store Implementierung dieser Konzepte ist wesentlich umfangreicher und robuster.

```
1  // Create a subject
2  const mySubject = new Rx.Subject();
3
4  // Add subscribers
5  const subscriberOne = mySubject.subscribe(val => {
6    console.log('***SUBSCRIBER ONE***', val);
7  });
8
9  const subscriberTwo = mySubject.subscribe(val => {
10   console.log('***SUBSCRIBER TWO***', val);
11 });
12
13 // Publish values to observers of subject
14 mySubject.next('FIRST VALUE!'); '***SUBSCRIBER ONE*** FIRST VALUE! ***SUBSCRIBER TWO*** FIRST VALUE!'
15 mySubject.next('SECOND VALUE!'); '***SUBSCRIBER ONE*** SECOND VALUE! ***SUBSCRIBER TWO*** SECOND VALUE!'
```

Listing 17 RxJS: Subject Abonnement Beispiel

Dem Redux Paradigma folgend werden Actions an den Applikationszustand (Store) gesendet. Innerhalb von @ngrx/store wird diese API folgendermaßen bereitgestellt:

```
// Inherit from subject
class Dispatcher extends Rx.Subject{
  dispatch(value : any) : void {
    this.next(value);
  }
}

// Create a dispatcher (just a Subject with wrapped next method)
const dispatcher = new Dispatcher();

// add subscribers
const subscriberOne = dispatcher.subscribe(val => {
  console.log('***SUBSCRIBER ONE***', val);
});

const subscriberTwo = dispatcher.subscribe(val => {
  console.log('***SUBSCRIBER TWO***', val);
});

// Publish values to observers of dispatcher
dispatcher.dispatch('FIRST DISPATCHED VALUE!');
dispatcher.dispatch('SECOND DISPATCHED VALUE!');
```

Listing 18 RxJS Subject als Dispatcher

Damit Komponenten und Services zur jederzeit Zugriff auf den initialen oder aktuellsten Applikationszustand haben, wird für den Store das sog. „BehaviorSubject“ verwendet, da diese Art von Subject, im Gegensatz zum herkömmlichen Subject, den letzten Wert direkt nach dem Abonnement zurückgibt.

```
class Store extends Rx.BehaviorSubject{
  constructor(initialState : any){
    super(initialState);
  }
}

const store = new Store('INITIAL VALUE!');

// Add a few subscribers
const storeSubscriberOne = store.subscribe(val => {
  console.log('***STORE SUBSCRIBER ONE***', val);
});
const storeSubscriberTwo = store.subscribe(val => {
  console.log('***STORE SUBSCRIBER TWO***', val);
});

// For demonstration, manually publish values to store
store.next('FIRST STORE VALUE!');

// Add another subscriber after 'FIRST VALUE!' published
// Output: ***STORE SUBSCRIBER THREE*** FIRST STORE VALUE!
const subscriberThree = store.subscribe(val => {
  console.log('***STORE SUBSCRIBER THREE***', val);
});
```

Listing 19 @ngrx/store Store als RxJS BehaviorSubject

In einer Store Applikation werden alle entsendeten Actions durch einen definierten Kontrollfluss (Pipeline) geleitet, bevor eine neue Repräsentation des Zustands in den Store weitergeleitet werden kann, um dann für alle Observer bekanntgegeben werden zu können. Die Erstellung der Pipeline passiert dadurch, dass der Dispatcher in den Store weitergereicht wird, wenn dieser erstellt wird. Anschließend wird die Store Methode „next()“ überschrieben, um alle Actions erst zu der Dispatcher Pipeline delegieren zu können, bevor der neue Zustand zurückgegeben wird. Das erlaubt es, dass der Store direkt den Action-Stream abonniert und die empfangenen Actions durch den Dispatcher reicht.

Das folgende Beispiel enthält noch nicht die Implementierung der Middleware und Reducer, an deren Stelle steht nur ein Kommentar.

```
/* All actions should pass through pipeline before newly calculated state is passed to store.
1.) Dispatched Action
2.) Pre-Middleware
3.) Reducers (return new state)
4.) Post-Middleware
5.) store.next(newState) */

class Dispatcher extends Rx.Subject{
  dispatch(value : any) : void {
    this.next(value);
  }
}

class Store extends Rx.BehaviorSubject{
  constructor(
    private dispatcher,
    initialState
  ){
    super(initialState);
    /* All dispatched actions pass through action pipeline before new state is passed
    to store */
    this.dispatcher
       // Pre-middleware
       // Reducers
       // Post-middleware
       .subscribe(state => super.next(state));
  }
```

```
28
29    // Delegate store.dispatch first through dispatched action pipeline
30    dispatch(value){
31      this.dispatcher.dispatch(value);
32    }
33
34    // Override store next to allow direct subscription to action streams by store
35    next(value){
36      this.dispatcher.dispatch(value);
37    }
38  }
39
40  const dispatcher = new Dispatcher();
41  const store = new Store(dispatcher, 'INITIAL STATE');
42
43  const subscriber = store.subscribe(val => console.log(`VALUE FROM STORE: ${val}`));
44
45  // Both methods are same behind the scenes
46  dispatcher.dispatch('DISPATCHED VALUE!');
47  store.dispatch('ANOTHER DISPATCHED VALUE!');
48
49  const actionStream$ = new Rx.Subject();
50  /* Overriding store next method allows us to subscribe store directly to action streams,
51  providing same behavior as manually calling store.dispatch or dispatcher.dispatch */
52  actionStream$.subscribe(store);
53  actionStream$.next('NEW ACTION!');
```

Listing 20 @ngrx/store Store & Dispatcher Beispiel Code

Dem Redux Paradigma folgend benutzt auch @ngrx/store das Reducer Konzept um bestimmte Teile des Applikationszustands zu verändern. Reducer nehmen als Parameter den aktuellen Zustand und eine Action, und folgen anschließend Switch Anweisungen. Jedes Mal, wenn eine Action entsendet wird, werden alle am Store registrierten Reducer aufgerufen (durch den root reducer, der in provideStore während Applikation Bootstrap erstellt wurde) und der selektierte Teil des Zustands (Akkumulator) und die entsendete Action übergeben werden. Wenn der Reducer registriert ist, um diese Art der Action zu behandeln, dann wird der entsprechend neue Zustand berechnet und zurückgegeben. Wenn der Reducer nicht für diese Art von Action verantwortlich ist, wird der unveränderte, aktuelle Applikationszustand zurückgegeben.

```
// Redux-Style Reducer
const person = (state = {}, action) => {
  switch(action.type){
    case 'ADD_INFO':
      return Object.assign({}, state, action.payload)
    default:
      return state;
  }
}

const infoAction = {type: 'ADD_INFO', payload: {name: 'Brian', framework: 'Angular'}}
const anotherPersonInfo = person(undefined, infoAction);
console.log('***REDUX STYLE PERSON***: ', anotherPersonInfo);

// Add another reducer
const hoursWorked = (state = 0, action) => {
  switch(action.type){
    case 'ADD_HOUR':
      return state + 1;
    case 'SUBTRACT_HOUR':
      return state - 1;
    default:
      return state;
  }
}


// Combine Reducers Refresher
const myReducers = {person, hoursWorked};
const combineReducers = reducers => (state = {}, action) => {
  return Object.keys(reducers).reduce((nextState, key) => {
    nextState[key] = reducers[key](state[key], action);
    return nextState;
  }, {});
};

const rootReducer = combineReducers(myReducers);
const firstState = rootReducer(undefined, {type: 'ADD_INFO', payload: {name: 'Brian'}});
const secondState = rootReducer({hoursWorked: 10, person: {name: 'Joe'}}, {type: 'ADD_HOUR'});
console.log('***FIRST STATE***:', firstState);
console.log('***SECOND STATE***:', secondState);
```

Listing 21 @ngrx/store Redux Reducer Code

Der Operator „scan()“ verhält sich ähnlich wie „reduce()“, jedoch wird der Akkumulator Wert im Verlauf der Zeit beibehalten, so dass auch in der Folge von neu entsendeten Actions der Zustand immer den letzten Stand repräsentiert.

```
const testSubject = new Rx.Subject();
// Basic scan example, sum over time starting with zero
const basicScan = testSubject.scan((acc, curr) => acc + curr, 0);

// Log accumulated values
const subscribe = basicScan.subscribe(val => console.log('Accumulated total:', val));

// Pass values into our test subject, adding to the current sum
testSubject.next(1); //1
testSubject.next(2); //3
testSubject.next(3); //6

const testSubjectTwo = new Rx.Subject();

// Scan example building an object over time
const objectScan = testSubjectTwo.scan((acc, curr) => Object.assign({}, acc, curr), {});

// Log accumulated values
const subscribe = objectScan.subscribe(val => console.log('Accumulated object:', val));

// Pass values into our test subject, adding properties to object
testSubjectTwo.next({name: 'Joe'}); // {name: 'Joe'}
testSubjectTwo.next({age: 30}); // {name: 'Joe', age: 30}
testSubjectTwo.next({favoriteFramework: 'Angular 2'}); // {name: 'Joe', age: 30, favoriteFramework: 'Angular 2'}
```

Listing 22 RxJS „scan()“ Beispielcode

Um „scan()“ im Store verwenden zu können, wird es als Operator des Dispatchers verwendet. Alle vom Dispatcher entsendeten Actions durchlaufen „scan()“, führen durch die Reducer mit dem aktuellen Zustand und der Action und geben eine neue Repräsentation des Applikationszustandes zurück. Dieser neue Applikationszustand gelangt dann per „next()“ in den Store und wird von hier aus an alle Abonnenten weitergeben.

```
class Store extends Rx.BehaviorSubject{
  constructor(
    private dispatcher,
    private reducer,
    initialState = {}
  ){
    super(initialState);
    this.dispatcher
       // Pre-middleware

       //Scan is a reduce over time.
       .scan((state, action) => this.reducer(state, action), initialState)

       // Post-middleware
       .subscribe(state => super.next(state));
  }
  // ...Store implementation
}
```

Listing 23 ngrx/store Dispatcher mit “scan()”

Um auf einen Teil des Stores zuzugreifen, bietet sich der Collection Operator „map()“ an. „map()“ wendet eine Funktion auf jedes Element einer Collection an und gibt eine neue Repräsentation dieses Elements zurück. Aufgrund der Tatsache, dass der Applikationszustand eine „Key/Value“ Map ist, ist es einfach, Hilfsfunktionen zu schrieben, die einen spezifischen Teil des Zustands zurückgeben.

```
class Dispatcher extends Rx.Subject{
  dispatch(value : any) : void {
    this.next(value);
  }
}

class Store extends Rx.BehaviorSubject{
  constructor(
    private dispatcher,
    private reducer,
    preMiddleware,
    postMiddleware,
    initialState = {}
  ){
    super(initialState);
    this.dispatcher
       // Pre-middleware

       .scan((state, action) => this.reducer(state, action), initialState)

       // Post-middleware
       .subscribe(state => super.next(state));
  }

  // Map makes it easy to select slices of state that will be needed for your components
  // This is a simple helper function to make grabbing sections of state more concise
  select(key : string) {
    return this.map(state => state[key]);
  }

  // ... Store implementation
}

// ... Create store

// Utilizing the store select helper
const subscriber = store
  .select('person')
  .subscribe(val => console.log('VALUE OF PERSON:', val));
```

Listing 24 @ngrx/store „select()“ Code Beispiel

Aus Performancesicht ist es sinnvoll, dass nur neue Daten an Komponenten weitergegeben werden, solang es neue Werte sind und nicht den alten Werten bzw. dem alten Zustand entsprechen. Genau für diesen Fall gibt es den RxJS Operator „distinctUntilChanged()“. Dieser

Operator entsendet nur einen Wert, wenn der nächste Wert nicht dem davor entsendetem Wert entspricht. Das ist besonders wichtig, da die Store Reducer im Standardfall immer den vorigen Zustand zurückgeben, so lang keine für Sie relevante Action vorhanden ist. Zusammenfassend kann man mit der Verwendung von „distinctUntilChanged()“ erreichen, dass nur eine Aktualisierung durchgeführt wird, wenn sich der Zustand wirklich geändert hat.

```
class Dispatcher extends Rx.Subject{
  dispatch(value : any) : void {
    this.next(value);
  }
}

class Store extends Rx.BehaviorSubject{
  constructor(
    private dispatcher,
    private reducer,
    preMiddleware,
    postMiddleware,
    initialState = {}
  ){
    super(initialState);
    this.dispatcher
       .let(preMiddleware)
       .scan((state, action) => this.reducer(state, action), initialState)
       .let(postMiddleware)
       .subscribe(state => super.next(state));
  }

  // distinctUntilChanged()!
  select(key : string) {
    return this
      .map(state => state[key])
      .distinctUntilChanged();
  }

  // ... Store implementation
}
```

```
// Add reducers...
// Configure store...

const subscriber = store
  // With distinctUntilChanged
  .select('person')
  .subscribe(val => console.log('PERSON WITH DISTINCTUNTILCHANGED:', val));

const subscriberTwo = store
  // Without distinctUntilChanged, will print out extra time
  .map(state => state.person)
  .subscribe(val => console.log('PERSON WITHOUT DISTINCTUNTILCHANGED:', val));

// Dispatch actions
dispatcher.dispatch({
  type: 'ADD_INFO',
  payload: {
    name: 'Brian',
    message: 'Exploring Reduce!'
  }
});

// Person will not be changed
dispatcher.dispatch({
  type: 'ADD_HOUR'
});
```

Listing 25 @ngrx/store „select()“ mit „distinctUntilChanged()“ Code Beispiel

Nachdem die grundlegenden Konzepte von @ngrx/store erläutert worden sind folgt nun ein beispielhafter Überblick über die zentralen Implementierungen in der Beispiel Applikation.

Insgesamt gibt es in der Redux Implementierung fünf Reducer:

- „books“ Reducer
- „collection“ Reducer
- „layout“ Reducer
- „search“ Reducer
- „index“ Reducer

Der Code der jeweiligen Reducer kann unter „ngrx-example-app/src/app/reducers/“ eingesehen werden. Beispielhaft folgt ein Auszug aus dem „collection“ Reducer (ngrx-example-app/src/app/reducers/collection.ts):

```
16  export function reducer(state = initialState, action: collection.Actions): State {
17    switch (action.type) {
18      case collection.ActionTypes.LOAD: {
19        return Object.assign({}, state, {
20          loading: true
21        });
22      }
23
24      case collection.ActionTypes.LOAD_SUCCESS: {
25        const books = action.payload;
26
27        return {
28          loaded: true,
29          loading: false,
30          ids: books.map(book => book.id)
31        };
32      }
```

Listing 26 Redux Beispiel Implementierung „Reducer“

Die Store Actions sind als String Konstanten definiert und im Repository zu finden unter „ngrx-example-app/tree/master/src/app/actions“. Es folgt ein Auszug aus den Collection Actions (ngrx-example-app/tree/master/src/app/actions/collection.ts).

```
6   export const ActionTypes = {
7     ADD_BOOK:             type('[Collection] Add Book'),
8     ADD_BOOK_SUCCESS:     type('[Collection] Add Book Success'),
9     ADD_BOOK_FAIL:        type('[Collection] Add Book Fail'),
10    REMOVE_BOOK:          type('[Collection] Remove Book'),
11    REMOVE_BOOK_SUCCESS:  type('[Collection] Remove Book Success'),
12    REMOVE_BOOK_FAIL:     type('[Collection] Remove Book Fail'),
13    LOAD:                 type('[Collection] Load'),
14    LOAD_SUCCESS:         type('[Collection] Load Success'),
15    LOAD_FAIL:            type('[Collection] Load Fail'),
16  };
17
18
19  /**
20   * Add Book to Collection Actions
21   */
22  export class AddBookAction implements Action {
23    type = ActionTypes.ADD_BOOK;
24
25    constructor(public payload: Book) { }
26  }
```

Listing 27 Redux Beispiel Implementierung Actions

In der Implementierung der „ngrx example app“ wird um die String Konstanten noch eine Hilfsfunktion gelegt, um die Prüfung der Typisierung zu stärken.

Die Daten gelangen über sog. Container Komponenten vom Store an die jeweiligen Komponenten. Außerdem sind die Container Komponenten dafür verantwortlich, dass durch Kind Komponenten ausgelöste Actions an die Reducer entsendet werden. Die implementierten Container Komponenten sind unter „ngrx-example-app/src/app/containers/“ zu finden und es folgt als Beispiel die „collection-page.ts“ Container Komponente mit Zugriff auf den Applikationszustand über den Store im Constructor Aufruf in Zeile 36 und 37 in Listing 28.

```
10  @Component({
11    selector: 'bc-collection-page',
12    changeDetection: ChangeDetectionStrategy.OnPush,
13    template: `
14      <md-card>
15        <md-card-title>My Collection</md-card-title>
16      </md-card>
17
18      <bc-book-preview-list [books]="books$ | async"></bc-book-preview-list>
19    `,
20    /**
21     * Container components are permitted to have just enough styles
22     * to bring the view together. If the number of styles grow,
23     * consider breaking them out into presentational
24     * components.
25     */
26    styles: [`
27      md-card-title {
28        display: flex;
29        justify-content: center;
30      }
31    `]
32  })
33  export class CollectionPageComponent {
34    books$: Observable<Book[]>;
35
36    constructor(store: Store<fromRoot.State>) {
37      this.books$ = store.select(fromRoot.getBookCollection);
38    }
39  }
```

Listing 28 Redux Beispiel Implementierung Container Komponente

Als Kind einer Container Komponente fungieren im Normalfall normale Komponenten, die ihre Inputdaten ausschließlich durch ihre Eltern / Container Komponenten beziehen, nachfolgend beispielhaft die „book-preview-list.ts“ Komponente aus „ngrx-example-app/src/app/components“:

```
4  @Component({
5    selector: 'bc-book-preview-list',
6    template: `
7      <bc-book-preview *ngFor="let book of books" [book]="book"></bc-book-preview>
8    `,
9    styles: [`
10     :host {
11       display: flex;
12       flex-wrap: wrap;
13       justify-content: center;
14     }
15   `]
16 })
17 export class BookPreviewListComponent {
18   @Input() books: Book[];
19 }
```

Listing 29 Redux Implementierung Kind Komponente mit @Input()

Es folgt ein Beispiel für eine Komponente die ein Ereignis an ihren Container Eltern Element sendet, um den Zustand zu ändern (ngrx-example-app/src/app/components/book-detail.ts), Zeile 21, 25 und ab Zeile 70 in Listing 30.

```
5  @Component({
6    selector: 'bc-book-detail',
7    template: `
8      <md-card *ngIf="book">
9        <md-card-title-group>
10         <md-card-title>{{ title }}</md-card-title>
11         <md-card-subtitle *ngIf="subtitle">{{ subtitle }}</md-card-subtitle>
12         <img md-card-sm-image *ngIf="thumbnail" [src]="thumbnail"/>
13       </md-card-title-group>
14       <md-card-content>
15         <p [innerHtml]="description"></p>
16       </md-card-content>
17       <md-card-footer class="footer">
18         <bc-book-authors [book]="book"></bc-book-authors>
19       </md-card-footer>
20       <md-card-actions align="start">
21         <button md-raised-button color="warn" *ngIf="inCollection" (click)="remove.emit(book)">
22         Remove Book from Collection
23         </button>
24
25         <button md-raised-button color="primary" *ngIf="!inCollection" (click)="add.emit(book)">
26         Add Book to Collection
27         </button>
28       </md-card-actions>
29     </md-card>
```

…

```
61  export class BookDetailComponent {
62    /**
63     * Presentational components receieve data through @Input() and communicate events
64     * through @Output() but generally maintain no internal state of their
65     * own. All decisions are delegated to 'container', or 'smart'
66     * components before data updates flow back down.
69     */
70    @Input() book: Book;
71    @Input() inCollection: boolean;
72    @Output() add = new EventEmitter<Book>();
73    @Output() remove = new EventEmitter<Book>();
74
```

Listing 30 Redux Implementierung Kind Komponente mit Event Emitter

3.4 Vergleichende Evaluierung der Umsetzung

Anzahl der Programmcode Zeilen

Eine sehr gut messbare und vergleichbare Metrik zweier Programmcode Implementierungen sind die Zeilen des Codes, die gebraucht wurden, um die jeweilige Beispielapplikation zu implementieren. Gleichermaßen möchte ich auch erwähnen, dass die Anzahl von Programmcodezeilen nicht zwingend Auskunft über die Programmcode Qualität oder Funktionalität gibt.

Die Messung wurde durchgeführt mit der NPM Bibliothek „sloc“ (https://www.npmjs.com/package/sloc). Zuerst wurde das Kommandozeilen Programm „sloc“ global installiert per

```
1   sudo npm install -g sloc
```

Listing 31 Installation von „sloc“

Nach der Installation kann „sloc“ auf den Quellcode Ordner wie folgt angewendet werden

```
1   sloc src/
```

Listing 32 Benutzung von "sloc"

Die Ergebnisse der Messung der beiden Implementierungen lauten wie folgt:

Die Messung der Programmcode Zeilen für die Implementierung eines Daten Service getriebenen Zustandsmanagement mit Angular der verwendeten Beispielapplikation hat ergeben, dass 981 Programmcodezeilen in 37 Dateien geschrieben wurden.

```
 master  /Library/WebServer/Documents/ngrx-example-app-dereduxed 
sloc src

---------- Result ------------

            Physical :  1246
              Source :  981
             Comment :  77
 Single-line comment :  52
       Block comment :  25
               Mixed :  5
               Empty :  193
               To Do :  0

Number of files read :  37

------------------------------
```

Abbildung 20 Messung der Programmcode Zeilen für die Implementierung eines Daten Service getriebenen Zustandsmanagement mit Angular

Die Messung der Programmcode Zeilen für die Implementierung eines Redux getriebenen Zustandsmanagement mit Angular der verwendeten Beispielapplikation hat ergeben, dass 1744 Programmcodezeilen in 50 Dateien geschrieben wurden.

```
 master  /Library/WebServer/Documents/ngrx-example-app 
sloc src

---------- Result ------------

            Physical :  2380
              Source :  1744
             Comment :  278
 Single-line comment :  21
       Block comment :  257
               Mixed :  6
               Empty :  364
               To Do :  0

Number of files read :  50

------------------------------
```

Abbildung 21 Messung der Programmcode Zeilen für die Implementierung eines Redux getriebenen Zustandsmanagement mit Angular

Tabelle 4 Vergleich der Programmcode Zeilen und Anzahl der Dateien beider Implementierungen

	Daten Service Implementierung	Redux Implementierung
Programmcode Zeilen	981	1744
Anzahl Dateien	37	50

Performancemessungen

Die folgenden Performance Messungen wurden durchgeführt auf einem MacBook Pro Anfang 2015, mit einem 3,1 GHz Intel Core i7 Prozessor und 16GB 1867 MHz DD3 Arbeitsspeicher. Der verwendete Browser ist Chrome in Version 57.

Zu Beginn wurde die Ladezeit beider Implementierungen gemessen und verglichen. Die Messung erfolgt dabei per „console.time()“ und „console.timeEnd()“ jeweils vor und nach der Einbindung des Programmcodes. Die Werte, die im Vergleich verwendet wurden, sind das Mittel aus fünf aufeinanderfolgenden Messwiederholungen.

Ergebnis:

Daten Service Implementierung: 715ms

Redux Implementierung: 756ms

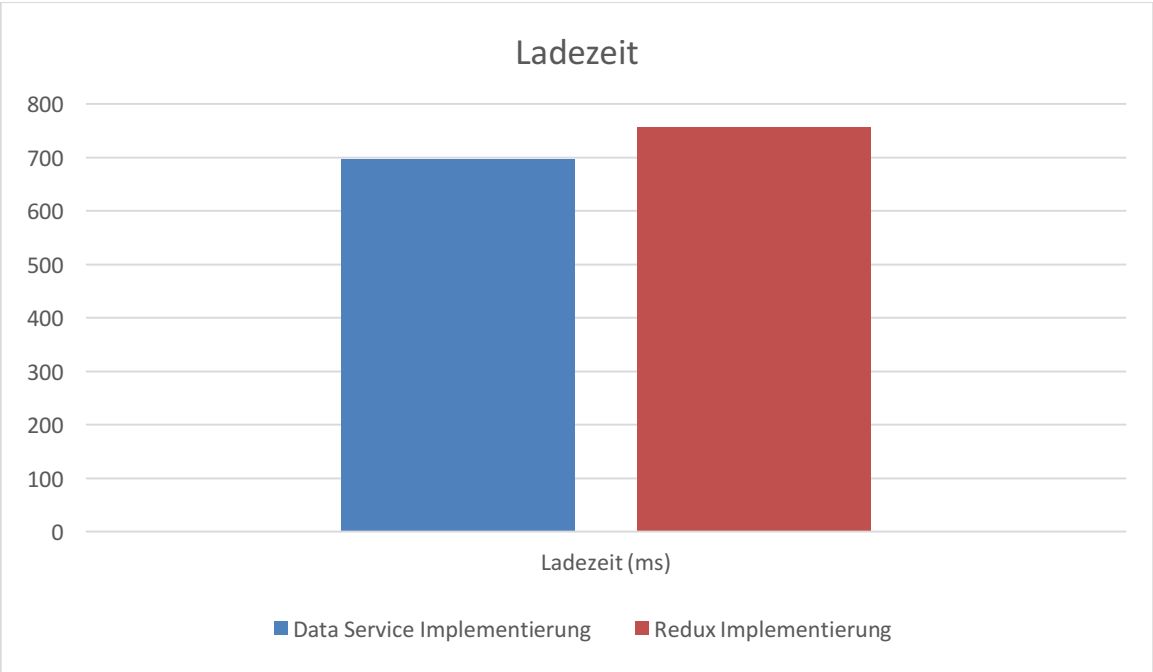


Abbildung 22 Vergleich der JavaScript Ladezeit beider Implementierungen

Bei diesem Vergleich ist zu erwähnen, dass es sich laut (Horie, 2017) um einen „Mikro Benchmark“ handelt, bei dem die absoluten Werte stark von der Hardware abhängen. Nichts desto trotz kann der relative Unterschied zwischen den Ladezeiten beider Programmcode Implementierungen aufschlussreich sein.

(Horie, 2017) nennt als sehr relevante Kennzahl für die Performance einer Single Page Application die Update Performance, da diese gegenüber der ersten Render Performance in einer langlebigen Applikation öfters zum Tragen kommt, nämlich immer dann, wenn sich der Applikationszustand ändert.

Für die Performance Messungen wurde in beiden Code Repositories einer neuer Git Branch erstellt, dieser findet sich unter dem Branch Bezeichner “performance”. Dort lässt sich die konkrete Implementierung der Performance Messung nachvollziehen, in der vorliegenden Masterarbeit werden das Vorgehen und die Ergebnisse dargestellt.

Ein nützliches Werkzeug um die Update Performance innerhalb einer Single Page Applikation zu messen, ist das vom Ember Team entwickelte „DbMonster“. Dieses Werkzeug aktualisiert eine Tabelle so schnell wie es möglich ist und misst die Frames per Second (FPS) und die folgenden JavaScript Timings: min, max und mean. Die ausschlaggebendste und im folgenden Fall betrachtete Zahl ist die durchschnittliche Renderzeit („mean“) (Horie, 2017).

Die Daten, die in den Beispiel Applikationen im Zustand gebunden sind, ergeben sich wie folgt. DbMonster erzeugt in den Performance Messungen eine Tabelle mit einer Anzahl konfigurierbarer Zeilen. Jeder dieser Zeilen enthält sieben Spalten. Jeweils zwei Tabellen Zeilen werden aus einem von DbMonster generierten Daten Objekt gerendert. Dieses Objekt hat sieben Eigenschaften, von denen eine wiederum ein Objekt enthält. Dieses Objekt hat vier Eigenschaften wovon zwei Eigenschaften Objekte sind, die insgesamt 17 Objekt Kinder haben. Diese Objekte haben wiederrum fünf Eigenschaften. Ein Beispiel eines einzelnen Objekts, welches von DbMonster generiert wurde, findet sich in Abbildung 23.

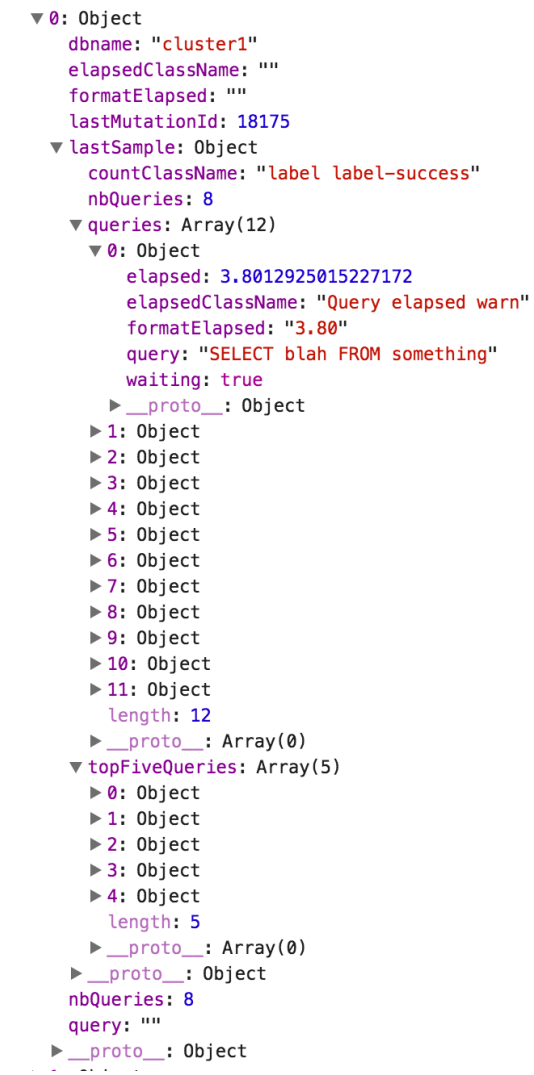


Abbildung 23 DbMonster Beispiel Objekt

Die Anzahl der gebundenen Daten innerhalb der jeweiligen Performance Test bestehen damit aus einem Vielfachen von 21 Objekten (1 + 1 + 2 + 17). Durch die Implementierung per „requestAnimationFrame“ ändert DbMonster alle Daten bis zu 60 Mal pro Sekunde, wenn eine Veränderungsrate von 100% eingestellt ist. Die Werte der Veränderungsrate finden sich in der Tabelle pro Performance Messung.

Die beiden Implementierungen der Beispiel Applikation werden während den Performance Messungen nicht gescrollt. Die durchschnittliche der Renderzeit wurde nach 15 Sekunden nach dem Neu Laden abgelesen.

Die Applikation wurde per „ng build --prod“ für eine Produktionsumgebung mit dem Standard Angular CLI Einstellungen vorbereitet.

Der Render-Profiler von „perf-monitor“ wird in der Methode „ngOnInit“ gestartet und in der Methode „ngAfterViewChecked“ gestoppt, beide gehören zur „AppComponent“ Klasse.

Die Angular Eigenschaft „changeDetection“ für die „AppComponent“ wurde in der Daten Service Implementierung auf dem Standardwert „Default“ belassen.

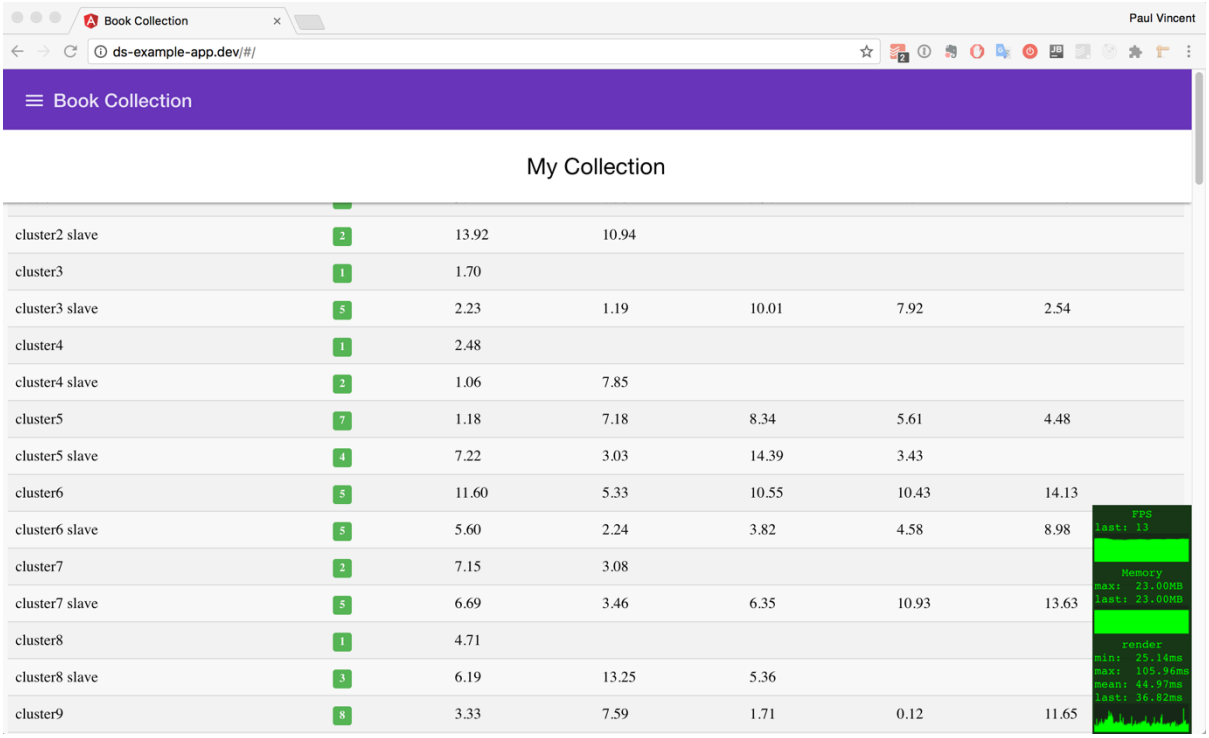


Abbildung 24 1. Performance Messung der Daten Service Implementierung (1050 Objekte, 50% Veränderungsrate)

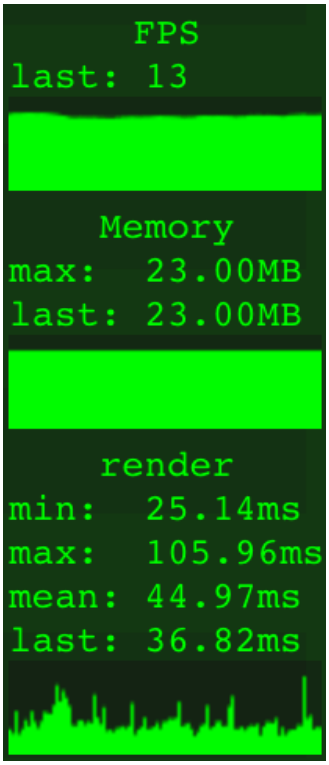


Abbildung 25 perf-monitor Ausgabe der Daten Service Implementierung (1050 Objekte, 50% Veränderungsrate)

Das zuvor geschriebene Vorgehen wurde gleichermaßen für die Redux Implementierung der Beispielapplikation umgesetzt und anschließend die Performance gemessen.

Die Angular Eigenschaft „changeDetection“ für die „AppComponent“ in der Redux Implementierung ist auf den Wert „OnPush“ eingestellt.

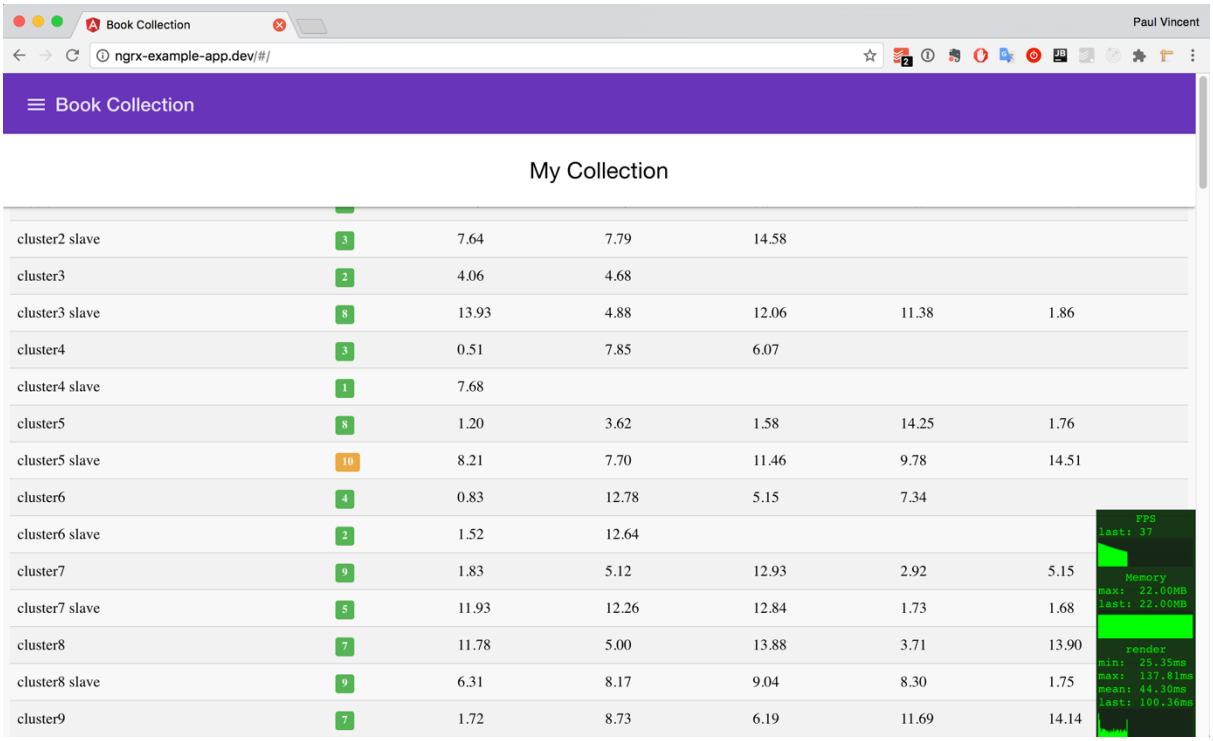


Abbildung 26 Performance Messung der Redux Implementierung (1050 Objekte, 50% Veränderungsrate)

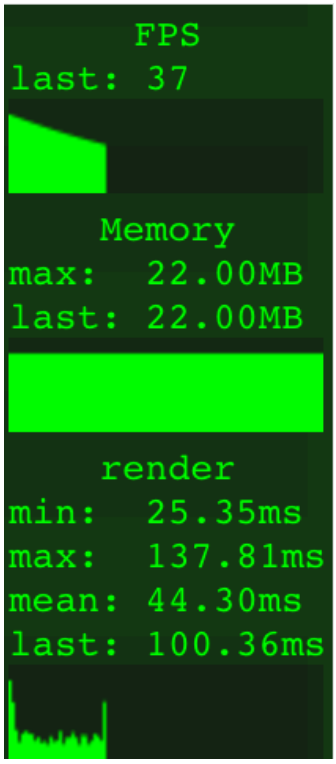


Abbildung 27 perf-monitor Ausgabe der Redux Implementierung (1050 Objekte, 50% Veränderungsrate)

Um einen besseren Überblick zu bekommen und auch die Messergebnisse im entsprechenden Kontext betrachten zu können, insbesondere über das Optimierungspotential von OnPush wurden Messungen mit variierenden Werten für die Anzahl der gebundenen Datenobjekte und auch der der Veränderungsrate durchgeführt. Es wurden Messungen für jeweils 210, 1050 und 3150 gebundene Objekte durchgeführt. Außerdem wurden für jede Objektanzahl Messungen bei drei verschiedenen Veränderungsraten durchgeführt. Es wurden Messungen mit 10%, 50% und 100% Veränderungsrate durchgeführt. 100% Veränderungsrate bedeutet, dass bei jedem Update, d.h. bis zu 60 Mal pro Sekunde, sich alle Daten verändern.

Tabelle 5 Vergleichende Performance Messung Durchschnittliche Renderzeit in verschiedenen Szenarien

Gebundene Objekte (DbMonster Zeilen)	**Veränderungs-rate**	**Daten Service Impl.: ⌀ Renderzeit (ms)**	**Redux Impl.: ⌀ Renderzeit (ms)**
210 (10)	10%	6,17	**3,46**
1050 (50)	10%	**12,27**	13,25
3150 (150)	10%	**27,82**	41,59
210 (10)	50%	13,34	**10,08**
1050 (50)	50%	44,97	**44,30**
3150 (150)	50%	**111,94**	113,15
210 (10)	100%	20,13	**18,12**
1050 (50)	100%	**70,31**	86,54
3150 (150)	100%	**190,66**	252,45

In Tabelle 5 sind die Ergebnisse der insgesamt 18 Messungen zu finden. Die jeweils geringere, durchschnittliche Renderzeit für jedes Szenario ist fett markiert. Insgesamt gab es 9 Szenarien, bei denen

- die Daten Service Implementierung in fünf Messszenarien eine geringere, durchschnittliche Renderzeit als die Redux Implementierung aufwies
- die Redux Implementierung in vier Messszenarien eine geringere, durchschnittliche Renderzeit als die Daten Service Implementierung aufwies.

Zusammenfassen lässt sich, dass bei einer geringen bis mittleren Datenanzahl und geringer bis mittlerer Veränderung dieser Daten die Redux Implementierung schneller bzw. ähnlich schnell wie die Daten Service Implementierung rendert. Bei einer hohen Veränderungsrate rendert die

Redux Implementierung für eine geringe Anzahl an gebundenen Daten schneller als die Daten Service Implementierung.

Die Messergebnisse zeigen, dass bei einer hohen Anzahl an gebundenen Daten die Daten Service Implementierung bei allen drei Veränderungsraten im Mittel schneller rendert als die Redux Implementierung.

Die aktuelle Implementierung von DbMonster hilft dabei, eine beliebige Anzahl von Daten zu generieren und diese, bei Bedarf, sehr schnell zu verändern und damit die Renderzeit einer Implementierung zu testen. Durch die Verwendung von „requestAnimationFrame“ kann eine Änderung der Daten bis zu 60 Mal pro Sekunde passieren, in Abhängigkeit von der eingestellten Veränderungsrate von DbMonster. Eine solche Aktualisierungsrate wird im Normalfall für Animationen (Irish, 2011) verwendet.

Die Umsetzung der Performance Messung der vorliegenden Arbeit nimmt an, dass bei einer Änderungsrate von 50% und 100% sich fast die Hälfte bzw. fast alle Daten innerhalb der JavaScript Web Applikation ändern, was nach Erfahrung des Autors wiederum ein eher spezieller Anwendungsfall ist. Je nach Anwendungsfall könne sich deutlich weniger Daten innerhalb der Applikation ändern. Um ein möglichst breites Spektrum an möglichen Fällen abzudecken wurden die vorgestellten neun Szenarien gemessen.

Weiterhin ist hier zu beachten, dass als Vertreter für das Redux Paradigma die Implementierung von @ngrx/store anhand von Observables gewählt wurde. Da das Redux Paradigma keiner spezifischen Implementierung anhängt, sind andere Messergebnisse in weiteren Implementierungen durchaus denkbar.

Befragung von Entwicklern

Um weitere Daten für den Vergleich der beiden Zustandsmanagement Implementierungen zu erheben, wurde eine Entwickler Umfrage mit den folgenden Fragen und Antworten erstellt.

Die komplette Umfrage findet sich im Anhang als PDF. Insgesamt haben 15 JavaScript Entwickler an der Umfrage im Zeitraum vom 03.04.2017 – 09.04.2017 teilgenommen und jeweils 12 Fragen beantwortet. Das Setting der Umfrage gestaltete sich wie folgt. Die Umfrage wurde in Form eines online Fragebogens an die Entwickler geschickt. Die Einführung direkt zu Beginn des Fragebogens enthielt lediglich Links zur Demo der Beispiel Applikation und zu den Git Repositories beider Beispiel Implementierungen. Es wurde bewusst darauf verzichtet, genau zu benennen, in welchen Dateien der Zustand in der jeweiligen Implementation verwaltet wird. Auch wurde bewusst darauf verzichtet, weitere Architekturdokumentation in Form von Diagrammen den beiden Implementationen für die Umfrage anzuhängen. Das möglichst neutrale Ausgangssetting sollte den Vergleich beider Implementierungen bestmöglich unterstützen. Außerdem haben die befragten Entwickler mit der Einladung zur Umfrage Informationen über das Thema und das Ziel der Masterarbeit bekommen.

Die Eingangsfragen 1 bis 5 helfen bei der Interpretation und Einschätzung der Antworten, da die Antworten im Zusammenhang mit der Erfahrung eines jeden Entwicklers stehen.

Frage 1: Seit wie vielen Jahren entwickeln Sie schon JavaScript Code? (15 Antworten)

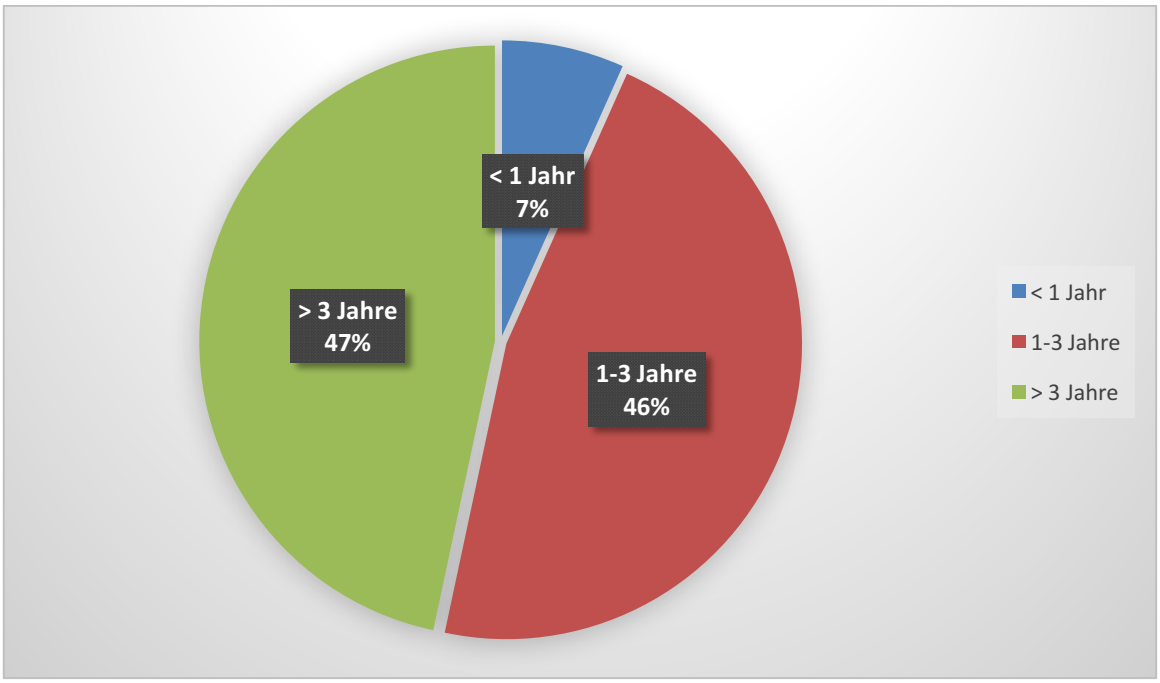


Abbildung 28 Auswertung Entwickler Umfrage Frage 1

Frage 2: Haben Sie bereits Angular Erfahrung? (15 Antworten)

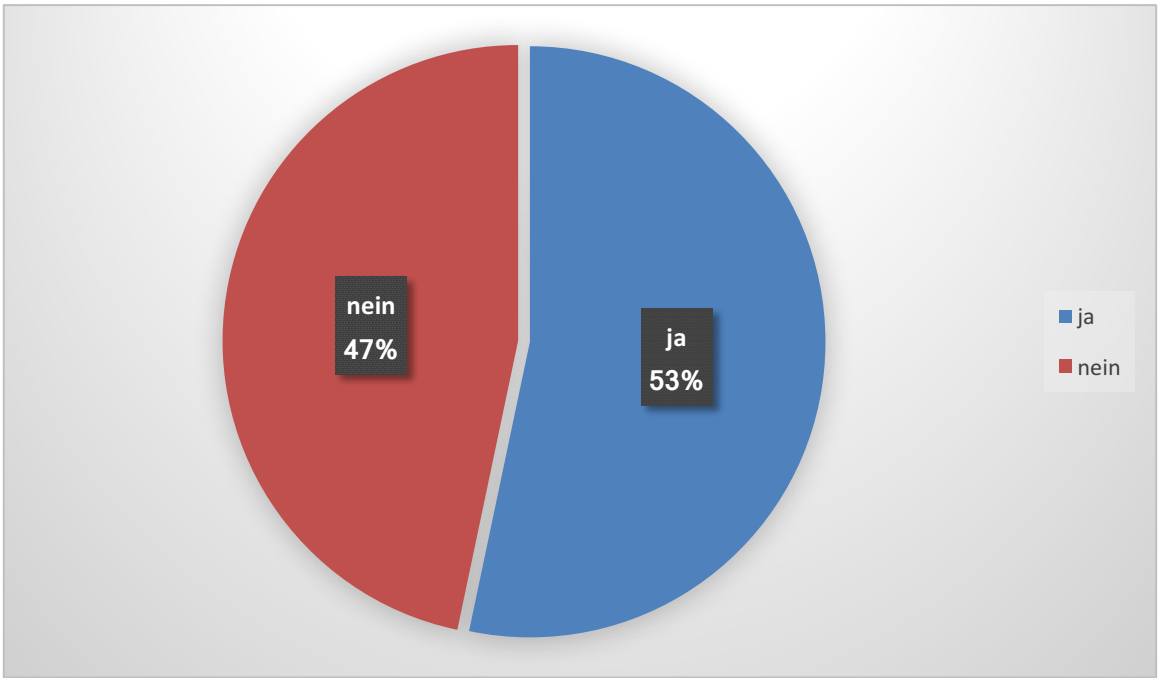


Abbildung 29 Auswertung Entwickler Umfrage Frage 2

Frage 3: Haben Sie bereits Erfahrung mit Redux basiertem Zustandsmanagement? (15 Antworten)

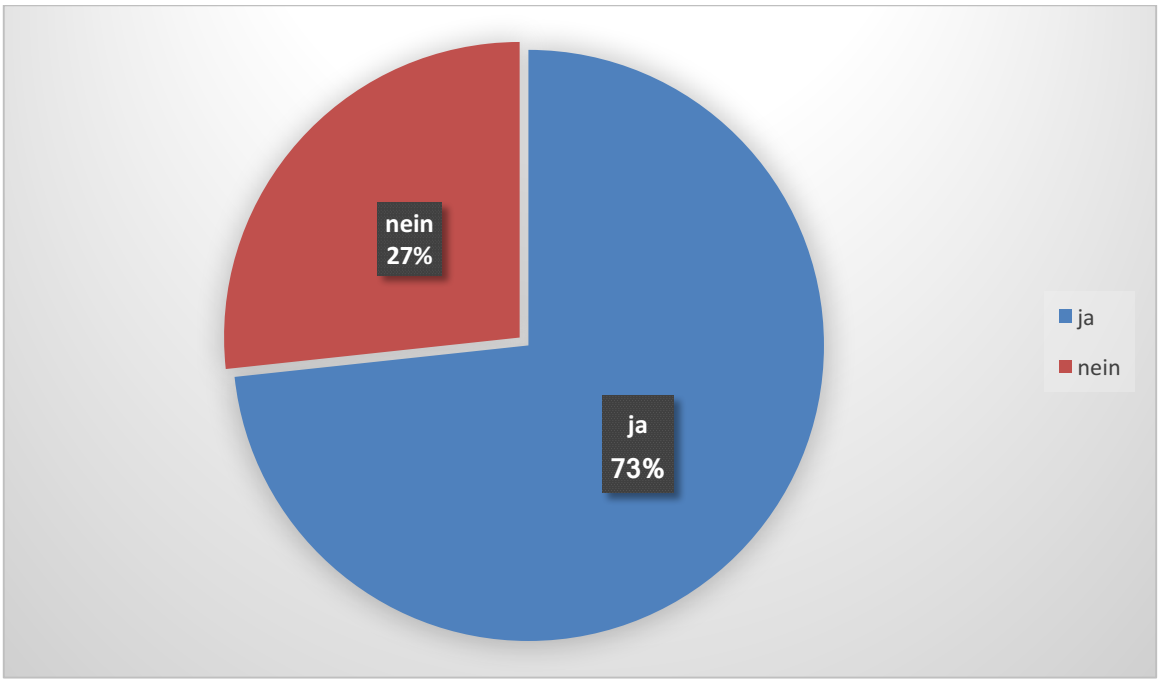


Abbildung 30 Auswertung Entwickler Umfrage Frage 3

Frage 4: Haben Sie bereits TypeScript Erfahrung? (15 Antworten)

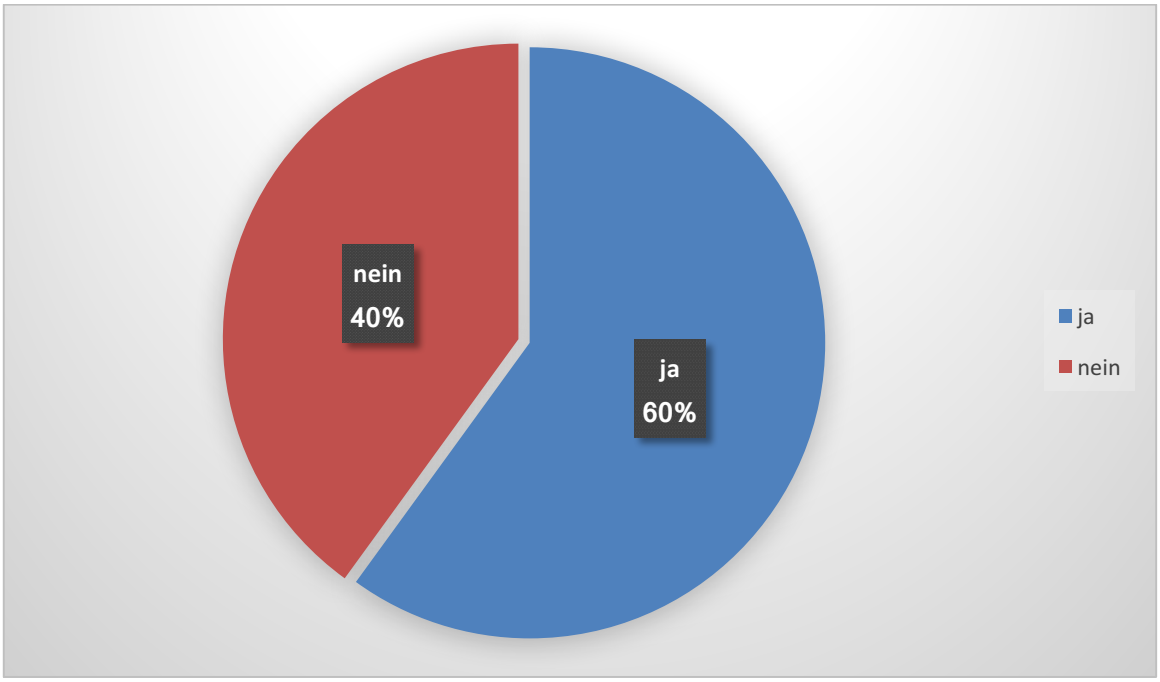


Abbildung 31 Auswertung Entwickler Umfrage Frage 4

Frage 5: Haben Sie bereits RxJS Erfahrung? (15 Antworten)

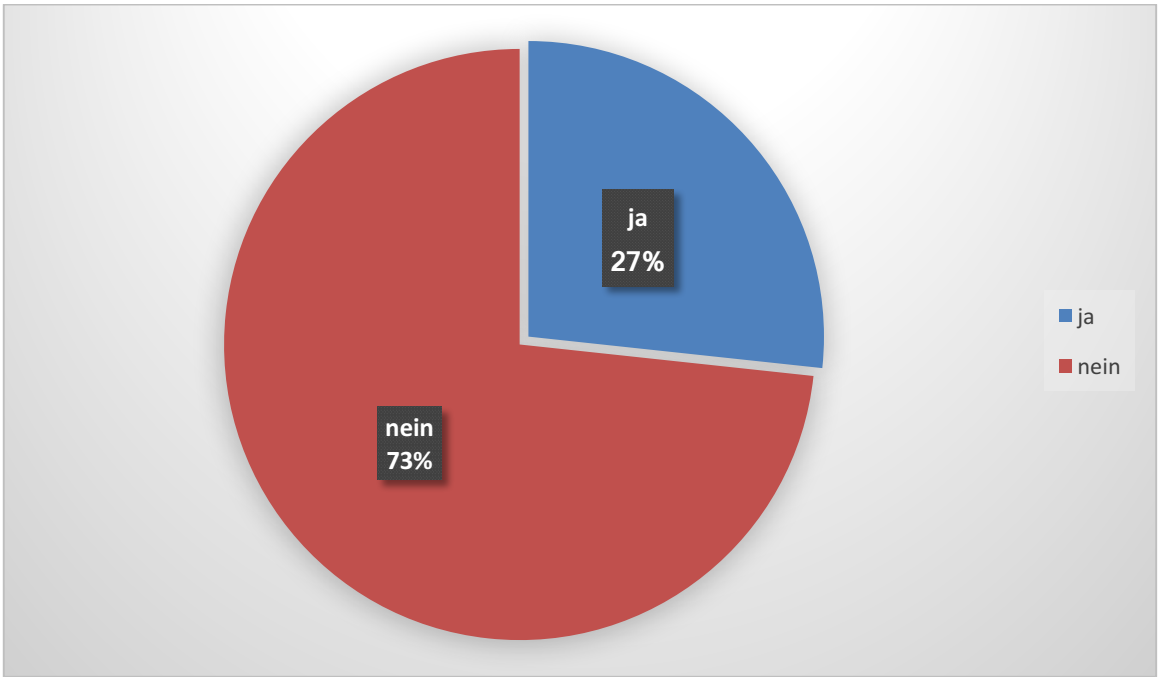


Abbildung 32 Auswertung Entwickler Umfrage Frage 5

Frage 6: In Bezug auf die Daten Service Implementierung: Stellen Sie sich vor, Sie müssen eine neue Anforderung entwickeln, die den Applikationszustand beeinflusst. Wie schätzen Sie den Entwicklungsaufwand aufgrund der derzeitigen Zustandsmanagement Architektur ein? (15 Antworten)

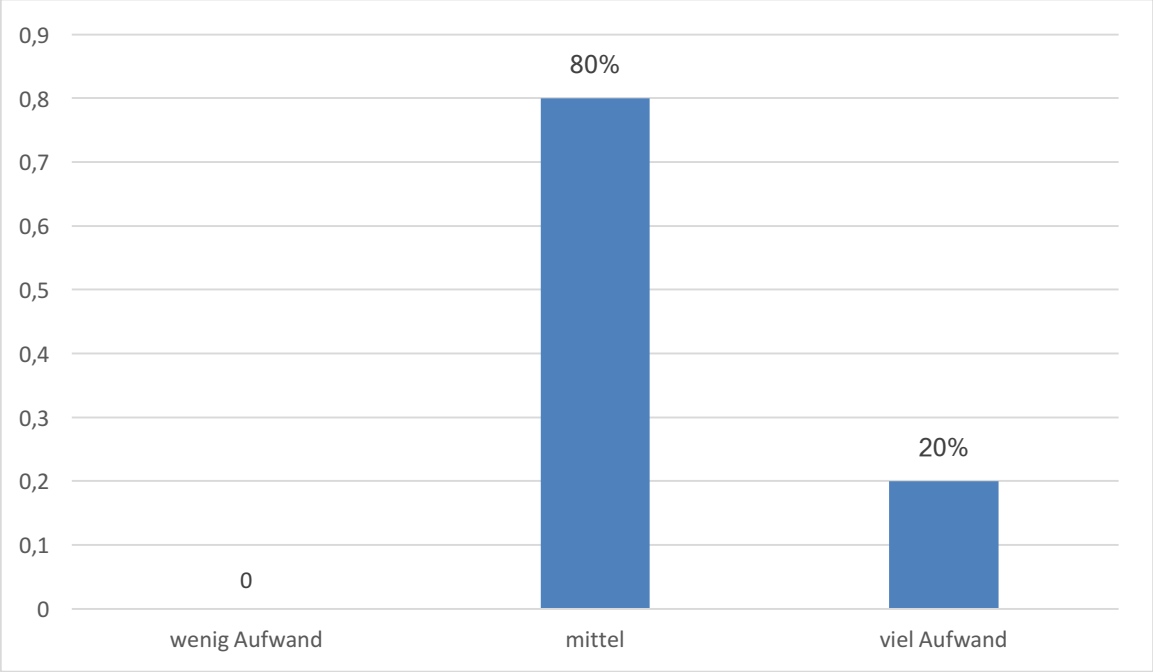


Abbildung 33 Auswertung Entwickler Umfrage Frage 6

Frage 7: In Bezug auf die Redux Implementierung: Stellen Sie sich vor, Sie müssen eine neue Anforderung entwickeln, die den Applikationszustand beeinflusst. Wie schätzen Sie den Entwicklungsaufwand aufgrund der derzeitigen Zustandsmanagement Architektur ein? (15 Antworten)

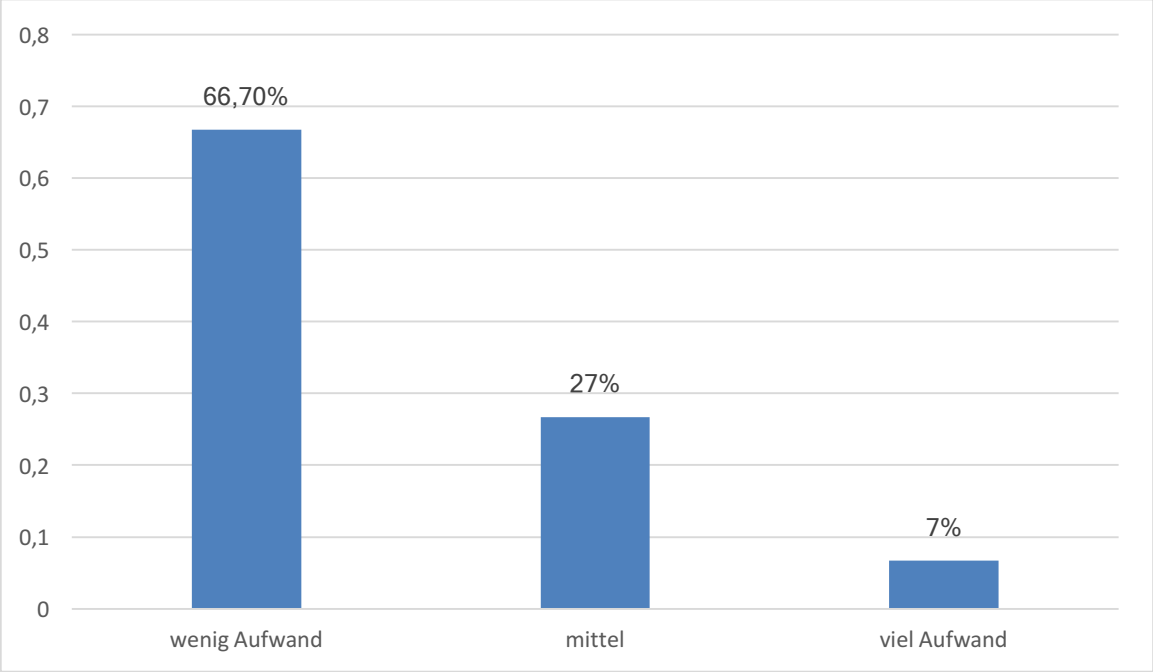


Abbildung 34 Auswertung Entwickler Umfrage Frage 7

Frage 8: In Bezug auf die Daten Service Implementierung: Stellen Sie sich vor, Sie müssen die Logik zum Speichern der Büchersammlung so ändern, dass auf einem Server statt lokal gespeichert wird. Wie schätzen Sie den Entwicklungsaufwand aufgrund der derzeitigen Zustandsmanagement Architektur ein? (15 Antworten)

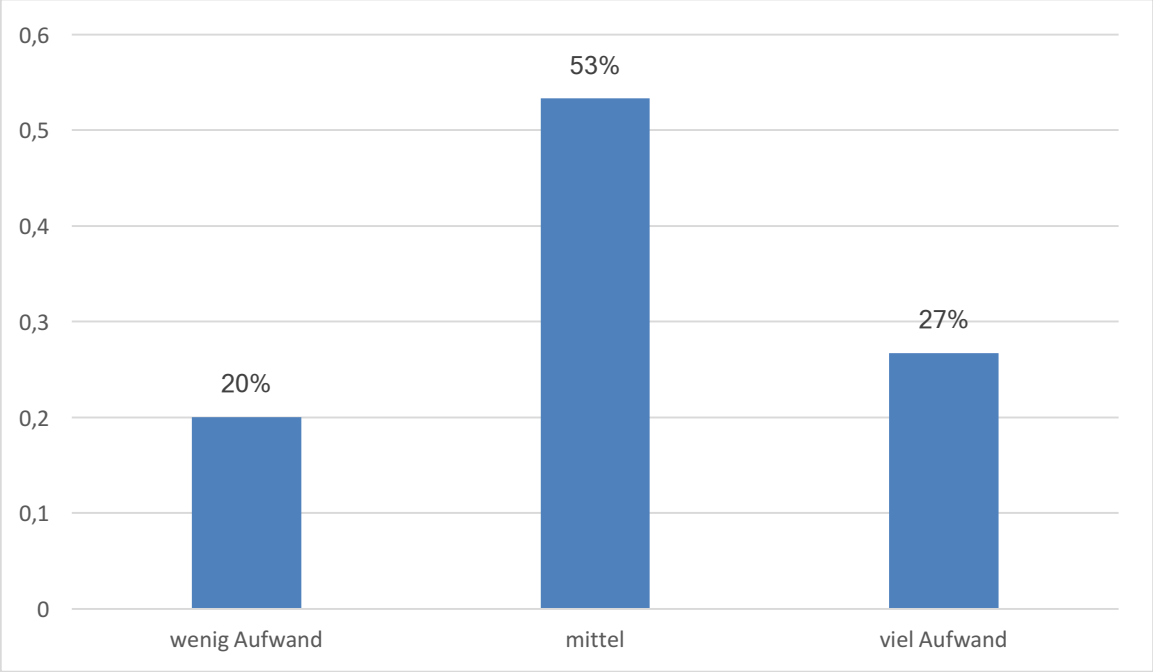


Abbildung 35 Auswertung Entwickler Umfrage Frage 8

Frage 9: In Bezug auf die Redux Implementierung: Stellen Sie sich vor, Sie müssen die Logik zum Speichern der Büchersammlung so ändern, dass auf einem Server statt lokal gespeichert wird. Wie schätzen Sie den Entwicklungsaufwand aufgrund der derzeitigen Zustandsmanagement Architektur ein? (15 Antworten)

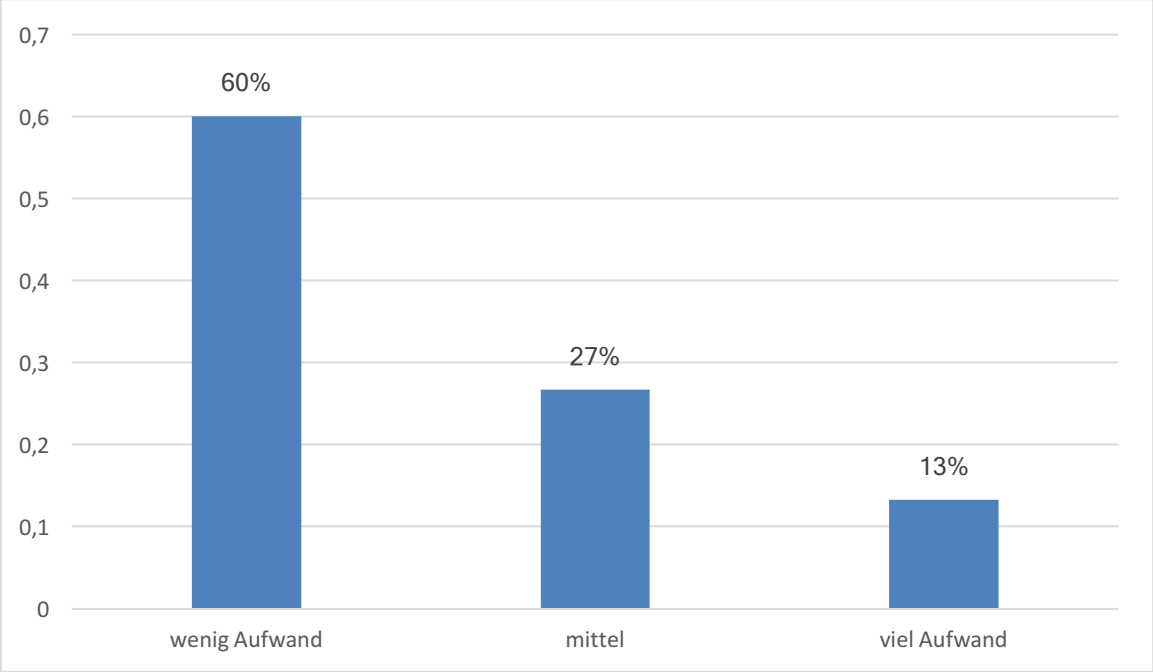


Abbildung 36 Auswertung Entwickler Umfrage Frage 9

Frage 10: In Bezug auf die Daten Service Implementierung: Stellen Sie sich vor, Sie müssen einen Unit Test für einen Teil des Applikationszustands entwickeln. Wie schätzen Sie den Entwicklungsaufwand aufgrund der derzeitigen Zustandsmanagement Architektur ein? (15 Antworten)

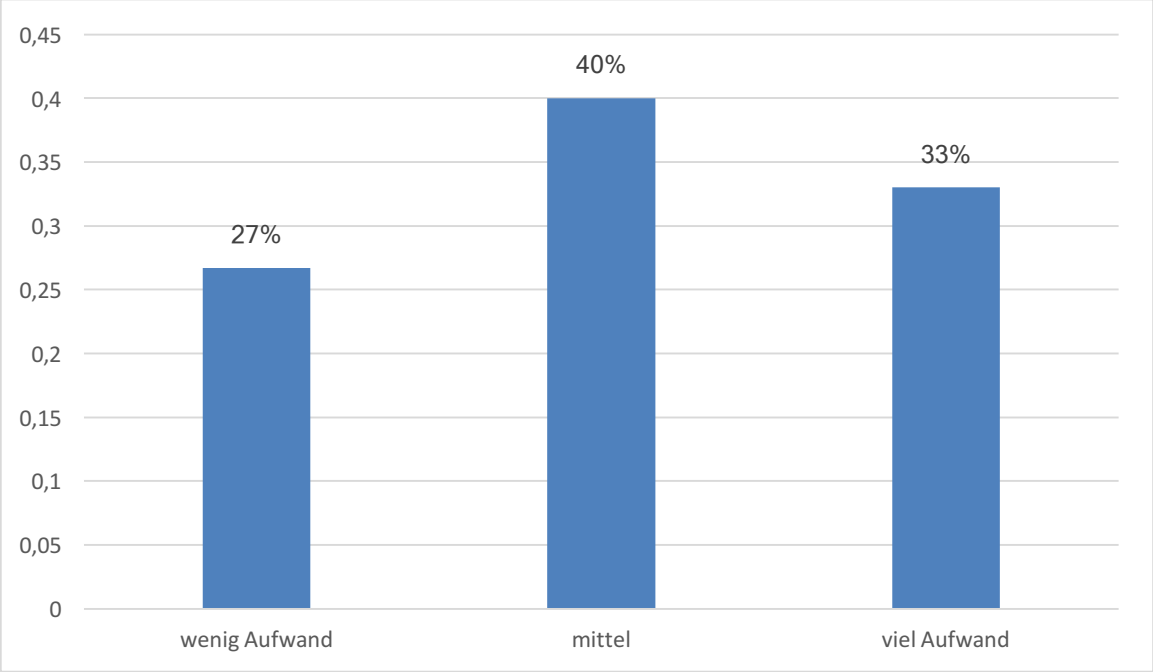


Abbildung 37 Auswertung Entwickler Umfrage Frage 10

Frage 11: In Bezug auf die Redux Implementierung: Stellen Sie sich vor, Sie müssen einen Unit Test für einen Teil des Applikationszustands entwickeln. Wie schätzen Sie den Entwicklungsaufwand aufgrund der derzeitigen Zustandsmanagement Architektur ein? (15 Antworten)

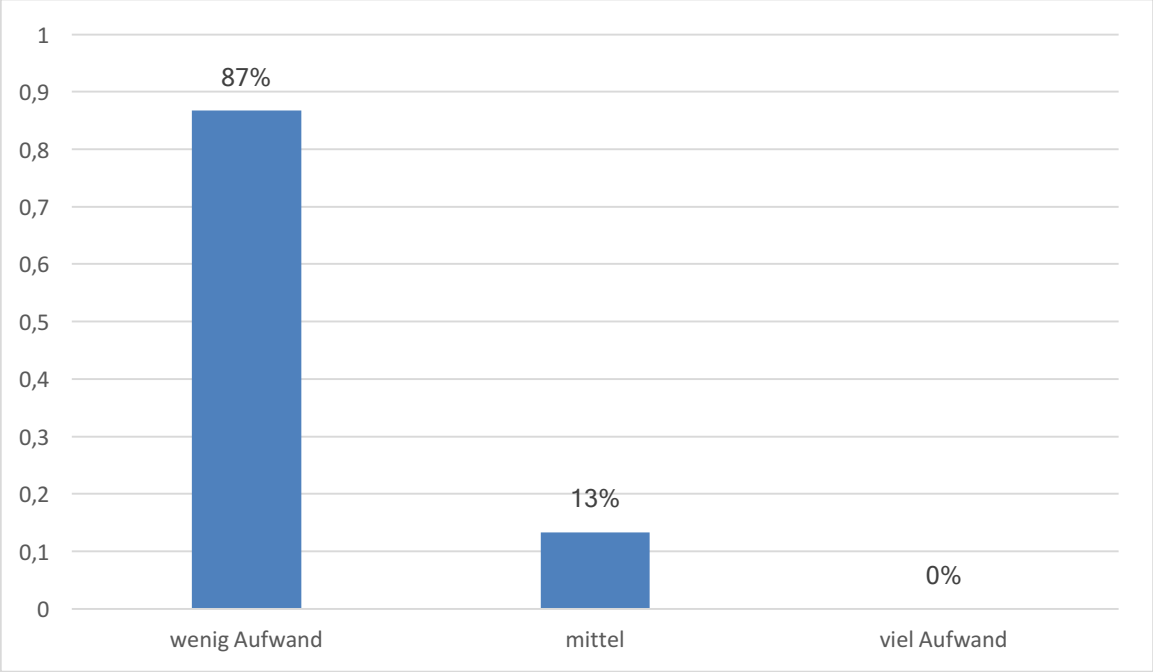


Abbildung 38 Auswertung Entwickler Umfrage Frage 11

Frage 12: Welche der beiden Implementierungen ist selbsterklärender? (15 Antworten)

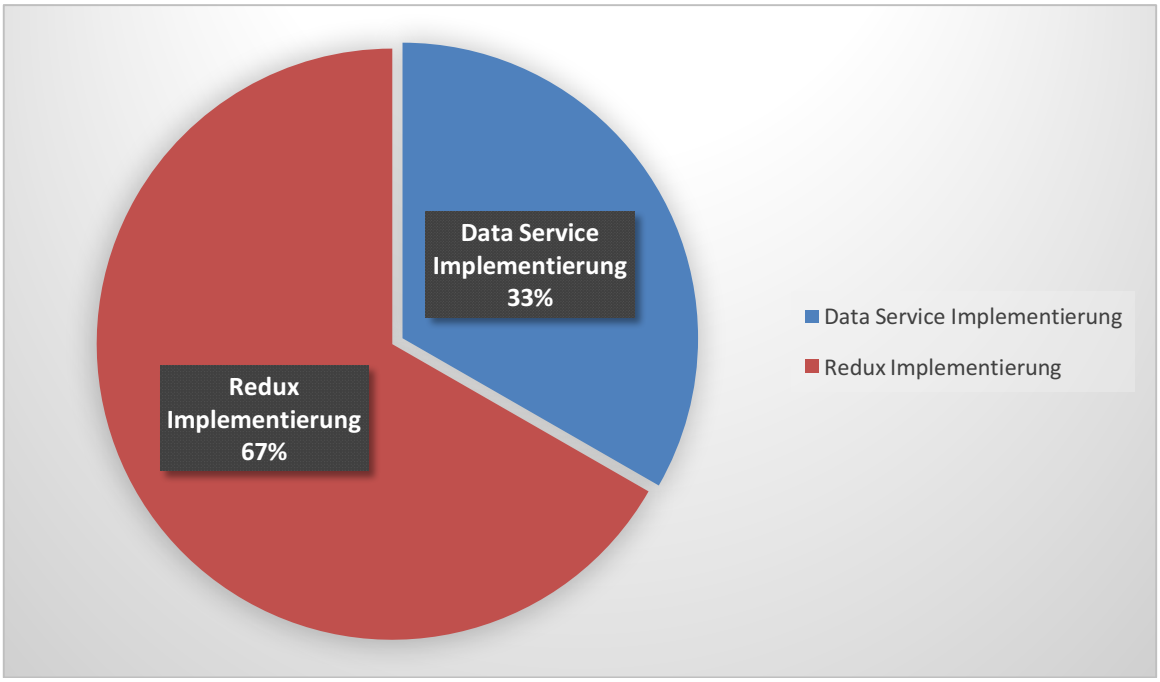


Abbildung 39 Auswertung Entwickler Umfrage Frage 12

Es folgt eine Aufschlüsslung der in Untergruppen „bisherige Redux Erfahrung“ der befragten Entwickler.

75% der Entwickler ohne vorige Redux Erfahrung nehmen an, dass die Daten Service Implementierung selbst erklärender ist.

82% der Entwickler mit voriger Redux Erfahrung nehmen an, dass die Redux Implementierung selbst erklärender ist.

Im gerundeten Durchschnitt schätzen die Entwickler ohne vorige Redux Erfahrung für die Fragen 6,8 und 10 (Daten Service Implementierung) einen mittleren Aufwand. Im gerundeten Durchschnitt schätzen die Entwickler ohne vorige Redux Erfahrung für die Fragen 7 und 9 und (Redux Implementierung) einen mittleren Aufwand, für die Frage 11 (Redux Implementierung) einen geringen Aufwand.

Im gerundeten Durchschnitt schätzen die Entwickler mit voriger Redux Erfahrung für die Fragen 6,8 und 10 (Daten Service Implementierung) einen mittleren Aufwand. Im gerundeten Durchschnitt schätzen die Entwickler mit voriger Redux Erfahrung für die Fragen 7,9 und 11 (Redux Implementierung) einen geringen Aufwand.

Evaluierung der Verbreitung des Redux Paradigmas

Ein weiterer Indikator für den qualitativen Vergleich ist die Verbreitung und Popularität des Redux Paradigmas. Über die Entwicklung der Verbreitung werde ich aufzeigen, dass der Ansatz auf großes Interesse und offene Ohren innerhalb der weltweiten Entwicklergemeinschaft gestoßen ist. In diesem Abschnitt werden folgende Kennzahlen betrachtet:

- Google Trends Entwicklung
- Anzahl GitHub Stars
- Anzahl Git Commits im Redux Repository
- Anzahl Stack Overflow Fragen

In Google Trends wurde der Suchbegriff „Redux JS“ verwendet, da der Suchbegriff „Redux “zu viele nicht relevante Ergebnisse assoziiert.

Das Diagramm aus Google Trends zeigt das „Interesse im zeitlichen Verlauf“, welches folgendermaßen definiert ist:

> *„Die Werte geben das Suchinteresse relativ zum höchsten Punkt im Diagramm für die ausgewählte Region im festgelegten Zeitraum an. Der Wert 100 steht für die höchste Beliebtheit dieses Suchbegriffs. Der Wert 50 bedeutet, dass der Begriff halb so beliebt war und der Wert 0 entspricht einer Beliebtheit von weniger als 1% im Vergleich zum Höchstwert.“ (Google, Google Trends, 2017)*

Die folgende Abbildung 40 zeigt deutlich, dass seit dem ersten Commit im April 2015 das Interesse am Redux Paradigma stetig zunimmt:

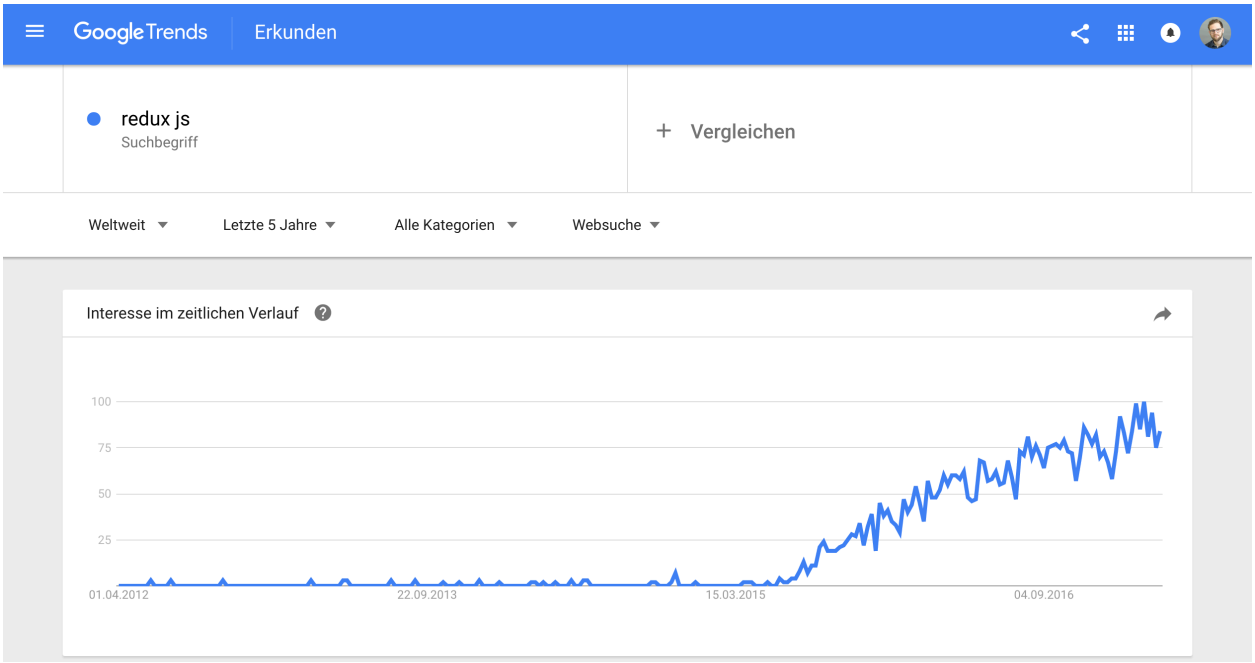


Abbildung 40 Google Trends Entwicklung für den Suchbegriff "redux js" (Google, Google Trends, 2017)

Um neben der Entwicklung der Popularität des Begriffs „Redux JS“ auch vergleichend zum Thema „JavaScript MVC“ einen Trend ablesen zu können, folgt in Abbildung 41 der direkte Vergleich zwischen den beiden Suchbegriffen.

Abbildung 41 Google Trends Vergleich der Entwicklung der Suchbegriffe "javascript mvc" und "redux js" (Google, Google Trends, 2017)

Natürlich kann der Vergleich der Trends für die beiden Suchbegriffe „Redux JS“ und „JavaScript MVC“ nur unter Einschränkung vorgenommen werden, da eine moderne JavaScript-MV* Architektur das Thema Zustandsmanagement mit abdeckt, so dass hier genau genommen zwei Begriffe auf unterschiedlichen Ebenen verglichen werden. Nichtsdestotrotz zeigt dieser Vergleich, wie stark das Redux Paradigma an Popularität gewonnen hat, während das Thema JavaScript MV* sich relativ gleichbleibend fortentwickelt hat.

(Swanepoel, 2013) beschreibt die Anzahl von **GitHub Stars** und die Anzahl, Häufigkeit und Alter von Commits auf GitHub als einen geeigneten Indikator, um die Popularität eines Frameworks oder in diesem Falle eines Paradigmas messbar machen zu können.

Am 29.03.2017 war das Redux Repository (reactjs/redux) mit 29.504 GitHub Stars das 21 beliebteste JavaScript Repository auf GitHub. Insgesamt gab es zu diesem Zeitpunkt 368.701 JavaScript Repositories auf GitHub.

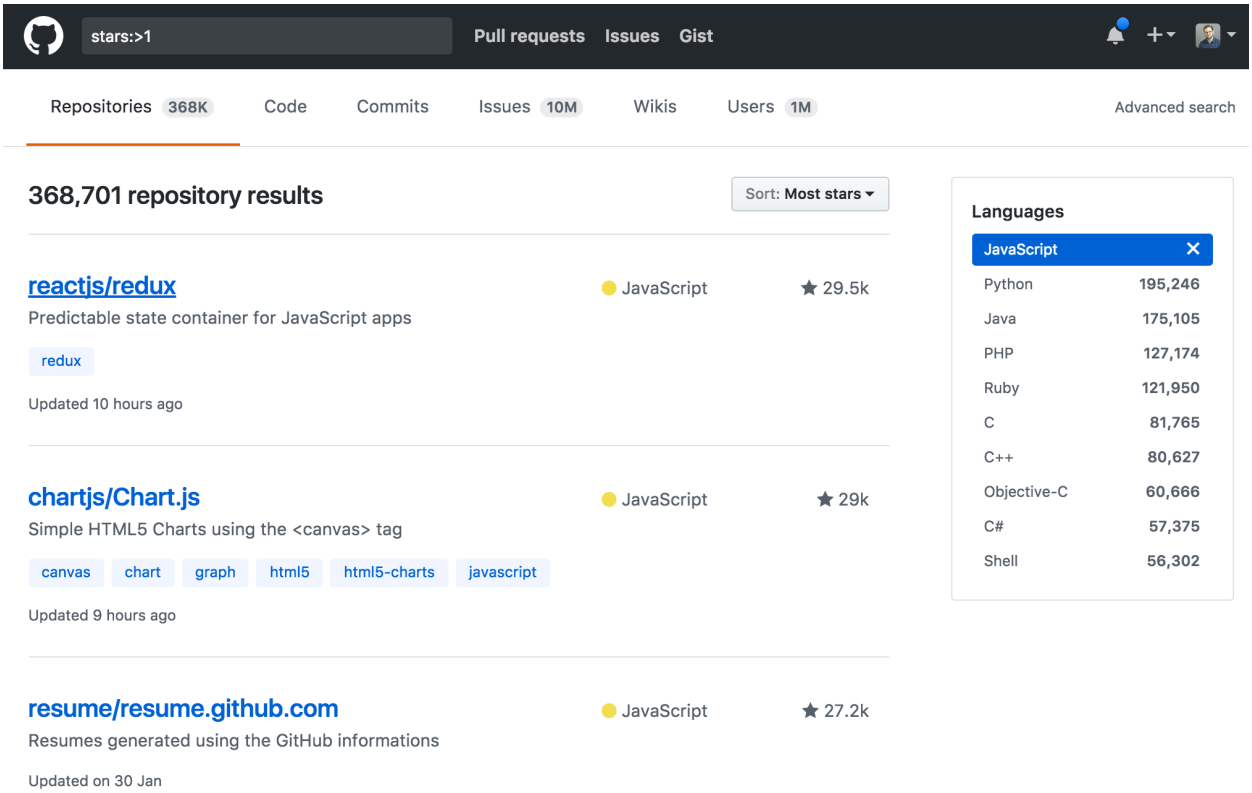


Abbildung 42 Redux als in den TOP 30 JavaScript Repositories auf (GitHub, 2017)

Die Anzahl der Commit auf GitHub im „reactjs/redux“ Repository lässt sich im zeitlichen Verlauf wie folgt darstellen:

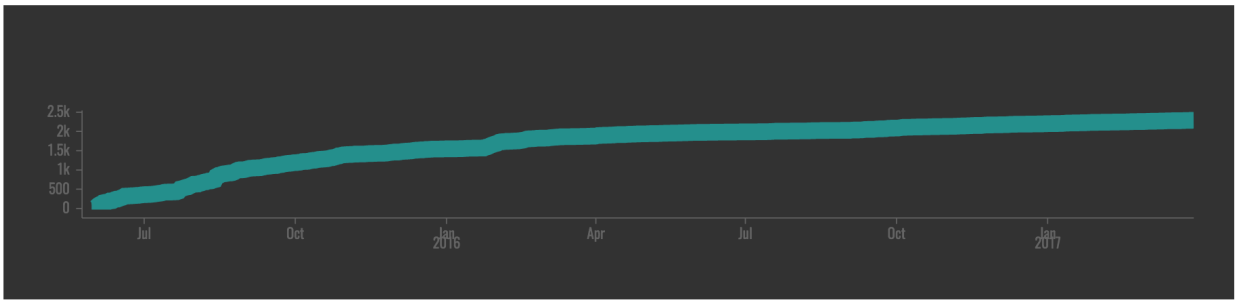


Abbildung 43 Anzahl der Commits im Redux Repository im zeitlichen Verlauf (GitHub Stats, 2017)

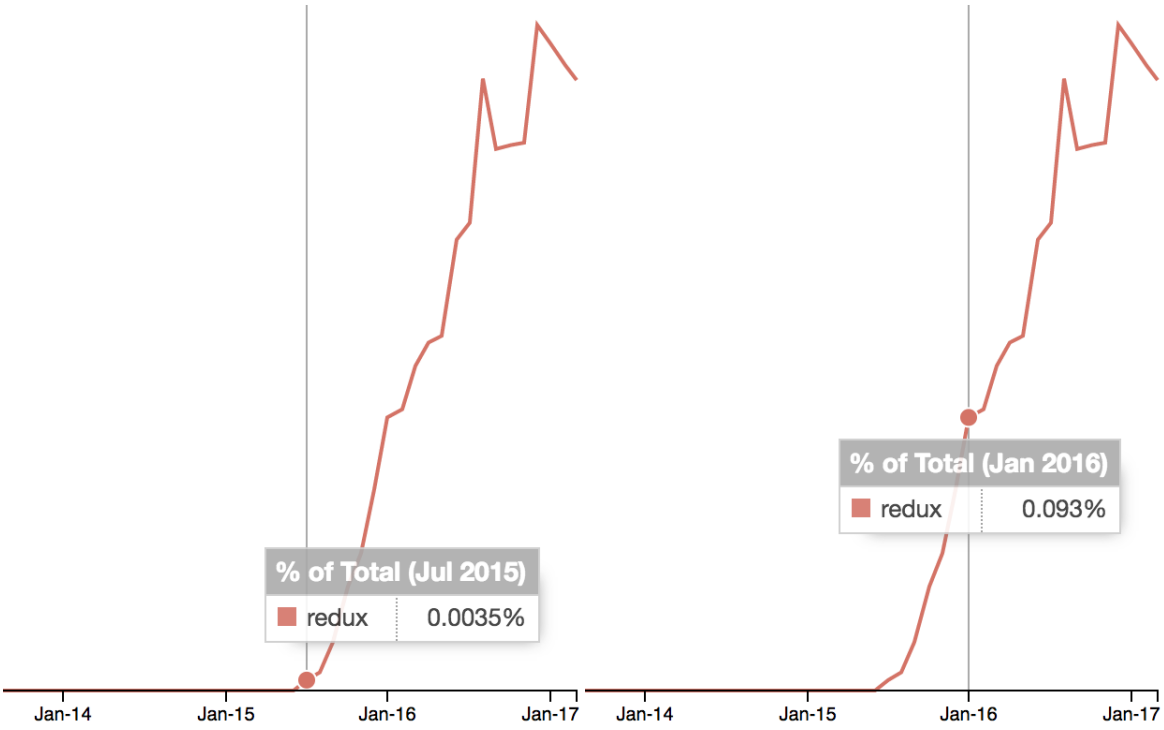
Um die konkreten Werte einsehen zu können folgt eine tabellarische Darstellung an ausgewählten Stichdaten.

Tabelle 6 Anzahl der Commits im Redux Repository, ausgewählte Stichdaten

Zeitpunkt	Anzahl Commits
30.05.2015	6
01.07.2015	359
06.07.2016	1989
30.12. 2016	2200

Um die Entwicklung innerhalb der Entwicklergemeinde zu untersuchen wurde mit Hilfe von der Website http://sotagtrends.com erfasst, wie viele Fragen auf Stack Overflow mit dem Tag „Redux“ pro Monat in Bezug auf die Gesamtheit aller Fragen auf Stack Overflow gestellt wurden. Die Daten, die sotagtrends.com auswertet, werden aus der Stack Exchange Netzwerk API bezogen.

Der prozentuale Anteil der Fragen im Verlauf der Zeit stellt sich wie folgt dar:



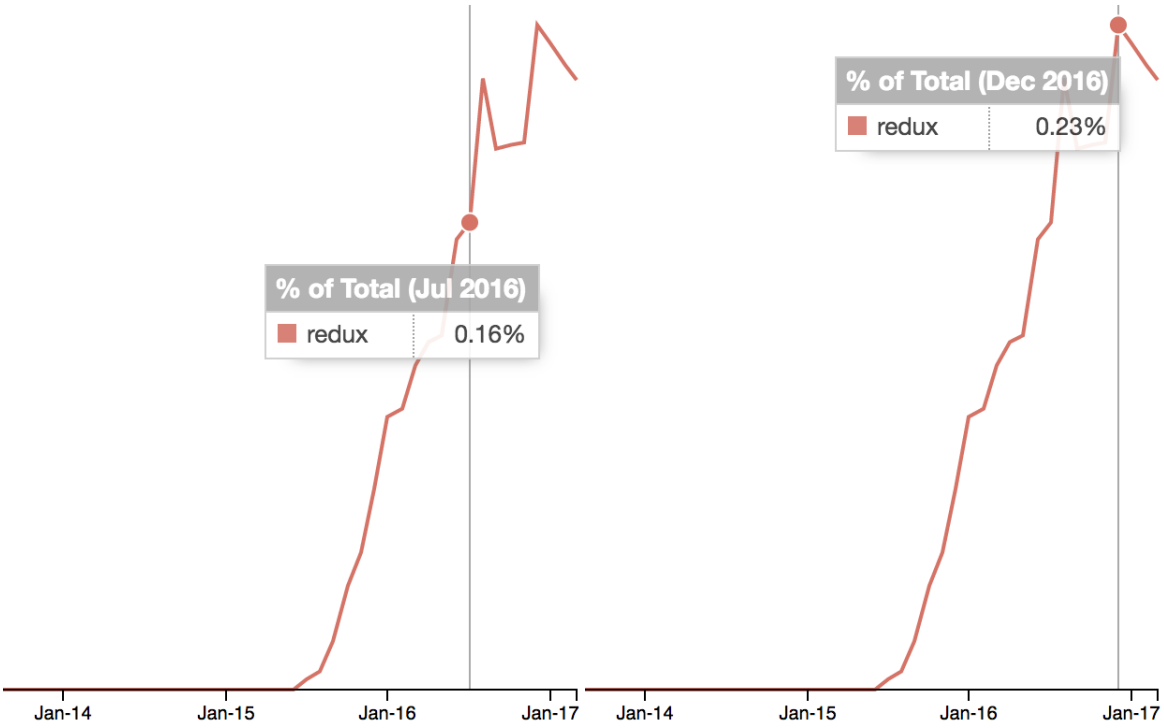


Abbildung 44 Prozentualer Anteil von Redux Fragen auf Stack Overflow ab 2015 (Diagramm) (sotagtrends.com, 2017)

Tabelle 7 Prozentualer Anteil von Redux Fragen auf Stack Overflow ab 2015 (Tabelle)

Zeitpunkt	% Anteil
Juli 2015	0,0035
Januar 2016	0,093
Juli 2016	0,16
Dezember 2016	0,23

Bezüglich des „Thinking Aloud“ bei der Konfrontation von Dritten mit beiden Ansätzen lässt sich insbesondere feststellen, dass die Themen Nachvollziehbarkeit, Plattform- bzw. Kontextunabhängigkeit und die gute Testbarkeit der Redux Implementierung benannt werden.

3.5 Vorteile der Implementierung des Redux getriebenen Zustandsmanagement

The Dao of Immutability

The true constant is change.
Mutation hides change.
Hidden changes manifests chaos.
Therefore, the wise embrace history.
(Elliot, Webcast – The Two Pillars of JS: Introduction to Functional Programming, 2015)

In diesem Kapitel werden die Vorteile, welche sich durch die Implementierung eines Zustandsmanagements anhand des Redux Paradigmas ergeben, beschrieben. Es wurden folgende Kategorien auf Vorteile untersucht:

- Vorhersehbarkeit
- Entwickler Werkzeuge (Debugging)
- Testbarkeit
- Performance
- Implementierung von fortgeschrittener Funktionalität
- Plattformunabhängigkeit
- Lebendige und unterstützende Community

Die zentralsten Vorteile der Redux Implementierung sind die genormte Sprache und die vorgegebenen Regeln, die eingehalten werden, um innerhalb eines Zustandsmanagements, welches das Redux Paradigma implementiert, mit dem Zustand zu interagieren. Jeder beteiligte Entwickler, der das Redux Paradigma kennt, weiß, dass der Zustand an genauer einer Stelle festgehalten wird (Paul & Nalwaya, React Native for iOS Development, 2016, S. 2) und befolgt die vorgegebenen Regeln, um diesen Zustand zu verändern. Diese genormte Sprache führt zur verbesserten Vorhersehbarkeit von Programmcode und einer erhöhten Datenkonsistenz (Akshat & Nalwaya, 2016). Auch die Wartbarkeit wird anhand der eingehaltenen Norm erhöht und es vereinfacht das Wachstum des Programmcode Umfangs (de Sousa Antonio, Pro React, 2015, S. 170). Man bezeichnet den Applikationszustand innerhalb einer Redux Implementierung als deterministisch (Lumpe, Purkhardt, Muller, Cravero, & Chentnik, Developing a Redux Edge, 2016, S. 16). Diese Ansicht kann auch mit den Ergebnissen der Umfrage aus Kapitel 3.4 gestützt werden, in der die befragten Entwickler angaben, dass sie den Aufwand für die Neu Entwicklung und Änderung von den Applikation Zustand betreffender Funktionalität durchweg in der Redux Implementierung für geringer als in der Daten Service Implementierung einschätzen. Des

Weiteren zeigt die Antwort auf die Frage 12 nach dem „Prinzip der geringsten Überraschung“ eine klare Richtung. Zwei Drittel der befragten Entwickler halten die Redux Implementierung für selbsterklärender.

Ein weiterer Vorteil bei der Entwicklung anhand des Redux Paradigmas sind die zur Verfügung stehenden Entwickler Werkzeuge. Innerhalb der Beispielapplikation konnte das Paket „@ngrx/store-devtools“ verwendet werden um mit der passenden Browser Erweiterung eine Visualisierung und Bearbeitung des Applikationszustands mit Hilfe einer komfortablen, grafischen UI zur Laufzeit zu ermöglichen.

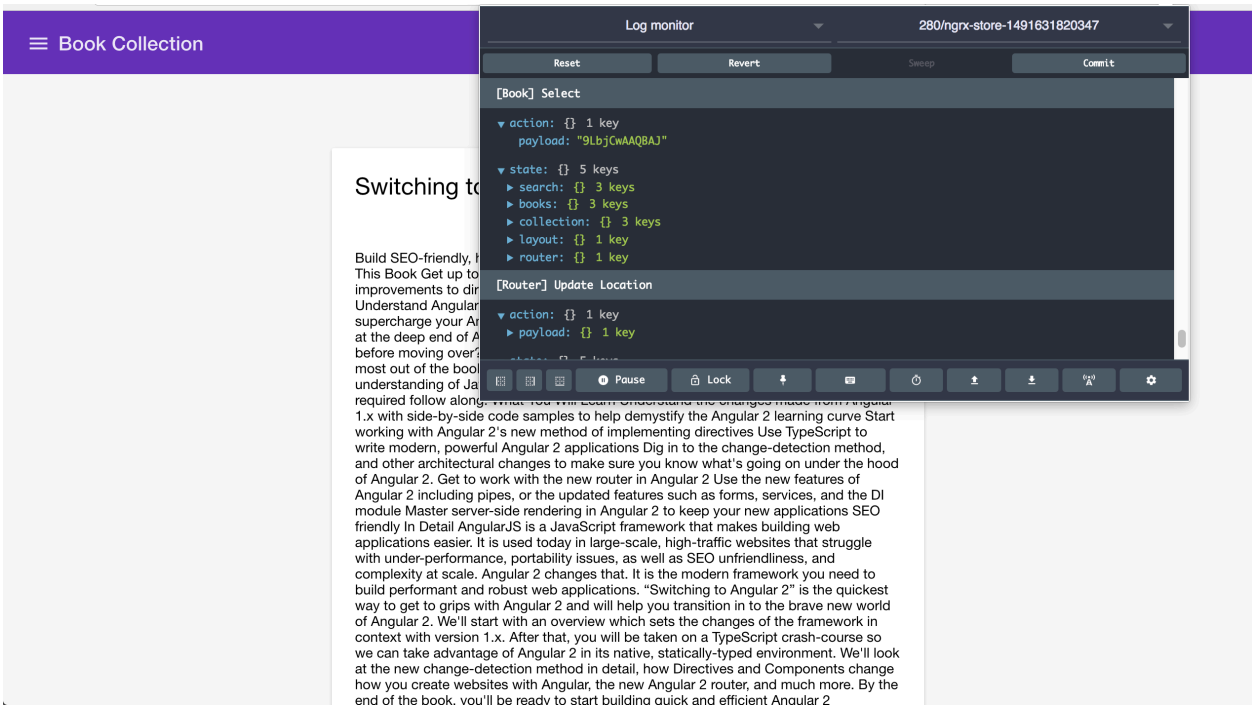


Abbildung 45 Verwendung der Redux Entwickler Werkzeuge (Log Monitor)

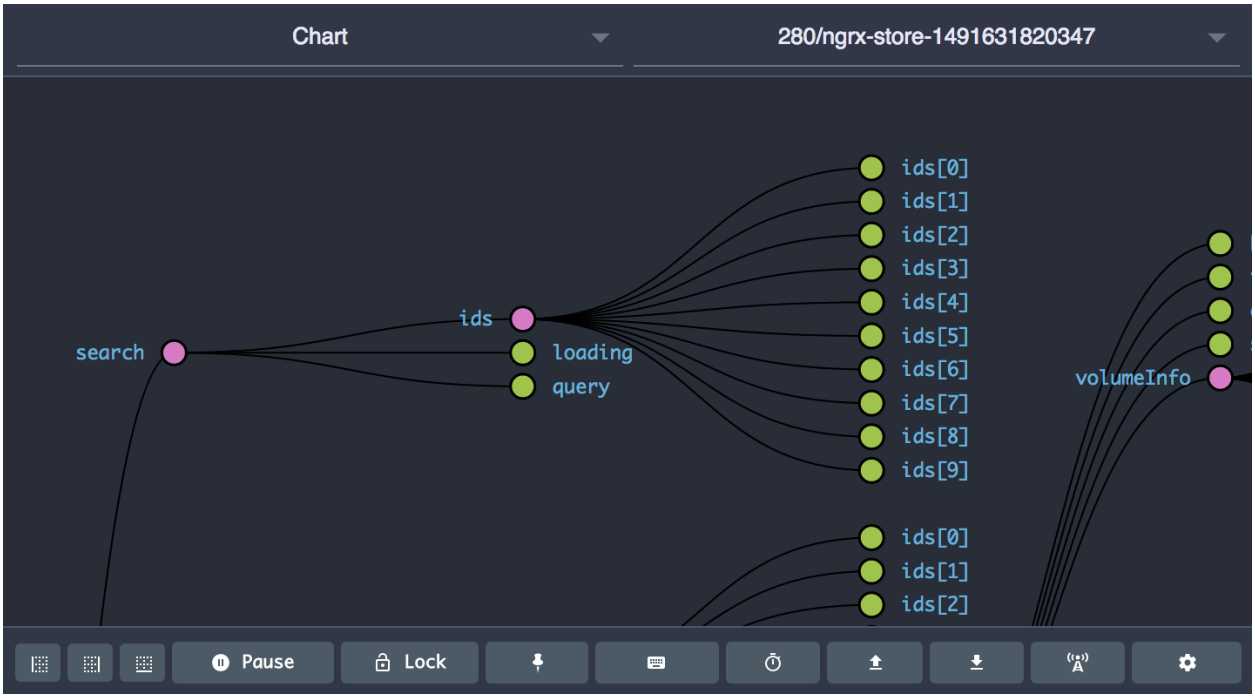


Abbildung 46 Verwendung der Redux Entwickler Werkzeuge (Chart)

Außerdem existiert eine leichtere Version des Redux Log Monitors in Form einer Logger Middleware, die die Zustandsänderungen innerhalb einer Redux Applikation in Form von „console.log“ aufrufen innerhalb der JavaScript Browserkonsole ausgibt. Diese Ausgabe kann in kann in der täglichen Entwicklungsarbeit enorm helfen.

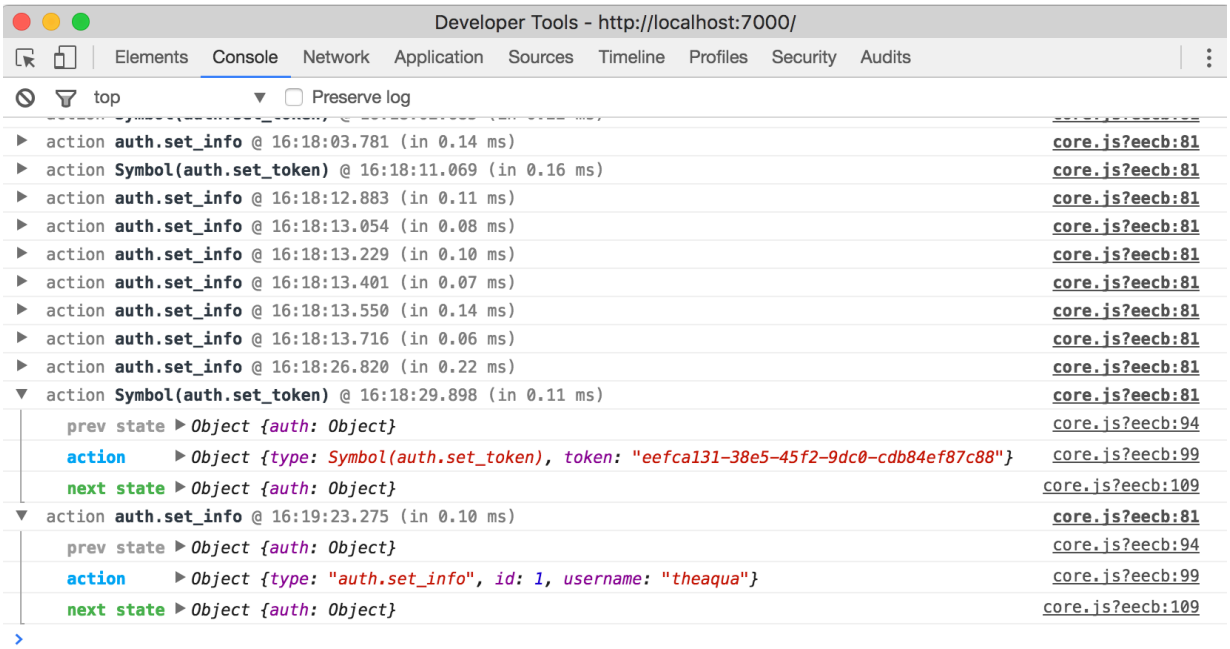


Abbildung 47 Redux Logger Middleware Beispiel

Da alle Benutzer Interaktionen innerhalb der Redux Applikation als einheitliche Actions gehandhabt werden, ist es möglich, den Applikationszustand über die Zeit festzuhalten und beispielsweise als Momentaufnahme einem Fehlerereignis anzuhängen (Troncone, Comprehensive Introduction to @ngrx/store, 2016).

Speziell die Verwendung des Redux Paradigmas in Angular kann Performancevorteile mit sich bringen, da der Applikationszustand zentralisiert in der Applikation vorliegt und über Container und Komponenten in die entsprechenden Applikationsteile gelangt. Angular 2 ist optimiert auf ein „Top-Down“ Datenfluss Szenario, indem Angular die Änderungsdetektion deaktivieren kann, solang eine Komponente ihre Daten aus Observables bezieht, die keine neuen Werte entsendet haben. Weitere Optimierungsszenarien in anderen Kontexten sind möglich. Die Performance Vorteile sind stark abhängig vom betrachteten Kontext, was durch die verschiedenen Messergebnisse der neun Szenaren in Kapitel 3.4 bestätigt wird. Innerhalb der Messungen konnte die Redux Implementierung bei wenig Daten durchweg schneller Aktualisierungen darstellen als die Daten Service Implementierung. Bei mittlerer gebundener Datenanzahl waren beide Implementierungen fast gleichschnell. Bei vielen Datenobjekten konnten sich bei der Redux Implementierung keine Performance Vorteile gegenüber der Daten Service Implementierung zeigen.

Wie bereits in Kapitel 2.5 erklärt wurde, werden alle Änderungen des Applikationszustands in Reducern gehandhabt, welche innerhalb des Redux Paradigmas als pure Funktionen ohne Seiteneffekte implementiert sind. Diese Art von Funktion ist sehr einfach zu testen, weil jeweils die gleichen Eingangsdaten die gleichen Ausgangsdaten produzieren (Geary, 2016). Des Weiteren ermöglicht diese Art des Zustandsmanagements, dass ein Test des Zustands ohne die Verwendung von „Mocks“ und „Spies“ auskommt, was die Testbarkeit weiterhin erhöht und weniger komplex und fehleranfällig ist (Elliot, How to Redux, 2016). Dies zeigt sich auch in der Auswertung der Entwickler Umfrage aus Kapitel 3.4, in der 87% der befragten Entwickler angaben, dass sie die Entwicklung eines Unit Tests innerhalb des Redux Paradigmas für wenig aufwändig einschätzen, wohin gegen 40% mittleren bzw. 33% der Entwickler viel Aufwand für die Entwicklung des gleichen Unit Tests innerhalb einer Daten Service getriebenen Architektur schätzen.

Ein weiter Vorteil des Redux Paradigmas ist es, dass fortgeschrittene Funktionen wie zum Beispiel unbegrenztes „zurück“ und „vor“ relativ einfach zu implementieren sind (Geary, 2016). Auch das Serialisieren und Speichern der Applikationszustandsgeschichte kann einfach durchgeführt und anschließend gespeichert werden (Parviainen, 2015). Dies kann wiederrum auch die Entwicklungszeit verkürzen, da während der Entwicklung auf einen gespeicherten Applikationszustand zurückgegriffen werden kann und so nicht nach jeder Änderung der Zustand wieder manuell durch den Entwickler hergestellt werden muss (Abramov & Contributors, Redux - Three Principles, 2016).

Redux als Zustandsmanagement Paradigma ist unabhängig vom jeweiligen Kontext und kann daher in Angular, React und vielen anderen Frameworks und Applikationen, die einen Zustand zu verwalten haben, Verwendung finden. Ein Entwickler, der zum Beispiel in einem Angular Kontext mit @ngrx/store arbeitet wird die Mechanismen im nächsten React Redux Projekt auch verstehen und anwenden können. Man kann sagen, dass das Redux Paradigma kontextunabhängig Anwendung finden kann.

Wie bereits in der Verbreitungsevaluation in Kapitel 3.4 aufgezeigt, umgibt das Redux Projekt eine unterstützende und lebendige Gemeinschaft aus Entwicklern aus der ganzen Welt. Einer der Co-Autoren von Redux, Dan Abramov, hat zwei qualitativ sehr hochwertige und kostenfreie Video Kurse auf egghead.io zum Redux Projekt veröffentlicht. Des Weiteren leistet Dan eine hervorragende Community Arbeit, zum Beispiel in Form einer umfangreichen und verständlichen Projekt Dokumentation und seiner Erreichbarkeit für Fragen und Diskussionen über Twitter.

3.6 Zusammenfassung des Vergleichs der Daten Service und Redux Zustandsmanagement Implementierung

Zusammenfassend lässt sich sagen, dass die Redux Implementierung gegenüber der Daten Service Implementierung in Form von Programmcode Zeilen und Anzahl der Dateien aufwändiger erscheint. Mit 1744 Programmcode Zeilen liegt die Redux Implementierung fast bei doppelt so viel Programmcode Zeilen wie die 981 Programmcode Zeilen der Daten Service Implementierung. Ein Trend in die gleiche Richtung lässt sich auch bei der Anzahl der Dateien festhalten, die Redux Implementierung ist aufgeteilt auf 50 Dateien, die Daten Service Implementierung kommt mit 37 Dateien aus.

Bei dem Performance Vergleich beider Implementierungen zeigt sich, dass die Daten Service getriebene Variante bei der Ladezeit um 41 ms schneller ist. Durch die Messungen der Render Performance von aktualisierten Daten konnte die Redux Implementierung bei wenig Daten durchweg schneller Aktualisierungen darstellen als die Daten Service Implementierung. Bei mittlerer gebundener Datenanzahl waren beide Implementierungen fast gleichschnell. Bei vielen Datenobjekten war die Daten Service Implementierung durchweg schneller als die Redux Implementierung.

Die Entwickler Umfrage hat ergeben, dass doppelt so viele Entwickler die Redux Implementierung im Vergleich zur Daten Service Implementierung für selbsterklärender halten. Die geschätzten Aufwände für die Neuentwicklung und Änderung bestehender Funktionalität mit Auswirkungen auf den Applikationszustand wird deutlich geringer innerhalb der Redux Implementierung eingeschätzt. Auch der geschätzte Aufwand für Unit Tests für den Applikationszustand fällt in der Redux Implementierung geringer aus.

Die Untersuchung der Entwicklung der Google Trends zeigt, dass seit der ersten Veröffentlichung des Redux Paradigmas ein ansteigendes Interesse vorherrscht und eine zunehmende Adaptierung des Paradigmas geschieht.

Die Anzahl der GitHub Stars als Pulsmesser innerhalb der Entwickler Szene zeigt, dass das Redux Paradigma sehr beliebt ist, dass sich viele Entwickler mit den Ideen und Konzepten des Paradigmas beschäftigen und die Implementierungen für aktuelle und zukünftige Projekte berücksichtigen wollen. Des Weiteren steht die Anzahl und Frequenz der Git Commits im Redux Repository für eine lebendige und aktive Gemeinschaft innerhalb des Redux Projekts.

Die Anzahl der Stack Overflow Fragen zeigt weiterhin, dass Redux Paradigma gut angenommen wurde, weil viele Entwickler sich damit auseinandersetzen und Fragen zu Verwendung haben.

Das „Thinking Aloud“ bei der Konfrontation von Dritten mit beiden Ansätzen hat auch verschiedene Aspekte der erarbeiteten Vorteile des Redux Paradigmas bestätigt.

Interessant zu sehen ist, dass beide Zustandsmanagement Implementierungen das Konzept der „smart“ und „dumb“ Komponenten verwenden können, obwohl dieses erst richtig mit dem Redux Paradigma populär wurde. Daran sieht man, dass ein fortschrittlicheres Paradigma auch positive Auswirkungen auf bereits bestehende Konzepte haben kann.

4 SCHLUSSBETRACHTUNG

„Redux, with its three principles of restrictions for updating state,
has proven to be invaluable in a number of projects we have implemented."
(ThoughtWorks, 2017)

4.1 Zusammenfassung

Das Ziel der vorliegenden Masterarbeit ist es, die Vorteile des Redux basierten Zustandsmanagement für JavaScript Applikationen gegenüber herkömmlichen JavaScript MV*-Architekturen zu erarbeiten und diese Vorteile kritisch reflektiert darzustellen.

Einleitend wird gezeigt, dass heut zu Tage das Thema Zustandsmanagement eine erfolgskritische Komponente bei der Entwicklung moderner Web Applikationen darstellt. Ausgehend von der MVC-Architektur entwickelten sich innerhalb der verschiedenen JavaScript Frameworks und Community zahlreiche Ansätze und Konzepte, um den Zustand einer Single Page Application möglichst nachvollziehbar, komfortabel und performant verwalten zu können. Je komplexer der Zustand einer Applikation wird, desto unübersichtlicher und schwerer nachvollziehbar werden klassische Zustandsmanagement Architekturen.

Das Redux Paradigma, welches aus den Ideen der Flux Architektur hervorging, greift einen Teil der MVC-Architektur auf und formuliert explizite Regeln, um den Zustand einer komplexen Web Applikation zu verwalten.

Im Praxisteil der vorliegenden Masterarbeit wurde erarbeitet, dass eine Redux Implementierung nicht weniger Programmcode als eine klassische Implementierung benötigt und eine Vielzahl an Konzepten beinhaltet, die sich auch in der Anzahl der Dateien wiederspiegeln. Aus Performancesicht konnte die @ngrx/store Redux Implementierung gegenüber der Daten Service Applikation nur bei einer geringen Anzahl von gebundenen Daten Vorteile erzielen.

Die verfügbaren Redux Entwicklerwerkzeuge sind eine große Hilfe bei der täglichen Entwicklung.

Die eigens erstelle Entwickler Umfrage hat ergeben, dass die Redux Implementierung im Vergleich zur Daten Service Implementierung von einem Großteil der befragten Entwickler für selbsterklärender gehalten wird. Auch die geschätzten Aufwände für die Neuentwicklung und Änderung des Applikationszustands werden innerhalb der Redux Implementierung deutlich geringer eingeschätzt.

Die Evaluation der Entwicklung der Verbreitung des Redux Paradigmas zeigt ein starkes Interesse der Entwicklergemeinde, welches sich insbesondere auch an der Aktivität im und um das Redux Projekt zeigt.

Des Weiteren hat das „Thinking Aloud" bei der Konfrontation von Dritten mit beiden Ansätzen die Nachvollziehbarkeit, die Kontextunabhängigkeit und die gute Testbarkeit der Redux Implementierung bestätigt.

4.2 Fazit und Ausblick

Wie aus der theoretischen Einleitung eindeutig ersichtlich geworden ist, lässt sich festhalten, dass das Thema Zustandsmanagement für moderne JavaScript Web Applikationen wichtig und erfolgskritisch ist. Das Ziel einer guten Zustandsarchitektur ist ein vorhersehbarer Applikationszustand. Alle genannten Autoren sind sich einig, dass der Komplex des Zustandsmanagements einer bedarfsgerechten und durchdachten Lösung bedarf. Die Implementierung des Redux Paradigmas kann eine probate Lösung sein, ist jedoch nicht als Allheillösung zu betrachten.

Wichtig ist die Betrachtung des Kontexts eines jeden Projekts. Die Größe, die Häufigkeit und Komplexität in Bezug auf den Zustand und dessen Änderungen der Web Applikation sind hier ausschlaggebend. Für wenig Funktionalität können die primären Vorteile des Redux Paradigmas eher unvorteilhaft wirken, da viele, neue Konzepte gelernt und verinnerlicht werden müssen, bevor man von der verbesserten Struktur profitieren kann.

Auf der anderen Seite steht das Potential, dass das Redux Paradigma in Bezug auf „Clean Code“ und moderne Programmierung mit sich bringt. Wer sich darauf einlässt, dass Redux Paradigma anzuwenden, lernt zentrale Elemente der funktionalen Programmierung und bekommt konkrete Einblick in ein sehr gut strukturiertes und natürlich testbares Paradigma.

Wie bereits im Kapitel 3.3 erwähnt, ist bereits das Vorhalten des Zustands im Daten Service fortgeschritten, so dass die Redux Variante eine Weiter-Entwicklung davon ist. Die Redux Implementierung ist sozusagen ein „Store Daten Service“ mit Redux Regeln zur Modifikation des Stores. Trotzdem ist die Daten Service Variante sehr Angular spezifisch und lässt sich nur in andere Szenarien und Kontexte übertragen, die ähnliche Komponenten zur Verfügung stellen. Dahingegen ist das Redux Paradigma als solches allgemeiner und damit viel universeller einsetzbar.

Redux vermag auf den ersten Blick aufwendig auszusehen und mehr Programmcode Zeilen für die Implementierung zu benötigen, hier ist zu prüfen, ob dieser Umstand wirklich ein Signal für Programmcode Qualität ist. Schaut man sich die Entwicklung des NPM an, dann geht der Trend für moderne und gut strukturierten Programmcode klar in die Richtung von einer Vielzahl gut abgetrennter, möglichst kleiner Module.

> *„Reusable and well-encapsulated UI components are the key to getting the code bloat in large single page web apps under control without the need for a major re-architecture“ (Shapiro, Web Component Architecture & Development with AngularJS, 2015, S. 6)*

Der Zustand einer modernen JavaScript Web Applikation sollte als einer der zentralen Entwicklungspfeiler („first class citizen“) behandelt werden, da in Umgebungen mit deklarativen Paradigmen die Systeme sich reaktiv zum Zustand verhalten. Es ist wichtig, sich die Zeit zu nehmen und sich bewusst zu machen, wie ausschlaggebend es sein kann, den Zustand einer Web Applikation zu designen und diesen auch im Laufe der Entwicklung kontinuierlich zu

verbessern. Zustandsmanagement sollte mit der gleichen Wichtigkeit wie der eigentliche Programmcode behandelt werden.

Wie bereits aus der Implementierung der Redux Variante anhand von @ngrx/store ersichtlich wurde, rückt das Thema funktionale, reaktive Programmierung immer mehr in den Vordergrund, bei dem bestimmte Datentypen wie zum Beispiel ein „Observable“ Werte im Verlauf der Zeit repräsentieren.

Die Bibliothek MobX ist eine von Redux inspirierte Bibliothek und verfolgt einen viel leichtgewichtigeren Ansatz für ein Zustandsmanagement welches auf Observables basiert. Für weniger komplexe Umgebungen kann MobX eine passende Lösung sein, jedoch bietet es per Standard keinen transaktionalen Zustand.

„Designing a Redux app often begins by thinking about the application state data structure.“ (Parviainen, 2015)

Ausgehend vom vorigen Zitat vertritt der Autor der vorliegenden Masterarbeit die Meinung, dass die explizite Sprache und Regeln des Redux Paradigmas die primären Vorteile sind. Anhand des Redux Paradigmas wurde eine Standard Sprache für das erfolgskritische Thema des Zustandsmanagements formuliert und populär gemacht. Dieses neue Denk- und Formulierungsmodell kann in den verschiedensten Kontexten Anwendung finden und ist dadurch sehr vielseitig einsetzbar und transportiert am konkreten Beispiel wertvolles Programmierwissen und Best Practices.

ANHANG A - 1. Anhang

Aussagen beim „Thinking Aloud“ bei der Konfrontation von Dritten mit beiden Ansätzen

- “*The plus of Redux is that you have to change one store with some reducers. It's much easier and clearer than services and patterns.”*
- “*I think that one change (a user input or API response) can affect the state of an application in many places in the code*”
- “*I give the Data Service implementation higher complexity due mainly to the fact that I don't know it at all. I am somewhat familiar with Redux so I at least know what to do with it, and making changes with it is generally pretty easy if I have taken basic steps to isolate functionality like creating selectors, etc*.”
- “*Redux can't change the state :D It can only replace it. But, negleting it, it's easier to change smth in Redux. If do not neglect and assume that you have to change smth in current state, than it's better to use data service since Redux can't change states as mentioned before.”*
- “*My assumption is that writing units tests should be pretty similar so I am giving Low to Redux because I have already done those easily. Probably would not be hard to do the same with the other implementation.”*
- “*A lot of differences. However, being short, in data service you will have to test everything all in depth while in Redux only layer with reducers*”

ANHANG B - 2. Anhang (digitale Form)

Sourcecode der Implementierung des Daten Service getriebenen Zustandsmanagement:

Online https://github.com/pablopaul/data-service-example-app

„data-service-example-app-master.zip”

Performance Branch “data-service-example-app-performance.zip”

Sourcecode der Implementierung des Redux getriebenen Zustandsmanagement:

Online https://github.com/pablopaul/ngrx-example-app

„ngrx-example-app-master.zip”

Performance Branch “ngrx-example-app-performance.zip”

Komplette Entwickler Umfrage mit allen Antworten:
als Pdf und in Tabellenform

Performance Messungen

Screenshots

ABKÜRZUNGSVERZEICHNIS

AJAX - Asynchronous JavaScript and XML

API - Application Programming Interface

CSS - Cascading Style Sheets

CQRS - Command Query Responsibility Segregation

DOM - Document Object Model

ES6 - ECMAScript 2015

HTML - Hypertext Markup Language

JS - JavaScript

JSON - JavaScript Object Notation

MVC - Model View Controller

MVP - Model View Presenter

MVVM - Model View View Model

NPM - Node Package Manager

OOP - Objektorientierte Programmierung

PM - Presenter Model

REST - Representational State Transfer

RIA - Rich Internet Application

RxJS - Reactive Extensions for JavaScript

SPA - Single Page Application

UI - User Interface

URL - Uniform Resource Locator

XHR - XMLHttpRequest

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGVERZEICHNIS

LITERATURVERZEICHNIS

A M, V., & Sonpatki, P. (2016). *ReactJS by Example - Building Modern Web Applications with React.*

Abramov , D., & Contributors. (06. 11 2015). *Redux Motivation*. Abgerufen am 23. 06 2016 von Redux Motivation: https://github.com/reactjs/redux/blob/master/docs/introduction/Motivation.md

Abramov, D. (2016). *Getting Started with Redux*. Abgerufen am 22. 06 2016 von Getting Started with Redux: https://egghead.io/courses/getting-started-with-redux

Abramov, D., & Contributors. (16. 01 2016). *Intro and 3 Principles of Redux*. Abgerufen am 23. 06 2016 von https://github.com/tayiorbeii/egghead.io_redux_course_notes/blob/master/01-Intro_and_3_Principles_of_Redux.md

Abramov, D., & Contributors. (19. 02 2016). *Redux - Prior Art*. Abgerufen am 23. 06 2016 von Redux - Prior Art: http://redux.js.org/docs/introduction/PriorArt.html

Abramov, D., & Contributors. (04. 06 2016). *Redux - Three Principles*. Abgerufen am 23. 06 2016 von Redux - Three Principles: http://redux.js.org/docs/introduction/ThreePrinciples.html

Abramov, D., & Redux Contributors. (2016). *Redux*. Abgerufen am 22. 06 2016 von Redux: http://redux.js.org/index.html

Akshat, P., & Nalwaya, A. (2016). *React Native for iOS Development.*

Alfoni, C. (08 2015). *Why we are doing MVC and FLUX wrong*. Abgerufen am 15. 11 2016 von Why we are doing MVC and FLUX wrong: http://www.christianalfoni.com/articles/2015_08_02_Why-we-are-doing-MVC-and-FLUX-wrong

Böhm, R. (19. 12 2015). *Angular 2 Tutorial für Einsteiger*. Abgerufen am 17. 11 2016 von Angular 2 Tutorial für Einsteiger: https://angularjs.de/artikel/angular2-tutorial-deutsch

Bailey, D. (23. 12 2011). *Backbone.js is not an MVC Framework*. Abgerufen am 14. 11 2016 von Backbone.js is not an MVC Framework: http://lostechies.com/derickbailey/2011/12/23/backbone-js-is-not-an-mvc-framework

Bainomugisha, Carreton, A., Cutsem, T., Mostinckx, S., & De Meuter, W. (2013). A Survey on Reactive Programming. *ACM Computing Surveys, Vol. 45, No. 4*.

Benguella, R. (28. 12 2015). *http://riadbenguella.com/*. Abgerufen am 22. 06 2016 von http://riadbenguella.com/: http://riadbenguella.com/from-actions-creators-to-sagas-redux-upgraded/

Bevacqua, N. (2015). *JavaScript Application Design.*

Chen, J. (04. 05 2014). Hacker Way: Rethinking Web App Development at Facebook.

Cooper, G., & Krishnamurthi. (2006). Embedding dynamic dataflow in a call-by-value language. *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06)*.

Coury, F., Lerner, A., Murray, N., & Taborda, C. (2016). *ng-book 2.*

Cravens, J., & Brady, T. (2014). *Building Web Apps with Ember.js.*

Davis, M., Ryan, M., & Salathe, F. (31. 12 2016). *ngrx/effects Introduction*. Abgerufen am 08. 03 2017 von ngrx/effects Introduction: https://github.com/ngrx/effects/blob/master/docs/intro.md

de Sousa Antonio, C. (2015). *Pro React.*

Elliot, E. (2013). *Programming JavaScript Applications.*

Elliot, E. (12. 11 2015). *Webcast – The Two Pillars of JS: Introduction to Functional Programming*. Abgerufen am 08. 04 2017 von Webcast – The Two Pillars of JS: Introduction to Functional Programming: https://ericelliottjs.com/premium-content/webcast-the-two-pillars-of-js-introduction-to-functional-programming/

Elliot, E. (28. 08 2016). *10 Tips for Better Redux Architecture*. Abgerufen am 28. 08 2016 von 10 Tips for Better Redux Architecture: https://medium.com/javascript-scene/10-tips-for-better-redux-architecture-69250425af44

Elliot, E. (Autor). (2016). *How to Redux* [Kinofilm].

Elliot, E. (2016). *The Best Way to Learn to Code is to Code: Learn App Architecture by Building Apps*. Abgerufen am 23. August 2016 von The Best Way to Learn to Code is to Code: Learn App Architecture by Building Apps: https://medium.com/javascript-scene/the-best-way-to-learn-to-code-is-to-code-learn-app-architecture-by-building-apps-7ec029db6e00#.uh7f6x1gz

Fedosejev, A. (2015). *React.js Essentials.*

Fink, G., & Flatow, I. (2014). *Pro Single Page Application Development.*

Fluin, S. (27. 01 2017). *Branding Guidelines for Angular and AngularJS*. Abgerufen am 02. 03 2017 von Branding Guidelines for Angular and AngularJS: http://angularjs.blogspot.de/2017/01/branding-guidelines-for-angular-and.html

Fowler, M. (2004). *Presentation Model* . Abgerufen am 15. 11 2016 von Presentation Model : http://martinfowler.com/eaaDev/PresentationModel.html

Fowler, M. (12 2015). *Event Sourcing*. Abgerufen am 26. 08 2016 von Event Sourcing: http://martinfowler.com/eaaDev/EventSourcing.html

Freeman, A. (2012). *Pro JavaScript for Web Apps.*

Gackenheimer, C. (2015). *Introduction to React.*

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1997). *Design Patterns - Elements of Reusable Object-Oriented Software.*

Geary, D. (18. 07 2016). *Introducing Redux*. Abgerufen am 15. 11 2016 von Introducing Redux: http://www.ibm.com/developerworks/web/library/wa-manage-state-with-redux-p1-david-geary/index.html

GitHub. (30. 03 2017). *Repository Search*. Abgerufen am 30. 03 2017 von Repository Search: https://github.com/search?l=JavaScript&p=3&q=stars%3A%3E1&type=Repositories&utf8=%E2%9C%93

GitHub Stats. (30. 03 2017). *Github Stats*. Abgerufen am 30. 03 2017 von Github Stats: https://githubstats.com/reactjs/redux

Google. (06. 04 2017). *Google Trends*. Abgerufen am 06. 04 2017 von Google Trends: https://trends.google.de/trends/explore?q=redux%20js

Google. (2017). *Services - ts - TUTORIAL*. Abgerufen am 28. 02 2017 von Services - ts - TUTORIAL: https://angular.io/docs/ts/latest/tutorial/toh-pt4.html

Gossmann, J. (08. 10 2005). *Introduction to Model/View/ViewModel pattern for building WPF apps*. Abgerufen am 15. 11 2016 von Introduction to Model/View/ViewModel pattern for building WPF apps: https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/

Guedes, C. (12 2015). *A Tour on React ecosystem*. Abgerufen am 23. 06 2016 von A Tour on React ecosystem: http://slides.com/cguedes/a-tour-on-react-ecosystem

Harrison, R., Samaraweera, L., Dobie, M., & Lewis, P. (2012). *Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs.*

Horie, L. (11. 04 2017). *Mithril 1.1.1 Framework comparison*. Abgerufen am 13. 04 2017 von Mithril 1.1.1 Framework comparison: https://mithril.js.org/framework-comparison.html

Hughes, J. (1984). Why functional programming matters. *The Computer Journal*.

Irish, P. (22. 02 2011). *requestAnimationFrame for Smart Animating*. Abgerufen am 20. 04 2017 von requestAnimationFrame for Smart Animating: https://www.paulirish.com/2011/requestanimationframe-for-smart-animating/

ISO/IEC IS 13250-2:2006. (2006). *Information Technology - Document Description and Processing Languages - Topic Maps - Data Model.* (International Organization for Standardization, Hrsg.) Abgerufen am 04. 05 2012 von http://www.isotopicmaps.org/sam/sam-model/

Kohn, F., Kutzmutz, O., & Larisch, P. (Hrsg.). (2010). *Destillate - Literatur Labor Wolfenbüttel 2010.* Wolfenbüttel: Bundesakademie für kulturelle Bildung.

Kol, T. (03. 10 2016). *Avoiding Accidental Complexity When Structuring Your App State.* Abgerufen am 20. 11 2016 von Avoiding Accidental Complexity When Structuring Your App State: https://hackernoon.com/avoiding-accidental-complexity-when-structuring-your-app-state-6e6d22ad5e2a#.i800253hx

Lumpe, J., Purkhardt, K., Muller, A., Cravero, D., & Chentnik, E. (2016). *Developing a Redux Edge.*

MacCaw, A. (2011). *JavaScript Web Applications.*

Maier, I., Rompf, T., & Odersky, M. (2010). *Deprecating the Observer Pattern.*

Mikowski, M., & Powell, J. (2014). *Single Page Web Applications.*

Minar, I. (12. 07 2012). *MVC vs MVVM vs MVP*. Abgerufen am 15. 11 2016 von MVC vs MVVM vs MVP: https://plus.google.com/+AngularJS/posts/aZNVhj355G2

Monteiro, F. (2014). *Learning Single-page Web Application Development.*

Mulder, P. (2014). *Full Stack Web Development with Backbone.js.*

Nesher, G. (23. 08 2016). From MVC to Flux.

Osmani, A. (2012). *Learning JavaScript Design Patterns.*

Osmani, A. (2015). *JavaScript Application Design - Vorwort.*

Osmani, A. (07. 06 2016). *Developing Backbone.js Applications*. Abgerufen am 11. 09 2016 von Developing Backbone.js Applications: https://addyosmani.com/backbone-fundamentals/

Parviainen, T. (2015). *Full-Stack Redux Tutorial*. Abgerufen am 19. 10 2016 von http://teropa.info/blog/2015/09/10/full-stack-redux-tutorial.html

Paul, A., & Nalwaya, A. (2016). *React Native for iOS Development.*

Pucella, R. (1998). Reactive programming in standard ml. *Proceedings of the International Conference on Computer Languages (ICCL'98)*.

RFC 2616. (1999). *Hypertext Transfer Protocol -- HTTP/1.1.* (IETF, Hrsg.) Von http://www.ietf.org/rfc/rfc2616.txt abgerufen

Saternos, C. (2014). *Client-Server Web Apps with JavaScript and Java.*

Savkin, V. (13. 07 2016). *Change Detection in Angular 2*. Abgerufen am 18. 11 2016 von Change Detection in Angular 2: https://vsavkin.com/change-detection-in-angular-2-4f216b855d4c

Schläpfer, R. (29. 09 2014). *Why you might not need MVC with React.js*. Abgerufen am 15. 11 2016 von Why you might not need MVC with React.js: http://www.code-experience.com/why-you-might-not-need-mvc-with-reactjs/

Schreiber, D. V., & Leser, A. (Hrsg.). (2008). *Wichtiges Werk.* Graz: Wissensverlag.

Scott, E. (2016). *SPA Design and Architecture.*

Shapiro, D. (2015). *Web Component Architecture & Development with AngularJS.*

sotagtrends.com. (30. 03 2017). *Tag Trends*. Abgerufen am 30. 03 2017 von Tag Trends: http://sotagtrends.com/?tags=[redux]

Sotelo, C. (06. 11 2014). *Migrating to AngularJS 2.0*. Abgerufen am 18. 11 2016 von Migrating to AngularJS 2.0: http://paislee.io/migrating-to-angularjs-2-0/

Stefanov, S. (2010). *JavaScript Patterns.*

Swanepoel, H. (20. 09 2013). *A quick comparison of modern client-side MV* frameworks*. Abgerufen am 03. 04 2017 von A quick comparison of modern client-side MV* frameworks: https://www.slideshare.net/HendrikSwanepoel/a-quic

ThoughtWorks. (01. 03 2017). *Redux | Technology Radar | ThoughtWorks*. Abgerufen am 10. 04 2017 von Redux | Technology Radar | ThoughtWorks: https://www.thoughtworks.com/radar/languages-and-frameworks/redux

Tol, K. (03. 10 2016). *Avoiding Accidental Complexity When Structuring Your App State.* Abgerufen am 20. 11 2016 von Avoiding Accidental Complexity When Structuring Your App State: https://hackernoon.com/avoiding-accidental-complexity-when-structuring-your-app-state-6e6d22ad5e2a

Troncone, B. (10. 11 2016). *Comprehensive Introduction to @ngrx/store*. Abgerufen am 19. 03 2017 von Comprehensive Introduction to @ngrx/store: https://gist.github.com/btroncone/a6e4347326749f938510

Troncone, B., & Wormald, R. (27. 01 2017). *@ngrx - Reactive Extensions for Angular 2*. Abgerufen am 28. 03 2017 von @ngrx - Reactive Extensions for Angular 2: http://ngrx.github.io/

Valdecantos, H. (2016). *An empirical study on code comprehension: DCI compared to OO.*

Wartburg, R. v., Steinbacher, S., Wittmer, R., & Schütze, S. (2011). *Zitieren und Referenzieren nach APA.* Abgerufen am 20. 4 2012 von Wissenschaftliches Schreiben in der Psychologie: http://etools.fernuni.ch/wiss-schreiben/apa/de/html/index.html

Weisse, B. (2016). *AngularJS & Ionic Framework.*

You, E., Beziuk , G., & Fritz, C. (13. 11 2016). *Comparison with Other Frameworks*. Abgerufen am 17. 11 2016 von Comparison with Other Frameworks: https://vuejs.org/v2/guide/comparison.html