Technology
Arts Sciences
TH Köln

25knots – Ein Tool zur Verbesserung der gestalterischen Qualität von Artefakten im Hochschulkontext

Umsetzung vom Proof of Concept zur marktfähigen Webanwendung

BACHELORARBEIT

vorgelegt an der Technischen Hochschule Köln

Campus Gummersbach

Im Studiengang Medieninformatik

ausgearbeitet von

Christian Alexander Poplawski

MATRIKELNUMMER 11088931

Erster Prüfer: Prof. Dipl. Des. Christian Noss

Technische Hochschule Köln

Zweiter Prüfer: Dipl. Des. Liane Kirschner

Railslove GmbH

Gummersbach, im August 2017

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die nachfolgende Arbeit beschäftigt sich mit der Entwicklung der Webanwendung *25Knots* zu einem Markfähigen Produkt, basierend auf bereits im Praxisprojekt erarbeiteten Konzepten und einem Proof of Concept.
Ziel ist dabei nicht nur, die Verwendung durch die Community[1] nach Abschluss dieser Arbeit möglich zu machen, sondern diese auch aktiv an der Weiterentwicklung der Anwendung teilhaben zu lassen.

Die Anwendung *25Knots* zielt darauf ab, die gestalterische Qualität von Artefakten im Hochschulkontext zu verbessern. Dieses Ziel soll durch die Unterstützung von Studierenden während des Entwicklungsprozesses von Artefakten erreicht werden. Die Anwendung soll den Studierenden dabei eine Möglichkeit bieten, interaktiv Ergebnisse zu produzieren, die für ihren aktuellen Entwicklungsprozess hilfreich sind. Wissen soll dabei eher implizit, durch ein *Learning by Doing* Konzept vermittelt werden.

Generell sollen in dieser Arbeit alle Aspekte behandelt werden, die auf dem Weg von einem Prototypen zu einem marktfähigen Produkt eine Rolle spielen. Diese umfassen zum Beispiel die Struktur und Gestaltung der Anwendung, technische Entscheidungen die bei der Umsetzung getroffen werden müssen, die generelle Entwicklung, aber auch Bereiche wie Hosting oder Möglichkeiten zur Weiterentwicklung. Im Folgenden Kapitel sollen zunächst einige grundle-

gende Ziele und das generelle Vorgehen bei der Umsetzung erläutert werden

1.1 Motivation

Wie bereits erwähnt basiert das Thema dieser Arbeit auf Konzepten, die im Rahmen des Praxisprojektes erstellt wurden. Bei der Erstellung dieser Konzepte wurde explizit auf eine spätere Umsetzbarkeit geachtet, so wurden zum Beispiel schon einige grobe Berechnungsmethoden entworfen, die in der Anwendung verwendet werden können.
Die Umsetzung des Proof of Concept im Bereich Typographie hat außerdem gezeigt, dass die erstellten Konzepte in einer Anwendung durchaus funktionieren.
Weiterhin konnte ich in persönlichen Gesprächen mit verschiedenen Personen innerhalb und auch außerhalb des Hochschulkontextes feststellen, dass eine Umsetzung der erarbeiteten Konzepte durchaus eine reale Zielgruppe besitzt.

Die Motivation für diese Arbeit ergibt sich daher aus dem Willen, aus diesem erarbeiteten Konzept einen Mehrwert schaffen zu wollen, der über den Erhalt von *Credit Points* hinaus geht. Dieser Mehrwert entsteht dabei zum einen für mich persönlich, zum anderen aber auch für andere Personen, die einen Mehrwert aus der Existenz einer solchen Anwendung ziehen können.
Die generelle Motivation für die Existenz einer Anwendung wie *25knots* lässt sich dem Kapitel *Relevanz des Themas* auf Seite 6 entnehmen.

1.2 Zielsetzung

Ziel der vorliegenden Arbeit soll es sein, ein von der Community verwendbares und erweiterbares Produkt zu entwickeln. Zunächst sei hier ein *verwendbares Produkt* im Rahmen dieser Arbeit definiert.
Ein verwendbares Produkt:

1. bietet eine befriedigende Nutzererfahrung
2. ist einfach zugänglich

[1] Als Teil der *Community* wird hier jede Person gesehen, die ein Interessen an der Anwendung besitzt.

3. ist bei potentiellen Nutzern als Hilfsmittel zum Erreichen eines Zieles bekannt

Eine befriedigende Nutzererfahrung bedeutet für die Anwendung konkret, dass diese gut Strukturiert, ansprechend Gestaltet und ohne Probleme verwendbar sein muss. *Ohne Probleme verwendbar* impliziert hierbei eine hohe Qualität des geschriebenen Codes, um Fehler zu vermeiden. Der Großteil der Arbeit wird sich mit der Umsetzung dieses Bereiches beschäftigen.
Um die Anwendung einfach zugänglich zu machen, bietet sich das Hosting auf einem Server an (im Gegensatz zu beispielsweise nur der Bereitstellung das Anwendungscodes). Weitere details zu diesem Thema werden im Kapitel *Hosting* auf Seite 41 erläutert.
Damit die Anwendung genutzt werden kann, müssen ihre potentiellen Nutzer von ihrer Existenz wissen. Mit Blick auf die Zielgruppe bietet sich zunächst eine Bewerbung direkt an der Hochschule, beispielsweise durch die Teilnahme am *Medieninfromatik Showcase* an. Aber auch eine Bewerbung im größeren Rahmen, beispielsweise durch einen Talk beim *DevHouse Friday Köln* ist denkbar.

Zuletzt sei auch die erweiterbarkeit der Anwendung konkret definiert. Eine mögliche Erweiterbarkeit bedeutet zunächst, dass der Code der Anwendung öffentlich verfügbar sein muss, beispielsweise auf der Plattform Github[2]. Weiterhin muss der Code gut dokumentiert und verständlich geschrieben sein, um einen Einstieg in das bestehende Projekt für Dritte so einfach wie möglich zu halten. Eine erweiterbarkeit bezieht sich aber nicht nur auf geschriebenen Code, sondern kann (und soll) auch auf konzeptioneller Ebene erfolgen. Auch hier muss die Möglichkeit zur Beteiligung so simpel wie möglich gehalten werden. Eine ausführlichere Diskussion findet sich im Kapitel *Release*

Aus diesen Punkten kann eine zentrale Forschungsfrage für die Arbeit formuliert werden:
Wie kann der Community ein nutzbares und erweiterbares Produkt auf Basis eines Konzeptes und Prototypen bereitgestellt werden?
Diese lässt sich in zwei Unterfragen unterteilen, die es in dieser Arbeit zu beantworten gilt:

1. Durch welche Maßnahmen kann eine aktive Nutzung und Weiterentwicklung durch die Community gewährleistet werden?

2. Welche technischen Entscheidungen müssen während der Entwicklung getroffen werden?

[2] `https://github.com`

Auch wenn es Teil der Zielsetzung ist, ein marktfähiges Produkt zu erstellen kann hier nicht davon ausgegangen werden, dass das Endergebnis der Arbeit ein Produkt ist, das als fertig angesehen werden kann. Auch mit Blick auf die spätere Weiterentwicklung durch die Community muss es viel mehr das Ziel sein, eine hochwertige erste Version des Produktes, die als Grundlage für weitere Features dient, also ein *Minimum Viable Product* zu erstellen.

1.3 Relevanz des Themas

Die Relevanz der Anwendung an sich wurde bereits im Praxisprojekt erläutert, daher soll hier nur eine Kurzfassung der Erläuterung folgen. Der Grundgedanke der Relevanz ist dabei folgender:

> Menschen bilden sehr schnell ein Urteil über die Gestaltung eines Artefaktes und dieses lässt sich nur schwer wieder ändern. Weiterhin wird dieser schlechte erste Eindruck auf andere Bereiche des Artefaktes übertragen und wirkt sich somit unter Umständen auch auf die Gesamtbewertung eines Artefaktes aus.

Gestützt wird dieser Gedanke durch Studien von [LFDB06], [CP96], und [Nic98].

Unterstützend seien hier noch zwei weitere Quellen aufgeführt, die die Relevanz weiter unterstreichen: [TCKS06] zeigen, dass die Ergebnisse der Studie von Lindgaard et al. auch mit anderen Parametern bestand haben und unterstreichen weiterhin die Wichtigkeit von guter Gestaltung für eine gute Nutzererfahrung [TKI00].

Neben der Relevanz des Produktes, das in Rahmen der Arbeit entstehen soll sei aber auch das übergreifende Thema der Arbeit angesprochen: Die Entwicklung von einem Konzept zu einem fertigen Produkt. Als abschließende Arbeit für den Studiengang Medieninformatik ist dieses ein passendes Thema, da hier viele Aspekte aus verschiedenen Modulen des gesamten Studiums vereint werden. Daher bietet das Thema eine gute Verbindung zwischen den verschiedenen Disziplinen innerhalb des Studiums und einer wissenschaftlichen Diskussion verschiedener Vorgehensweisen und Abläufe.

1.4 Zielgruppe

Während der Konzeption im Praxisprojekt wurden Studenten der Technischen Hochschule Köln im Studiengang Medieninformatik als Zielgruppe festgelegt. Es wurde aber bereits dort deutlich, dass diese Zielgruppe leicht erweiterbar ist. Somit kann jede Person einen Mehrwert aus diesem Tool ziehen, die ein Artefakt erstellen muss dessen Hauptaugenmerk eigentlich nicht auf der Gestaltung, sondern auf einer bestimmten Funktion liegt. Die fehlende Aufmerksamkeit für die Gestaltung kann im Studium an einer fehlenden Bewertung dieser oder in der Wirtschaft an einem mangelndem Budget liegen. Es entstehen also häufig Situationen, in denen eine solide Gestaltung nicht als wichtig erachtet wird, diese jedoch, wie im vorhergehenden Kapitel erwähnt, durchaus Vorteile mit sich bringt.

Für diese Arbeit werden aber zunächst weiterhin die Studenten des Studienganges Medieninformatik als Zielgruppe definiert, um eine Disparität zwischen dem Konzept und der Umsetzung auszuschließen.

1.5 Struktur der vorliegenden Arbeit

Außerhalb dieser Einleitung teilt sich die Arbeit in drei weitere Kapitel.
Im zweiten Kapitel werden zunächst einige Theoretische Grundlagen und Entscheidungen auf einer taktischen Ebene erläutert. Dort findet sich neben einigen strukturellen Erläuterungen auch eine Diskussion verschiedener Technologien, mit denen eine Umsetzung möglich gewesen wäre und ein Einstieg in verschiedene Aspekte der Programmierung mit React.js.
Im dritten Kapitel werden einige Aspekte der Umsetzung und Gestaltung der Anwendung angesprochen. Es finden sich beispielsweise Erläuterungen zu den einzelnen Bereichen der Anwendung, aber auch Bereichsübergreifende praktische Themen wie die CSS-Architektur.
Das vierte Kapitel beschäftigt sich mit Fragen die im Bezug mit der Veröffentlichung der Anwendung stehen.

NOTIZ: Hier wird am Ende noch einmal drüber geschaut, wenn die tatsächliche Struktur etwas deutlicher ist.

Kapitel 2

Theoretische Grundlagen

Das nachfolgende Kapitel soll die theoretischen Grundlagen erläutern, die für die Umsetzung der Anwendung von Bedeutung sind. Diese Grundlagen teilen sich in verschiedene Bereiche auf. Zum einen soll der Aufbau der Anwendung festgelegt werden. In diesem Beriech müssen außerdem einige Bereich noch Konzipiert werden. Auch dieser Prozess und dessen Ergebnisse sollen hier erläutert werden. Weiterhin gilt es, die verschiedenen technischen Möglichkeiten der Umsetzung zu diskutieren. Abschließend findet sich ein konzeptioneller Einstieg in React.js und in einige damit verbundene und wichtige Bibliotheken, sowie eine Darstellung der Design-Prozesse.

2.1 Struktur der Anwendung

In diesem Abschnitt soll zunächst die generelle Struktur der Anwendung definiert werden. Weite Teile der Struktur können dabei aus dem Praxisprojekt übernommen werden. Im Praxisprojekt wurden die folgenden Themengebiete behandelt:

- Typographie
- Layout & Struktur
- Whitespace
- Farben
- Bilder

- Interaktive Elemente

Nach einer erneuten evaluation der Ergebnisse des Praxisprojektes konnten die in der Abschlussarbeit zu behandelnden Themengebiete auf zunächst drei eingegrenzt werden (der Bereich *Layout & Struktur* wurde dabei in *Layout & Grids* umbenannt):

- Typographie
- Layout & Grids
- Farben

Diese Abgrenzung begründet sich auf verschiedene Weisen. Im Fall des Bereiches *Whitespace* konnte im Rahmen des Praxisprojektes kein zufriedenstellendes Konzept erarbeitet werden, wodurch sich dieser Bereich per se nicht für eine Umsetzung eignet. Weiterhin muss es Ziel dieser Arbeit sein, am Schluss eine vollständige Anwendung zu erhalten. Um dieses Ziel im zeitlichen Rahmen erreichen zu können mussten weitere Themengebiete vernachlässigt werden. Hier boten sich die Bereiche *Bilder* und *Interaktive Elemente* an, da diese für die gestalterische Grundqualität eine vergleichsweise niedrige Rolle spielen. **NOTE: Vielleicht muss das hier noch belegt werden, wie ein Brötchen**. Diese Bereiche wurden aber ausreichend konzeptioniert und bieten sich als erste Erweiterungen für die Anwendung nach beenden der Arbeit an.

Außerdem müssen für die Anwendung jeweils ein nutzerfreundlicher Einstieg und Ausstieg gefunden werden. Diese wurden im Praxisprojekt nicht explizit ausgearbeitet und fallen somit auch Konzeptionell in den Bereich der Abschlussarbeit und werden später in diesem Kapitel behandelt.

Die finale Struktur der Anwendung für den Rahmen dieser Arbeit sieht also wie folgt aus:

- Einstieg
- Typographie
- Layout & Grids
- Farben
- Ausstieg

2.1.1 Einstieg in die Anwendung

Bereits im Praxisprojekt wurde festgestellt, dass es sinnvoll ist, das Zielmedium des Nutzers zu kennen. Mit Blick auf die Zielgruppe wurden hier drei mögliche Bereiche definiert: Native App, Website und Textdokument. **NOTE: Hier verweis auf Untersuchung im PP**. Diese Bereiche können jedoch auch in sich verschiedene Eigenarten aufweisen, so kann ein Textdokument beispielsweise für das Lesen an einem Bildschirm oder das Lesen in gedruckter Form entworfen werden. Eine komplette Auflistung der möglichen Bereiche oder *scopes* der Anwendung findet sich in Abbildung 2.1 auf Seite 10.

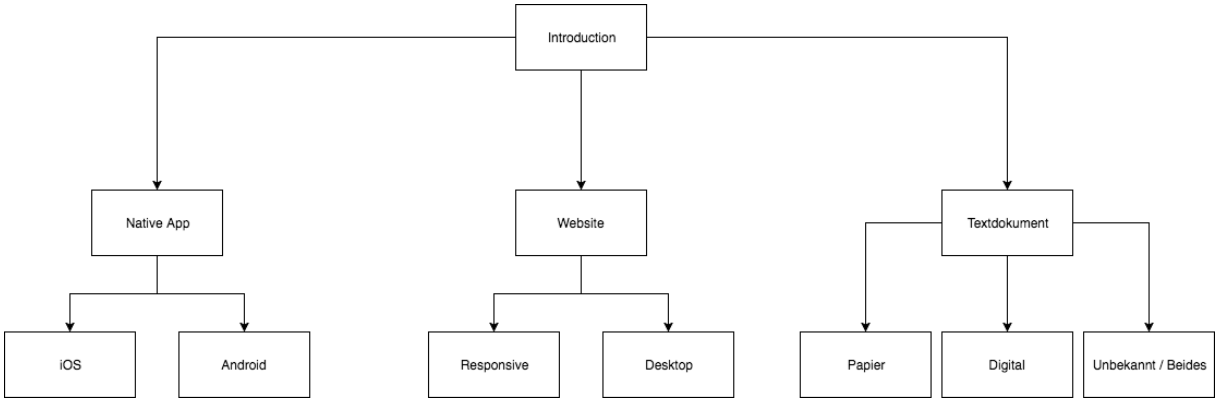

Abbildung 2.1: Mögliche Entscheidungen im Einstieg der Anwendung

Obwohl die Abgrenzung der Bereiche für die hier definierte Zielgruppe ausreichend ist, lassen sich bereits jetzt einige Stellen erkennen, die bei einer möglichen späteren Erweiterung der Zielgruppe überarbeitet werden müsste. Vorrangig betrifft das den Bereich *Website*. Hier ist die vorhandene Unterteilung in *Responnsive* und *Desktop* für ein Echtwelt-Szenario unter Umständen zu allgemein gehalten.

Um den kognitiven Aufwand **Beleg** für den Nutzer möglichst gering zu halten, bietet es sich an, ihn Schrittweise durch das Festlegen des für ihn passenden Bereiches zu führen.

2.1.2 Ergebnisse der Benutzung

Eines der Ziele der Anwendung ist es, dem Nutzer währen der Nutzung auf interaktive weise Wissen zu vermitteln. Da der Nutzer jedoch während der Nutzung auch konkrete Ergebnisse erarbeitet wäre es hier kontraproduktiv, ihm diese Ergebnisse nicht am Ende der Anwendung noch einmal explizit zukommen zu lassen (zusätzlich zum implizit gesammelten Wissen).

Dieser Bereich wurde im Rahmen des Praxisprojektes nicht ausdefiniert, ist aber für die Wahrnehmung der Anwendung als fertiges Produkt durchaus wichtig. Eine gute Darstellung der Ergebnisse des Nutzers definieren einen ausschlaggebenden Teil der Nutzungserfahrung. Hier sollen also mögliche Darstellungen der Ergebnisse diskutiert werden und vorrangig zwei Fragen beantwortet werden:

1. Welche Darstellung der Ergebnisse ist für den Nutzer am Vorteilhaftesten?
2. Welche Darstellung bietet den besten Kompromiss aus Umsetzbarkeit und Mehrwert für den Nutzer?

Die einfachste Darstellung ist eine Transistente Darstellung innerhalb der Anwendung am Ende der Verwendung. Eine Darstellung ist wegen der Haltung der ermittelten Werte im Redux-Store der Anwendung einfach. Obwohl einfach umzusetzen, ist diese Lösung nicht optimal: Nachdem der Nutzer die Anwendung schließt sind die erarbeiteten Daten verloren, da diese nicht persistent gespeichert werden.

Ein naheliegender Schritt ist also eine Persitierung des Wissens für den Nutzer. Hier bieten sich verschiedene Möglichkeiten, wie zum Beispiel das Speichern in Cookies oder das Entwicklen eines Backends, an. Ein Kompromiss zwischen Usability und Entwicklungsaufwand wäre hierbei die persistieren des Wissen in einer herunterladbarer Datei, beispielsweise als PDF.

Eine weitere Frage beschäftigt sich mit dem Aufbau dieser Datei. Auch hier wäre der einfachste Ansatz, die Ergebnisse des Nutzers einfach aufzulisten. Optimal wäre eine Aufbereitung der Daten, sodass der Nutzer diese möglichst ohne weitere Manipulation in seinen Workflow übernehmen kann. Obwohl das Zielmedium des Nutzers bekannt ist, zeigt sich hier das Problem, dass innerhalb dieser Zielmedien weiterhin verschieden Tools verwendet werden können, die eine unterschiedliche Aufbereitung der Daten erfordern. Beispielsweise kann bekannt sein, dass der Nutzer eine Webandwendung entwickelt und das Styling für seine texte in CSS vornimmt. Trotzdem kann der Nutzer zum Beispiel verschieden Preprozessoren wie SCSS, SASS oder LESS verwenden, die alle eine unterschiedliche Syntax verwenden. Es liegt dabei durchaus im Rahmen des Möglichen, diese Informationen vom Nutzer zu erhalten und die Daten entsprechend aufzubereiten, jedoch liegen diese Anforderungen außerhalb des Zeitlichen Rahmens dieser Abschlussarbeit.

Hier wird dem Nutzer daher zunächst eine PDF zur Verfügung gestellt. Ein exemplarischer

Aufbau kann ABB. XYZ entnommen werden.

2.2 Diskussion verfügbarer Technologien

Im Nachfolgenden Kapitel sollen mögliche Technologien diskutiert werden, die für die Umsetzung der Abschlussarbeit genutzt werden können. Für die Umsetzung einer Webanwendung bieten sich eine Vielzahl von Programmiersprachen und Frameworks an. Diese können durch die geplante Struktur der Anwendung jedoch bereits weiter eingegrenzt werden. Da die Anwendung keine persistente Datenhaltung implementiert und auch keine übermäßig aufwendigen Berechnungen durchgeführt werden müssen, kann auch ein klassisches Client-Server-Modell verzichtet werden. Aufgrund des hohen Grades der Interaktivität der Anwendung kann außerdem davon ausgegangen werden, dass in jedem Falle auf JavaScript zurück gegriffen werden muss.

In den letzten Jahren wurden viele JavaScript-Frontend-Frameworks veröffentlicht, die genau auf die Anforderung der gestiegenen Interaktivität im Browser reagieren. Im Praxisprojekt wurde bereits versucht, den Proof of Concept nur mithilfe des Frameworks jQuery umzusetzen. Hier wurde schnell deutlich, dass die Komplexität der Anwendung den Rahmen von jQuery übersteigt.

Im folgenden sollen also einige der bekanntesten JavaScript-Frontend-Frameworks verglichen werden. Die Auswahl dieser Frameworks erfolgt nach Beliebtheit auf GitHub.com. Weiterhin wurden Frameworks ausgeschlossen, die ein komplettes MCV-Pattern implementieren, da zum aktuellen Stand dieser Anwendung keine Models angemacht sind.

2.2.1 Vue.js

Zuerst wurde Vue.js im Dezember 2013 veröffentlicht. Mittlerweile ist Vue.js in der zweiten Version zugänglich. Vue beschreibt sich selbst in der Einführung der Documentation wie folgt:

> Vue [...] is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is very easy to pick up and integrate with other libraries or existing projects. [JS17a]

Vue.js ist dabei (wie die anderen hier behandelten Libraries und Frameworks auch) Komponentenbasiert. Um das User Interface effektiv zu aktualisieren, verwendet es eine Virtual DOM. Dieser ist eine Abstraktion des Normalen DOMs, wie ihn das w3c spezifiziert [Rob17]:

> The Document Object Model (DOM) is a programming API for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

Der DOM im klassischen Sinn ist also eine Baumdarstellung der Elemente, die sich beispielsweise in einer HTML-Datei wiederfinden. Abb. XYZ zeigt eine beispielhafte Darstellung einer HTML-Datei im Browser, die Struktur ihrer Elemente und die Darstellung als DOM-Baum. Der Virtual DOM wird von Frameworks wie Vue.js dafür verwendet, Änderungen am DOM performances durchführen zu können. Die genaue Funktionsweise sei im Rahmen dieser Arbeit nicht erläutert, jedoch beschreibt die Vue.js Dokumentation die Vorgänge wie folgt [JS17b]:

> Under the hood, Vue compiles the templates into Virtual DOM render functions. Combined with the reactivity system, Vue is able to intelligently figure out the minimal amount of components to re-render and apply the minimal amount of DOM manipulations when the app state changes.

Für das Darstellen von Inhalten verwendet Vue.js eine Templating-Engine, Erweiterungen wie Routing oder Verwaltung des Zustandes der Anwendung müssen über externe Bibliotheken eingefügt werden.

2.2.2 Angular.js

Das von Google Entwickelte Angular.js ist das älteste der hier behandelten Frameworks. Es wurde zuerst im Oktober 2010 veröffentlicht und ist ebenfalls in der 2. Version angelangt.
Insgesamt ist Angular deutlich komplexer als Vue.js und React. Dies zeigt sich zum Einen beispielsweise durch die inkludierung von Routing (wenn auch nicht im angular core package), zum Anderen aber auch die die deutlich Komplexere Architektur, die unter anderem aus Teilen wie Modulen, Komponenten und Templates besteht. Auch hier wird also eine Templating-Engine verwendet.

2.2.3 React.js

React.js wurde im Juli 2013 von Facebook veröffentlicht.
Wie Vue.js verwendet auch React.js einen Virtual DOM. Auch wenn die Implementierung sich hier unterscheiden sollte, ist das Prinzip das Gleiche. Anders als Vue.js und Angluar.js verwendet React keine Templating Engine, React Applikationen sind daher als komplett in JavaScript geschrieben. Hier wird also HTML in JavaScript geschrieben, und nicht JavaScript in HTML, wie es im klassischen Sinne geschieht. Ähnlich wie Vue.js ist auch React.js in seiner Funktionalität sehr komprimiert, Funktionen wie Routing oder Application State müssen also über externe Bibliotheken eingebunden werden.

2.3 Einstieg in React.js

Der Folgende Abschnitt soll einen überblick über wichtige Konzepte und Vorgehensweisen bei der Entwicklung mit React.js geben. Ziel soll es dabei sein, genug Wissen zu vermitteln, um die später in dieser Arbeit gezeigten Codebeispiele verstehen zu können.
Dieser Abschnitt wird auf das Konzept von Komponenten im Allgemeinen sowie deren Verwendung in und in React.js eingehen und einige Grundlagen in der Auszeichnungssprache JSX vermitteln. Weiterhin werden die Übertragung von Daten und Zuständen innerhalb von React Komponenten behandelt.

2.3.1 Komponenten

Der Komponentenbasierte Aufbau ist eines der Hauptaugenmerke von React.js. Komponenten werden auf der offiziellen React.sj-Website [1]als der wichtigsten Merkmale aufgeführt und Gackenheimer [Gac15, S. 28] nennt sie einen der Hauptbestandteile einer React-Anwendung.
Dieser Abschnitt beschäftigt sich mit dem Aufbau, der Verwendung, den verschiedenen Formen und den Besonderheiten von Komponenten in React.
Natürlich kann und soll es nicht Ziel sein, einen umfassenden überblick über alle Informationen zu geben, die mit Komponenten in React in Verbindung stehen. Daher soll im Folgenden vorrangig auf in der Umsetzung als am relevantesten empfundene Aspekte eingegangen werden.

2.3.1.1 Was ist eine Komponente?

Bevor auf die Besonderheiten von Komponenten in React.js eingegangen wird, soll im Folgenden kurz ein Überblick über das Konzept der Komponente an sich gegeben werden.

Das Oxford Dictionary definiert eine Komponente als

> A part or element of a larger whole [...]

Gerade im Bezug auf Softwareentwicklung werden Komponenten aber noch einige weitere Eigenschaften zugeschrieben. So schreiben [Szy02] über Komponenten in der Softwareentwicklung:

> One thing can be stated with certainty: components are for composition. [...] Composition enables prefabricated "things" to be reused by rearranging them in ever-new composites.

Das Hauptaugenmerk bei der Entwicklung einer Komponente in der Softwareentwicklung sollte also auf der Möglichkeit der Wiederverwendbarkeit innerhalb der Anwendung liegen. Dabei sollte der Ort für die Wiederverwendung keine Rolle spielen. Eine Komponente sollte also unabhängig von ihrer Umgebung die gleichen Ergebnisse liefern [DAH01]:

> It does not constrain the environment but describes the behavior of the component in an arbitrary environment: "for all inputs x and y, if y $\neq$ 0, then the output is z = x/y"

2.3.1.2 Komponenten in React.js

Jede Komponente in React ist eine Subklasse der Basisklasse `React.Component` [Inc16b]. Konzeptuell besitzt jede Komponente in React einen Lifecycle. Die offizielle Dokumentation [Inc16b] beschreibt in diesem Lifecycle drei Zustände:

1. Mounting
2. Updating

[1] `https://facebook.github.io/react/`

3. Unmounting

Mounting beschreibt dabei den Vorgang des erstellen der Komponente und des einfügens in den DOM. Funktionen, die beim Mounting aufgerufen werden, werden also nur ein einziges mal im gesamten Lifecycle aufgerufen (die Ausnahme bildet dabei die Funktion `render()`, die auch beim Update Ereignis aufgerufen wird).

Ein **Update** wird durch die Veränderung von `props` oder `state` herbeigeführt. Der Auslöser für ein Update kann dabei also sowohl von der Komponente selbst, als auch von einer anderen Komponente kommen, die diese Komponente aufgerufen hat.

Unmounting wird unmittelbar vor dem entfernen der Komponente aus dem DOM aufgerufen.

Abbildung XYZ zeigt eine Schematische Darstellung des Lifecycle einer React Komponente.

Für jeden dieser verschiedenen Zustände bietet die Basisklasse `React.Component` verschiedene Funktionen an, die überschreiben werden können um diese zu verwenden. Die Funktionen werden dabei entweder vor oder nach dem jeweiligen Ereignis im Lifecycle aufgerufen. Die einzelnen Funktionen sollen im Folgenden nicht im einzelnen erläutert werden, da diese ausreichend Dokumentiert sind.

Für das Arbeiten mit Komponenten in React ist weiterhin das Konzept der `props` und des `state` relevant. `props` sind Daten, die von einer Komponente zur anderen übergeben werden können. Diese können beliebige JavaScript Objekte oder Primitive sein, somit können auch Referenzen auf Funktionen and Kind-Komponenten übergeben werden. Die Kind-Komponente kann dann via `this.props.propName` auf die ihr mitgegebenen props zugreifen. Listing XZY verdeutlich dieses Prinzip

```
1  class Parent extends REact.Component {
2    calculateSomething() {
3      // Claculates something
4      return result
5    }
6
7    render() {
8      let result = this.calculateSomething()
9
10     return (
11       <Child value={result} />
```

```
12      )
13    }
14  }
15
16  class Child extends React.Component {
17    render() {
18      return (
19        <div>{this.props.value}</div>
20      )
21    }
22  }
```

Das Prinzip des `state` wird im Kapitel Stateless & Stateful auf Seite 17 näher erläutert.

2.3.1.3 Stateful & Stateless

Komponenten in React.js können entweder Stateful oder Stateless sein. Wie der Name bereits vermuten lässt, haben diese Komponenten entweder einen state, oder nicht.
Der state in React.js beschreibt den Zustand eine Komponente. Hier findet sich auch eines der Hauptfeatures von React.js: Ändert sich der state einer Komponente, so wird diese Komponente neu gerendert, also auch das User Interface aktualisiert um den neuen Zustand darzustellen. Der state kann (und sollte) dabei durch das Aufrufen der Funktion `setState()` geändert werden. Das nachfolgende Codebeispiel zeigt eine simple Komponente, die clicks auf einen Button zählt:

```
1   class ClickCounter extends React.Component{
2     constructor(props) {
3       super(props)
4
5       this.state = {clicks: 0}
6     }
7
8     handleClick() {
9       this.setState((prevState) => {
10          return {clicks: prevState.clicks + 1};
11      });
12    }
13
```

```
14      render() {
15        return (
16          <button onClick={this.handleClick}>Click me!</button>
17          {this.state.clicks}
18        )
19      }
20    }
```

Hier passiert folgendes: Im Konstruktor der Komponente (der nur ein mal, beim initialen rendern aufgerufen wird) wird die initiale Anzahl der Clicks im state auf 0 gesetzt. Die Komponente rendert ein `<buttton>` element und die Anzahl der gezahlten Klicks. Wird der Button geklickt, wird die Funktion `handleClick()` in der Komponente aufgerufen. Diese Funktion erhöht die Anzahl der Klicks im state um eins. Durch den aktualisierten state wird auch die Komponente neu gerendert, so dass nun auch die neue Klickzahl angezeigt wird.

Die Komponente im obigen Beispiel ist also Stateful. Wie schon früher in diesem Kapitel erwähnt, ist es aber Ziel der Komponentenbasierten Softwareentwicklung, wiederverwendbare Komponenten zu schreiben.
Diese Komponente ließe sich an jeder Stelle der Anwendung wiederverwenden, an der Klicks über einen Button gezählt werden müssen, jedoch ist die Wahrscheinlichkeit hoch, dass diese Anwendungsfall eher selten vorkommt. Außerdem ist die Wahrscheinlichkeit recht hoch, dass ein Button oder ein Element zur Darstellung von Werten in der Anwendung häufiger verwendet werden (natürlich würde eine einfache Darstellung wie im obigen Beispiel kein Sinn ergeben, für dieses Beispiel soll daher eine komplexere Darstellung der Daten angenommen werde, beispielsweise durch eine Progress-Bar). Eigentlich liegen hier also drei Komponenten vor: Eine Komponente, die Logik enthält und zwei weitere, die lediglich Daten darstellen, somit also stateless sind.

Mit dem release von React 0.14[2]wurde eine simplifizierte Schreibweise für stateless components eingeführt. Die beiden Komponente aus dem Beispiel könnten somit wie folgt definiert werden:

```
1   export default function Button = (props) => {
```

[2]https://facebook.github.io/react/blog/2015/09/10/react-v0.14-rc1.html

```
    return (
      <button onClick={props.onClick}>Click me!</button>
    )
  }
```

und

```
export default function ValueDisplay = (props) => {
  return (
    <div>{props.value}</div>
  )
}
```

Die Werte für diese Komponenten werden ihnen in der Oberkomponente als `props` übergeben:

```
class ClickCounter extends React.Component{
  constructor(props) {
    super(props)

    this.state = {clicks: 0}
  }

  handleClick() {
    this.setState((prevState) => {
        return {clicks: prevState.clicks + 1};
    });
  }

  render() {
    return (
      <Button onClick={this.handleClick} />
      <ValueDisplay value={this.state.clicks} />
    )
  }
}
```

Ziel muss es also sein, so viele Komponenten wie möglich so klein wie möglich, optimaler Weise mit nur einer einzigen Aufgabe zu schreiben, um die Wiederverwendbarkeit zu erhöhen.

Hier irgendwo eine Referenz auf Beispielhafte Verwendung von Komponenten in der Anwendung

2.3.2 JSX

Hier soll nicht tiefer auf die Kompilierung und Prozess von JSX eigegangen werden, jedoch ist JSX ein elementarer Teil von React-Anwendung und sei der Vollständigkeit halber hier erwähnt. Im folgenden soll vorrangig auf die Unterschiede zwischen JSX und HTML und die Besonderheiten und Pitfalls während der Entwicklung mit JSX eingegangen werden.

JSX ist eine (eigens für die Verwendung mit React entwickelte) Syntax-Erweiterung für JavaScript, die es Erlaubt, HTML-Ähnliche Tags in der `render()`-Methode von React Komponenten zu verwenden (der Einsatz von JSX konnte bereits in den vorhergehenden Beispielen beobachtet werden). JSX ist dabei lediglich eine syntaktische Verschönerung der Funktion `React.createElement(component, props, ...children)` [Inc16a], auch wenn die Syntax der HTML-Syntax ähnlich sieht, werden alle JSX-Elemente in die genannte JavaScript-Funktion kompiliert.

JSX unterstütz die Deklaration von regulären HTML-Elementen, als auch die von React-Komponenten. Unterschieden werden die beiden Varianten dabei nach Groß- und Kleinschreibung, wobei React-Komponenten groß- und reguläre HTML-Elemente klein geschrieben sind. React-Komponenten können dabei, wie oben bereits erwähnt, `props` mitgegeben werden. Der Schritt von HTML zu JSX stellt in der Entwicklung keine große Herausforderung dar. Der wichtigste Punkt ist die Verwendung von `class`. Auch bei der Entwicklung mit React werden für DOM-Elemente häufig Klassen vergeben. Da JSX-Code aber in JavaScript kompiliert wird, kann das in JavaScript reservierte Wort[3] `class` nicht verwendet werden. Es muss stattdessen auf `className` zurück gegriffen werden.

2.4 Einstieg in Redux

Redux[4]erlaubt die zentrale Verwaltung des state bzw. des Zustandes der Anwendung im sogenannten redux store. Dabei ersetzt der redux store nicht zwangsweise den state von einzelnen Komponenten. Es gibt keine genauen Richtlinien dafür, wann ein state in der Komponente und wann im redux store verwaltet werden sollte, jedoch bietet es sich als Richtlinie an, nur

[3] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar`

Zustände im redux store zu verwalten, die von mehr als nur einer einzigen Komponente verwendet werden. Redux besteh dabei aus drei Grundbestandteilen: Dem *Store*, den *Actions* und den *Reducern*.

Wie bereits erwähnt wird im Store der Zustand der Anwendung verwaltet. Der *Store* an sich kann dabei ein beliebiges JavaSCript Objekt (Funktionen ausgenommen) oder Primitiv sein. Für diesen *Store* hat die Anwendung nur Leserecht, er darf nicht direkt manipuliert werden. Für eine Änderung des Stores werden die anderen beiden Bestandteile benötigt.

Actions beschreiben Aktionen und sind JavaScript Objekte. Sie müssen mindestens einen `type` Parameter haben, der beschreibt, welche Aktion ausgeführt werden soll. Weiterhin können sie Daten definieren, die für die Änderung des stores von Bedeutung sind. Das absenden dieser actions ist der einzige weg, mit dem die Anwendung den *Store* verändern kann.

Reducer definieren, wie der Zustand der Anwendung je nach erhaltener Aktion verändert werden muss. Ist beispielsweise der `type` einer *Action* `INCREMENT_DOWNLOADS_COUNT` kann im *Reducer* definiert sein, dass beim erhalten dieser action das Feld `downloadsCount` im *Store* um eins erhöht werden muss.

Ein *Provider* macht den *Store* dann für Container-Komponenten verfügbar. Diese verteilen den *Store* und die actions als props an representative Komponenten. Ändert sich der *Store* erhalten die entsprechen betroffenen Komponenten also neue *props* und werden somit neu gerendert. Abb XYZ zeigt einen schematischen Ablauf einer Aktualisierung des redux stores.

Detailliertere Erläuterungen zur Implementation von redux im Rahmen der Anwendung finden sich in Kapitel ABC.

2.5 Komponentenbasierte Gestaltung

Bei einer so stark komponentenbasierten Entwicklung ergibt auch eine Anpassung des Design-Prozesse an diese Strategie Sinn. Dabei ist jedoch ein rein komponentenbasierter Designprozess nicht die Optimale Lösung, denn Aspekte wie Interaktion im Gesamtkontext der Anwendung und auch gestalterisches Zusammenspiel der verschiedenen Komponenten müssen getestet werden. Daher werden zunächst Wireframes mit Hilfe des Tool UXPin[5]erstellt, um die allgemeine

[4] `http://redux.js.org/`

Struktur der Anwendung zu entwerfen. Auf basis dieser Wireframes können dann bereits einzelne Komponenten definiert werden, die anschließend Gestaltet werden können. Jedoch muss die endgültige Gestaltung dieser Komponenten im Gesamtverbund einer Seite erfolgen, um garantieren zu können, dass die Komponenten untereinander harmonieren.

Bei einer Komponentenbasierten Gestaltung (und gerade auch mit Blick auf die Weiterentwicklung durch die Community) bietet sich außerdem immer ein Styleguide an, der einen überblick über die verschiedenen Komponenten gibt. Dieser Styleguide soll in diesem Projekt in Form von Storybook[6]abgebildet werden. Storybook bietet dabei nicht nur die Möglichkeit, einen überblick über die vorhandenen Komponenten zu bilden, sonder erlaubt es auch, den Code direkt aus der Anwendung einzubinden. Das heißt, es muss nicht extra für den Styleguide doppelter Code beschreiben werden. Weiterhin können Komponenten zunächst in Storybook entwickelt werden, der Fokus kann also rein auf der zu entwickelnden Komponente liegen und es muss nicht auf die umliegende Anwendung geachtet werden.

[HIER NOCH BILD]

[5]urlhttps://www.uxpin.com/

[6]urlhttps://storybook.js.org/

Kapitel 3

Entwicklung der Anwendung

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

3.1 Gestaltung

Generell nimmt die Gestaltung, wie bereits im Kapitel “Relevanz” deutlich wurde, eine zentrale Rolle in allen Projekte ein. Auch dieses Projekt bildet keine Ausnahme. Um eine gute Benutzbarkeit des Tools zu gewährleisten, ist eine solide Gestaltung unabdingbar.

Insgesamt ist die Anwendung eine sehr interaktive, in der Informationen eher Grafisch und über Interaktionen übertragen werden, als Beispielswiese über Text. Daher wurde hier bei der Gestaltung großer Wert darauf gelegt, klar zu kommunizieren, welche Bereiche interaktiv sind und welche Auswirkungen diese haben.
Weiterhin liegt hier der Fall vor, dass der Nutzer die Anwendung verwendet, um eine Gestaltung für sein Projekt zu erstellen. Der Fokus der Anwendung sollte daher, auch in der Gestaltung, darauf liegen, dem Nutzer die von ihm erstellte Gestaltung darzustellen. Die Anwendung sollte sich nicht un bestimmten Fällen (beispielsweise beim Auftreten eines Fehlers oder wenn eine

Interaktion notwendig ist) in den Vordergrund stellen.

Dieses Vorgehen lässt sich anhand von zwei Beispielen gut verdeutlichen. Zum Einen seien hier die Fehlermeldungen im Bereich Typographie genannt. Diese sind in einem Auffälligen Gelb hinterlegt, dass nur an dieser Stelle (im Sinne von: Zum anzeigen von Fehlern) verwendet wird und dem Nutzer dafür schnell auffällt (s. Abb. ZXY). Zum Anderen zeig die Auswahl von Farben gut, wie Interaktive Elemente der Anwendung gleichzeitig auch die Gestaltung des Nutzer zeigen können. Die Buttons zur Auswahl einer Farbe (s. Abb. ASH) bestehen hier lediglich aus der Farbe selbst, die Gestaltung der Anwendung wird hier als visuelle Zwischenebene also komplett entfernt.

Um die Design-Sprache der Anwendung nicht in den Vordergrund zu stellen wurde ein sehr schlichtes Farbschema verwendet, in dem die zwei Farben (Blau und Rot) nur zur Anzeige von Interaktivität verwendet wurden. Der größte Teil der Anwendung ist weiß gehalten, um sie die Gestaltung des Nutzers und dessen Entscheidungen besser in den Vordergrund stellen zu können.
Die beiden Farben haben dabei festgelegte Rollen: Blau zeigt Interaktivität innerhalb eines Schrittes der Anwendung an, Rot zeigt zwischen Schritten übergreifende Aktivitäten an (z.B. ein Button, der von einem Schritt zum nächsten führt).

3.2 Besonderhieten im Technologie-Stack

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

3.2.1 Redux

- Trennung in verschiedene Sub-Stores

- Container
- Kein direktes dispatch in Components mehr

3.2.2 React Storybooks

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

3.2.3 CSS-Architektur

Für den Umgang mit CSS in React.js bieten sich verschiedene Möglichkeiten an. Nativ sind zwei Vorgehensweisen möglich: Anwenden von CSS über ausgelagerte Stylesheets (wie es in der Regel bei allen Webseiten gemacht wird) oder das verwenden von Inline-Styles.

Das verwenden von klassischen Stylesheets unterscheidet sich nicht sehr von der Verwendung bei einer statischen Webseite. Auch in React werden Klassennamen oder IDs festgelegt, über die dann im Stylesheet das Aussehen definiert wird (natürlich sind auch alle anderen validen CSS Selektoren anwendbar, Klassen und IDs seine hier nur als die populärsten Varianten beispielhaft genannt). Auch die Verwendung von CSS-Präsporzessoren wie zum Beispiel SCSS stellt kein Problem dar, die kompilierung dieser Dateien muss lediglich in den Build-Prozess von React.js mit eingebunden werden. Eine simplifizierte Anwendung sähe beispielsweise wie folgt aus:

```
1 <ReactComponent className='myClass'>
2  Content
3 </ReactComponent>
```

Die kompilierte Dom-Node wäre dabei

```
1   <ReactComponent class='myClass'>
2    Content
3   </ReactComponent>
```

Und könnte im Stylesheet über

```
.myClass {
  color: red
}
```

Angesprochen werden. Auch die Verwendung von Methodiken wie BEM oder SMACCS ist mit diesem Ansatz ohne Probleme möglich. Für den Rahmen dieses Projektes wurde sich jedoch gegen diese Vorgehensweise entschieden, da bewusst ein Fokus auf eine komponentenbasierte Architektur gelegt werden sollte. Auch mit einer komponentenbasierten CSS-Architektur ist eine Komponente immer noch auf zwei Orte aufgeteilt: Die Funktion und das Markup, und das Styling.

[HIER LIESSE SICH AUCH DIE DISKUSSION NOCH AUSFÜHREN, DASS EIGENTLICH LEUTE LANGE DAFÜR GESPROCHEN HABEN, AUSSEHEN UND MARKUP ZU TRENNEN UND WARUM DAS HIER OKAY IST]

Um das Styling und die Funktion einer Komponente an einem Ort zu halten, bieten sich inline-styles an. Wie auch bei statischen Webseiten werden inline-styles direkt im `style`-Attribut eines Elementes definiert. In React.js werden diese als JavaScript-Objekt übergeben. Eine Beispielhafte Anwendung findet sich in Quellcode XYZ

```
const styles = {
  backgroundColor: 'red',
  fontSize: '12px'
}

export default function myComponent(props) {
  return (
    <div styles={styles}) >
    {props.children}
    </div>
  )
}
```

Auf den ersten Blick wirkt die Verwendung von inline-styles problematisch, vielleicht weil diese auf statischen Seiten viele Nachteile mit sich bringen. In einem Komponentenbasierten System wie React.js sind diese Nachteile jedoch nicht present. Durch die Komponentenbasierte Struktur und das damit einhergehende Ziel der Wiederverwendung von Komponenten, müssen Stylingänderungen auch hier nur an einer Stelle vorgenommen werden. Da jede Komponente

nur für ihr eigenes Styling verantwortlich ist und nicht für das von Kindern, und auch kein CSS außerhalb der inline-styles verwendet wird (abgesehen von CSS-Reset und einigen globalen Regeln wie der Schriftart), kann es zu keinen Problemen mit der Spezifität von CSS-Regeln kommen.

Allerdings weisen inline-styles einige Limitierungen auf. So können beispielsweise keine Pseudo-Elemente wie `:after` oder `:before` verwendet werden. Auch das definieren von hover-states via `:hover` wird nicht unterstützt.

[HIER NOCH VERWEIS AUF SPEZIFIKATION]

Zwar sind diese Limitierung zum aktuellen Stand des Projektes noch nicht von allzu großer Bedeutung, mit Blick auf eine Weiterentwicklung der Anwendung nach der Abschlussarbeit können diese in Zukunft jedoch eine größere Rolle spielen. Um diese Limitierungen zu umgehen, wurde das Framework Aphrodite verwendet. Das Framework erlaubt das Festlegen von styles innerhalb der Komponente ähnlich wie inline-styles, erzeugt aber für jedes neue style-objekt eine einzigartige CSS-Klasse, die via `className` angewendet wird. Die Einzigartigkeit der Klasse wird durch das hinzufügen eines Hauses am Ende des Klassennamens gewährleistet. Listing XYZ zeigt ein Beispiel

```
1  import React from 'react'
2  import { StyleSheet, css } from 'aphrodite'
3
4  function myComponent(props) {
5    <div className={css(styles.componentStyles)} >
6      {props.children}
7    </div>
8
9    const styles = StyleSheet.create({
10     componentStyles: {
11       color: 'blue'
12     }
13   })
14 }
```

Die Struktur der style-objekte ist dabei der von inline-style-objekten gleich.

Mit der Verwendung von Aphrodite können also Aussehen und Funktion von Komponenten in der gleichen Datei gehalten werden, ohne dabei den Limitierungen von inline-styles zu unter-

liegen.

3.3 Einstieg

Auf die Konzeptionellen Aspekte des Einstiegs in die Applikation wurde bereits im Kapitel Einstieg in die Anwendung [VERWEIS] eingegangen. In diesem Abschnitt soll ergänzend auf einige konkrete Aspekte der Gestaltung und Entwicklung eingegangen werden.

3.3.1 Gestaltung

Bereits im Kapitel Einstieg in die Anwendung [VERWEIS] wurde angesprochen, dass ein schrittweises Führen des Nutzers durch das Festlegen seines Entwicklungskontextes sinnvoll ist. Hierfür wurde im ersten Schritt ein Wizard entworfen, der den Nutzer schrittweise durch die Knoten der verschiedenen Ebenen des Baumes in ABB XYC auf Seite ABC [VERWEIS] führt. Hierbei wurden Buttons mit Icons und Beschreibung verwendet, um eine Unterscheidung zwischen den verschiedenen Optionen für den Nutzer so einfach wie möglich zu machen [BELEG]. Am Ende des Wizards wird dem Nutzer der von ihm gewählte Entwicklungskontext noch einmal zusammengefasst dargestellt. Außerdem wird ihm die Option geboten, den Vorgang zu wiederholen, sollten die Ergebnisse nicht den gewünschten entsprechen. Zum jetzigen Zeitpunkt ist diese Funktion nicht unbedingt eine Notwendige, jedoch ist nicht ausgeschlossen, dass die Entwicklungskontexte zu einem späteren Zeitpunk zahlreicher oder tiefer verschachtelt werden. Der Gestaltete Wizard kann ABB XZY entnommen werden [BILD FEHLT].

Hier wird auch die Verwendung der beiden Farben in der Applikation deutlich: Das Blau verändert Werte innerhalb eines einzelnen Schrittes, das Rot dient zu Navigation zwischen den verschiedenen Schritten der Applikation.

3.3.2 Entwicklung

Das Hauptaugenmerk in der Entwicklung lag in diesem Schritt im setzen der Scopes für die Anwendung. Da auf die Scopes in allen späteren Teilen der Anwendung zugegriffen werden können muss wurde schnell deutlich, dass diese im Application Sate, also dem Redux Store ge-

halten werden müssen. Hier stellte sich jedoch vor allem die Fragen nach der Datenstruktur der Scopes, wobei die Möglichkeiten auf die Haltung in einem Array oder als JavaScript Objekt begrenzt wurden. Ein Array bietet dabei eine höhere Flexibilität, jedoch eine deutlich ungenauere Beschreibung der Scopes. So können andere Komponenten herausfinden, ob der für ihren Anwendungsfall relevanten Scope sich im Array der Scopes befindet, ohne Kenntnis darüber haben zu müssen, wie der Schlüssel des Scopes innerhalb des JavaScript Objektes lautet. Auch spielt die Länger der verschiedenen Äste des Baumes bei der Verwendung des Arrays eine weniger große Rolle, während bei der Verwendung eines Objektes immer auch geprüft werden muss, ob der key im store überhaupt vorhanden ist. Daher wurde an dieser Stelle ein Array verwendet. [HIER SPÄTER NOCH MAL SCHAUEN, OB DAS NOCH STIMMT]

An dieser Stelle wird außerdem der in Kapitel Stateful & Stateless [VERWEIS] angesprochene Unterschied zwischen Component State und Application State deutlich. Die Interaktion des Nutzers mit dem Wizard findet wie folgt statt: Der Nutzer wählt einen Scope aus, dieser wird von der Anwendung hervor gehoben. Mit einem Klick auf den “Next”-Button wird die Selektion gespeichert und darauf aufbauend neue Optionen angezeigt. Abb XYZ [BILD FEHLT] zeigt noch einmal den Aufbau des Wizards in Verbindung mit den beteiligten Komponenten. Dabei ist nur der SetupContainer [NAME STIMMT NICHT] stateful, dieser kümmert sich um die Anzeige des vom Nutzer ausgewählten Scopes, der im state dieser Komponente gespeichert ist

```
1  constructor(props) {
2    super(props)
3
4    // shortended for readability
5
6    this.state = {
7      activeOption: false // currently active option, 'false' by default
8  }
```

Da die aktuell gewählte Option nur für diese Komponente von Bedeutung ist, wird diese auch nicht in den Application State geschrieben. Bestätigt der Nutzer die Auswahl jedoch mit dem “Next”-Button, wird der Wert des aktuell aktiven Elements in den Store geschrieben.

3.4 Typographie

Wie bereits erwähnt wurde für den Bereich Typographie bereits im Rahmen des Praxisprojektes ein Proof of Concept entwickelt, der weite Teile der erarbeiteten Logik implementierte. Im Folgenden soll daher vor allem auf die Bereiche eingegangen werden, die im Vergleich zum Praxisprojekt neu oder verändert sind. Vorrangig sind dies die Gestaltung und generelle Konzeption, sowie eine bessere Implementierung des Redux Stores und von React-Komponenten an sich.

3.4.1 Konzeption

Das Grundlegende Konzept zur Interaktion mit der Anwendung wurde aus dem Praxisprojekt übernommen, auch im Rahmen der Abschlussarbeit ist dieser Teil der Anwendung zweigeteilt: Zum einen wird das Ergebnis der Einstellungen in Form eines gesetzten Textes angezeigt, zum anderen werden verschiedene Möglichkeiten zur Manipulation des Textes in einer Tab-Navigation angezeigt.
Eine Andere Idee hier war die Möglichkeit zur direkten Interaktion mit Elementen. So hätte beispielsweise eine Überschrift angeklickt und deren Attribute daraufhin editiert werden können (Abb. ABC zeigt ein Beispiel). Gegen diesen Ansatz wurde sich aus verschieden Gründen entschieden.
Zum Einen wären für allgemeinere Einstellungen (wie z.B. die Wahl der Schriftart oder die Einstellung zur Laufweite des Textes gesonderte Bedienelemente nötig gewesen. Dem wurde eine geordnete Darstellung der Bedienelemente an einer zentralen Stelle vorgezogen. Damit einher wäre das Problem einer übersichtlichen Fehleranzeige gegangen: Es stellte sich als kompliziert dar, deutlich zu machen, zu welchem Attribut ein Fehler zugehörig ist.

Eine Konzeptuelle Neuerung stellt der Reset-Button dar. Dieser erlaubt ein einfaches zurücksetzen auf die Standardeinstellung und nimmt es dem Nutzer damit ab, im Zweifel 10 oder mehr Werte verändern zu müssen, um auf eine gute Einstellung zurück zu kommen.

Im Bezug auf die Standardeinstellungen stellt sich die Fragen: Welche Einstellungen sind hier zu bevorzugen? Es wurde sich bewusst gegen eine fehlerfreie Standardeinstellung entschieden,

da es dem Nutzer nicht möglich ist, in der Anwendung einen Schritt vorwärts zu gehen, wenn noch Warnungen angezeigt werden. So kann sicher gestellt werden, dass der Nutzer sich auf jeden Fall mit dem Bereich Typographie befasst.

[EVTL DARK PATTERN?]

Auf der anderen Seite sollten die Standardeinstellung nicht zu viele Fehler enthalten, um den Nutzer nicht zu demotivieren.

Weiterhin wurde auf mobile Geräte als Zielmedium eingegangen: Hier funktioniert die Rechnung der optimalen Zeilenlänge nach Anzahl der Wörter nicht mehr (wenigstens nicht in einem Rahmen, der einen Lesbaren Text liefern kann), außerdem sind die Schriftgrößen anders zu bewerten.

[HIER NOCH SCHAUEN, WELCHE RICHTLINIEN ES DA GIBT]

Andere Werte, wie zum Beispiel der Zeilenabstand bleiben davon aber unberührt. Weiter Informationen zur Umsetzung der verschiedenen Bereiche für verschiedene Zielmedien finden sich im Kapitel Entwicklung.

3.4.2 Gestaltung

Wie bereits erwähnt wurde das grundlegende Layout aus dem Praxisprojekt beibehalten, jedoch wurde die Anordnung verändert: Statt übereinander sind die beiden Bereiche nun nebeneinander angeordnet. Dies hat den Effekt, dass sowohl die Bedienelemente, als auch der größte Teil des gesetzten Textes in einem Viewport dargestellt werden können.

Während der Geltung wurde versucht, den Fokus auf den zu setzenden Text zu legen und die Bedienelemente eher in den Hintergrund zu rücken. Durch die dunklere Hintergrundfarbe sollte aber auch eine klare Trennung der beiden Element nach ihren Aufgaben (Output oder Input) kommuniziert werden (Abbildung ASD zeigt ein High-Fidelity-Mockup dieses Bereiches).

[BILD FEHLT]

Außerdem wurde als Einstieg in den Bereich Typographie eine statische Seite entworfen, die kurz die Funktion des Bereiches aufzeigt (ABB. XYV zeigt eine schematische Ablauf der Interaktion). Zwar ist die Funktion dieses Bereiches recht eindeutig und simpel, jedoch sollte es Ziel sein, den Nutzer so gut wie möglich während seiner Nutzung der Anwendung zu begleiten.

[BILD FEHLT]

3.4.3 Entwicklung

Die größte Neuerung in der Entwicklung im Vergleich zum Praxisprojekt war die Anwendung der Scopes, also der Zielmedien für die der Benutzer gestaltet. Hauptsächlich Interessant für den Bereich Typographie war dabei, ob der Nutzer für das Lesen am Bildschirm oder auf Papier gestaltet und das Mobile Betriebssystem, falls er eine Native Applikation entwickelt. Hauptsächlich wirken sich diese Scopes auf die wählbaren Schriftfamilien aus.

Die Umsetzung ist dabei recht simpel: Die verfügbaren Schriftfamilien sind als in einer Konstante gespeichert., sie sind dabei nach dem Namen ihres Scopes aufgeschlüsselt.

```
1  export const FONTS = {
2    DISPLAY: [
3      'Verdana', 'Arial', 'Tahoma', 'TrebuchetMS'
4    ],
5    RESPONSIVE: [
6      'Verdana', 'Arial', 'Tahoma', 'TrebuchetMS'
7    ],
8    NOT_RESPONSIVE: [
9      'Verdana', 'Arial', 'Tahoma', 'TrebuchetMS'
10   ],
11   PAPER_DISPLAY: [
12     'Verdana', 'Arial', 'Tahoma', 'TrebuchetMS', 'Times New Roman', '
         Georgia', 'Palatino'
13   ],
14   PAPER: [
15     'Times New Roman', 'Georgia', 'Palatino'
16   ],
17   ANDROID: [
18     'Roboto', 'Noto'
19   ],
20   IOS: [
21     'San Francisco'
22   ]
23 }
```

In der entsprechenden Komponente, die ein Dropdown-Menu mit den verschiedenen Optionen darstellt, wir dann einfach das zum aktuellen Scope passende Array ausgewählt:

```
determineFontFamilies() {
  let scope = this.props.scopes[1]
  return FONTS[scope]
}
```

Dieser Aufbau ermöglicht sowohl ein einfach hinzufügen und entfernen von Schriftfamilien, also auch das hinzufügen und entfernen von ganzen Scopes, was gerade mit Blick auf die Weiterentwicklung durch die Community interessant ist.

[HIER FEHLT NOCH DER MOBILE SCOPE]

Im Vergleich zum Praxisprojekt wurde weiterhin der Komponentenbasierte Aufbau verbessert. Währen im Praxisprojekt noch 21 Komponenten für die Darstellung verwendet wurden, konnte diese Zahl im rahmen der Abschlussarbeit auch 10 verringert werden.
[WIE KONNTE DAS ERREICHT WERDEN?]

Dies lässt sich beispielhaft an der Komponente aufzeigen, die verschiedene Möglichkeiten zur Manipulation von Attributen von Überschriften darstellt. In der Anwendung gibt es drei Überschriften verschiedener Ordnung, die in den Werten Größe, Abstand nach oben und Abstand nach unten verändert werden können. Im Praxisprojekt gab es hier für jede Überschrift verschiedener Ordnung eine eigene Komponente, im Rahmen der Abschlussarbeit konnte dies auf nur eine Komponente reduziert werden.
Der Hauptgrund hierfür ist das auslagern der Logik in andere Bereiche der Anwendung. So werden die Fehler nicht mehr innerhalb der Komponente errechnet, sondern lediglich dargestellt. Alle anderen Werte werden der Komponente beim Aufruf aus der Elternkomponente übergeben. Die prop area gibt dabei an, welchen Bereich des Stores diese Komponente verändert.

```
<HeadlineControls
  onChange={this.props.setValueInArea}
  area={'headline1'}
  title={'Headline 1'}
  componentErrors={errors.headline1}
  {...this.props.headline1}
/>
```

3.4.4 Ausblick

Auch wenn hier eine solide und Marktfähige Version dieses Bereiches erstellt werden konnte, wurden währen der Entwicklung noch weitere Bereiche deutlich, die in Zukunft potentielle Verbesserung bieten können.
So wäre eine bessere visuelle Darstellung, welche Bereich beim Manipulieren eines Werte verändert werden, wünschenswert. Aktuell wird dies zwar bei Veränderungen in großen Schritten schnell deutlich, bei kleineren Schritten ist die Darstellung jedoch nicht optimal.
Weiterhin kann eine bessere Umsetzung für den Scope Responsive erreicht werden, Hier werden in der Regel mehrere Versionen von gesetztem Text benötigt.

3.5 Farben

Der Bereich Farben bietet einige neue Aspekte in der Entwicklung, vor allem, weil dieser Bereich im Praxisprojekt nur in der Theorie definiert wurde. Hier standen also weder Code noch Struktur zur Verfügung, auf denen aufgebaut werden konnte (die Ausnahme bilden einige Wireframes. Der folgende Abschnitt erläutert die Gedankengänge hinter der Konzeption und einige interessante Aspekte, die während der Entwicklung auftraten.

3.5.1 Konzeption

Dieser Bereich wird sehr stark vom Entwicklungskontext des Nutzers beeinflusst, vor allem weil beide abgebildeten mobilen Betriebssysteme sehr genaue Vorgaben im Bereich der zu verwendenden Farben machen. Daher wurde dieser Bereich in zwei Schritten konzipiert: Das finden einer Grundfarbe und das finden einer Akzentfarbe.

Generell wurde die Idee aufgegriffen, Farben nach bestimmten Adjektiven wählen zu können. Dabei bietet sich, je nach Entwicklungskontext ein unterschiedlich vielfältiger Pool aus verfügbaren Farben. Für alle Scopes außerhalb der mobilen Betriebssysteme sind die Farben nicht begrenzt, es steht dem Nutzer also Frei, jede hexadezimal darstellbare Farbe zu wählen. Für die Auswahl nach Adjektiven wurde die Zahl der Farben jedoch auch etwa 20 [ZAHL NOCH GENAU NACH SCHAUEN] reduziert, um hier eine gut Übersicht erhalten zu können. Eine Übersicht der Adjektive und Farben bietet ABB. XZY [BILD FEHLT]

[HIER FEHLT NOCH: WIE KOMMEN DIE ADJEKTIVE ZUSTANDE?]

Die beiden mobilen Betriebssysteme sind in ihren Guidelines hier deutlich restriktiver. Android definiert 18 [ZAHL NOCH MAL NACHSEHEN] Farben in verschiedenen Farbtönen. Den Guidelines folgend werden dem Nutzer hier die Farben im Farbton 500 dargestellt, aus denen er wählen kann. In den Guidelines von iOS ist die Zahl der Farben mit 8 [ZAHL NOCH GUCKEN] am geringsten. In beiden Fällen konnte aber Trotzdem eine Auswahl nach Adjektiven, wenn auch limitierter, weiterhin angeboten werden.

Im zweiten Schritt soll dem Nutzer dann beim finden einer Akzentfarbe geholfen werden. Auch hier unterscheiden sich die Vorgänge je nach Entwicklungskontext. Im Bereich iOS fällt das finden einer Akzentfarbe ganz weg, das Apple eine Gestaltung mit nur einer Farbe (die in sich also eher eine Akzentfarbe als wie hier definiert eine Grundfarbe darstellt) nahe legt [VERWEIS]. Im Bereich Android gibt es in allen Farbtönen drei Abstufungen, die zur Verwendung als Akzentfarbe gedacht sind. Diese werden dem Nutzer hier zur Auswahl gestellt. In allen anderen Bereichen ist die Anzahl der Farben, wie schon im ersten Schritt, nicht eingeschränkt. Hier soll dem Nutzer die Möglichkeit geboten werden, Farbkombinationen in der Kontrasten Komplementär, Triadisch und Monochromatisch zu erstellen. Zu den ermittelten Farben werden dann weiterhin weiss und schwarz hinzugefügt. Für Android sind auch diese definiert, in allen anderen Bereichen (iOS ausgenommen) sind diese als `#ffffff` und `#333333` festgelegt.

3.5.2 Gestaltung

[TBD]

3.5.3 Entwicklung

[HIER KÖNNTE NOCH EIN BISSCHEN MEHR EINLEITUNG]

Da die Berechnung der Farben im HSL-Farbmodell erfolgen sollten, die Anwendung aber an allen anderen Stellen mit der Hexadezimal-Darstellung arbeitet, müssen die Farben zunächst umgerechnet werden. Für die Umwandlung wurde die Bibliothek tinycolor [FUSSNOTE] verwendet, die aufgrund anderer verwendeter Bibliotheken schon im Projekt eingebunden war. Die Bibliothek wandelt die Hexadezimaldarotellung in ein JavaScript-Objekt mit den Attri-

buten hue, saturation, lightness und alpha am. Somit können einzelne Werte dieses Objektes problemlos manipuliert und anschließend wieder in einen Hexadezimal-String umgewandelt werden.

Die Grundlegende Logik zum errechnen von bestimmten Kontrasten wurde bereits im Praxisprojekt definiert: Für einen Komplementärkontrast muss der hue-Wert um 180° verändert werden, für ein triadisches Farbschema werden die Werte jeweils um 30° nach links und rechts verändert und für ein Monochromatische Farbschema können der lightness- und saturation-Wert von einer Menge von Farben verändert werden. Im Folgenden sollen hier die konkreten Implementierungen dieser Berechnungen erläutert werden.

Die Berechnung der Komplementärfarbe erwies sich als recht einfach, hier musste lediglich darauf geachtet werden, dass der Hue-Wert nicht größer als 360° [VIELLEICHT AUCH 359?] werden durfte.

```
1  export function calculateCompelemntary(baseColor) {
2    let complementary = Object.assign({}, baseColor)
3    let hue = complementary.h
4
5    hue += 180
6    if (hue > 360) {
7      hue -= 360
8    }
9
10   complementary.h = hue
11   return complementary
12 }
```

Ähnlich gestaltete es sich bei der Berechnung des triadischen Farbschemas. Hier musste der Hue-Wert lediglich um den gegebenen Gradwert erhöht oder verringert werden. Um die Farbschemata dynamischer zu gestalten, wurde sich dazu entschieden, hier nicht statische Werte von 30° zu verwenden, sondern diese dynamisch in einem Rahmen von 25° bis 35° zu generieren.

Deutlich komplexer gestaltet die Generierung von monochromatischen Farbschemata. Zum einen waren hier mehr Farben nötig, zum anderen können diese in den Werten Saturation und Lightness verändert werden. Weiterhin steht hier nicht fest, ob beide oder nur einer der Werte verändert werden soll, und wie die Veränderung aussieht. Auch hier wurde im Sinne der Dynamik auf zufallsgenerierte Zahlen zurück gegriffen. In der Funktion wird zunächst zufällig

bestimmt, welche Werte verändert werden. Sowohl Saturation als auch Lightness können Werte zwischen 0 und 1 annehmen. Im zweiten Schritt wurden also eine oder zwei Zufallszahlen zwischen 0 und 1 generiert (gerundet auch zwei Dezimalstellen). Weiterhin muss gewährleistet werden, dass die einzelnen Farben sich nicht zu ähnlich sehen. Hier wurde eine Differenz von 0.1 als Minimum festgelegt (dabei reicht es, wenn einer der Werte außerhalb dieses Minimum liegt, um eine ausreichend hohe Differenz zwischen den Farben zu gewährleisten). Wird dieser Wert unterschritten, wird eine neue, zufällige Farbe generiert.

```
checkForSimilarColors(colors, candidate) {
  let similarValues = false
  for (var j = 0; j < colors.length; j++) {
    let saturationDifference = Math.abs(colors[j].s - candidate.s)
    let lightnessDifference = Math.abs(colors[j].l - candidate.l)

    if (saturationDifference < 0.1 && lightnessDifference < 0.1) {
      similarValues = true
    }
  }

  return similarValues
}
```

```
changeValuesOfColor(values, color) {
  let manipulatedColor = Object.assign({}, color)
  // Switch case
  // Do the changes that need to be DropdownController
  switch (values) {
    case 0:
      // change lightness
      manipulatedColor.l = Math.random().toFixed(2)
      break
    case 1:
      // change Saturation
      manipulatedColor.s = Math.random().toFixed(2)
      break
    case 2:
      // change lightning and saturation
      manipulatedColor.l = Math.random().toFixed(2)
      manipulatedColor.s = Math.random().toFixed(2)
```

```
        break
      default:
        throw new 'Oops, seems like the randomizer messed something up.'
    }

    return manipulatedColor
  }
```

```
calculateMonochromaticColors(amount) {

    let colors = []

    for (var i = 0; i < amount; i++) {
      let currentColor = Object.assign({}, convertToHsl(this.props.
          baseColor))
      // Figure out if only one or two values should be changed
      // Random: 0 = Lightness, 1 = Saturation, 2 = Both
      let randomOption = Math.floor(Math.random() * 3)
      let changedColor = this.changeValuesOfColor(randomOption,
          currentColor)

      if (colors.length < 1) {
        colors.push(changedColor)
      } else {
        let similarColors = this.checkForSimilarColors(colors, changedColor
            )
        while (similarColors) {
          changedColor = this.changeValuesOfColor(randomOption,
              changedColor)
          similarColors = this.checkForSimilarColors(colors, changedColor)
        }
        colors.push(changedColor)
      }
    }

    return colors
  }
```

[HIER FEHLT NOCH DER AUSSCHLUSS VON SCHWARZ UND WEISS]

3.6 Layouts & Grid

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

3.7 Offboarding

Wie im Kapitel 2.1.2 “Ergebnisse der Benutzung” bereits angesprochen, dient dieser letzte Schritt der Anwendung dazu, dem Nutzer eine Zusammenfassung der Ergebnisse zu liefern. Da alle vom Nutzer getroffenen Entscheidungen im redux store gespeichert werden ist es kein Problem, hier eine Zusammenfassung dieser Daten darzustellen. Diese Daten werden dem Nutzer in der aktuellen Version nach Themengebiet aufgeschlüsselt, untereinander dargestellt. Für die aktuell implementierten Gebiete und die daraus resultierende Menge an Informationen ist diese Lösung ausreichend, jedoch muss mit weiterer Entwicklung der Anwendung hier vermutlich auch ein anderer Lösungsansatz gefunden werden.

Als Anspruchsvoller stellte sich die Implementierung der Möglichkeit heraus, die Ergebnisse auch als PDF-Datei speichern zu können. Hier bieten sich verschiedene Möglichkeiten der PDF-Generierung an. Da die Applikation momentan eine reine Client-Anwendung ist, war die Auswahl an Möglichkeiten dadurch limitiert. Hier sollte nicht nur für die Möglichkeit der PDF-Generierung auch ein Server in die Anwendung eingespielt werden.

Die simpelste der Möglichkeiten ist ein Drucken als PDF Datei. Hierbei müsste für die Seite lediglich ein entsprechendes Stylesheet hinterlegt werden, dass das Layout für den Druck anpasst. Diese Lösung ist allerdings nur beschränkt verfügbar. Das Betriebssystem macOS bieten den Druck als PDF nativ an, das Betriebssystem Windows beispielsweise aber erst seit der neusten Version, Windows 10. Eine weitere Möglichkeit stellt die Bibliothek html2canvas[1] dar. Die Bibliothek erlaubt das Speichern von Seiten als Bilddateien. Die Verfügbarkeit ist hier deutlich höher als beim Drucken als PDF, jedoch bringt das Speichern als Bild einige Restriktionen mit sich. So können beispielsweise Werte nicht markiert und kopiert werden, was einen erhöhten

Arbeitsaufwand für den Nutzer bedeutet.

Die Entscheidung file aus diesen Gründen hier auf die Bibliothek jsPDF[2]. Diese erlaubt das erstellen von PDF-Dateien im Browser und auch das einfügen von DOM-Elementen in PDF-Dateien. Das einfügen bringt allerdings einige Limitierungen mit sich, so werden teilweise Texte, die zu tief in DOM-Elementen verschachtelt sind, nicht dargestellt. Daher wurde die PDF per hand zusammen gesetzt. jsPDF bietet dafür eine API, mit der verschiedene Elemente (wie Text oder geometrische Formen), unter Angabe der Position auf der x- und y-Achse, in die Datei eingefügt werden können. Listing XZY zeigt das beispielhafte einfügen einer Textzeile und das anschließende Speichern der Datei.

Hier besteht jedoch ein hoher manueller Aufwand, der mit potentiellen weiteren Themengebieten in der Anwendung erneut evaluiert werden muss. Abb. XZY zeigt eine beispielhafte PDF-Datei, die von der Anwendung erstellt wurde.

Hier wurde außerdem deutlich, dass eine mögliche Nomenklatur des Projektes durch den Nutzer Vorteile hätte. So könnte dieser Name auf der PDF-Datei vermerkt werden, um diese eindeutiger zuordnen zu können. Außerdem wird dem Nutzer somit eher das Gefühl vermittelt, dass er einen Prozess durchläuft, an dessen Ende er ein Ergebnis für sein Projekt erzielt hat.

3.8 Tests

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

[1] `https://github.com/niklasvh/html2canvas`
[2] `https://github.com/MrRio/jsPDF`

Kapitel 4

Veröffentlichung der Anwendung

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

4.1 Hosting

Um den Zugang zur Anwendung für die Community möglichst einfach zu gestalten, bietet es sich an, diese zu hosten (einer andere, weniger geeignete Möglichkeit, wäre beispielsweise nur die Veröffentlichung des Quellcodes und das lokale hosten durch den Nutzer selbst). Da es sich bei der Anwendung um eine rein Clientseitige handelt, stehen eine Vielzahl von Möglichkeiten für das hosting zu Verfügung, da der Server lediglich statische html- und JavaScript-Dateien ausliefern muss.

In die nähere Auswahl kamen in diesem Projekt Github Pages und Heroku. Ein hosting auf einem privaten Server wurde schnell ausgeschlossen, um die mögliche zusätzliche Arbeit möglichst gering zu halten. Da der Quellcode der Anwendung bereits auf Github veröffentlich wurde, scheint das Hosting über Github Pages zunächst die naheliegendste und einfachste Lösung. Ein Deployment auf Github Pages erfolgt nach entsprechenden Einstellungen im Repository einfach durch einen push auf den gh-pages branch. Die Anwenudung ist danach über die

Domain username.github.io/rpository erreichbar. Größter Vorteil ist dabei, dass kein externer Service benötigt wird. Es treten aber auch einige Limitierungen auf. Beim testen der Hosting-Möglichkeiten wurde deutlich, dass das Modul React-Router den zusätzlichen Path nicht ohne weiteres unterstützt. Zwar bieten sich hier relative einfach Lösungen wie das verwenden von hashes anstatt von Pfaden im React-Router an, jedoch wurde mit blick auf die bereits bei simplen dingen auftretenden Probleme diese Möglichkeit zunächst als weniger geeignet eingestuft.

Für das Hosting wurde sich für den Service Heroku entschieden. Heroku ist ein Saas, das ein Deployment und hosting ohne Setup und Konfiguration erlaubt. Ein Deployment auf Heroku läuft dabei über einen remote branch im git repository, kann also mit dem Befehl `git push heroku master` gestartet werden. Heroku erkennt die verwendete Sprache automatisch und stellt alles nötige automatisch ein. Jedoch unterstützt Heroku lediglich Serverseitige Progrmmiersprachen und beitet keinen Support für das Hosting von statischen files. Um diese Anwendung zu hosten, muss also ein Server geschreiben werden. Da dieser nur statische Files hosten muss, ist dieser sehr simple in Node.js und dem Framework express geschrieben:

```
1  const express = require('express')
2  const app = express()
3
4  app.use(express.static(__dirname + '/dist'))
5  app.listen(process.env.PORT || 8080)
```

Der Server hostet dabei den ordner `dist`, in dem sich die gebuildeten Files befinden (unter anderem auch die Datei `index.html`, auf die ein Browser automatisch zurück greift). Mit der Datei `Procfile` wird Heroku dann mitgeteilt, welche Befehle beim Deployment der Anwendung ausgeführt werden soll. Im Falles dieser Anwendung ist das die Datei `server.js`. Das `Procfile` besteht alsolediglich aus der Zeile

```
1  web: node server.js
```

Da sowohl Github Pages, als auch Heroku von sich aus eine recht kryptische URL verwenden, wurde weiterhin die Domain `25knots.de` gekauft, um auch hier den Einstieg für potentielle Nutzer so einfach wie möglich zu gestalten.

4.2 Weiterentwicklung

Nachfolgend soll außerdem auf eine mögliche Weiterentwicklung der Anwendung nach dem Abschluss dieser Arbeit eingegangen werden. In diesem Zusammenhang stellen sich vor allem drei Fragen:

1. Wie können Personen aus der Community dazu gebracht werden, an einer Weiterentwicklung mitzuarbeiten?
2. Wie kann eine durchgängig hohe Qualität des Quellcodes garantiert werden, auch wenn viele verschiedene Menschen daran arbeiten?
3. Wie kann das Mitwirken für Interessenten möglichst einfach gestaltet werden?

Eine konkrete Beantwortung der ersten Frage gestaltet sich recht schwierig. Es lässt sich jedoch ein logischer Zusammenhang zwischen der Anzahl der Nutzer und der Anzahl der Mitwirkenden eines Projektes feststellen. Am Beispiel der Plattform Github können *stars* als relativ sichere Mindestzahl der Nutzer gesehen werden. Laut [BVHC15] besteht ein Zusammenhang zwischen *stars* und der anzahl der Mitwirkenden an einem Projekt:

> In git-based systems, forks are used to either propose changes to an application or as a starting point for a new project. In both cases, the number of forks can be seen as a proxy for the importance of a project in GitHub. [...] Two facts can be observed in this figure. First, there is a strong positive correlation between stars and forks (Spearman rank correlation coefficient = 0.55). Second, only a few systems have more forks than stars.

Somit ist eine Steigerung der Mitwirkenden also automatisch mit einer Steigerung der Nutzer der Anwendung verbunden. Der Inhalt diese Abschnittes soll also vorrangig mit der Beantwortung der zweiten und dritten Frage beschäftigen.

4.2.1 Sicherung der Code-Qualität

Grober Aufbau für diese Sektion:

1. Travis-CI
2. es-lint

4.2.2 Dokumentation

Grober Aufbau für diese Sektion:

1. Was zeichnet eine gute Dokumentation aus?
2. Dokumentation innerhalb des Codes
3. Dokumentation außerhalb des Codes

4.2.3 Contribution Guidelines

zahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

4.2.4 Tests

zahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

4.3 Vermarktung

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

Kapitel 5

Ausblick

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

Kapitel 6

Schluss

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

6.1 Fazit

Bei der Textgestaltung und automatischen Änderung von Abbildungsnummern, Querverweisen, Seitenzahlen, Gliederungen, Literaturhinweisen etc. bietet sich der Rückgriff auf moderne Textverarbeitungsprogramme an. Nutzen Sie diese zur besseren Lesbarkeit und Strukturierung des Textes, aber vermeiden Sie überflüssige Spielereien. Da besonders bei Textdokumenten mit eingebundenen Objekten wie Bildern, Formeln

Literaturverzeichnis

[BVHC15] Hudson Borges, Marco Tulio Valente, Andre Hora, and Jailton Coelho. On the popularity of github applications: A preliminary note. *arXiv preprint arXiv:1507.00604*, 2015.

[CP96] Arnold Campbell and Susan Pisterman. A fitting approach to interactive service design: The importance of emotional needs. *Design Management Journal (Former Series)*, 7(4):10–14, 1996.

[DAH01] Luca De Alfaro and Thomas A Henzinger. Interface theories for component-based design. In *EMSOFT*, volume 1, pages 148–165. Springer, 2001.

[Gac15] Cory Gackenheimer. *Introduction to React*. Apress, 1st ed. edition, 9 2015.

[Inc16a] Facebook Inc. Jsx in depth, 2016. `https://facebook.github.io/react/docs/jsx-in-depth.html`.

[Inc16b] Facebook Inc. React.component, 2016. `https://facebook.github.io/react/docs/react-component.html`.

[JS17a] Vue JS. Introduction - vue.js, 2017. `https://vuejs.org/v2/guide/`.

[JS17b] Vue JS. Template syntax - vue.js, 2017. `https://vuejs.org/v2/guide/syntax.html`.

[LFDB06] Gitte Lindgaard, Gary Fernandes, Cathy Dudek, and Judith Brown. Attention web designers: You have 50 milliseconds to make a good first impression! *Behaviour & information technology*, 25(2):115–126, 2006.

[Nic98] Raymond S Nickerson. Confirmation bias: A ubiquitous phenomenon in many guises. *Review of general psychology*, 2(2):175, 1998.

[Rob17] Jonathan Robie. What is the document object model?, 2017. https://www.w3.org/TR/WD-DOM/introduction.html.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley Professional, 2 edition, 11 2002.

[TCKS06] Noam Tractinsky, Avivit Cokhavi, Moti Kirschenbaum, and Tal Sharfi. Evaluating the consistency of immediate aesthetic perceptions of web pages. *International journal of human-computer studies*, 64(11):1071–1083, 2006.

[TKI00] Noam Tractinsky, Adi S Katz, and Dror Ikar. What is beautiful is usable. *Interacting with computers*, 13(2):127–145, 2000.