Technology
Arts Sciences
TH Köln

Erstellung einer isolierten Vue-Komponenten Bibliothek im Zusammenspiel mit dem Utility-First-Framework Tailwind CSS

Entwicklung und Dokumentation anpassbarer Vue-Komponenten für kontinuierlich wachsende Webplattformen am Beispiel einer instanzierbaren Plattformlösung der AVACO GmbH.

Dokumentation zum Praxisprojekt

ausgearbeitet von

Marvin Christian Klimm

vorgelegt an der

Technischen Hochschule Köln
Campus Gummersbach
Fakultät für Informatik und
Ingenieurwissenschaften

im Studiengang
Medieninformatik

Hochschul-Betreuer: Prof. Christian Noss
Technische Hochschule Köln

Gummersbach, im September 2020

Adressen: Marvin Christian Klimm
Am Hepel 61
51643 Gummersbach
studies@marvinklimm.de

Prof. Christian Noss
Technische Hochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
christian.noss@th-koeln.de

Vorbemerkung

Diese Arbeit wurde im Rahmen eines Firmenprojektes angefertigt und dokumentiert die Probleme, Anforderungen, Herangehensweisen und Lösungen die während der Bearbeitung herausgestellt wurden. Das Endprodukt dieser Arbeit, eine Komponentenbibliothek, ist Eigentum der AVACO GmbH und wurde noch vor Anfertigung dieses Dokumentes fertiggestellt.

Die Ideen, Grundkonzepte sowie die Codebasis wurden jedoch vor und während der Bearbeitung des Firmenprojektes in einem persönlichen Freizeitprojekt erarbeitet und parallel zur Anfertigung dieser Dokumentation in einem ungebundenen Referenzprojekt angewandt.

Der Referenzcode dieses Projektes kann unter folgenden URLs aufgerufen werden:

- **Vollständiges Referenzrepository**
  `https://github.com/netzfluencer/tailwindstories`
- **Version mit Storybook Setup**
  `https://github.com/netzfluencer/tailwindstories/releases/tag/v0.1.1-alpha`
- **Version mit Tailwind CSS Setup**
  `https://github.com/netzfluencer/tailwindstories/releases/tag/v0.2.0-alpha`
- **Version mit Beispielkomponenten**
  `https://github.com/netzfluencer/tailwindstories/releases/tag/v0.3.0-alpha`
- **Frontend: Tailwindstories in Verwendung**
  `https://github.com/netzfluencer/tailwindstories-frontend`

Zum Zeitpunkt der Abgabe, ist das Referenzrepository womöglich noch nicht öffentlich zugänglich. Bei Interesse kann ein Zugang per E-Mail an studies@marvinklimm.de angefragt werden.

Inhaltsverzeichnis

1 Einleitung

> *"While the utility-class-focused approach of new frameworks like Tailwind CSS make you question everything you know about writing "proper" semantic CSS, its 81% satisfaction ratio means that it might be time to reconsider our old preconceived notions..."* (Greif u. Benitte, 2019)

In der aktuellen Studie *The State of CSS 2019* haben Raphaël Benitte und Sacha Greif herausgestellt, dass das neuartige Utility-Frist CSS-Framework *Tailwind CSS* neben allen anderen verglichenen Frameworks die höchste Entwicklerzufriedenheit verzeichnen konnte.

Seit 2017 entwickelt die AVACO GmbH das anpassbare Webportal *My.PORTAL*, das als Handels-, Kommunikations- und Kollaborationslösung in unterschiedlichen Nutzungskontexten eingesetzt werden kann. Dabei dienten bisher das Framework Bootstrap und der CSS-Preprozessor *Syntactically Awesome Style Sheets* (i.F. SASS) als Hauptwerkzeuge für das Styling der wachsenden Vue-Applikation. Mit dem kontinuierlichen Wachstum begegnete das Frontend-Team neuen Herausforderungen in der Frontendentwicklung und insbesondere in der Gestaltung von wiederverwendbaren Vue-Komponenten sowie Style-Elementen. Diese Herausforderungen sind insbesondere in den Bereichen des Komponentenmanagements, der Komponentenentwicklung und des Komponentenstylings zu identifizieren und sollen nun, durch die Einführung einer neuen Komponentenbibliothek sowie Tailwind CSS und PostCSS als neue Gestaltungsbasis, gelöst werden.

Diese Arbeit befasst sich mit der Erstellung der neuen Vue-Komponenten-Bibliothek und der Einführung von Tailwind CSS im Frontend der aktuellen Vue-Applikation. Dabei sollen Komponenten aus der Bibliothek isoliert bzw. unabhängig vom Hauptprojekt entwickelt, gestaltet und wiederverwendet werden können. Langfristig soll die neue Komponentenbibliothek fähig sein die bereits existierende Komponentenbibliothek zu ersetzten.

Dazu sollen zunächst die Herausforderungen und Probleme in der aktuellen Frontendarbeit von *My.PORTAL* herausgestellt werden. Anschließend erfolgt eine Ermittlung der Anforderungen an die neue Komponentenbibliothek auf dessen Grundlage ein passendes Framework ausgewählt werden soll. Desweiteren sollen die Chancen und Risiken die mit der Einführung von Tailwind CSS verbunden sind ermittelt werden um dann schließlich Tailwind CSS im gesamten Frontend einzuführen.

Anschließend erfolgt eine Bewertung der Implementierung anhand aller vorher herausgestellten Herausforderungen, Anforderungen, Chancen und Risiken. Neue Herausforderungen, Chancen und Risiken werden dargelegt um schließlich auch einen Ausblick auf die neue Vue Version 3 in Bezug auf diese Arbeit anfertigen zu können.

2 Kontext der Arbeit

2.1 Über My.PORTAL

Mit My.PORTAL bietet die AVACO GmbH eine anpassbare Webplattform, die als Handels-, Kommunikations- und Kollaborationslösung in unterschiedlichen Nutzungskontexten eingesetzt werden kann. Besonders hervorgehobene Einsatzzwecke sind die Nutzung als Stadtportal, Unternehmensportal und Bildungsportal, in denen die Plattformbetreiber eine eigene für sie angepasste DSGVO-konforme Portallösung erhalten. (AVACO GmbH, 2020)

2.2 Architekturen und Technologien

2.2.1 Kommunikation

Das in .NET implementierte Backend basiert auf einer Microservices-Architektur in welcher jeder Microservice gesondert voneinander agiert. Zu Projektbeginn wurden Services über RESTful APIs zugänglich gemacht. Seit Ende 2018 werden neue Services mit GraphQL-API eingeführt, weshalb das System derzeit noch in beiden Weisen kommuniziert.

2.2.2 Frontend

Das Frontend wird als Vue Single Page Application in einem modularisierten Stil entwickelt. Entscheidend für die Einführung von Vue.js waren laut Angaben der AVACO GmbH die Entwicklungsfreundlichkeit, die solide Community, die Performance, die vielen nützlichen Bibliotheken, die leichte Erlernung sowie die Unternehmensunabhängigkeit des Frameworks gewesen.

Derzeit besteht das Frontend aus den folgenden Modulen:

- **portal-frontend**: Haupt-Applikation, die außerdem auch alle Instanzkonfigurationen beinhaltet.
- **portal-global-components**: Bibliothek von wiederverwendbaren Vue-Komponenten
- **portal-tracker**: Framework um Aktivitäten dem Backend zu übermitteln
- **portal-statistics**: Weitere Vue-Applikation um Portal Administratoren Nutzungsstatistiken anzuzeigen

3 Herausforderungen im Frontend

Im folgenden Abschnitt werden Herausforderungen im Frontend von My.PORTAL aufgezeigt. Diese Herausforderungen bilden wichtige Bewertungskriterien für die neue Komponentenbibliothek und die Einführung von Tailwind CSS.

3.1 Redundante CSS Definitionen

Das Frontendstyling wird seit Entwicklungsbeginn in SASS[1] gestaltet. Jede Komponente wird als *Single File Component*[2] mit Component-Scoped CSS innerhalb der .vue Datei gestaltet. Es gibt keine globalen CSS-Klassen, die von mehreren Komponenten verwendete werden. Konstante Gestaltungswerte wie beispielsweise Abstände, Grids, Farben und komplexere Stylings werden per SASS-Mixins[3] oder SASS-Variablen eingefügt was zu redundanten Definitionen im transpilierten Code führt.

Codebeispiel 3.1: UserList.vue (abstrakt)

```
1  <template>
2    <section class="user-list-section">
3      <ul class="user-list">
4        <li v-for="n in 3" class="user-list__item">User {{n}}</li>
5      </ul>
6    </section>
7  </template>
8
9  <style lang="sass" scoped>
10 .user-list-section
11   +make-container
12
13 .user-list
14   +make-row
15   &__item
16     +make-col-ready
17     +make-col(4)
18     +md\-
19       +make-col(6)
20     +sm\-
21       +make-col(12)
22 </style>
```

[1] Syntactically Awesome Stylesheets (SASS) ist ein CSS Präprozessor
[2] Vue Single File Components: `https://vuejs.org/v2/guide/single-file-components.html`
[3] Mixins sind Styledefintionen die in ein Stylesheet eingefügt werden

Codebeispiel 3.2: SiteNav.vue (abstrakt)

```
<template>
  <header class="main-header">
    <ul class="site-nav">
      <li v-for="n in 3" class="site-nav__item">Icon {{n}}</li>
    </ul>
  </header>
</template>

<style lang="sass" scoped>
.main-header
  +make-container

.site-nav
  +make-row
  &__item
    +make-col-ready
    +make-col(4)
</style>
```

Codebeispiel 3.3: Transpiliertes CSS Ergebnis

```
.main-header {
  width: 100%;
  padding-right: 15px;
  padding-left: 15px;
  margin-right: auto;
  margin-left: auto;
}

.user-list-section {
  width: 100%;
  padding-right: 15px;
  padding-left: 15px;
  margin-right: auto;
  margin-left: auto;
}

.site-nav {
  display: flex;
  flex-wrap: wrap;
  margin-right: -15px;
  margin-left: -15px;
}

.user-list {
  display: flex;
  flex-wrap: wrap;
  margin-right: -15px;
  margin-left: -15px;
}

```

```
31 .site-nav__item {
32   position: relative;
33   width: 100%;
34   padding-right: 15px;
35   padding-left: 15px;
36   flex: 0 0 33.33%;
37   max-width: 33.33%;
38 }
39
40 .user-list__item {
41   position: relative;
42   width: 100%;
43   padding-right: 15px;
44   padding-left: 15px;
45   flex: 0 0 33.33%;
46   max-width: 33.33%;
47 }
48
49 @media (max-width: 991.98px) {
50   .user-list__item {
51     flex: 0 0 50%;
52     max-width: 50%;
53   }
54 }
55
56 @media (max-width: 767.98px) {
57   .user-list__item {
58     flex: 0 0 100%;
59     max-width: 100%;
60   }
61 }
```

Die dargestellten Code Beispiele zeigen die Entstehung redundanter CSS-Definitionen. Zur weiteren Veranschaulichung des Problems wurden die gezeigten Mixins im Frontendcode gezählt und der daraus entstandenen Code kalkuliert. Neben den aufgelisteten Mixins gibt es noch 8 weitere Mixins, die mehr als drei mal verwendet werden.

- **+make-container Mixin**: 37x verwendet [Desktop][4]
- **+make-row**: 50x verwendet [Desktop]
- **+make-col-ready**: 69x verwendet [Desktop]
- **+make-col(*)**: 303x verwendet [Desktop, Tablet, Mobile]

Zu beachten ist, dass die Werte von *+make-col(Parameter)* in verschiedenen Display-Kontexten greifen sollen und die Parameterwerte dabei von 1 - 12 (Integer) reichen können. Dies ist für die Berwertung in Abschnitt 6.1.1 relevant.

[4] Displaygrößen in welchen das Mixin verwendet wird

3.2 Ungenutzte CSS Klassen

Über die Entwicklungszeit hinweg hat sich gezeigt, dass es Komponenten gibt, die gegenüber ihrer ersten Ausführung geändert wurden. Dabei ist es passiert, dass ungenutzte CSS Klassen im Code verblieben sind und somit die Applikation unnötig vergrößern.

3.3 Property Hell

Zu Beginn des Projektes wurden viele Komponenten spezifisch für den ersten Einsatzkontext entwickelt. Für ein konsistentes UI-Design war es immer wieder nötig bestehende Komponenten in einer abgewandelten Form im Portal wiederzuverwenden. Dazu wurden regelmäßig neue Properties zu Komponenten hinzugefügt, um sie an neue Kontexte anzupassen. Diese Properties steuern oft nur einen einzigen Usecase und erschweren es den Entwicklern die Dokumentation sowie den Code einer Komponente überblicken zu können. Besonders unübersichtlich ist es, wenn Komponenten das Verhalten von Kind-Komponenten per Properties verändern und ein Property durch verschiedenen Komponenten durchgereicht werden muss.

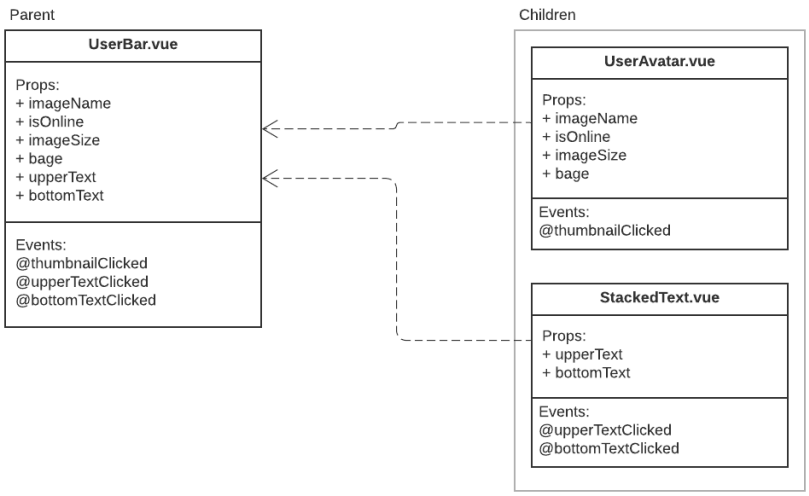

Abbildung 3.1: Abstraktes Beispiel der Property Hell

Abbildung 3.1 zeigt eine Eltern-Komponente die aus zwei weiteren Komponenten besteht. Property-Werte der Eltern-Komponente werden an Kinder-Komponenten weitergegeben. Je mehr Properties die Komponenten besitzen, desto unübersichtlicher wird der Code. Um das Beispiel nachvollziehbar darzustellen, wurden die Property-Namen und Events in allen Komponenten gleich genannt. In den orginallen Komponenten können Property-Namen voneinander abweichen und erschweren die Lesbarkeit und Nachvollziehbarkeit ein Komponente zusätzlich.

3.4 Komponentenmanagement

Bisher wurden Komponenten gemäß dem Atomic Design Prinzip von Frost (2013) entwickelt und geordnet. Dabei werden Komponenten in die Kategorien „Atom“, „Molekül“, „Organismus“, „Template“ und „Page“ kategorisiert. Hier haben sich für dieses Projekt zwei Schwächen herausgestellt:

1. Der Unterschied zwischen Molekülen und Organismen ist oft für die Komponenten nicht strikt definierbar.
2. Es kann passieren, dass größere Komponenten, wie beispielsweise Organismen Moleküle benötigen, die sonst keine andere Komponente benötigt. Diese Moleküle werden jedoch in dem Ordner Moleküle abgespeichert, was zu einer Unübersichtlichkeit der Repositories geführt hat. Der folgende Code soll dies veranschaulichen.

Codebeispiel 3.4: components/organisms/Navigation/Navbar.vue

```
<template>
  <nav>
    <nav-tab v-show="showTab" />
    <nav-tab-button @click="showTab = !showTab">Toggle</
       nav-tab-button>
  </nav>
</template>

<script>
// Für eine bessere Lesbarkeit wurde die Navbar.vue modularisiert
// Die importierten Komponenten werden jedoch nur von dieser
   Datei benötigt.
import NavTab from '../../molecules/Navigation/NavTab.vue'
import NavTabButton from '../../molecules/Navigation/
   NavTabButton.vue'
// Allerdings werden die Komponenten in den Molekül-Ordner ü
   bergeordnet, was die Übersichtlichkeit des Ordners im Hinblick
    auf alle weiteren Komponenten negativ beeinflußt und den
   Ordner "aufbläht".

export default {
  data() {
    return {
      showTab: false
    }
  },
  components: {
    NavTab,
    NavTabButton
  }
}
</script>
```

3.5 Post-Modularisierung von Komponenten

Es kommt immer wieder vor, dass Abschnitte einer Vue-Komponente zur Wiederverwendung eines Abschnitts in eine neue Komponente ausgelagert werden sollen. Neben der Herausforderung zugehörige Methoden und Variablen aus der Logik herauszufiltern müssen auch bereits spezifische Stylings in genisteten SASS-Abschnitten identifiziert und verschoben werden. Dies ist mühsam und fehleranfällig.

3.6 Manuelle Dokumentation der Komponenten

Eine Dokumentation von Komponenten mit Code Beispielen und Property Auflistungen muss manuell vom Entwickler verantwortet werden. Dies führt dazu, dass Dokumentationen durch menschliche Fehler unvollständig, veraltet oder fehlerhaft sind. Zudem ist die manuelle Dokumentation aufwendig.

3.7 Abhängigkeit der Portal-Konfiguration in portal-global-components

Jede Portal Instanz wird im portal-frontend-Repository konfiguriert und dort gespeichert.
Das Repository portal-global-components ist zur Verwendung auf eine gültige Portal-Konfiguration aus portal-frontend angewiesen. Dadurch sind die Komponenten in Zukunft nicht in anderen Projekten/Architekturen ohne eine solche Portal-Konfiguration nutzbar.

4 Erstellung der neuen Komponentenbibliothek

4.1 Anforderungen

Die derzeit eingesetzte Komponentenbibliothek aus dem Repository *portal-global-components* wurde zuerst auf Basis von vue-cli 2 und nach späterer Migration auf vue-cli 3 entwickelt. Es werden bis jetzt keine Dokumentations-Frameworks verwendet und die Dokumentierung der erstellten Bibliothek wird manuell als eigene Vue-App entwickelt.

Aufgrund der bestehenden Komplexität, die unteranderem Resultat der in Kapitel 3 aufgelistet Herausforderungen ist, und dem bevorstehenden UI-Redesigns des gesamten Portals soll eine neue Komponentenbibliothek der Frontend-Infrastruktur hinzugefügt werden. Diese soll zu einer Verbesserung der in Kapitel 3 genannten Herausforderungen beitragen und außerdem das Utility-Class Framework Tailwind CSS verwenden und somit im Frontend einführen. Sowohl die bestehende Bibliothek als auch die neue Bibliothek sollen zunächst im Frontend verwendet werden. Ab Release der neuen Bibliothek soll diese jedoch das *portal-global-components* Repository langfristig vollständig ersetzen.

Anforderungen, die sich für die neue Vue-Komponentenbibliothek ergeben sind folglich:

- **Einführung von Tailwind CSS**: Unter Verwendung der Bibliothek wird Tailwind CSS auch in *portal-frontend* eingeführt. Die Bibliothek liefert eine Standard-Tailwind-Konfiguration, die in Portal-Instanzen individuell angepasst werden kann.
- **Isolierte Komponentenentwicklung**: Die Komponenten sollen unabhängig von *portal-frontend* und einer Portal-Koniguration entwickelt werden sowie nutzbar sein (vgl. Abschnitt 3.7).
- **Automatisierte Komponentendokumentation**: Die Komponenten sollen nicht mehr vollständig manuell von Entwickelnden dokumentiert werden. Stattdessen soll der Entwickler eine *Single-File-Component* mit mindestens einem Codebeispiel neu einführen können. Anschließend sollen dann beispielsweise Properties und Codebeispiele benutzerfreundlich in einer Vue-Single-Page-Application dargestellt werden 3.6).
- **Einfache Komponentenkategorisierung**: Komponenten sollen weiterhin gemäß ihrer Komplexität in entsprechende Ordner gespeichert werden. Die Probleme aus Abschnitt 3.4) sollen dabei weitesgehend gelöst werden.

4.2 Frameworkwahl

Damit die neue Bibliothek möglichst robust ist und die Einführung möglichst einfach umzusetzen ist, soll ein passendes Framework als Projektbasis eingesetzt werden. Zunächst wird ein Vergleich zwischen drei möglichen Frameworks präsentiert um schließlich eine Auswahl treffen zu können.

4.2.1 Vergleich

Nach einer Recherche wurden für dieses Projekt drei Open-Source Frameworks näher betrachtet.

1. **Vue Styleguidist** (1,8k Follower auf Github (d))
   Vue-Styleguidist ist ein Fork des bekannten react-styleguidist (9,2k Follower auf Github (a)). Vue Styleguidist wird offiziell als *"Isolated Vue component development environment with a living style guide."* (Vue Styleguidist) beschrieben. Das Framework kann basierend auf Code-Kommentare einer Komponente eine Dokumentation dieser automatisiert anfertigen. Dokumentationsseiten werden in Markdown verfasst. Außerdem ist dieses Framework offiziell Teil des Vue-Community-Ecosystems[1], was einen langen Support mit regelmäßigen Updates wahrscheinlich macht.

2. **Vue Design System** (1,9k Follower auf Github (c))
   Das Vue Design System wurde von Viljami Salminen, einem Design System Architekten aus Finland, erstellt und wurde bereits in der Vergangenheit von der AVACO GmbH für eine neue Bibliothek in Betracht gezogen. Das Projekt basiert auf Vue.js und Vue-Stylguidist. Dabei unterscheidet es sich besonders darin eigene Design-System-Praktiken und Strukturen dem Anwendenden anzubieten. Eine Praktik stellt dabei eine abstraktere Atomic Design Alternative dar. Diese ist besonders im Hinblick auf Komponentenmanagement Herausforderung aus Abschnitt 3.4 interessant.

   Die Terminologie des abgewandelten Atomic Design Ansatzes werden wir folgt definiert (Salminen, b):

   - **Design Tokens**: Werden in einer .yml Datei definiert und sind im gesamten Design System als SASS-Variablen nutzbar.
   - **Elements**: Komponenten, die nicht weiter heruntergebrochen werden können. Beispielsweise Buttons.
   - **Patterns**: Komponenten, die aus anderen *Elements* und *Tokens* bestehen.
   - **Templates**: Komponenten, die aus *Elements*, *Patterns* und *Tokens* bestehen

   Der letzte Release erfolgte im Jahr 2018 weshalb man derzeit nicht beurteilen kann inwieweit das Projekt noch gepflegt werden wird. Vue-Styleguidst, das Basisframework, erhält jedoch weiterhin Updates.

[1] `https://vue-community.org/guide/ecosystem/documentation.html#vue-styleguidist`

3. **Storybook** (53,7k Follower auf Github (b))
   Storybook ist ein populäres aus der React-Community entstandendes Framework. Die offizielle Beschreibung lautet wie folgt:

   > *"Storybook is an open source tool for developing UI components in isolation for React, Vue, Angular, and more. It makes building stunning UIs organized and efficient."* (Storybook)

   Obwohl das Framework ursprünglich für React entwickelt wurde unterstützt es mittlerweile in weiten Umfängen viele weitere Frameworks. Komponenten werden dabei in *Stories* dokumentiert, welche in einer eigenen JavaScript-Datei gespeichert werden. Eine Vue-Story wird dann als Stringified Single Page Component verfasst. Dieser Prozess ist nicht vollkommen automatisiert, da zu einer neuen Komponente zunächst eine Story verfasst werden muss. Die Story kann in Form eines Beispielcodes verfasst werden. Jede Komponente, die in eine Story als Modul hineingeladen wird, wird dann automatisiert analysiert und Properties, Slots, etc. benutzerfreundlich dargestellt.

   Insgesamt ist Storybook durch seine Schnittstellen und Pluginsystem sehr mächtig. Dies geschieht jedoch auf Kosten der initialen Aufsetzung.

   Zwei weitere interessante Eigenschaften sind zum einen die Svelte[2] Unterstützung und zum anderen die Figma-Einbindungsmöglichkeit, da die AVACO GmbH diese beiden Werkzeuge in Zukunft zusätzlich einführen könnte.

4.2.2 Auswahl Storybook

Frost (2016) definiert in seinem Artikel die Bezeichnungen *Workshop* und *Storefront* in Bezug auf Design-Systeme. Demnach sind *Workshops* Entwicklungsumgebungen für das effektive Erstellen neuer UI-Komponenten im Team. Der *Workshop* bietet alle Werkzeuge und Strukturen für diese Arbeit.

Die *Storefront* präsentiert hingegen das fertige Design-System mit allen weiteren Ressourcen. Ähnlich zu einem Styleguide sollen in der *Storefront* Informationen erscheinen, die für Entwickler und Benutzer aus anderen Disziplinen hilfreich sind.

Laut Nguyen (2018) dem Initiator von Storybook haben Styleguidist und Storybook viele Gemeinsamkeiten und bedienen beide jeweils Anforderungen an Workshop und Storefront. Jedoch klassifiziert er letztendlich Styleguidist als Storefront und Storybook als Workshop. Auch Frost bezeichnet Storbook wie folgt:

> *"Storybook is a powerful frontend workshop environment tool that allows teams to design, build, and organize UI components (and even full screens!) without getting tripped up over business logic and plumbing."*

Da es in diesem Projekt besonders darum geht eine robuste Komponentenbibliothek für Frontend-Entwickler zu erstellen, welche die Entwickler in ihren Entwicklungen unterstützt

[2] Mehr zu Svelte: `https://svelte.dev/`

und die Entwickler nur einfache Storefront-Eigenschaften benötigen wird das Projekt mit Storybook entwickelt. Jedoch sollen die von (Salminen, b) definierten Terminologien und Prinzipien bei der Umsetzung auf das Tailwind-Ecosystem projiziert werden. Sofern in Zukunft die Bibliothek um ein vollständiges Design System erweitert werden sollte, sollte ein zusätzliches Werkzeug zur Dokumentierung und Darstellung des Systems gesucht werden.

4.3 Entwicklung

4.3.1 Repositoryverwendung

Die Komponenten Bibliothek wird in einem eigenen Repository unter dem Namen *portal-storybook* erstellt und kann anschließend als *Node-Module* in anderen Projekten als Vue-Plugin installiert werden. Dabei soll die Bibliothek drei unterschiedliche Komponenten-Registrierungen ermöglichen.

1. **Installation aller Komponenten im globalen Vue Namespace**: Komponenten müssen zur Verwendung nicht mehr in andere Dokumente importiert werden, sondern können direkt im Markup unter ihrem Komponenten-Namen genutzt werden.

Codebeispiel 4.1: Globale Installierung

```
1 // bspw. main.js in einem vue-cli-3 Projekt
2 // ...
3 import * as storybook from 'portal-storybook'
4 // ...
5 Vue.use(storybook, {useAllComponents: true})
```

2. **Installation einzelner Komponenten als Modul durch *Named Exports* aus der Dependency heraus**: Komponenten werden nicht im globalen Namespace installiert sondern werden erst spezifisch in den Dokumenten wo sie genutzt werden durch *Named Imports* als Komponente installiert. In der *main.js* aus 4.1 würde jedoch weiterhin *Vue.use(storybook,)* ohne dem useAllComponents-Key aufgerufen werden um andere notwendige Abhängigkeiten zu registrieren.

Codebeispiel 4.2: Named Exports/Imports in einer SFC

```
1 // bspw. eine Datei mit dem Namen Dashboard.vue im
2 // ...
3 <script>
4 import { MyComponent1, MyComponent2 } from 'portal-storybook'
5
6 export default {
7     components: { MyComponent1, MyComponent2 }
8 }
9 </script>
```

3. **Installation einzelner Komponenten als Modul durch import der SFC Vue-Dateien aus dem Dependecy-Verzeichnis**: Sofern Treeshaking in einem Projekt

nicht unterstützt wird und eine globale Installation nicht erfolgen soll können Komponenten außerdem auch durch einzelne Imports installiert werden.

Codebeispiel 4.3: Default Exports/Imports in einer SFC

```
1  // bspw. eine Datei mit dem Namen Dashboard.vue im
2  // ...
3  <script>
4  import MyComponent1 from 'portal-storybook/src/elements/
       MyComponent1'
5  import MyComponent1 from 'portal-storybook/src/elements/
       MyComponent2'
6
7  export default {
8      components: { MyComponent1, MyComponent2 }
9  }
10 </script>
```

4.3.2 Repositoryaufbau

Wie bereits im Vorwort erwähnt wird in dieser Arbeit nicht das Orginal-Repository *portal-storybook* aus dem Firmeneigentum der AVACO GmbH referenziert sondern eine andere Komponentenbiliothek (*tailwindstories*) die auf den selben Konzepten und Lösungen aufbaut, nicht die Firmenkomponenten enthält und neu aufgetauchte Probleme aus *portal-storybook* (vgl. 6.3.1) bereits mitbehandelt.

Die Abbildung 4.1 am Ende dieses Kapitels zeigt einen Screenshot des *tailwindstories*-Repository und stellt den Verzeichnisaufbau nach der Einführung von storybook in ein neues *vue-cli-3* Vue-Projekt (v.2.) da.

Neben diversen Konfigurationsdateien gibt es die folgenden hervorzuhebenden Verzeichnisse und Dateien:

- **.storybook**
  Enthält alle Storybook spezifisch Workbench-Konfigurationen und Funktionen. Diese sind wichtig für die Nutzung der Workbench, werden aber für den Import fertiger Komponente in einem anderen Projekt nicht benötigt.

- **src/main.js**
  Entrypoint-Datei für die Nutzung als Bibliothek mit Setup-Methoden und Bibliothek-Konfigurationen.

- **src/elements**
  Verzeichnis mit Komponenten die als *elements* klassifiziert werden. Komponenten werden als Vue-SFC-Dateien angelegt, in der index.js importiert und dann als Named-Modul in der main.js importiert. Eine README.md erklärt die Klassifizierungsmerkmale laut Salminen:

  > *"Elements utilize decisions made on the token level. A simple example of an element would be a button, a link, or an input. Anything that cannot be*

> *broken down further. I use the name 'element' since everything in Vue and React world is nowadays 'a component.' Using that term for anything else would be confusing."* (Salminen, 2018a)

- **src/patterns**
  Verzeichnis mit Komponenten die als *patterns* klassifiziert werden. (wie *src/elements*)

  > *"Patterns are UI Patterns that fall on the more complex side of the spectrum. So for example things like a date picker, a data table, or a visualization. Patterns utilize both elements and tokens. If you wonder whether something should be called an element or a pattern, ask yourself this question: "Can this be broken down into smaller pieces?" If the answer is yes, it should most likely be a pattern."* (Salminen, 2018b)

- **src/templates**
  Verzeichnis mit Komponenten die als *templates* klassifiziert werden. (wie *src/elements*)

  > *"Templates exist to document the layout and structure of a section. I am not calling these pages since semantically that would be incorrect. While they can be pages, that's not their only functionality. Templates consist of the three things mentioned above: tokens, elements, and patterns."* (Salminen, 2018c)

- **src/plugins**
  Verzeichnis mit Tailwind-Plugins die unter anderem als *Tokens* klassifiziert werden können.

  > *"Design tokens are the visual design atoms of the design system — specifically, they are named entities that store visual design attributes. We use them in place of hard-coded values (such as hex values for color or pixel values for spacing) in order to maintain a scalable and consistent visual system for UI development."* (Salesforce)

  Die Tailwind CSS Plugins können Tokens als neue Utility-Klassen einführen.[3]

4.3.3 Storybook Setup

Zu Beginn des Projektes wurde Storybook Version 4 in *portal-storybook* installiert und eingesetzt. Zur automatisierten Dokumentation der erstellten Vue-Komponenten wurde das Addon *storybook-addon-vue-info*[4] eingesetzt. Dieses Addon kann die genutzten Komponenten in einer Story analysieren und wichtige Eigenschaften sowie den Quellcode in dem Storybook-GUI nutzerfreundlich darstellen.

Seit der Storybook Version 6 wird das Storybook Addon *@storybook/addon-info* auf welchem das Addon *storybook-addon-vue-info* basiert nicht mehr unterstützt und Storybook bietet nun offiziell das Addon *@storybook/addon-docs* für den Zweck automatisier-

[3] Mehr zu Tailwind CSS Plugins: `https://tailwindcss.com/docs/plugins`
[4] Repository: `https://github.com/pocka/storybook-addon-vue-info`

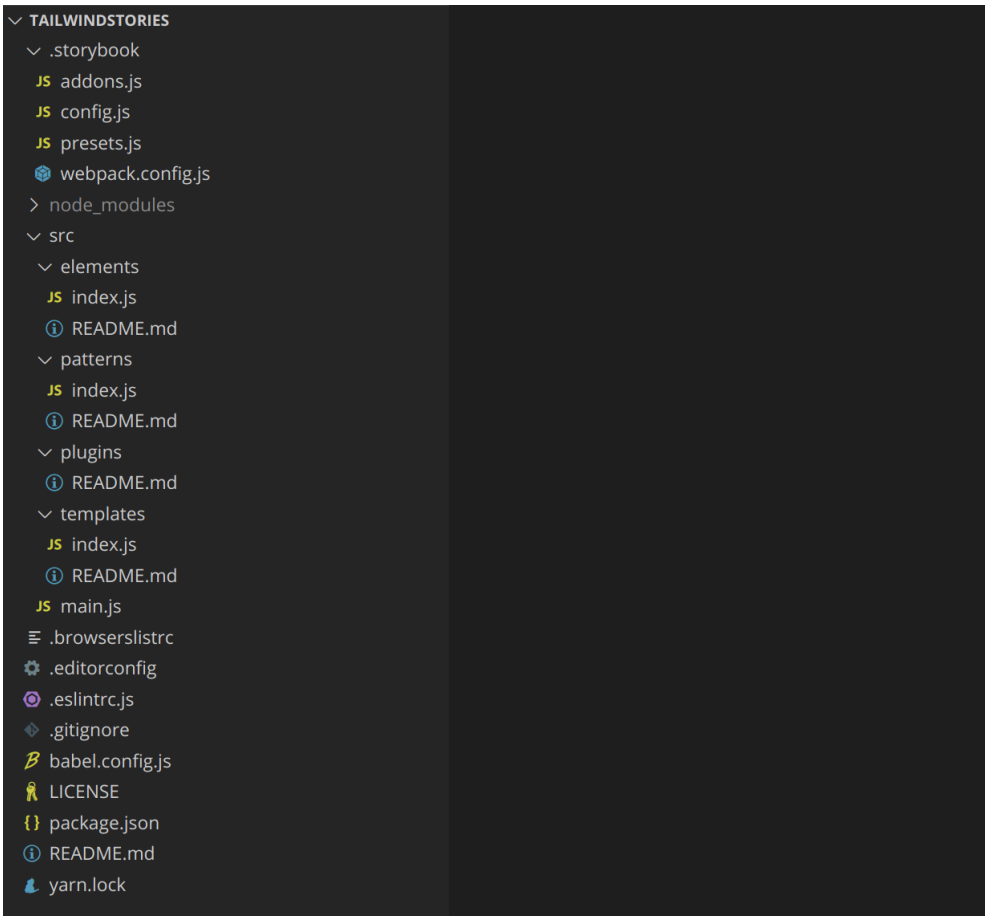

Abbildung 4.1: Repositoryaufbau mit Storybook

ter Komponenten-Analysen an. Dieses Addon arbeitet derzeit besonders gut mit React Komponenten aber auch Vue Komponenten können analysiert und mit ihren Eigenschaften dargestellt werden. Es gibt die Möglichkeit Property-Werte direkt aus der GUI heraus zu manipulieren. Der einzige Nachteil im Vergleich zu dem nun veralteten (deprecated) *storybook-addon-vue-info* ist, dass Code-Snippets derzeit nicht den eigentlichen Code der dargestellten Komponente enthalten, sondern den Code der Story (Github Issue). In React mit JSX tritt dieses Problem jedoch nicht auf.

Da dies jedoch für eine gute Developer-Expierence behindernd ist, wurde *tailwindstories* zunächst auch nicht auf der Storybook Version 6 aufgebaut sondern auf der Version 5 zusammen mit *storybook-addon-vue-info*. In der Zukunft soll jedoch ein Upgrade auf Version 6 mit *@storybook/addon-docs* ausgeführt werden.

4.3.4 Komponentengruppierung

In Abschnitt 4.3.2 wurde bereits der generelle Aufbau des Repositorys erläutert jedoch soll im folgenden ein weiteres Prinzip zum Management mit Komponenten präsentiert werden

um die globalen Komponentenordner *elements*, *patterns* und *templates* nicht mit kontextgebundenen Komponenten, wie in Codebeispiel 3.4 gezeigt zu füllen.

Generell können sollen Komponenten nach Charakteristiken in Unterordnern eingeordnet werden. So können beispielsweise Formular-Elemente wie Inputs, Textareas, Datum-Selektoren in *elements/form* eingeordnet werden. Jede Vue-Komponenten-Datei soll mit einem Großbuchstaben anfangen, dass beispielsweise in *elements/form* die Dateien *FormInput.vue* und *FormTextarea.vue* befinden. Ein weiteres Merkmal ist der Prefix *Form* in den Dateinamen. Dieser hilft den Entwicklern den Ort der Komponente zu identifizieren. Der Prefix orientiert sich immer an den letzten Ordnername des Dateiweges. Würde man den Ordner *form* in einem weiteren Ordner einnisten so würden sich die Dateinamen nicht ändern. Ein Gruppierungsordner soll immer mit einem Kleinbuchstaben anfangen.

Die Lösung für das Problem mit der Einordnung von kontextgebundenen Komponenten gestaltet sich nun so, dass ein Komponente wie *Navbar.vue* beispielsweise in *patterns/Navbar/Navbar.vue* eingeordnet werden kann. Der Ordner *Navbar* beginnt wie eine Komponete mit einem Großbuchstaben und darf in einem *_components*-Ordner *elements*, *patterns* und *templates* unkategorisiert enthalten. Diese Komponenten sind jedoch nur im Kontext von *Navbar.vue* zu verwenden, entweder als Kind in einem Slot einer Navbar.vue Instanz oder in der SFC von Navbar.vue. Der Prefix dieser Komponente richtet sich an den Ordnernamen über *_components*; in diesem Fall *Navbar*. Das folgende Beispiel soll den Aufbau verdeutlichen.

Codebeispiel 4.4: components/patterns/Navbar/Navbar.vue

```
1  <template>
2    <nav>
3      <navbar-tab v-show="showTab" />
4      <navbar-tab-button @click="showTab = !showTab">Toggle</
           nav-tab-button>
5    </nav>
6  </template>
7
8  <script>
9  import NavbarTab from './_components/NavbarTab.vue'
10 import NavbarTabButton from './_components/NavbarTabButton.vue'
11
12 export default {
13   data() {
14     return {
15       showTab: false
16     }
17   },
18   components: {
19     NavbarTab,
20     NavbarTabButton
21   }
22 }
23 </script>
```

5 Einführung von Tailwind CSS

Tailwind CSS ist ein CSS Framework, dass auf eine *Utlity-First* Methodik basiert. Dabei werden CSS-Klassen überwiegend als atomare Attribute definiert, die anschließend bei Bedarf in Komponenten-Klassen abstrahiert werden können.

5.1 Allgemeine Vorteile

> *"We've had great success with going utility-first in the Algolia documentation, reducing our CSS from over 125 KB (and continuously growing) to less than 50 KB (9 KB GZipped, 7 KB with Brotli compression),* ***which represents a 60% size decrease!*** *From a user perspective, this is a guarantee of lightning-fast loading styles with extremely low overhead."* (Dayan, 2019)

Die Vorteile von Tailwind CSS gegenüber anderer CSS-Frameworks sind vielseitig. Einige der von Dayan (2019) hervorgehobenen Eigenschaften sind insbesondere:

- **Anpassbarkeit und Nachvollziehbarkeit**: Utility Klassen werden in einer JavaScript-Datei definiert und sind somit zentral einsehbar, was einfacher und schneller zu warten ist und eine gute Anpassbarkeit an individuelle Design-Systeme ermöglicht. Die Dokumentation von Tailwind CSS ist außerdem leicht verständlich für Lernende.
- **Kontrollierbare Dateigröße**: Durch einmalig definierte Utility Klassen wächst der CSS Code nicht mehr an. Styles müsssen nicht mehr durch Spezifitäten überschrieben werden, was bessere Performance bewirkt.
- **Automatisierte *CSS-Spülung*** (engl. purge): Da Tailwind CSS initial viele zunächst ungenutzte Utility-Klassen basierend aus einer Konfiguration erstellt (Initial Größe beträgt meist unkomprimiert c.a. 3400 kB[1]) wurde das Framework mit dem Framework PurgeCSS für die Generierung von Production-Code erstellt. PurgeCSS vergleicht die gesamte Codebasis mit allen CSS-Klassen und entfernt CSS-Klassen die nicht genutzt wurden. Dadurch wird die CSS-Dateigröße meistens auf c.a. 10kB (komprimiert) reduziert (Tailwindcss, 2020).
- **Komprimierungsfreundlichkeit**: Durch Tailwind CSS erfolgt die eigentliche Produktgestaltung meist direkt im Markup Code unter redundanter Verwendung von Utility-Klassennamen. Komprimierungsalgorithmen sind so gestaltet das gleiche Strings komprimiert werden, wodurch der Markup-Code durch Utlity-Klassen unwesentlich größer wird.

[1] vgl. `https://tailwindcss.com/docs/controlling-file-size`

5.2 Chanceneinschätzung für My.PORTAL

Durch die Einführung von Tailwind CSS erwartet die AVACO GmbH die folgenden Probleme (vgl. Kapitel 3) zu lösen oder zu verbessern:

- **Redundante CSS Definitionen**: Scoped-CSS wird durch Tailwind CSS auf ein Minimum reduziert. Redudante CSS Defintionen sollten (nahezu) vollständig entfernt werden können.
- **Ungenutzte CSS Klassen**: Neben Tailwinds Utlity-Klassen wird es nur in Ausnahmen andere Klassen geben. Die Chance, dass ungenutze CSS Klassen geschrieben werden wird deutlich reduziert. Ungenutze Tailwind-Klassen werden automatisch von PurgeCSS entfernt.
- **Post-Modularisierung von Komponenten**: Da die Styledefinitionen sich im Markup als CSS Klassen befinden sollte ein Verschieben von Code Abschnitten nur noch eine Analyse des Markups und der Script-Logik erfordern.
- **Tailwindkonfiguration statt Portalkonfiguration in portal-storybook** (vgl Abschnitt 3.7): Es wird keine Portal-Konfiguration für die Bibliothek benötigt. Stattdessen enthält die Bibliothek eine eigene *tailwind.conf.js* die sich nach der Tailwind CSS Dokumentation ausrichtet. Andere Projekte nehmen diese Konfiguration dann als Basis und können sie entsprechend eigener Anforderungen zielgerichtet anpassen. (vgl. 5.3.2)
- **Property Hell**: Tailwind CSS löst Vue-Properties ab die bisher dynamisch Klassen mit höherer Spezifität zur umgestaltung einer Komponentenerscheinung angelegt wurden. Stattdessen sollen Erscheinungsanpassungen mit Utility-Klassen in *class*-Attributen erfolgen. Die Property Hell wird in Kapitel 6.1.3 näher betrachtet.

Diese Einführung ist jedoch auch mit Risiken verbunden:

- **Wenig allg. Langzeiterfahrung mit Tailwind CSS**: Unbekannte Probleme können in der Tailwind-Community oder spezifisch bei der AVACO GmbH auftreten. Jedoch fällt in der Entwickler-Community aktuell Tailwind CSS verglichen mit anderen populären Frameworks (bspw. Bootstrap) auf den höchsten Zufriedenheitsgrad (Greif u. Benitte, 2019).
- **Technologie Konflikte**: Besonders in der Übergangsphase werden alte Technologien (z.B. SASS und Bootstrap) zusammen mit Tailwind CSS und PostCSS (inkl. PurgeCSS) im Projektcode koexistieren. Die Folgen dessen können sich im Hinblick auf Bugs, Performance und Code-Komplexität stärker oder schwächer auswirken.
- **Bevorstehendes Upgrade auf Node 12**: Derzeit wird das Frontend auf der Node Version 11 aufgesetzt. Mit dem nächsten großen Release wird Tailwind CSS Version 2 nur noch Node ab den Versionen 12 unterstützen. Da jedoch die Gebundenheit von My.PORTAL zu Version 11 derzeit primär von SASS abhängt und SASS langfristig durch Tailwind CSS und PostCSS ersetzt werden wird, ist dies kein dauerhafter Risikozustand.

5.3 Entwicklung

Das Setup erfolgte gemäß der offiziellen Dokumentation[2] von Tailwind CSS. Damit diese Komponenten sich in anderen Projekten richtig verhalten, muss Tailwind CSS auf Basis der tailwind.config.js in diesen verfügbar sein und die tailwind.css Datei aus der Bibliothek importiert werden. (vgl. Abschnitt 5.3.2)

5.3.1 Plugins

Die Tailwind Standardkonfiguration soll durch Plugins und Wertüberschreibung an Design-Systeme anpassbar sein. Aktuell gibt es drei Plugins:

- **Color-Plugin**: Führt Corporate Identiy Farben als Utlity-Klassen ein. Es kann eine Primary Color gesetzt werden aus welcher automatisch eine passende Farbpalette für Status-Farben (Success, Error, ...) erzeugt wird. Als Basis wurde das Paket *chroma-js* gewählt welches die Farbpaletten im LAB-Farbraum mischt[3]. Da dieses Plugin die Konfiguration mit weiteren Object-Keys erweitert, wird es in der Konfigurationsdatei nicht als Plugin registriert sondern wird dem Konfigurationsobjekt per Spread-Syntax angehangen.
- **Container-Plugin**: Führt responsive Margin/Padding Größen für Container ein. Die Utility-Klasse *.px-c* setzt das Padding eines Layout-Containers. Dies bietet eine größere Flexibilität beim entwickeln von Markup unter Einhaltung von Linienführungen in dem GUI.
- **Box-Plugin**: Definiert das Aussehen von GUI-Boxen (z.B. Karten) indem es Box-Utilitly-Klassen und eine Komponenten-Klasse (*.box*) einführt.

5.3.2 Anpassungsmethode in Projekten

Damit die Komponenten sich in die diversen Erscheinungen unterschiedlicher Portal Instanzen einpassen können, wurde eine Tailwind Setup-Methode angefertigt. Diese gibt das der Komponentenbibiliothek zugrundeliegende Tailwind-Konfigurationsobjekt zurück und bietet durch definierte Parameter die Möglichkeit das Erscheinungsbild (derzeit die Farbpalette) zu verändern. Eine tiefere Anpassung ist durch normale Tailwind Konfiguration möglich, indem Werte der Bibliothekskonfiguration verändert, entfernt oder erweitert werden. Die folgenden Codebeispiele referenzieren das Repository *tailwindstories* (vgl. Vorbemerkung der Arbeit).

Codebeispiel 5.1: tailwind.config.js in einem Vue-Cli 3 Projekt

```
1 const twSetup = require('tailwindstories/tailwind.setup.js')
2
3 module.exports = twSetup({ color: '#891b55' })
```

[2] vgl. `https://tailwindcss.com/docs/installation`
[3] chroma-js LAB: `https://gka.github.io/chroma.js/#chroma-lab`

Codebeispiel 5.2: Erweiterte tailwind.config.js in einem Vue-Cli 3 Projekt

```
const twSetup = require('tailwindstories/tailwind.setup.js')({
    color: '#891b55' })

module.exports = {
  ...twSetup,
  extend: {
    ...twSetup.extend,
    screens: {
      '2xl': '1440px',
    }
  }
}
```

5.3.3 My.Portal Konfiguration

In einem einfachen Vue-Cli 3 Projekt ist das Setup von Postcss und Tailwind allgemein gut dokumentiert verfügbar[4]. Die MyPortal Vue-Applikation basiert jedoch noch auf vue cli 2 mit diversen individuellen Webpack Konfigurationen sowie Transpilierungsskripten die auf dem Vue Webpack Template aufbauen[5]. Wie bereits in Abschnitt 3.7 erwähnt enthält das Hauptrepository zudem die einzelnen Instanzkonfigurationen.

Die Struktur ist in Abbildung 5.1 modelliert. In Englisch beschriebene Abschnitte stammen von der Webpack-Template Dokumentation und der *portals* Ordner enthält die Instanzspezifischen Konfigurationen. Der Struktur wurde außerdem bereits eine *tailwind.config.js* angehängt auf welche im Folgenenden eingegangen wird.

```
├── build/                    # webpack config files
│   └── ...
├── config/
│   ├── index.js              # main project config
│   └── ...
├── portals/
│   ├── Instanz1/             # Instanzordner
│   │   ├── advanced/
│   │   │   └── ...
│   │   ├── index.js          # Konfiguration des Portals (Locales, Modules, Meta ...)
│   │   ├── style.sass        # Portal Style Anpassungen (Farben, Logogröße, ...)
│   │   └── tailwind.config.js  # NEU: Portal Style Anpassungen (Farben, Logogröße, ...)
│   └── ...
├── src/
│   ├── main.js               # app entry file
│   ├── App.vue               # main app component
│   ├── components/           # ui components
│   │   └── ...
│   └── assets/               # module assets (processed by webpack)
```

Abbildung 5.1: Portal-Frontend Repository Struktur

[4] vgl. https://cli.vuejs.org/guide/css.html#postcss und https://tailwindcss.com/docs/installation

[5] vgl http://vuejs-templates.github.io/webpack/

Ein Portal kann anschließend durch ein Node-Skript mit einem Portalordnernamen als Parameter transpiliert werden:

npm run build Instanz1

Durch den Befehl werden die Dateien aus dem Ordner *Instanz1* in die Build und Konfigurationsdateien importiert. Äquivalent könnte man eine neue Portalinstanz in dem Verzeichnis *portals/Instanz2* anlegen und starten.

Da Tailwind CSS standardmäßig im Root Verzeichnis nach der optionalen *tailwind.config.js* sucht sind für diese Art der Instanziierung besondere Anpassungen für das Tailwind Setup im *portal-frontend* notwendig.

Dazu wurde zunächst im *build* Verzeichnis ein Modul anlegt, die eine normale Tailwindkonfiguration anhand des Portalnamens auflösen kann und mit der Tailwindkonfiguration aus der Komponentenbibliothek zusammenführt sowie ein erweitertes PurgeCSS Setup festlegt (vgl. Codebeispiel 5.4).

Codebeispiel 5.3: Modul zur Auflösung und Zusammenführung der tailwind.config.js

```
1  // building tailwinds config for selected instance
2  // ------
3  const _ = require('lodash')
4  const path = require('path')
5  const tailwindBase = require('portal-storybook/tailwind.config.js
       ')
6
7  function resolve(dir) {
8    return path.join(__dirname, '..', dir)
9  }
10
11 module.exports = portalName => {
12   const pathToRoot = resolve('portals/' + portalName)
13   const pathToTailwind = resolve('portals/' + portalName + '/
       tailwind.config.js')
14   const tailwindConf = require(pathToTailwind)
15
16   function emptyConfig(obj) {
17     return Object.keys(obj).length === 0 && obj.constructor ===
         Object
18   }
19
20   const tailwindConfig = (emptyConfig(tailwindConf)) ?
       tailwindBase : _.merge(tailwindBase, tailwindConf)
21
22   return {
23     ...tailwindConfig,
24     purge: [
25       `./public/**/*.html`,
26       `./src/**/*.vue`,
27       `./node_modules/portal-global-components/src/**/*.vue`,
28       `./node_modules/portal-storybook/src/**/*.vue`,
```

```
29        `${pathToRoot}/**/*.vue`,
30        `${pathToRoot}/**/*.html`
31      ]
32    }
33  }
```

Damit dieses Modul nun für das Setup von Tailwind CSS innerhalb von PostCSS angewandt wird muss die *postcss.config.js* wie folgt angepasst werden.

Codebeispiel 5.4: Modul zur Auflösung und Zusammenführung der tailwind.config.js

```
1  module.exports = ctx => ({
2    "plugins": [
3      require('tailwindcss')(require('./build/tailwind.js')(ctx &&
           ctx.options && ctx.options.portalName || '_default')),
4      require('autoprefixer')(),
5    ]
6  })
```

Dabei ist das Setup in Form eines Rückgabewertes einer Funktion für PostCSS nicht so häufig verbreitet. Grund dafür ist jedoch das benötigte *Context (ctx)* Objekt[6] was in der Webpack Konfiguration wie folgt definiert wurde:

Codebeispiel 5.5: Webpack.conf.js

```
1  // portalName wird durch das DevServer Script welches die
2  // Terminalparameter ausliest mit dem Portalnamen initialisiert.
3  module.exports = portalName => {
4  // ...
5    {
6      loader: 'postcss-loader',
7      options: {
8        config: {
9          ctx: {
10           portalName
11         }
12       }
13     }
14   }
15 // ...
16 }
```

[6] vgl. https://github.com/postcss/postcss-load-config#context

6 Resultate

6.1 Chancenbewertung Tailwind CSS

Der folgende Abschnitt bewertet die in Abschnitt 5.2 gelisteten Chancen und Risiken im Hinblick auf die derzeitige Implementierung von Tailwind CSS.

6.1.1 Redundante CSS Definitionen

Durch den Einsatz von Tailwind CSS wird ein Alternative gegenüber der Nutzung von SASS-Mixins geschaffen. Da SASS nach dem Tailwind CSS Refactoring vollständig entfernt werden wird sind Entwickler bereits jetzt dazu angehalten keine Mixins mehr zu verwenden.

Um Einschätzen zu können inwieweit sich das Entfernen von Mixins lohnt wurde eine exemplarische Berechnung des von *+make-col()* erzeugten Codes angefertigt.

Mit dem Wissen, dass die drei Displaygrößen Mobile, Tablet und Desktop im Projekt unterschiedliche Spezifitäten haben müssen führt der Einsatz des Mixins zu redundanten Code, der sich derzeit mit äquivalenten Utility-First CSS Klassen um c.a. 88 - 90% reduzieren ließe.

> **10 Parameterwerte * 3 Screengrößen = 30 Klassen bzw. bei 12 Parameterwerten wären es dauerhaft 36 Klassen** (vgl. 3.1)

Somit versursacht der Einsatz dieses Mixins allein 34 KB (28KB minified) (bezogen auf 303-maligen Gebrauch). Das Tailwind CSS Äquivalent hat jedoch lediglich eine Größe von 3,9 KB (3,3KB minified).

Da bisher noch kein vollständiges Refactoring des Frontends umgesetzt wurde, sondern diese Arbeit sich mit der Umsetzung einer geeigneten Komponentenbibliothek sowie der generellen Einführung von Tailwind CSS befasst, kann die gesamte CSS-Ersparnis im Vergleich zu SASS noch nicht ermittelt werden. Jedoch ist das Portal-Frontend durchaus komplexer als die ehemalige Algolia-Dokumentation (Dayan, 2019) gestaltet, weshalb man ziemlich sicher von einer ähnlich guten Einsparung von 60% Code ausgehen kann.

6.1.2 Ungenutzte CSS Klassen

Diese Chance hat sich durch Tailwind CSS und PurgeCSS bewahrheitet. Ungenutzte CSS Klassen werden durch *PurgeCSS* entfernt. Jedoch ist dies nicht die robusteste Art unge-

nutzten Code zu behandeln, da der Code immernoch in den Quelldateien verweilt. Durch den vollständigen Wechsel zu Tailwind CSS wird dieser Problematik jedoch ebenfalls entgegengewirkt, da in der Regel keine neuen CSS Klassen definiert werden. Wird ein Markupabschnitt bearbeitet oder entfernt, werden die gesetzten Utility-Klassen gleichmaßen behandelt (entfernt/bearbeitet). Diese Chance hat sich bereits jetzt schon in einigen Komponenten bewahrheitet.

6.1.3 Property Hell

Der Verhalt bzgl. der Property-Hell hat sich nur teilweise durch Tailwind CSS verbessert. Die Gründe für die Entstehung einer Property Hell sind vielseitig und zunächst nicht durch Storybook oder Tailwind CSS zu beseitigen. Um eine Verbesserung zu erlangen muss ein anderer Architekturstil von Vue-Komponenten aufgesetzt werden, der einer Property-Orientierung entgegenwirkt.

Eine Alternative zu Properties bieten *Slots*. Anders als Properties werden bei Slots nicht JavaScript-Typen weitergegeben sondern Codeabschnitte definierte in welche weiterer Code eingefügt werden kann (Vue.js, 2020b). Durch diese Art der Verschachtelung in der Eltern-Komponente wird ein hindurchreichen von Properties entgegengewirkt. Zudem lässt sich der Inhalt der Slots dynamisch gestalten, was Komponenten deutlich flexibler einsetzen lässt.

Meyer (2019) befürwortet in seinem Artikel eine ähnliche Herangehensweise. Die Vorteile listet er wie folgt:

- **Flexibilität**: Für neue Slot-Inhalte müssen keine neuen Properties definiert werden.
- **Komponentenweitergabe**: Auch Komponenten können in einen Slot eingefügt werden.
- **Keine If-Statements**: Properties müssen nicht mehr validiert werden.
- **Bessere Lesbarkeit**

Codebeispiel 6.1: ChatWindowContacts.vue

```
1  <template>
2    <div>
3      <entity v-for="(contact, i) in contacts" :key="i" :class="[
         roundedClass]">
4        <template #image>
5          <portal-avatar :imageName="contact.image">
6            <template #badge v-if="contact.unreadMessages > 0">
7              <badge>{{ contact.unreadMessages }}</badge>
8            </template>
9          </portal-avatar>
10       </template>
11       <template #caption>{{ contact.name }}</template>
12       <template #description>{{ contact.lastMessage}}</template>
13     </entity>
14   </div>
```

```
15 </template>
16
17 <script>
18 export default {
19   props: {
20     roundedClass: {
21       type: [String, Object, Array],
22       default: 'rounded'
23     }
24   }
25   // API Calls and more
26 }
27 </script>
```

Im Codebeispiel wurde außerdem eine *rounedClass* Property eingeführt. Diese dient dazu dynamisch Utility-Klassen an ein Markup-Element zu setzen. Dadurch müssen nicht mehr weiter dynamische Klassen definiert werden wie zum Beispiel eine Klasse *entity–rounded-bigger* die durch eine boolische Property *isRoundedBigger = true* gesetzt werden. Stattdessen kann der Entwickler eine *roundedClass* wie ein normales Class Attribut behandeln, dass *entity* gesetzt wird und den Standardwert mit genau den Werten überschreiben, die in seinem Kontext benötigt werden.

Ein Beispiel wäre, der *Entity* nur die *rounded* Erscheinung auf mobilen Geräten verleihen zu wollen:

> *roundedClass = "rounded md:rounded-none"* (mit diesem Vorgehen muss der Entwickler keine neue Prop wie *isRoundedOnlyMobile* definieren)

6.1.4 Post-Modularisierung von Komponenten

Das Verschieben von Markup-Abschnitten zusammen mit Utility-Klassen ist bei weiten nicht mehr so komplex. Jedoch ist es weiterhin schwierig die richtigen Script-Abschnitt herauszufiltern um eine Komponente inkl. Funktionalität zu extrahieren. In Abschnitt 7 soll geprüft werden ob Vue Version 3 neue Möglichkeiten bietet, die sich positiv auf diesen Prozess auswirken.

6.1.5 Tailwindkonfiguration statt Portalkonfiguration in portal-storybook

Tailwind CSS hat diese Chance letztendlich erfüllt. Die Repositorystruktur ist nun nicht mehr vom Client (portal-frontend) abhängig. Das Setup von *portal-storybook* ist jedoch für ein Projekt wie *portal-frontend* durch die verschiedenen Konfigurationsdateien etwas komplizierter gestaltet als zunächst für Standard-Vue-Applikationen angenommen.

6.2 Risikobewertung Tailwind CSS

6.2.1 Technologie Konflikte

Dieses Risiko hat sich mit der Einführung von Tailwind CSS bewahrheitet. Da die Frameworks Bootstrap und Tailwind CSS beide auf eigenen Reset.css Dateien basieren, die beide die Standardbrowser Styles anpassen, gab es viele Konflikte bei Style-Elementen, wie Überschriften, Textabschnitte, Links und Listen. Da jedoch Tailwind CSS langfristig als Stylebasis dienen soll, war es für My.PORTAL besser die Bootstrap Reset-Styles zu entfernen und die neuen Bugs im Portal manuell mit Tailwind CSS Utility Klassen wieder in die ursprüngliche Erscheinung zu gestalten. Zudem mussten einige Third-Party-Plugins auf CSS-Klassen geprüft werden, die nicht von PurgeCSS entfernt werden dürfen. Dies war zeitaufwendig jedoch auch nicht schwierig.

6.2.2 Weitere Risiken

Inwieweit die geringe Langzeiterfahrung mit Tailwind CSS und das bevorstehende Upgrade auf Node 12 Herausforderungen mit sich bringen, lässt sich schwierig prognostizieren und bewerten. Jedoch soll in Zukunft ein größeres Technologieupgrade im gesamten Portal geschehen, weshalb das Upgrade auf Node 12 ohnehin mittlerweile als Vorteil zu bewerten ist und die Upgrades an sich unter anderem für dieses Projekt als neue Chancen wie Risiken kategorisiert werden können.

6.3 Weitere Beobachtung

Nach der Implementierung und Nutzung von *portal-storybook* konnten unteranderem neue Herausforderungen, Chancen und Risiken aufgedeckt werden.

6.3.1 Neue Herausforderungen

- **Debugging: Codemarkup identifizieren**: Wenn man transpilierten Code im Browser zum untersuchen identifizieren möchte, kann durch die fehlenden individuellen Klassennamen eine schnelle Identifizierung des gesuchten Codeabschnitts schwierig sein. Ein Hilfe gegen dieses Problem sind der konsequente Einsatz von semantisch richtigen HTML5-Tags, jedoch tritt das Problem dennoch gelegentlich auf.
- **Längere *Build Time* auf Production**: Durch den zusätzlichen Einsatz von PurgeCSS hat sich die Buildtime eines production-stage Portals verdreifacht. Da SASS ebenfalls noch im Portal enthalten ist, ist eine Verbesserung abzusehen, da mit dem entfernen von SASS der Preprozessor und viele CSS Klassen entfernt werden. Jedoch ist es nicht einschätzbar wie positiv der Effekt im Hinblick auf die Build-Time ausfallen wird.

6.3.2 Neue Chancen und Risiken

Für Storybook, Tailwind CSS, Node.js und Vue.js stehen Upgrades auf neue Hauptversionen an. Diese Upgrades bringen allgemein neue Features und bessere Performance sind aber auch für eine dauerhafte Aktualität des Portals notwendig.

Tailwind CSS wird in Version 2 einige Deprecation-Änderungen enthalten, jedoch wird auf diese Änderungen bereits in den aktuellen Versionen hingewiesen und es gibt die Möglichkeit Tailwind bereits mit den neuen Änderungen zu verwenden.

Storybook wäre bereits möglich zu aktualisieren, allerdings besteht derzeit noch immer das Dokumentationsproblem aus Abschnitt 4.3.3. Solange dieses Problem nicht gelöst ist, wäre ein Upgrade unpraktisch. Daher ist dieses Upgrade trotz Chancen auf ein besseres (offizielles) Dokumentationsbild auch als ein Risiko zu betrachten, welches das Aktualisieren generell blockt.

Vue.js Version 3 ist bereits veröffentlicht. Jedoch sind die meisten eingesetzten Vue-Community-Projekte noch nicht entsprechend aktualisiert worden. Die AVACO GmbH plant mit dem Launch von Nuxt.js Version 3, welches auf Vue 3 aufbauen soll, ein Upgrade auf Vue Version 3 mit Nuxt.js zu tätigen. Mit diesem Upgrade sind viele Chancen und Risiken verbunden, die nicht direkt im Zusammenhang mit Tailwind CSS oder Storybook stehen werden. In Kapitel 7 wird ein Ausblick auf die Chancen gegeben.

7 Ausblick auf Vue.js Version 3

Mit einem Upgrade auf Vue Version 3 kommt es zu einigen Veränderungen und neuen Features für Vue-Entwickler.

Ein Feature welches optional aber für den Kontext dieser Arbeit besonders relevant ist, ist das Composition API. Dieses wurde für die Entwicklung von Komponenten in größeren Vue-Applikationen gestaltet und stellt eine Alternative zu der weiterhin verfügbaren Options API dar. Das Composition API wurde insbesondere im Hinblick auf die zwei folgenden Probleme entwickelt (Vue.js, 2020a):

- **Komplexe Komponenten verstehen**: Komponenten mit viel JavaScript-Code werden in der Options API nach Optionen kategorisiert. Für komplexe Komponenten ist es jedoch oft verständlicher die Komponente nach logischen Abschnitten zu organisieren.
- **Keine einfachen Mechanismen für Code-Extrationen**: Es gibt keine optimalen Möglichkeiten Code aus Komponenten so zu extrahieren um ihn in anderen Komponenten zu verwenden.

Beide Probleme sind auch die Hauptursachen warum in die Post-Modularisierung von Komponenten (vgl. Abschnitt 6.1.4) erschwert wird. Mit der Einführung der *setup()* Methode und logischen Strukturierung des Codes würde somit ein Mechanismus entstehen, der das extrahieren von Code erleichtert und generell zu einer anderen Struktur und besseren Lesbarkeit des Komponenten-Codes führt.

8 Fazit

Durch die Einführung von Tailwind CSS zusammen mit der neuen Komponentenbibliothek, wurden viele Herausforderungen in der Frontendarbeit von My.PORTAL angegangen.

Komponenten, die auf Tailwind CSS basieren sind nun im Hinblick auf ihre CSS Gestaltung leichter zu überblicken, da die Styledefintionen direkt im Markup nachvollziehbarer sind. Dynamisches setzten von Styleattributen ist durch Utility Klassen leichter geworden und insgesamt müssen kaum noch neue CSS Klassen erstellt werden. Das extrahieren von Markupabschnitten ist (ohne Logik-Teil) nun einfacher möglich und beugt zudem redundanten oder ungenutzten CSS Code vor. Insbesondere Mixins die vorher an unterschiedlichen Stellen vermehrt für ähnliche Zwecke verwendete wurden hatten viel redundanten Code erzeugtet und können nun durch Utility Klassen vollständig ersetzt werden.

In einem stetig wachsenden Projekt wie My.PORTAL war es bisher der Regelfall das mit jeder neuen Funktionalität auch neuer CSS Code erstellt wurde. Durch Tailwind CSS wird auf einer CSS Basis gearbeitet, die kaum weiter wächst und sich sehr gut komprimieren lässt.

Dadurch, dass die Komponentenbibliothek nun unabhängig von einer Portalkonfiguration nutzbar ist und stattdessen auf einer Tailwind-Konfiguration basiert, kann das Frontendteam robuste Vue-Komponenten erstellen, die unabhängig vom Projekt auch in anderen Projekten eingesetzt und angepasst werden können.

Insgesamt konnten in diesem Projekt alle Herausforderungen positiv verändert werden. Jedoch ist die Einführung von Tailwind CSS besonders in einem laufenden Projekt auch mit Risiken, die zusätzlicher Arbeit bedürfen, verbunden. So musste durch die neuen Reset-Styles eine umfangreiche visuelle Korrektur im Frontend vorgenommen werden. Zudem wirkt sich die parallele Verwendung mit zusätzlichen Frontendtechnologien wie SASS auf die Build-Performance des Frontends aus und ist durch die Utility-First-Prinzipien von Tailwind CSS langfristig auch nicht zu vertreten. Ein umfangreiches Refactoring des SASS Codes zu Tailwind CSS hat deshalb bereits begonnen und verfolgt das Ziel den SASS-Preprozessor komplett aus dem Projekt zu entfernen.

Weitere neue Herausforderungen, Chancen und Risiken ergeben sich außerdem auf die bevorstehenden Technologie-Upgrades. Ein Storybook Upgrade auf Version 6 kommt für dieses Projekt erst mit Einführung einer besseren offiziellen automatisierten Vue-Dokumentation in Frage und Tailwind CSS Version 2 wird auf Node Version 12 laufen müssen.

Neue Chancen für besser organisierte Komponenten ergeben sich durch die Einführungen der neuen Composition API in Vue.js Version 3.

Codeverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[AVACO GmbH 2020] AVACO GmbH: *Website*. https://www.avaco.io/portalloesung. Version: 2020. – Zuletzt aufgerufen am 20. September 2020

[Dayan 2019] Dayan, Sarah: *Redesigning Our Docs – Part 4 – Building a Scalable CSS Architecture*. https://blog.algolia.com/redesigning-our-docs-part-4-building-a-scalable-css-architecture/. Version: 2019. – Zuletzt aufgerufen am 09. Oktober 2020

[Frost] Frost, Brad: *Zitat von Brad Frost*. https://storybook.js.org/. – Zuletzt aufgerufen am 25. September 2020

[Frost 2013] Frost, Brad: *Atomic Design*. https://bradfrost.com/blog/post/atomic-web-design/. Version: Juni 2013. – Zuletzt aufgerufen am 22. September 2020

[Frost 2016] Frost, Brad: *The Workshop and the Storefront*. https://bradfrost.com/blog/post/the-workshop-and-the-storefront/. Version: Mai 2016. – Zuletzt aufgerufen am 25. September 2020

[Github a] Github: *React Styleguidist - Github Seite*. https://github.com/styleguidist/react-styleguidist. – Zuletzt aufgerufen am 25. September 2020

[Github b] Github: *Storybook - Github Seite*. https://storybook.js.org/. – Zuletzt aufgerufen am 25. September 2020

[Github c] Github: *Vue Design System - Github Seite*. https://vueds.com/. – Zuletzt aufgerufen am 25. September 2020

[Github d] Github: *Vue Styleguidist - Github Seite*. https://github.com/vue-styleguidist/vue-styleguidist. – Zuletzt aufgerufen am 25. September 2020

[Github Issue] Github Issue: *Storybook Issue auf Github*. https://github.com/storybookjs/storybook/issues/12495. – Zuletzt aufgerufen am 09. Oktober 2020

[Greif u. Benitte 2019] Greif, Sacha ; Benitte, Raphaël: *The State of CSS 2019*. https://2019.stateofcss.com/technologies/css-frameworks/. Version: November 2019. – Zuletzt aufgerufen am 20. September 2020

[Meyer 2019] Meyer, Nico: *Vue.js: slots vs. props*. https://medium.com/@nicomeyer/vue-js-slots-vs-props-af87078a8bd. Version: Februar 2019. – Zuletzt aufgerufen am 19. Oktober 2020

[Nguyen 2018] Nguyen, Dominic: *Storybook vs Styleguidist*. https://www.chromatic.com/blog/storybook-vs-styleguidist/. Version: Mai 2018. – Zuletzt aufgerufen am 25. September 2020

[Salesforce] Salesforce: *Lighting Design System - Design Tokens*. https://www.lightningdesignsystem.com/design-tokens/. – Zuletzt aufgerufen am 25. September 2020

[Salminen a] Salminen, Viljami: *Vue Design System*. https://vueds.com/. – Zuletzt aufgerufen am 25. September 2020

[Salminen b] Salminen, Viljami: *VUEDS - Dokumentation - Terminologie*. https://github.com/viljamis/vue-design-system/wiki/terminology. – Zuletzt aufgerufen am 25. September 2020

[Salminen 2018a] Salminen, Viljami: *VUEDS - Elements*. https://viljamis.com/2018/vue-design-system/#elements. Version: März 2018. – Zuletzt aufgerufen am 25. September 2020

[Salminen 2018b] Salminen, Viljami: *VUEDS - Patterns*. https://viljamis.com/2018/vue-design-system/#patterns. Version: März 2018. – Zuletzt aufgerufen am 25. September 2020

[Salminen 2018c] Salminen, Viljami: *VUEDS - Templates*. https://viljamis.com/2018/vue-design-system/#templates. Version: März 2018. – Zuletzt aufgerufen am 25. September 2020

[Storybook] Storybook: *Storybook*. https://storybook.js.org/. – Zuletzt aufgerufen am 25. September 2020

[Tailwindcss 2020] Tailwindcss: *Controlling File Size*. https://tailwindcss.com/docs/controlling-file-size. Version: 2020. – Zuletzt aufgerufen am 09. Oktober 2020

[Vue Styleguidist] Vue Styleguidist: *Vue Styleguidist*. https://vue-styleguidist.github.io/. – Zuletzt aufgerufen am 25. September 2020

[Vue.js 2020a] Vue.js: *Vue.js Composition API: Logic Reuse & Code Organization*. https://composition-api.vuejs.org/#logic-reuse-code-organization. Version: 2020. – Zuletzt aufgerufen am 20. Oktober 2020

[Vue.js 2020b] Vue.js: *Vue.js: Slots*. https://vuejs.org/v2/guide/components-slots.html. Version: 2020. – Zuletzt aufgerufen am 19. Oktober 2020