



Hochschule Bochum
Bochum University
of Applied Sciences
Campus
Velbert/Heiligenhaus



Erstellung eines Theme-Creators

Erweiterung der Barrierefreiheit der MEVIweb
HMI-Suite und Visualisierung durch selbsterstellte Themes

Autor: Philip Maas

Matrikelnummer: 015 200 898

Erstprüfer: Prof. Dr. rer. nat. Peter Gerwinski

Zweitprüferin: M. Sc. Benedikt Wildenhain

Abgabedatum: 12. April 2022

Abstract

Die IMS Messsysteme GmbH arbeitet aktuell an einem Paket verschiedener Microservices. Die Kernsoftware stellen die HMI-Suite, sowie die Visualisierung dar. Erstere bietet einen Editor an, mit der die Visualisierung des zweiten Services frei editierbar ist. So können durch eine Reihe von Komponenten wie Knöpfe, Textflächen oder Diagramme eigene Seiten erstellt werden, mit der automatisierte Anlagen in der Stahlindustrie überwacht und gesteuert werden können. Die Visualisierung ist aktuell in drei Farbvarianten erhältlich, welche nicht vollständig barrierefrei sind. In jeder Farbvariante – auch Theme genannt – werden beispielsweise Rot und Grün zur Signalkodierung genutzt, was Personen mit Rot-Grün-Schwäche vor eine Herausforderung bei der Bedienung und vor sicherheitskritischen Aspekten im Betrieb stellt. Obwohl einzelne Komponenten anpassbar sind, ist es nicht möglich ohne großen Aufwand ein gesamtes Theme zu erstellen. Deswegen soll in dieser Arbeit ein Theme-Creater erschaffen werden, mit dem Themes hinzugefügt, gelöscht oder editiert werden können.

The IMS Messsysteme GmbH is currently working on a package of various microservices. The core software consists of the HMI suite and the visualization. The former offers an editor in which the visualization of the second service can be freely edited. In this way, a series of components such as buttons, text areas or diagrams can be used to create own pages in order to monitor and control automated systems in the steel industry. The visualization is currently available in three color variants, which are not completely barrier-free. Each color variant – also called theme – uses red and green for signal encoding, which presents a challenge to people with red-green deficiencies and is safety-critical aspect of operation. Although individual components are customizable, it is not possible to create an entire theme without a great deal of effort. Therefore a theme creator is to be created in which themes can be added, deleted and edited.

Inhaltsverzeichnis

Inhaltsverzeichnis	i
Abkürzungsverzeichnis	iii
1 Einleitung	1
1.1 Projektumfeld	1
1.2 Motivation	1
1.3 Ist-Zustand	2
1.4 Projektziel	2
1.5 Intrinsische Motivation	3
2 Grundlagen	4
2.1 Einführung in die Webentwicklung	4
2.1.1 Genutzte Webtechnologien	5
2.1.2 HTML, CSS und SASS	5
2.1.3 JavaScript und TypeScript	9
2.1.4 jQuery	10
2.1.5 Das IMS-Komponentenmodell	11
2.1.6 JSON	14
2.1.7 REST	15
2.1.8 Websockets	17
2.1.9 Nutzung der DevTools	17
2.1.10 Node.js	20
2.2 Barrierefreiheit	20
2.2.1 Definition und Anforderungen	20
2.2.2 Farbenblindheit	22
2.2.3 Kontrastschwächen	24
3 Implementierung	26
3.1 Ausarbeitung eines technischen Konzeptes	26
3.2 Implementierung der View	30
3.3 Ausweitung zum Proof of Concept	34
3.4 Iterative Implementierung	36

3.5	Reviews und inkrementelle Verbesserungen	39
4	Fazit und Ausblick	43
4.1	Persönliches Fazit	43
4.2	Ausblick	44
	Abbildungsverzeichnis	I
	Tabellenverzeichnis	III
	Literatur	IV
A	Anhang	VII
A.1	Design-Visionen des Theme Creators	IX

Abkürzungsverzeichnis

Ajax	Asynchronous JavaScript and XML
BGG	Behindertengleichstellungsgesetz
CI	Corporate Identity
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheets
DOM	Document Object Model
GUI	Graphical User Interface
HMI	Human Machine Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IMS	IMS Messsysteme GmbH
JSON	JavaScript Object Notation
JS	JavaScript
MEVInet	Measuring and Visualization based on .NET
MEVIweb	Measuring and Visualization based on web technologies
QoL	Quality of Life
REST	Representational State Transfer
SASS	Syntactically Awesome Style Sheets
SMACCS	Scalable and Modular Architecture for CSS
SPS	Speicherprogrammierbare Steuerung
TS	TypeScript

URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WCAG	Web Content Accessibility Guidelines
WYSIWYG	What-You-See-Is-What-You-Get

1 Einleitung

Die Firma *IMS Messsysteme GmbH (IMS)* entwickelt und produziert berührungslose Messsysteme für die Prozess- und Qualitätskontrolle in Walzwerken der Metallindustrie.

1.1 Projektumfeld

Die IMS arbeitet aktuell an einem Softwarepaket, welches *Measuring and Visualization based on web technologies (MEVIweb)* genannt wird [1]. Es beruht, wie der Name es andeutet, auf den Webtechnologien *Hypertext Markup Language (HTML)*, *Cascading Style Sheets (CSS)*, *JavaScript (JS)* und *TypeScript (TS)*. Bei der Umsetzung wird auf ein eigenes Framework zurückgegriffen, welches wiederum auf jQuery basiert. In den Messstraßen findet die Datenerfassung und -speicherung durch speicherprogrammierbare Steuerungen statt, dessen Funktionalität in diesem Projekt als gegeben angenommen werden. Die erfassten Daten werden anschließend in einer Visualisierung angezeigt. Ebenso können die eingesetzten Akteure der Messstrecke über diese gesteuert werden. Das Layout der Visualisierung ergibt sich durch die *Human Machine Interface (HMI)*-Suite. Hier werden die einzelnen Seiten der Visualisierung durch einen *What-You-See-Is-What-You-Get (WYSIWYG)*-Editor per Drag & Drop kreiert. Dieser soll den Anwender*innen eine möglichst vollständige Freiheit bei der Kreierung der Visualisierung bieten.

1.2 Motivation

Die IMS verkauft ihre Messsysteme weltweit an verschiedene Stätten der Stahlproduktion. Dabei ist jede Messstraße individuell geplant und erbaut worden, sodass auch eine individuelle Visualisierung vonnöten ist. Hier ist der WYSIWYG-Editor der Suite, auch genannt *Page-Composer*, bereits ein mächtiges Werkzeug. Dennoch kommt es in regelmäßigen Abständen zu spezifischen Wünschen der Kundschaft, welche derzeit nur durch eine Änderung im Code erfüllt werden können. Ein Beispiel ist der gängige Wunsch die eigene *Corporate Identity (CI)* in der Visualisierung umzusetzen.

1.3 Ist-Zustand

Aktuell ist es in der HMI-Suite möglich die Attribute einzelner Kontrollelemente, auch *Bricks* genannt, zu verändern. Wird beispielsweise ein Button einer Seite hinzugefügt, kann in diesem die Farbe des optionalen Icons eingestellt werden. Die Schrift- und Hintergrundfarbe sind nicht veränderbar. Selbst bei gegebener Funktionalität müsste für die Erstellung eines konsistenten Themes diese Farbe in jedem einzelnen Button geändert werden. In der aktuellen Visualisierung gibt es 35 verschiedene Bricks, die in manchen Fällen ihre Farbeinstellungen voneinander erben oder ineinander verschachtelt sind. Bis dato ist es nicht möglich alle Farben der Kontroll- oder Hilfselemente der Visualisierung präzise einzustellen.

Alle Farbeinstellungen sind statisch im Code hinterlegt. Dabei wird ein Großteil des CSS in einer globalen Datei gepflegt, während die einzelnen Bricks ihre Stil-Attribute aus verschiedenen Quellen beziehen. Das widerspricht dem Kerngedanken des Komponentenmodells aus Abschnitt 2.1.5. Die aktuelle Implementation des Themings hat zudem den Nebeneffekt, dass bei einem Umschalten der Themes die gesamte Visualisierung neu geladen werden muss.

Ein Einbau der kundenspezifischen CI aus Abschnitt 1.2 würde demnach bedeuten, dass der Code angepasst werden muss. So entsteht für jeden Kunden mit dem Wunsch einer eigenen CI auch ein eigenes Setup, welches dokumentiert und gewartet werden muss. Dies würde die Ressourcen die IMS überschreiten.

1.4 Projektziel

In diesem Projekt soll ein Theme-Creator erstellt werden, mit dem alle möglichen Farben der Visualisierung global eingestellt werden können. So soll es neben den aktuell 35 Bricks möglich sein die Farben der Navigationsleiste, des Headers, des Hintergrundes, der Flächen für Kontrollelementgruppen, sowie verschiedener modale Fenster einzustellen.

Dafür soll in der HMI-Suite ein neuer Tab namens *Themes* erstellt werden. In diesem befindet sich die Möglichkeit der Farbeinstellung, sowie eine Live-Vorschau der Änderungen. Dabei muss eine einfache und strukturierte Auswahl der Komponenten über einen Menübaum möglich sein. Standardmäßig ineinander verschachtelte Elemente sollen hier sichtbar und getrennt voneinander editierbar sein. Beispielsweise soll ein Button in einer Tabelle individuell zum klassischen Button angepasst werden können. Ebenfalls soll die Möglichkeit

geschaffen werden neben den vier Standard-Themes eigene Custom-Themes zu erstellen. Dabei soll eine Vererbung bei der Neuerstellung implementiert werden. Die somit erstellten Themes werden dann in Form einer *JavaScript Object Notation (JSON)*-Datei im Konfigurationsverzeichnis des Kundenprojektes abgespeichert, während die Basis-Themes durch das Setup bezogen werden. So wird sichergestellt, dass firmenspezifische CIs erstellt werden können, während die Basis-Themes weiterhin durch die IMS gepflegt werden.

Des Weiteren soll ein Expertenmodus geschaffen werden, mit dem der Funktionsumfang des Theme-Creators für die Designer*innen der IMS erweitert werden kann. In diesem können zum Beispiel die Basis-Themes angepasst werden, während außerhalb des Expertenmodus nur Custom-Themes erstellt werden können.

Zuletzt soll das Umschalten zwischen zwei Themes so verändert werden, dass die Visualisierung nicht neu gestartet werden muss.

1.5 Intrinsische Motivation

Neben der oben genannten Motivation, welche aus betriebswirtschaftlicher Sicht Sinn ergibt, möchte ich an dieser Stelle den wissenschaftlichen Stil der Dokumentation brechen und eine persönliche Motivation in das Entwicklungsprojekt einbringen. Ich habe selbst starke visuelle Beeinträchtigungen. Neben der allgemeinen Kurzsichtigkeit und einer Hornhautverkrümmung, bin ich einseitig blind. Somit entfällt eine Tiefenwahrnehmung. Mein größtes Problem am PC ist allerdings oftmals meine Nachtblindheit. So kann ich vor allem bei schlechten Lichtverhältnissen nur schwer Kontraste erkennen. Dies macht sich besonders bei einer Kaskadierung von verschiedenen Helligkeitsabstufungen einer Farbe bemerkbar. Die Kombination all dieser Probleme macht es für mich schwierig die Informationen, welche die Visualisierung – oder auch andere Software – mir zur Verfügung stellt, in kurzer Zeit zu verarbeiten. Deswegen bin ich selbst immer wieder auf Barrierefreiheit in Software, vor allem auf sogenannte *High-Contrast-Themes* [2] und dem präzisen Nutzen von Warnfarben angewiesen. In meiner Freizeit beschäftige ich mich immer wieder mit dem Thema Barrierefreiheit in Software. Ich arbeite pädagogisch im GG eSport Jugendzentrum [3]. Während der Lockdowns habe ich in mehreren Terminen und zusammen mit dem *Spieleratgeber NRW* Online-Gesprächsrunden zum Thema Barrierefreiheit im Gaming für Jugendliche veranstaltet.

Aus diesen Gründen liegt mir die Erstellung eines Theme-Creators aus der Sicht der Inklusion und Barrierefreiheit persönlich am Herzen.

2 Grundlagen

In diesem Kapitel wird erklärt, warum die IMS mit dem MEVIweb auf Webtechnologien aufbaut. Dazu werden die genutzten Sprachen, Programmierparadigmen, Laufzeitumgebungen, Datenformate, Frameworks und Architekturmuster erklärt, um eine Wissensbasis für die Implementierung zu schaffen. Ebenso wird die Bedeutung der Barrierefreiheit im Rahmen der User Experience erforscht. Es soll geprüft werden, welche Punkte der Barrierefreiheit mit dem Theme-Creator erschlossen werden können.

2.1 Einführung in die Webentwicklung

Webtechnologien funktionieren in der heutigen Zeit auf einer Vielzahl von Endgeräten und Betriebssystemen. Sei es ein PC, Einplatinencomputer, Smartphone oder sogar eine Smartwatch. Die meisten modernen elektronischen Geräte sind mit mindestens einem Internetbrowser kompatibel. Da ein Großteil der aktuellen Betriebssysteme solche Browser sogar bereits beinhalten, bietet sich bei Neuentwicklung von Software das Programmieren per Webtechnologien an. Somit entfällt ein Optimieren für einzelne Systeme und es kann sich auf eine übergeordnete Plattform konzentriert werden. Die logikverarbeitende Anwendung läuft meist auf dem Server, anstatt auf dem Client im Browser. So muss eine benutzende Person die verwendete Software nicht installiert werden. Der Aufwand zum Aufruf der Anwendung beträgt lediglich das Öffnen eines *Uniform Resource Locator (URL)* im Browser. Auch die manuelle Aktualisierung von Software schwindet, da diese ebenfalls im Backend vorgenommen wird. In den genannten Vorteilen sieht auch die IMS ihre Zukunft, weshalb MEVIweb im Gegensatz zu *Measuring and Visualization based on .NET (MEVInet)* auf Webtechnologien basiert. So besteht das MEVIweb aus einer Ansammlung von Microservices, bei dem jeder seinen eigenen Zweck erfüllt. Als Hauptseite dient der ServiceManager, mit dem die anderen Services verwaltet und aufgerufen werden. Der SoftwareUpdater übernimmt die automatische Aktualisierung des MEVIwebs. Herzstück dieses Service-Pakets bieten allerdings die HMI-Suite und die Visualisierung, um die es hauptsächlich in diesem Projekt geht. Mithilfe der HMI-Suite kann eine vollumfängliche Visualisierung per Drag & Drop Editor erstellt werden, mit der die hauseigenen Stahlmesssysteme in der Produktion gesteuert und dazugehörigen Messungen eingesehen werden können.

2.1.1 Genutzte Webtechnologien

Für Webanwendungen werden hauptsächlich die Sprachen HTML, CSS und JS verwendet. Diese sind bereits viele Jahre in Nutzung, werden immer wieder an aktuelle Innovationen angepasst und erfreuen sich großer Beliebtheit [4]. HTML und CSS sind koexistent und können quasi nicht mehr unabhängig voneinander genutzt werden, um marktfähige Produkte zu schaffen. In diesem Projekt werden *Syntactically Awesome Style Sheets (SASS)* anstelle von nativem CSS genutzt. SASS ist ein CSS-Präprozessor und wirbt mit einer besseren Syntax, sowie dem Nutzen von Variablen, Grundrechenarten, Bedingungen und Schleifen innerhalb von CSS. Die drei genannten Sprachen werden zusammen mit TypeScript und JQuery zu dem sogenannten *IMS-Component-Framework* vereint. TS ist eine eigene Programmiersprache, welche JS erweitert. Im Backend wird Node.js genutzt [5]. In diesem Projekt wird der Server hauptsächlich zum Speichern und Laden der Konfigurationsdateien, sowie zum Auslösen von serviceübergreifenden Events genutzt. In den folgenden Abschnitten werden die verwendeten Technologien genauer erläutert.

2.1.2 HTML, CSS und SASS

HTML ist eine Auszeichnungssprache welche vorwiegend im World Wide Web genutzt wird. Sie befindet sich aktuell in der Version 5 und basiert auf dem XML-Standard [6]. So ist auch in HTML die wichtigste Struktureinheit das Element, welches durch Attribute in Form eines Schlüssel-Wert-Paares näher beschrieben werden kann. Elemente können wiederum Texte oder weitere Elemente enthalten. Es ist zu beachten, dass HTML eine reine Layoutsprache ist, in der Inhalte präsentiert werden. Für eine individuelle Darstellung der genutzten Elemente wird daher empfohlen eine CSS-Datei anzulegen. Dazu kann jedem HTML-Element ein Attribut `id` und/oder `class` samt Schlüssel zugeordnet werden. Während eine `id` einzigartig sein muss und nur mit einem spezifischen Element verknüpft werden darf, kann man eine `class` mehreren Elementen zuordnen. In Abbildung 2.1 wird dargestellt, wie die Schriftfarbe aller Elemente mit der Klasse `myClass` auf Rot und die Schriftfarbe der Elemente mit der ID `myID` auf Blau geändert wird.

```
1 .myClass { // Klassen werden mit einem Punkt referenziert...
2   color: red;
3 }
4
5 #myID { // ... und IDs mit einer Raute
6   color: blue;
7 }
```

Abbildung 2.1: Beispiel CSS

Wenn sich `myClass` und `myID` auf das gleiche Element beziehen, stellt sich die Frage, welche der definierten Stile Priorität hat. In dem Fall der Abbildung 2.1 ist die Frage noch einfach zu beantworten. Eine ID genießt eine höhere Gewichtung als eine Klasse, da sie spezifischer ist. In einer großen Software wie dem MEVIweb, in dem eine Mischung aus Vererbungen, globalen und komponenten-spezifischen CSS-Dateien, Inline-Styles, Style-Tags und eine Mischung aus Klassen und IDs verwendet wird, lässt sich die Prioritätsfrage nicht einfach beantworten. In Abbildung 2.2 ist umfangreich dargestellt, welche Stile eine höhere Priorität genießen. Doch auch mit dieser Abbildung ist es schwierig in einem großen Projekt den Überblick zu bewahren. Zum einen kann bei der Verschaffung eines Überblicks die Entwicklungskonsole aus Abschnitt 2.1.9 helfen. Zum anderen können innerhalb eines Teams Programmierparadigmen vereinbart werden. Einigt man sich zum Beispiel darauf immer nur css-Regeln auf Klassen anzuwenden, kann man den orange-umrandeten Teil aus Abbildung 2.2 um $\frac{2}{3}$ reduzieren. Auch von einer häufigen Verwendung des `!important`-Befehls ist abzuraten. Dies lässt sich nicht immer verhindern, da manchmal Stile aus Bibliotheken von Dritten überschrieben werden müssen. Eine häufige Verwendung im eigenen Projekt führt allerdings meist dazu, dass die CSS-Dateien mit diesem Befehl überflutet werden und dadurch die Autorität des Befehls reduziert wird.

Durch das alte Theming wurden häufig globale CSS-Regeln und Inline-Styles genutzt. Ebenso wurde an diversen Stellen auf den `!important`-Befehl zurückgegriffen. Mit der Implementierung des neuen Themings soll dies so weit wie möglich reduziert werden. So wird in Zukunft sichergestellt, dass die meisten CSS-Stile in der Datei der dazugehörigen Softwarekomponente liegen und klassenbezogen eingesetzt werden. Dadurch wird die Komplexität reduziert und Zeit bei zukünftigen Veränderungen und der damit eingehenden Einarbeitung in den Code gespart.

The definitive guide to CSS styling order

Includes CSS stylings for SVG

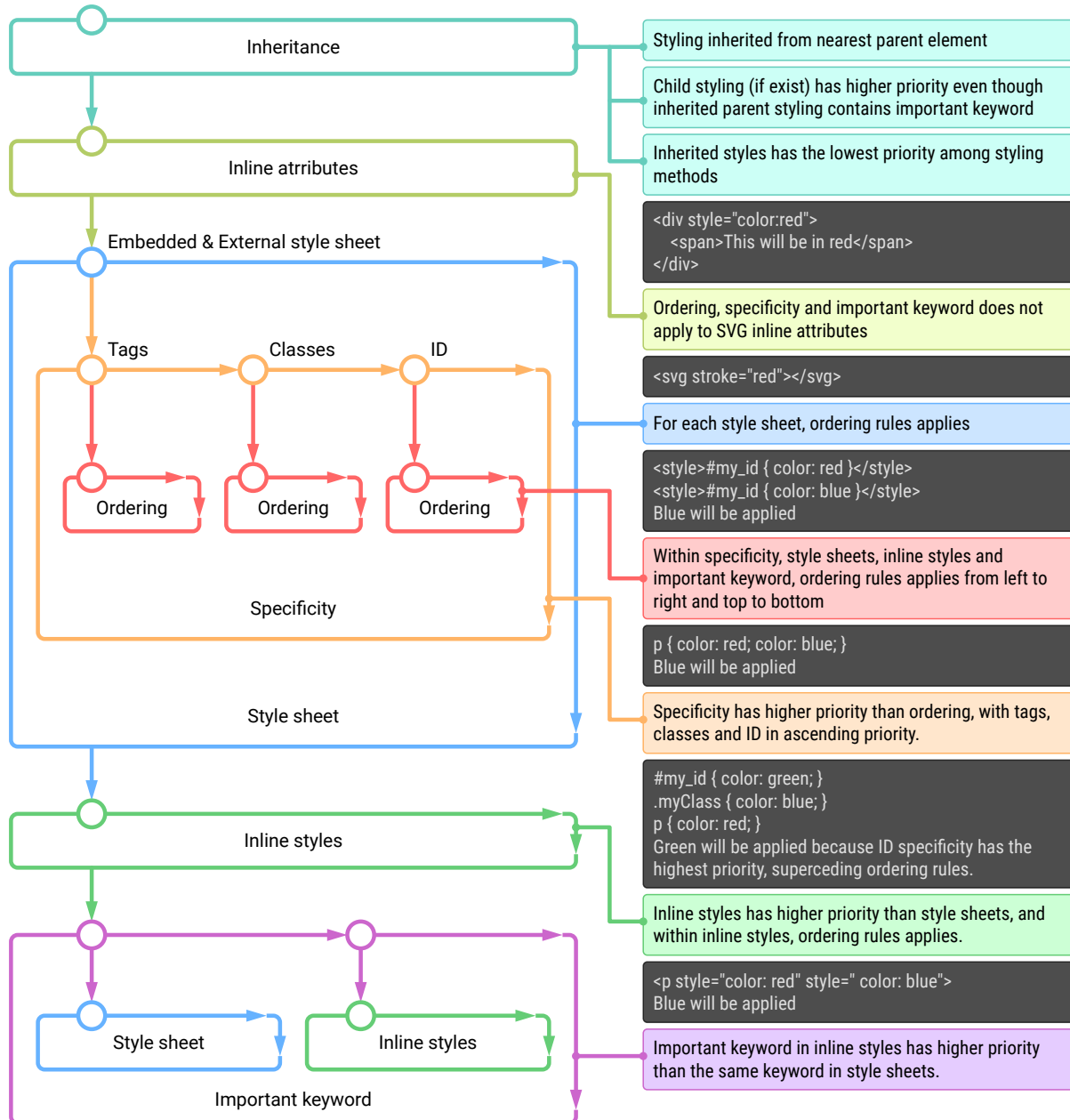


Abbildung 2.2: Prioritätsordnung CSS [7]

CSS ist eine mächtige Gestaltungssprache. Durch die Verwendung von Pseudo-Elementen [8] und Funktionen [9] kann Dynamik in sonst statisches Layout gebracht werden. Es

ist ohne die Verwendung von JS möglich hover-Effekte, radiale Farbverläufe und einfache Animationen zu erschaffen. Auch das Nutzen von Variablen ist wie in Abbildung 2.3 dargestellt möglich. In dem Pseudo-Element namens **root**, identifizierbar durch den Doppelpunkt, wird die Variable **myBlue** mit dem dazugehörigen Hex-Code erstellt. Dieser wird in Zeile 6 mit der Funktion **var** abgerufen. Zudem wird als zweiter Funktionsparameter ein *Fallback* hinterlegt, falls die Variable nicht gefunden werden kann.

```
1 :root {  
2   --myBlue: #1e90ff;  
3 }  
4  
5 body {  
6   color: var(--myBlue, blue);  
7 }
```

Abbildung 2.3: Nutzung von Variablen in CSS

Zuletzt kann die Struktur von CSS durch die Nutzung des Präprozessors SASS verbessert werden. SASS erweitert das klassische CSS durch Verschachtelung, eine einfachere Variablen-Erstellung und Schleifen. Vor allem die Schachtelung wird in diesem Projekt häufiger genutzt. In Abbildung 2.4 wird das Design einer Navigationsleiste links durch SASS und rechts durch CSS beschrieben. Durch die Verwendung von SASS wird erzwungen, dass alle Stile, die sich auf das HTML-Element **nav** beziehen, an einem Ort stehen. In CSS kann es passieren, dass nicht alle Regeln, die sich auf **nav** beziehen an einer Stelle stehen. Meistens bezieht sich hier das Dateilayout auf die Implementationsreihenfolge. In einer langen Datei kann durch die Verwendung von SASS schneller die richtige Stelle für eine Design-Änderung oder -Erweiterung gefunden werden.

<pre>1 nav { 2 ul { 3 margin: 0; 4 padding: 0; 5 list-style: none; 6 } 7 8 li { display: inline-block; } 9 10 a { 11 display: block; 12 padding: 6px 12px; 13 text-decoration: none; 14 } 15 }</pre>	<pre>1 nav ul { 2 margin: 0; 3 padding: 0; 4 list-style: none; 5 } 6 7 nav li { 8 display: inline-block; 9 } 10 11 nav a { 12 display: block; 13 padding: 6px 12px; 14 text-decoration: none; 15 }</pre>
---	--

Abbildung 2.4: Beschreibung einer Navigation: SASS vs. CSS

2.1.3 JavaScript und TypeScript

JavaScript ist eine Skriptsprache mit dem Ziel Webseiten dynamische Aspekte zu verleihen. Aus diesem Grund ist JS im Gegensatz zu anderen populären Sprachen eine asynchrone Programmiersprache. Normalerweise wird Code linear von der ersten Zeile bis zur letzten ausgeführt, während nur eine Aktion zu einem gegebenen Zeitpunkt passiert. In JavaScript ist dies nicht der Fall [10]. Es wird, ausgehend von der Nutzung im Web, vorzugsweise mit Callbacks, Timern, Events und asynchroner Funktionalität, wie Promises oder `async/await`, gearbeitet. So wird zum Beispiel bei einem Knopfdruck nicht die gesamte Webseite blockiert, während eine rechenintensivere Operation im Hintergrund läuft. Durch die asynchronen Konzepte und der vorwiegenden Nutzung eines einzelnen Threads kann ein blockierungsfreier Ablauf gewährleistet werden.

Die Programmiersprache TypeScript wurde 2012 von Microsoft unter der Apache-Lizenz veröffentlicht und basiert auf JavaScript. Code in TypeScript wird von einem Compiler in eine JavaScript-Datei übersetzt. Der Unterschied beider Sprachen ist, dass TypeScript mehr an objektorientierte Programmiersprachen angelehnt ist, was den Einstieg erleichtern soll. Hierbei wird im Gegensatz zu JavaScript vom Compiler darauf geachtet, dass Datentypen angegeben und korrekt genutzt werden. Das aus JavaScript und Python bekannte Duck-Typing, also die automatische Interpretation von Datentypen, soll dadurch vermieden werden [11]. So wird ein syntaktisch korrekter Programmierstil erzwungen, was

die Arbeit in Teams erleichtert und die universelle Lesbarkeit des Codes sichern soll. Aus diesen Gründen wächst die Beliebtheit von TS firmenintern und wird in neuen Projekten im Gegensatz zu JS bevorzugt genutzt.

In diesem Projekt wird im Frontend mit TypeScript und im Backend mit JavaScript gearbeitet. Dieser Umstand ist historisch gewachsen. In Zukunft sollen die Backends der einzelnen Microservices auf TypeScript aktualisiert werden.

2.1.4 jQuery

jQuery ist die populärste und am meisten genutzte JavaScript-Bibliothek [12]. Sie existiert seit 2005 und befindet sich aktuell in der Version 3.6 und erlaubt das Abrufen Manipulieren des HTML-Dokumentes, auch *Document Object Model (DOM)* genannt. Dabei ist die Schreibweise kompakter als klassisches JS. Jedes HTML-Element, welches durch jQuery abgerufen wird, ist von einem eigenen jQuery-Wrapper-Objekt umschlossen.

In Abbildung 2.5 wird ein On-Click-Event an einen Button mit der ID `myBtn` gebunden. Hier wird die kompakte Schreibweise durch jQuery sichtbar, da es möglich ist den Zugriff auf einzelne oder mehrere Elemente mit den aus CSS bekannten Selektoren zu vollziehen.

Zudem ist es mit jQuery möglich neue Elemente zur Laufzeit ins DOM zu injizieren, Daten per *Asynchronous JavaScript and XML (Ajax)* zu übertragen oder Animationen zu implementieren. In diesem Projekt werden hauptsächlich die ersten beiden beschriebenen Funktionen verwendet.

```
1 // klassisches JS
2 document.getElementById('myBtn').onclick = () => {
3     console.log('Hello World!');
4 }
5
6 // jQuery
7 $('#myBtn').on('click', () => {
8     console.log('Hello World!');
9 });
```

Abbildung 2.5: klassisches JS vs. jQuery

2.1.5 Das IMS-Komponentenmodell

In HTML werden bestimmte Elemente mit einer vorgesehenen Funktionsweise verknüpft. So wird beispielsweise beim `input`-Element immer noch ein Attribut `type` erwartet. Durch die Kombination des Elements und des Typs wird dem Browser mitgeteilt, wie das Element zu interpretieren und darzustellen ist. Dadurch unterscheiden sich die, in diesem Projekt verwendeten, Input-Typen *button*, *color*, *text* und *number* in Aussehen und Funktionsweise.

Die vom Browser vorgegebene Darstellung der klassischen Elemente ist nicht immer von Vorteil. Hier setzen verschiedene Browser auf unterschiedliche Darstellungsweisen oder sogar Funktionalitäten. Für eine plattformübergreifende Konsistenz muss mit spezifischen Sonderregeln gearbeitet werden. Ansonsten muss die Software beim Start auf Browserkompatibilität überprüft werden. Deshalb wird oft auf eigens implementierte Komponenten zurückgegriffen. Dafür werden Gruppierungselemente wie `div` und `span`, mit eigenen Stilen zu neuen Elementen kombiniert.

Der Programmcode der MEVIweb-Services ist in *Views* und *Components* aufgeteilt. Eine View entspricht einer sichtbaren Seite im Client. So besitzt die HMI-Suite zum Beispiel fünf Views, die als fünf Tabs dargestellt sind:

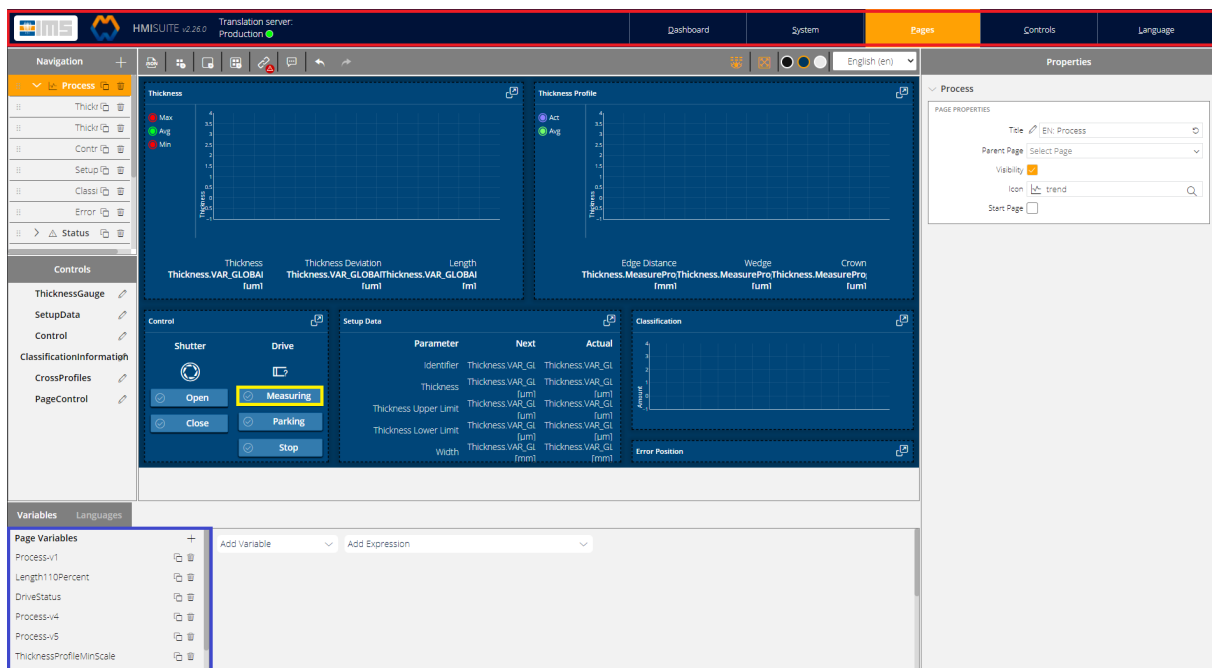


Abbildung 2.6: Beispielhafte Components in der View Page-Composer

In der Abbildung 2.6 befindet sich ein Ausschnitt der View *Page-Composer*, welche verschiedene Komponenten enthält. Eine Komponente ist ein modularer, systemunabhängiger Baustein in der Webtechnologie, dessen Nutzen vergleichbar zu einer Klasse in der Objektorientierten Softwareentwicklung ist. In der blauen Box sieht man die Komponente `ims-binding-variable-tree`, welche sich wiederum in einem `ims-panel` befindet. Dieses stellt einen generischen, skalierbaren Container für ein konfigurierbares Layout dar. Der gelb markierte `ims-ccu-button-brick` ist eines der Bricks welche in der Visualisierung sichtbar sind. Bricks sind also Komponenten, die durch die nutzende Person manipuliert werden können. Dieser CCU-Button befindet sich in der `ims-grid`-Komponente, welche das Herzstück des WYSIWYG-Editors darstellt und alle Bricks enthalten kann.

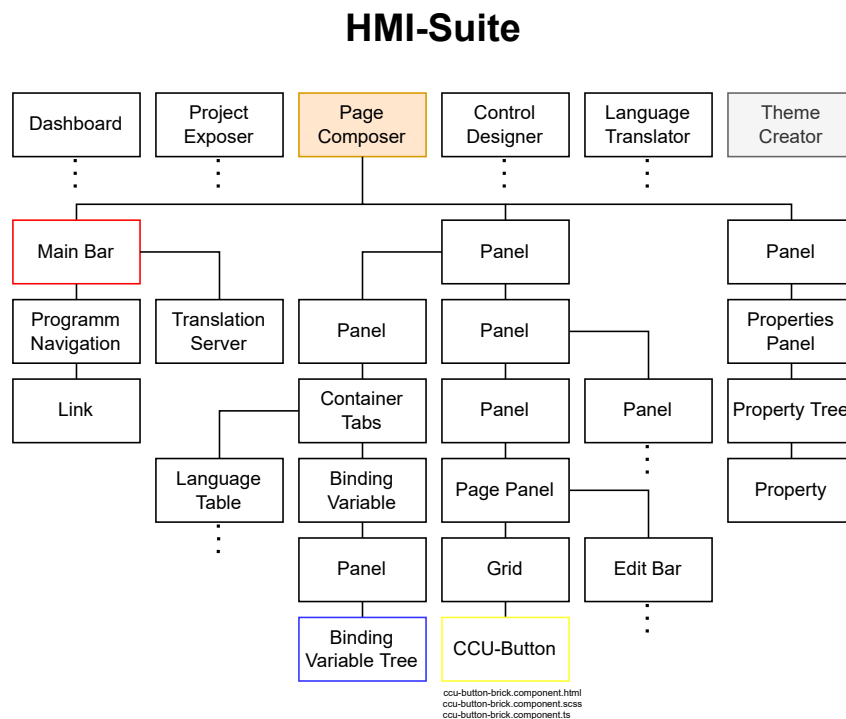


Abbildung 2.7: Beispielhafte Components in der View-Component-Sicht

Diese Nutzung von ineinander verschachtelten Komponenten innerhalb einer View wird Komponentenmodell genannt und findet in Webanwendungen häufig nutzen. Bekannte Frameworks wie Angular, React, Vue.js oder Svelte bauen ebenfalls auf diesem Modell auf [13].

In Abbildung 2.7 ist die Ansicht aus Abbildung 2.6 aus der Komponentensicht aufge-

Komponenten gegeben. Eine größere Darstellung ist im Anhang in Abbildung A.1 sichtbar. Das Wichtigste ist, dass die einzelnen Bricks im unteren Bereich der Abbildung eine Vielzahl an Abhängigkeiten zu anderen Komponenten aufweisen. Diese müssen bei der Erstellung des Theme-Creators aus zwei Gründen beachtet werden: Zum einen darf das CSS der Bricks, welche sich auf die gleiche Komponente beziehen, sich nicht gegenseitig überschreiben. Das heißt, dass Farbeinstellungen die im CCU-Button vorgenommen werden andere Buttons nicht beeinflussen dürfen. Andererseits muss der anwendenden Person die Möglichkeit gegeben werden ineinander verschachtelte Komponenten frei zu gestalten. Die Farbgebung eines Buttons im größeren Event-Table-Filter Brick muss unabhängig vom normalen Button individualisierbar sein.

Entsprechend der Architektur des Komponentenmodells soll in Kapitel 3 für den Theme-Creator eine neue *View* erstellt werden.

2.1.6 JSON

Die JavaScript Object Notation, kurz JSON, ist wie die Extensible Markup Language, kurz XML, ein text-basiertes, hierarchisch aufgebautes und vom Menschen lesbares Datenformat. JSON wird im Web zum Austausch von Daten, meist in Form von Objekten, verwendet. Dabei werden Objekte immer zwischen zwei geschweiften und Arrays zwischen zwei eckigen Klammern gespeichert. In diesen befinden sich Schlüssel-Wert-Paare, also Objektattribute mit einem Wert. Die Werte können als Ganz- oder Gleitkommazahl, Boolean, String oder null vorliegen. Da Objekte und Arrays in jeder möglichen Art und Weise ineinander verschachtelt werden können, kann schnell ein komplexes Gebilde entstehen. Es folgt ein JSON-Beispiel mit einem Eltern-Objekt, welches ein Array von zwei Kind-Objekten beinhaltet


```
1 {  
2   "name": "Georg",  
3   "alter": 47,  
4   "verheiratet": false,  
5   "beruf": null,  
6   "kinder": [  
7     {  
8       "name": "Lukas",  
9       "alter": 19,  
10      "schulabschluss": "Gymnasium"  
11    },  
12    {  
13      "name": "Lisa",  
14      "alter": 14,  
15      "schulabschluss": null  
16    }  
17  ]  
18 }
```

Abbildung 2.9: Beispiel JSON [json]

Der Vorteil von JSON ist neben der Lesbarkeit, welche beim Debuggen hilft, die universelle Nutzung des Formats in den meisten der gängigen Programmiersprachen. So können JSON-Dateien nativ in JavaScript eingelesen, direkt als Objekt genutzt und weiterverarbeitet werden. JSON wird in diesem Projekt für das Übertragen der Konfigurationen vom Client zum Server, sowie das Laden und Speichern dieser verwendet.

2.1.7 REST

Das Sicherheitskonzept eines Browsers sieht vor, dass die Webseiten in einer Art Sandbox laufen und möglichst geringe Zugriffsrechte besitzen. Meist beschränken sich diese darauf, ein Seitenlogo oder Bilder auf Root-Ebene des Projektes laden zu können. Ein fehlendes Sandboxing stellt ein Sicherheitsrisiko dar, da so Angreifer*innen leichten Lese- und Schreib-Zugriff auf den Computer hätten. Aus diesem Grund sind die Microservices in Client und Server aufgeteilt. Da das Backend nicht im Browser, sondern über Node.js läuft, kann es auf den Rechner zugreifen. Für die Kommunikation zwischen Server und Client wird im MEVIweb der *Representational State Transfer (REST)* genutzt. REST ist ein Programmierparadigma basierend auf dem *Hypertext Transfer Protocol (HTTP)*. Mögliche Befehle sind GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS und TRACE. In diesem Projekt werden nur GET, POST und PUT genutzt. Diese werden anhand vom Beispiel in Abbildung 2.10 erklärt:

- POST wird genutzt, um einen neuen Eintrag zu erstellen. So wird in Abbildung 2.10 die Arbeitskraft mit dem Namen Paul vom Client im Server angelegt.
- GET verlangt nach einer Information oder Inhalt vom Server. In Abbildung 2.4 fragt der Client die Arbeitskraft mit der Nummer 21 ab und bekommt das zuvor angelegte Paul-Objekt zurück.
- PUT wird genutzt, um eine Information, die zuerst per POST erstellt wurde, zu aktualisieren. So kann der *status* von Paul zu *unemployed* geändert werden. Bei jeder Anfrage erwartet die Gegenstelle eine Antwort in Form eines HTTP-Statuscode, zum Beispiel ein *OK (200)*.

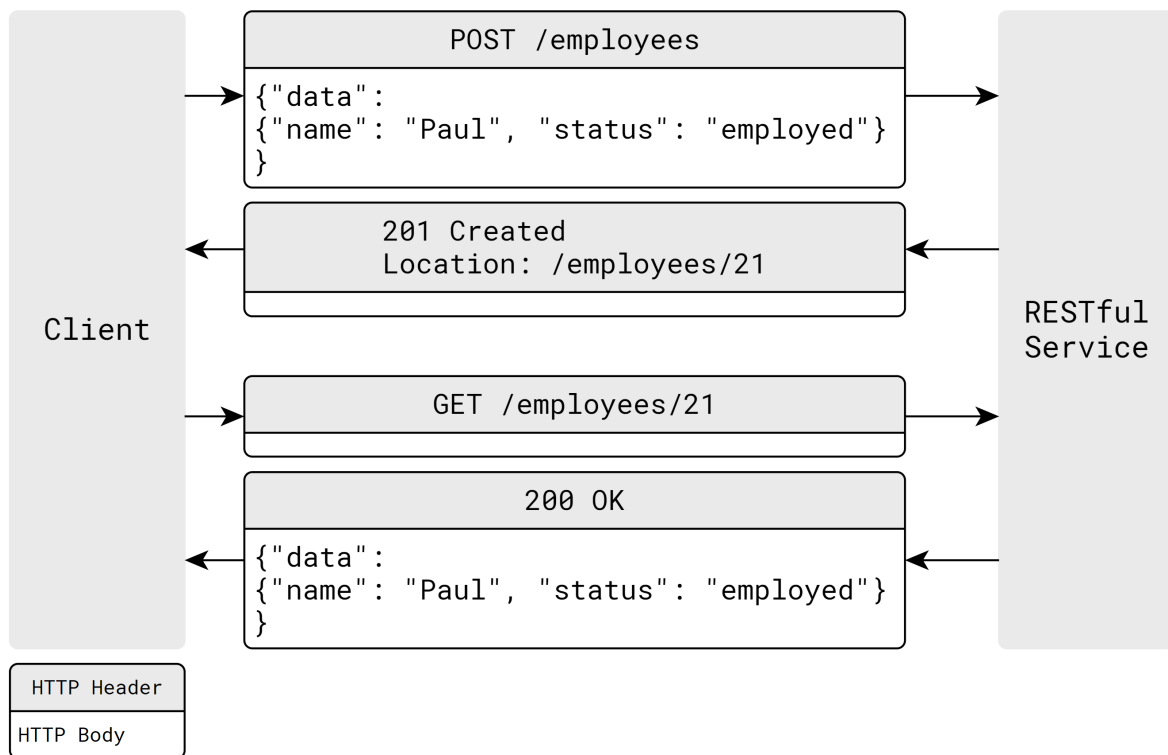


Abbildung 2.10: Beispielhafter Ablauf von REST

In diesem Projekt werden hauptsächlich die Befehle GET und PUT verwendet, sodass der Server zum Start direkt prüft, welche Dateien der Client benötigt und bei fehlenden Ressourcen Standards erstellt. REST wird genutzt, um die JSON Dateien aus Abschnitt 2.1.6 abzurufen oder einen Speicherwunsch zu vermitteln.

2.1.8 Websockets

REST eignet sich in Fällen, in denen eine Ressource vereinzelt abgefragt oder aktualisiert werden muss. Werden in kurzer Zeit viele kleine Ressourcen bearbeitet, dann entsteht bei REST das Problem, dass für jede Ressource die Verbindung zwischen Server und Client neu auf- und abgebaut werden muss. In einem solchen Fall bietet sich die Nutzung von Websockets an. Hier wird einmalig eine bidirektionale Verbindung aufgebaut und bleibt bis zur Schließung bestehen. Abbildung 2.11 zeigt die Kommunikation von einer Server-Client-Architektur via Websockets.

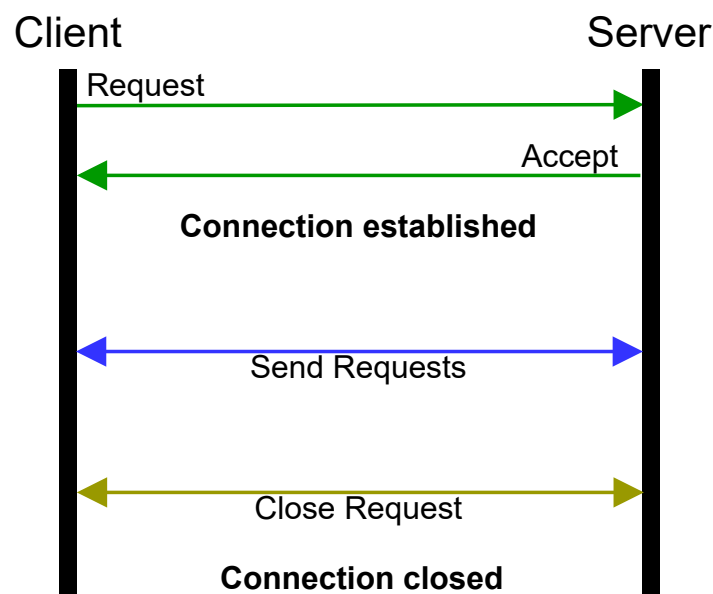


Abbildung 2.11: Kommunikation via Websockets [15] WEBSOCKET PROTOCOL

2.1.9 Nutzung der DevTools

In der Webentwicklung werden häufig auf die DevTools des Browsers zurückgegriffen, da mit dieser eine Fülle an Informationen abgerufen werden kann. In diesem Projekt wird der Chrome-Browser und die damit gelieferte Konsole genutzt. Der Funktionsumfang ist bei den gängigen Browsern ähnlich und die Prinzipien übertragbar. In diesem Abschnitt sollen die für dieses Projekt häufig verwendeten Funktionen ergründet werden.

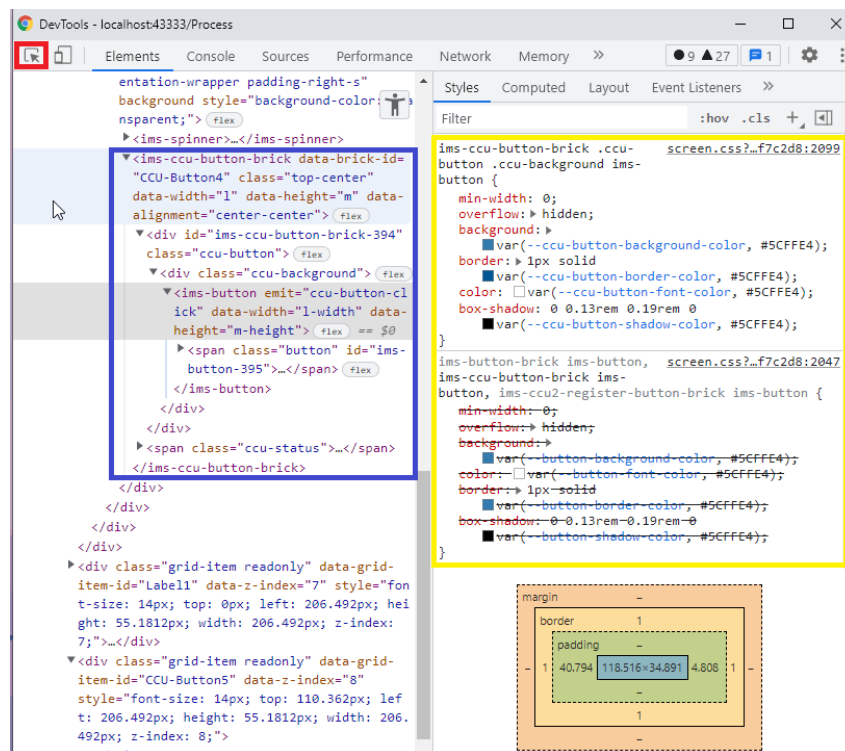


Abbildung 2.12: Tab *Elemente* der DevTools

Die DevTools sind über die F12-Taste erreichbar. Es ist direkt der Tab *Elements* sichtbar, in dem auf der linken Seite das HTML-Dokument zu erkennen ist. Klickt man ein bestimmtes Element an, werden auf der rechten Seite die dazugehörigen CSS-Styles sichtbar. Alternativ kann über den rot umrandeten Cursor ein Element direkt aus der Webseite ausgewählt werden. Diese Funktion ist in Abbildung 2.13 sichtbar. Hier kann auch direkt der Kontrastwert eingesehen werden, welcher in 2.2.3 näher behandelt wird. In diesem Beispiel wurde ein CCU-Button ausgewählt, wie man in der blau umrandeten Fläche in Abbildung 2.12 sehen kann. Innerhalb des CCU-Buttons befindet sich das HTML-Element `ims-button`. Dies deckt sich mit den Informationen aus Abbildung 2.8 bzw. A.1, da der CCU-Button die Button-Komponente verwendet. Hier sieht man auch im gelben Kasten, wie in Abschnitt 2.1.5 beschrieben, dass sich verschiedene CSS-Regeln überschreiben und woher die einzelnen CSS-Regeln stammen. Die dabei verwendeten Regeln entsprechen der Implementierung aus Kapitel 3. Zuletzt kann unten rechts in Abbildung 2.12 die Größe, der Außen- und Innenabstand und die Dicke der Umrandung des CCU-Buttons betrachtet werden.

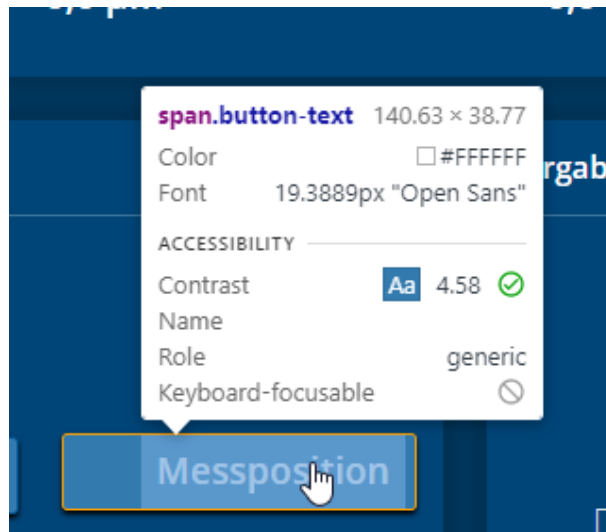


Abbildung 2.13: Barrierefreiheit in den DevTools

In dem Tab *Console* kann man die Konsole einsehen. Hier erscheinen alle Meldungen, die im Code mit dem `console`-Objekt erzeugt werden. Die Konsole bietet zwei wichtige Vorteile: Mehrfach geworfene Fehler werden mit einem Counter versehen, anstatt sie mehrmals in die Konsole zu schreiben. Außerdem können ausgegebene Objekte direkt untersucht werden. So ist es beispielsweise möglich sich ein jQuery-Wrapper-Objekt aus Abschnitt 2.1.4 mit allen möglichen Attributen und Funktionen zu untersuchen.

Der letzte, in diesem Projekt wichtige, Reiter ist der **Application**-Tab. Hier können die gespeicherten *Cookies* und der *Local Storage* eingesehen werden. Letzterer ist in Abbildung 2.14 zu sehen. Die rot markierten Schlüssel-Wertpaare sind in diesem Projekt hinzugefügt worden, um View- und serviceübergreifend das eingestellte Theme zu speichern.

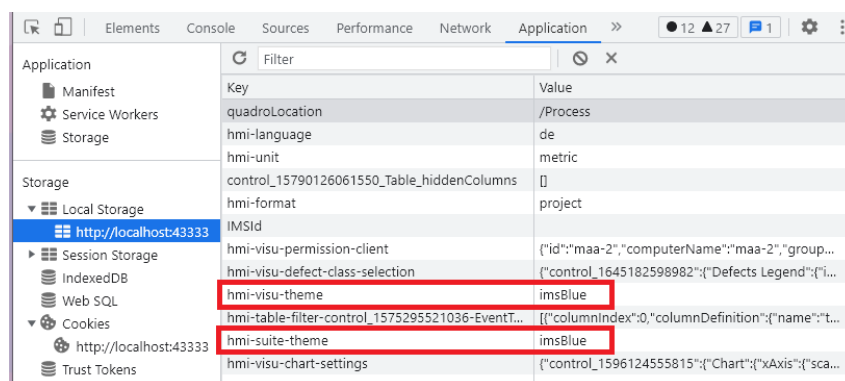


Abbildung 2.14: Speicherfunktion des Browsers

2.1.10 Node.js

Node.js ist eine, auf JS basierende, Laufzeitumgebung, die am 27. Mai 2009 veröffentlicht wurde. Mittlerweile befindet sich diese in der experimentellen Version 17.6 bzw. in der empfohlenen Version 16.14 mit Langzeitunterstützung. In diesem Projekt wird aus Kompatibilitätsgründen die Version 12.18 genutzt. Aktuell wird Node.js von der OpenJS Foundation gewartet und weiterentwickelt. Der Vorteil von Node.js ist, dass ein gesamtes Server-Client-Projekt mit nur einer Programmiersprache, nämlich JavaScript, bearbeitet werden kann. In erster Linie wurde Node.js für das Internet of Things entwickelt und genutzt, um netzwerkbasierte Anwendungen effizient zu gestalten. Deshalb arbeitet Node.js, ebenso wie JS im Browser, asynchron und benötigt für Anfragenverarbeitungen keine weiteren Threads, welche sich gegenseitig blockieren könnten. Es basiert auf der Google V8 Engine, welche auf Performance ausgelegt ist und auch im Chromium-Browser genutzt wird. Da Node.js in Form einer node.exe unter Windows gestartet werden muss, kann hier ein Lese- und Schreibzugriff auf die Festplatte gewährleistet werden. Somit ist es möglich, dass der Server Konfigurationsdateien abspeichern und wieder einlesen kann. Aus diesen Gründen und der theoretischen Echtzeitfähigkeit von Node.js wird diese bei der IMS als Basis für die Serverlogik der Microservices genutzt [5]. Da Node.js natives JS an einigen Stellen erweitert, wird es auch zu den Frameworks gezählt.

2.2 Barrierefreiheit

Die Barrierefreiheit ist Teil der menschenzentrierten Gestaltung aus der DIN EN ISO 9241-220 [16]. Sie trägt im Rahmen der Vermeidung nutzungsbedingter Schäden zur Qualität bei, indem sie Risiken negativer Auswirkungen minimiert. Kann beispielsweise im laufenden Betrieb eines Stahlwerkes eine Fehler- nicht von einer Erfolgsmeldung unterschieden werden, so entsteht ein sicherheitsrelevantes Risiko in der Nutzung.

2.2.1 Definition und Anforderungen

Die Barrierefreiheit ist in Deutschland durch §4 das *Behindertengleichstellungsgesetz (BGG)* definiert:

“Barrierefrei sind bauliche und sonstige Anlagen, Verkehrsmittel, technische Gebrauchsgegenstände, Systeme der Informationsverarbeitung, akustische und visuelle Informationsquellen und Kommunikationseinrichtungen sowie andere gestaltete Lebensbereiche, wenn

sie für Menschen mit Behinderungen in der allgemein üblichen Weise, ohne besondere Erschwernis und grundsätzlich ohne fremde Hilfe auffindbar, zugänglich und nutzbar sind. Hierbei ist die Nutzung behinderungsbedingt notwendiger Hilfsmittel zulässig.” [17]

Die Anforderungen der Barrierefreiheit entscheiden sich demnach anhand des Kontextes. Dieses Projekt kann aus der vorliegenden Gesetzesgrundlage am besten als System der Informationsverarbeitung, sowie als visuelle Informationsquelle eingeordnet werden. Da es sich bei den Microservices der IMS gleichzeitig um eine Webtechnologie handelt, wird in diesem Projekt auf Anforderungskatalog *Web Content Accessibility Guidelines (WCAG)* des *World Wide Web Consortium (W3C)* zurückgegriffen.

Der WCAG beschreibt Richtlinien, um Inhalte im Internet möglichst barrierefrei zu gestalten. Im folgenden wird der englische Begriff, sowie eine deutsche Erklärung zum Inhalt des WCAG 2.1 [18] aufgezählt.

1. Perceivable - Wahrnehmbar

- 1.1 **Text Alternatives:** Verfügbarkeit von Textalternativen für alle Nicht-Text-Inhalte.
- 1.2 **Time-based Media:** Verfügbarkeit von Alternativen für multimediale Inhalte, beispielsweise Audiotranskription.
- 1.3 **Adaptable:** Inhalte, die auf verschiedene Arten dargestellt werden können, ohne dass Informationen oder Strukturen verloren gehen.
- 1.4 **Distinguishable:** Einfachheit Inhalte zu sehen und zu hören, klare Trennung zwischen Vorder- und Hintergrund.

2. Operable - Bedienbar

- 2.1 **Keyboard Accessible:** Alle Funktionalitäten sind per Tastatur verfügbar.
- 2.2 **Enough Time:** Ausreichend Zeit, Inhalte zu lesen und zu benutzen.
- 2.3 **Seizures and Physical Reactions:** Vermeidung von Inhalten, die physische Reaktionen wie Epilepsie auslösen.
- 2.4 **Navigable:** Mittel, welche die Navigation und das Finden von Inhalten unterstützen.
 - a) Verfügbarkeit von Alternativen zur Tastatureingabe.

3. Understandable - Verständlich

3.1 **Readable:** Verständlich- und Lesbarkeit von Textinhalten.

3.2 **Predictable:** Vorhersehbarkeit von Webseiten in Funktionalität und Aussehen.

3.3 **Input Assistance:** Unterstützung zum Vermeiden und Korrigieren von Fehlern.

4. Robust - Robust

4.1 **Compatible:** Maximierung der Kompatibilität mit aktuellen und zukünftigen Werkzeugen, einschließlich assistierender Techniken.

Der Theme-Creator fällt unter den Punkt 1.4 des WCAG. Genauer gesagt werden mit der Einführung des Theme-Creators die Punkte 1.4.1 **Use of Color** und 1.4.3 **Contrast (Minimum)** beziehungsweise **Contrast (Maximum)** abgedeckt. In den Abschnitten 2.2.2 und 2.2.3 sollen Ursache dieser Punkte, sowie mögliche Probleme bei der Nutzung näher erörtert werden.

2.2.2 Farbenblindheit

Die Farbenblindheit ist ein Sammelbegriff diverser Farbschwächen, die auf ein Fehlen bestimmter Zäpfchen zurückzuführen sind.

Die jeweiligen Zäpfchen sind für die damit eingehende Farbwahrnehmung zuständig, somit bedeutet ein Fehlen bestimmter Stäbchen im Umkehrschluss die Blindheit für eine bestimmte Farbe. Die klassische Farbenblindheit, auch **Achromatie** genannt, bezieht sich auf ein Fehlen jeglicher Zäpfchen. So ist lediglich die Wahrnehmung von Kontrasten möglich. Die Achromatie tritt mit einer Chance von $1/30.000$ extrem selten auf [19]. Meistens ist mit der Verwendung des Begriffs Farbenblindheit fälschlicherweise eine Rot-Grün-Sehschwäche oder **Dichromatie** gemeint. Hierbei fallen entweder die roten oder grünen Zäpfchen weg, sodass die beiden Farbkanäle zu einem gelben Kanal zusammengefasst werden. In Abbildung 2.15 ist eine vereinfachte schematische Darstellung der Netzhaut, samt der roten, grünen und blauen Zäpfchen, sowie den fehlenden Zäpfchen bei den oben genannten Beeinträchtigungen dargestellt. In Abbildung 2.16 ist der Unterschied verschiedener Farbwahrnehmungen anhand von vier Beispielen zu erkennen.

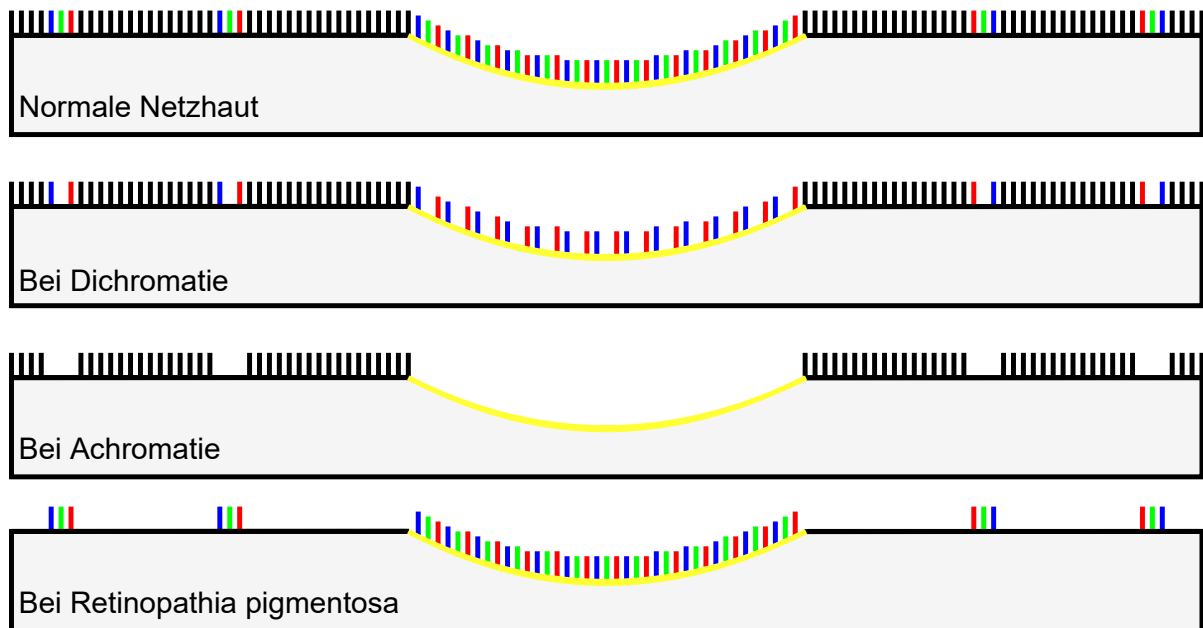


Abbildung 2.15: Vereinfachte Darstellung der Netzhaut bei verschiedenen Sehschwächen

Durch den Theme-Creator kann das Problem der Achromatie nicht direkt gelöst werden. Natürlich können diverse Grautöne zur Darstellung der Visualisierung gewählt werden. Signalfarben sollten hier im besten Fall durch den Einsatz von Icons oder Strichstärken ersetzt werden. Auch der Einsatz von Alternativen zur farblichen Darstellung sind in Punkt 1.4.1 des WCAG gefordert. Diese sind bereits in der Visualisierung vorhanden, sodass sich in diesem Projekt ausschließlich auf die Farbgebung konzentriert wird.

Die Dichromatie kann durch den Einsatz eines eigenen Themes ausgeglichen werden. Hier müssen Rot und Grün als Signalfarben durch Gelb und Blau ersetzt werden. Durch den Theme-Creator kann also eine Visualisierung für Menschen mit Rot-Grün-Schwäche erstellt werden.



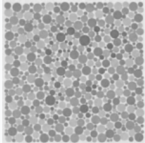









Motiv	Trichromatisches Bild	Dichromatisches Bild ohne Rot-Grün-Unterscheidung	Achromatisches Bild in Graustufen
Pseudoisochromatische Farbtafel			
Obststand			
Mosaikfenster			
Regenbogen			

Abbildung 2.16: Verschiedene Effekte der Farbenblindheit [20]

2.2.3 Kontrastschwächen

In Abbildung 2.15 erkennt man neben den Zäpfchen für die Farbwahrnehmung auch die schwarz dargestellten Stäbchen. Diese sind für die Helligkeitswahrnehmung, vor allem beim *Nachtssehen* oder *Dämmerungssehen* zuständig [21]. Mit den Stäbchen können dementsprechend keine Farben, sondern nur Grautöne wahrgenommen werden. Die Kontrastempfindlichkeit des Menschen wird durch diverse Faktoren gestört, beispielsweise aufgrund einer Linsentrübung, einer Pupillenverengung, Kurzsichtigkeit oder der Störung beziehungsweise dem Fehlen der Stäbchen [22]. Die letztere Beeinträchtigung kann durch die seltene Krankheit *Retinopathia pigmentosa* ausgelöst werden und ist ebenfalls in Abbildung 2.15 dargestellt.

Die visuelle Aufnahme von Informationen erfolgt immer durch einen Hinter- und einen Vordergrund. In diesem Abschnitt wird beispielsweise schwarzer Text auf einem weißen Hintergrund dargestellt. Diese beiden Farben besitzen ein Verhältnis zueinander, welches *luminance ratio* – oder fälschlicherweise *contrast ratio* – genannt wird [23]. In diesem Fall

ist der einheitsfreie Wert 21. In manchen Fällen wird die *luminance ratio* mit 21:1 betitelt, um die Existenz eines Verhältnisses zu verdeutlichen. Dabei ist 21 der höchstmögliche, während 1 den niedrigsten Wert und somit die Nicht-Existenz eines Kontrastes darstellt. Dieser Kontrastwert wird in den Entwicklungstools angezeigt und ist damit in Abbildung 2.13 zu sehen. In den WCAG wird die *Color Tutor demonstration* von Tom Jewett [24] empfohlen, um das Kontrastverhältnis zweier Farben selbst zu erkunden.

In der Visualisierung befindet sich Information, meist in Form von Texten, Icons oder Diagrammen auf einem Hintergrund. Da weder im Hinter- noch im Vordergrund Muster verwendet werden, muss laut Punkt 1.4.6 des WCAG ein Kontrastwert von 7:1 erreicht werden. Mit einem Kontrast von 7:1 können auch Personen mit Kontrastschwäche alle Texte, unabhängig der Größe, identifizieren können. Durch die Nutzung des Theme-Creators können schnell und einfach Themes mit hohen Kontrasten erzeugt werden, welche sogar das Verhältnis 7:1 überschreiten.

3 Implementierung

In diesem Kapitel sollen die erarbeiteten Grundlagen aus Kapitel 2 umgesetzt werden, um einen Theme-Creator zu erschaffen. Dafür muss erforscht werden, mit welchen Mitteln generell ein Theming umgesetzt werden kann. Nach der Ausarbeitung des technischen Konzepts sollen die View und ein Proof of Concept implementiert werden. Im Anschluss können interaktiv alle Bricks auf das neue Theming umgestellt werden. Aufgrund der weitreichenden Code-Änderungen, müssen alle Neuerungen ausführlich getestet werden, bevor es weltweit produktiv genutzt wird.

3.1 Ausarbeitung eines technischen Konzeptes

In diesem Abschnitt soll ein technisches Konzept für das neue Theming erarbeitet werden. Dabei wird betrachtet, wie das alte Theming aufgebaut war, welche Probleme dadurch entstanden sind und wie diese behoben werden können.

Das alte Theming ist an das Buch *Scalable and Modular Architecture for CSS (SMACCS)* von Jonathan Snook [25] und einem Artikel von Zell Liew angelehnt [26].

Der Kerngedanke des alten Themings ist, dass es im Projekt nur eine `Theme.scss`-Datei gibt. Dort werden über eine SASS-Map die Farben der einzelnen Themes abgelegt. Im Anschluss wird mit der SASS `@each`-Schleife eine CSS-Datei erzeugt, die Style-Regeln für alle Klassen und Komponenten enthält. Abbildung 3.1 enthält beispielhafte Implementierung des alten Themings. Somit wird das Theming klar von den einzelnen Komponenten getrennt.

```
1 // Color definitions for every theme in a SCSS-Map
2 $themes: (
3   blue-theme: (
4     primary-background: rgba(1, 67, 109, 0.66),
5     secondary-background: rgba(1, 67, 109, 1),
6   ),
7   light-theme: (
8     primary-background: rgb(243, 243, 243),
9     secondary-background: rgb(229, 229, 229),
10  ),
11  dark-theme: (
12    primary-background: rgba(24, 43, 58, 0.66),
13    secondary-background: rgba(24, 43, 58, 1),
14  ),
15 );
16
17 // This each-loop will build all css rules for every theme and class:
18 @each $theme, $map in $themes {
19   .#{$theme} {
20     .gradient-background {
21       background: radial-gradient(map-get($map, primary-background),
22       map-get($map, secondary-background));
23     }
24   }
25 }
26
27 // CSS-Output after compiling SCSS:
28 .blue-theme .gradient-background {
29   background: radial-gradient(rgba(1, 67, 109, 0.66), rgba(1, 67, 109, 1));
30 }
31
32 .light-theme .gradient-background {
33   background: radial-gradient(rgb(243, 243, 243), rgb(229, 229, 229));
34 }
35
36 .dark-theme .gradient-background {
37   background: radial-gradient(rgba(24, 43, 58, 0.66), rgba(24, 43, 58, 1));
38 }
```

Abbildung 3.1: Umsetzung des alten Themings

In Zeile 2 der Abbildung 3.1 wird die SCSS-Map mit dem Namen `$themes` erstellt. In dieser befinden sich für jedes Theme die Variablen für einzelne Farben. In der `@each`-Schleife ab Zeile 18 werden die vorher eingestellten Farben dann in die CSS-Regeln für bestimmte Klassen, (Pseudo-)Elemente oder IDs eingesetzt. Das daraus resultierende kompilierte CSS ist ab Zeile 27 zu betrachten.

Das alte Theming weißt theoretisch den Vorteil auf, dass alle Theme-spezifischen Regeln an einer Stelle definiert sind. Dadurch soll die Pflege erleichtert und die Code-Komplexität

reduziert werden.

In der Anwendung werden eine handvoll Nachteile sichtbar: Dadurch, dass die Kernsoftware in zwei Microservices mit einem shared-Ordner aufgeteilt ist, gibt es drei `Theme.scss`-Dateien. In allen drei Dateien sind die selben Farbvariablen hinterlegt. Die Änderung einer Farbe in einem Theme hat also das Bearbeiten von drei Dateien zur Folge.

Da das Theming in einer eigenen Datei gepflegt wird, müssen in den einzelnen Bricks eigene SCSS-Regeln implementiert werden, welche direkt durch das Theming überschrieben werden. Da ein Brick nicht unabhängig von einem Theme existieren kann entsteht so redundanter Code. Unerfahrene Entwickler*innen wissen nicht, in welcher Datei sie das CSS anpassen müssen, um eine sichtbare Änderung zu erzielen. Anstatt alle Bricks in den `Themes.scss`-Dateien zu überschreiben und damit schwer überblickbare Abhängigkeiten zu schaffen, können die Theme-spezifischen Regeln auch direkt in den verwendeten Komponenten definiert werden. Dieser Umstand wird bereits im Artikel von Liew angesprochen [26].

Zuletzt muss für eine farbliche Änderung immer der Programmcode geändert und neu kompiliert werden. Bei einem Theme-Wechsel ist ebenfalls ein Neustart nötig. Im neuen Theming ist also gefordert, dass das Theming direkt in den Komponenten passiert. Zudem soll es eine einzige Theme-Datei geben, mit der eventuell nötige Funktionalität zur Verfügung gestellt wird. Das neue Theming soll es möglich machen eine dynamische Veränderungen einzelner Farben zu vollziehen.

Das neue Theming soll also möglichst auf globale CSS-Dateien, sowie redundante Style-Regeln verzichten und eine Änderung des CSS zur Laufzeit möglich machen. Dafür werden die *CSS Custom Properties* verwendet [27].

Durch die CSS Custom Properties können Variablen definiert werden, ohne SCSS zu nutzen. Diese sollen in der `index.html` am Ende des `head`-Elementes innerhalb eines eigenen `style`-Elementes automatisiert definiert werden. Dafür wird eine statische Klasse geschrieben, welche das `style`-Element samt aller Theme-Variablen zum aktuell aktiven Theme als String zurück gibt. Die erzeugte Ausgabe in der `index.html` soll vergleichbar zu Abbildung 3.2 sein.

```
1 <head>
2   <!-- [...] Alle bisherigen Einträge im head -->
3
4   <style id="themes">
5     :root {
6       --primary-background: rgba(1, 67, 109, 0.66);
7       --secondary-background: rgba(1, 67, 109, 1);
8
9       --table-header-background: #003C67;
10      --table-row-background-even: #004678;
11      --table-row-background-odd: #003C67;
12
13      /* Weitere Theme-Variablen */
14    }
15  </style>
16 </head>
17 <body>
18   <!-- Sichtbarer Inhalt der HMI-Suite oder Visu -->
19 </body>
```

Abbildung 3.2: HMTL-Aufbau im neuen Theming

Die statische Klasse zur Generierung der CSS-Variablen soll `themes.ts` heißen. In dieser Klasse wird die Funktion `initThemes` benötigt, welche die Werte aus der `Themes.json` Datei über den Server ausliefert und einen *Promise* zurück gibt. Diese Funktion wird vor der Initialisierung des restlichen Systems ausgeführt. Erst wenn die Theme-Variablen geladen wurden, wird die Suite oder die Visualisierung gestartet.

Die `initThemes` soll dann das `style`-Element generieren und am Ende des `head`-Elements einfügen. Durch die Verwendung des Promise wird sichergestellt, dass das System erst lädt, wenn alle Styles initialisiert wurden, um ein unerwünschtes Flackern bei der Veränderung des CSS zur Laufzeit zu vermeiden.

Die Visualisierung der Veränderung eines aktuell ausgewählten Themes oder das Umschalten von Themes kann dann durch ein Austauschen des `style`-Elementes mittels jQuery realisiert werden. Somit muss die Seite nicht neu geladen werden.

Der große Vorteil der Lösung mittels CSS Custom Properties besteht darin, dass während der Definition der Styles im Programmcode weder die Farbwerte noch die verwendeten Themes bekannt sein müssen. Ebenso müssen keine separaten Regeln pro Theme definiert werden. Dennoch wird das Theming direkt in den Komponenten implementiert, sodass redundanter Code und das gegenseitige Überschreiben von Style-Regeln verhindert werden.

Innerhalb der SCSS-Dateien der Komponenten können die Variablen und ein Default-Wert vergleichbar zu Abbildung 3.3 zugewiesen werden. Dieser Default-Wert wird einmalig als globale CSS-Variable definiert, damit sie in jeder Komponente nutzbar ist. Dabei wird ein auffälliger Farbton gewählt, der eine fehlende Farbe signalisieren soll. Dieser Farbton wird ansonsten nicht in den Basis-Themes zu finden sein.

```
1 .view {  
2   background-color: var(--background-color, $missingColor); // missingColor = #5CFFE4  
3 }
```

Abbildung 3.3: Zuweisung der Theme-Variablen in CSS

Aus dem technischen Konzept des neuen Themings resultiert ein Nachteil: die bereits verwendeten CSS-Funktionen `lighten` und `darken` sind nicht mehr verwendbar. Somit ist ein zusätzlicher Aufwand beim Umbau auf das neue Theming erforderlich. Allerdings zwingt dieser Umstand die Designer*innen und Entwickler*innen fest definierte Farben zu verwenden anstatt mit Funktionen zu arbeiten, bei denen der genaue Effekt und die zukünftige Stabilität nicht gegeben sind. Dies sorgt für einen konsistenten und nachvollziehbaren Code. Somit kann dieser Nachteil auch als Vorteil betrachtet werden.

In den folgenden Abschnitten soll dieses technische Konzept schrittweise erarbeitet werden.

3.2 Implementierung der View

Um das technische Konzept aus Abschnitt 3.1 zu einem Proof of Concept zu evolvieren, muss ein geeignetes Frontend zur Wahl eines Themes und zum Festlegen der Theme-Farben erstellt werden. In diesem Abschnitt soll die Vision des Designers aus Abbildung 3.4 umgesetzt werden. Weitere Mock-Ups sind im Anhang zu finden. Insbesondere die Abbildungen A.3, A.6, A.7 und A.8 werden in diesem Abschnitt beachtet.

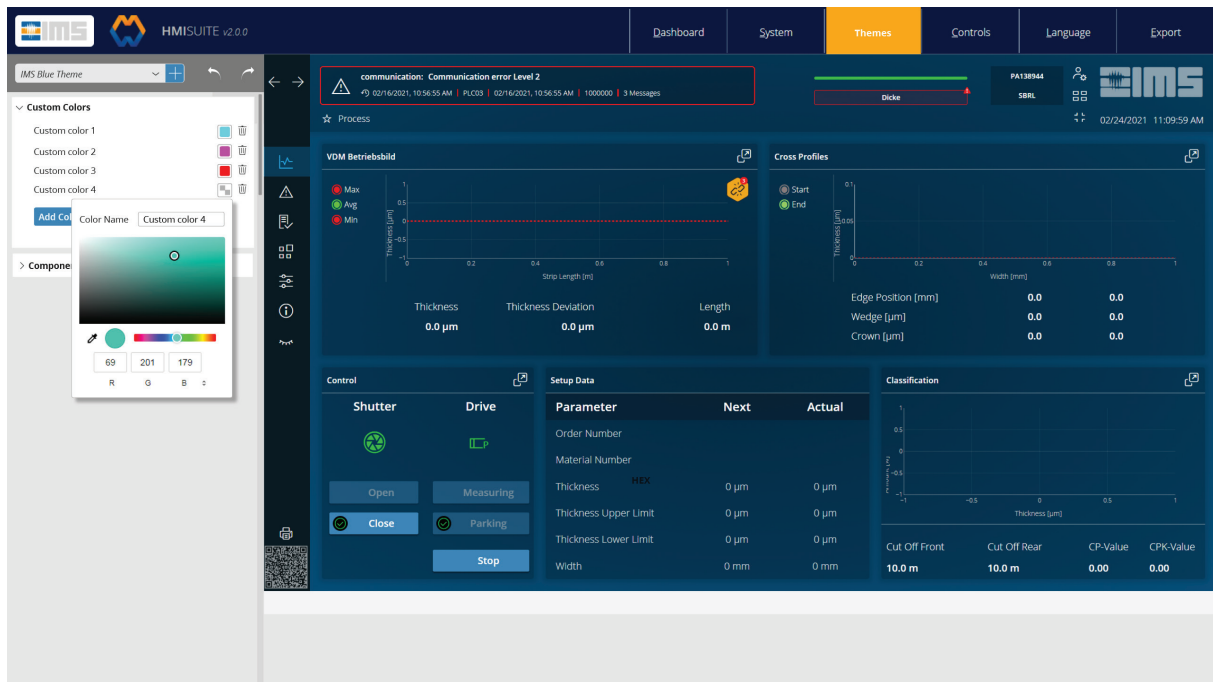


Abbildung 3.4: Mock-Up des Theme-Creators

In Abbildung 3.4 sind zwei wichtige Features des Theme-Creators zu erkennen: ein Auswahlbaum der einzelnen Komponenten, sowie eine Vorschau der Visualisierung. Für die Vorschau muss eine geeignete Lösung gefunden werden. Eine Möglichkeit ist es eine repräsentative Attrappe der Visualisierung zu bauen. Dabei entsteht das Problem, dass man auf einer Seite nicht alle wichtigen Bricks darstellen kann. Es müsste je nach Auswahl im Baum der richtige Brick samt Verwendungskontext angezeigt werden. Dies bedeutet einen hohen zusätzlichen Programmieraufwand. Da die tatsächliche Visualisierung die Aufgabe einer Vorschau besser als eine Attrappe erfüllt, kann diese per `iframe`-Element in die Seite eingebunden werden. So soll spätestens ab dem Proof of Concept aus Abschnitt 3.3 die farbliche Änderung im tatsächlichen Betrieb sichtbar sein. Da es möglich sein soll den URL des `iFrames` zu verändern, wurde das Mock-Up aus Abbildung 3.4 um eine editierbare URL-Leiste erweitert.

Die linke Seite aus Abbildung 3.4 ist das Herzstück des Theme-Creators. Hier soll es möglich sein ein bestehendes Theme auszuwählen oder ein Neues zu erstellen. Wie man in Abbildung A.7 sehen kann, darf ein neues Theme nur durch Vererbung realisiert werden. Das erste Custom-Theme erbt dementsprechend von einem Basis-Theme. Alle folgenden Themes können dann auch von Custom-Themes erben. Bei Custom-Themes soll ein Löschen des gesamten Themes möglich sein.

Wurde ein Theme ausgewählt, werden alle bearbeitbaren Komponenten und Bricks aufgelistet. Pro editierbares Attribut muss es einen Eintrag im Baum geben, welcher einen Color-Picker enthält. Ebenso soll die Möglichkeit geschaffen werden globale Farben anzulegen, welche in Abbildung 3.4 als *Custom Colors* bezeichnet werden. Somit soll das Hinterlegen und Wiederverwenden von eigenen Farb-Variablen, wie zum Beispiel einer primären Text- oder Hintergrundfarbe, möglich gemacht werden. Die Variablen können dann an einzelne Attribute der Komponenten gebunden werden. So muss bei einem (Re-)design des ausgewählten Themes die Farbe einmal umgestellt werden, anstatt alle Komponenten einzeln zu bearbeiten.

Für den Auswahlbaum wurde die neue Komponente `ims-theme-tree` erstellt, welche vom `ims-tree` erbt. Der Aufbau des Baumes erfolgt durch die Klasse `themeDictionary`, welche das Objekt `themeDict`, sowie Funktionen zum Hinzufügen, Verwalten und Entfernen der globalen Farben enthält. In Abbildung 3.5 ist ein beispielhafter Aufbau des `themeDict`-Objektes erkennbar.

```
1  'category': {
2    'title': 'Category',
3    'parentId': null,
4    'expert': false,
5  },
6  'component': {
7    'title': 'Component',
8    'parentId': 'category',
9    'expert': false,
10 },
11 'Attribute': {
12   'title': 'Default Font Color',
13   'parentId': 'component',
14   'expert': false,
15   'controls': [
16     'alpha-picker',
17     'empty',
18     'color-picker',
19     'select-internal'
20   ],
21 },
```

Abbildung 3.5: Beispielhafter Aufbau des ThemeDict-Objekts

Die einzelnen Objekte im `ThemeDict` können unterschiedliche Schlüssel-Wert-Paare besitzen. Diese sind abhängig von der Funktionalität, die der dazugehörige Knoten im Baum anbieten soll. Besitzt ein Objekt drei Einträge handelt es sich um einen Knoten, der wei-

tere Kind-Elemente aufweisen wird. Dies ist in den ersten beiden Einträgen in Abbildung 3.5 zu sehen, in dem eine *Kategorie* mit dem Kind-Knoten *Komponente* erstellt wird, welcher wiederum Kind-Blätter aufweisen wird. Beide Objekte enthalten einen Titel, den Schlüssel eines möglichen Eltern-knotens, sowie den Expert-Schlüssel. Dieser boolsche Wert gibt an, ob mit dem dazugehörigen Eintrag im Baum interagiert werden kann. Der Expert-Modus soll in Zukunft per Konfigurationsdatei umschaltbar sein, damit die IMS-Designer*innen einen höheren Grad an Freiheit als die Kund*innen genießen können. So sollen beispielsweise die Basis-Themes nicht ohne die Expert-Berechtigung bearbeitbar sein. In Zukunft sollen einige Attribute, zum Beispiel die Größe des Schattens einer Komponente, durch den Expertenmodus freischaltbar sein. Somit wurde der Schlüssel bereits jetzt im **ThemeDict** eingebaut, auch wenn dieser im Projektverlauf noch nicht genutzt wird.

Eine Komponente enthält n Attribute, welche durch den Theme-Creator bearbeitet werden sollen. Damit in Zukunft auch Attribute editiert werden können, die unabhängig von der Farbe sind, enthalten diese Objekte den **controls**-Array, indem verschiedene Manipulationsmöglichkeiten an die Blätter des Baumes gebunden werden können. In diesem Projekt werden hauptsächlich der alpha-picker für das Einstellen eines Alpha-Kanales, der color-picker zum Wählen der Farbe und select-internal für das Auswählen der globalen Farben genutzt.

Im **ims-theme-tree** wird der Baum anhand des **ThemeDict** initialisiert. Daraufhin werden die Blätter des Baumes mit den jeweiligen Controls versehen. Zuletzt werden für die Controls Events initialisiert, um beispielsweise das Speichern eines Themes bei der Änderung einer Farbe oder eines Alphakanals auszulösen.

Die globalen Farbvariablen, die in Abbildung 3.4 mit *Custom Colors* heißen, sollen in diesem Projekt *Internal Colors* genannt werden. Diese stellen im **ims-theme-tree** und damit auch im **ThemeDict** eine Ausnahme dar, da diese nicht statisch vorgegeben sind, sondern von der anwendenden Person hinzugefügt beziehungsweise gelöscht werden können. Somit muss es ein **add-internal-control** geben, welches das **ThemeDict** und damit den Baum manipulieren kann. Dies erfolgt allerdings nur zur Laufzeit. Auch die *Internal Colors* werden mit dem jeweiligen Theme in einer JSON-Datei abgespeichert und erst beim Initialisieren des Baumes geladen. Ein Hinzufügen oder Löschen der *Internal Colors* resultiert dementsprechend in einer permanenten Veränderung der JSON-Datei und nur in einer temporären Manipulation des **ThemeDicts** für die aktuelle Laufzeit der Theme-Creator-View.

Zuletzt soll es möglich sein die Änderungen im Theme-Creator rückgängig zu machen. Dafür wurde die bereits bestehende Komponente `ims-history-controls` genutzt. In Abbildung 3.6 kann die fertiggestellte View des Theme-Creators gesehen werden. Im nächsten Schritt soll die Möglichkeit geschaffen werden, dass eine Farbänderung an der Node *Page Background* eine Änderung in der Visualisierung hervorruft.

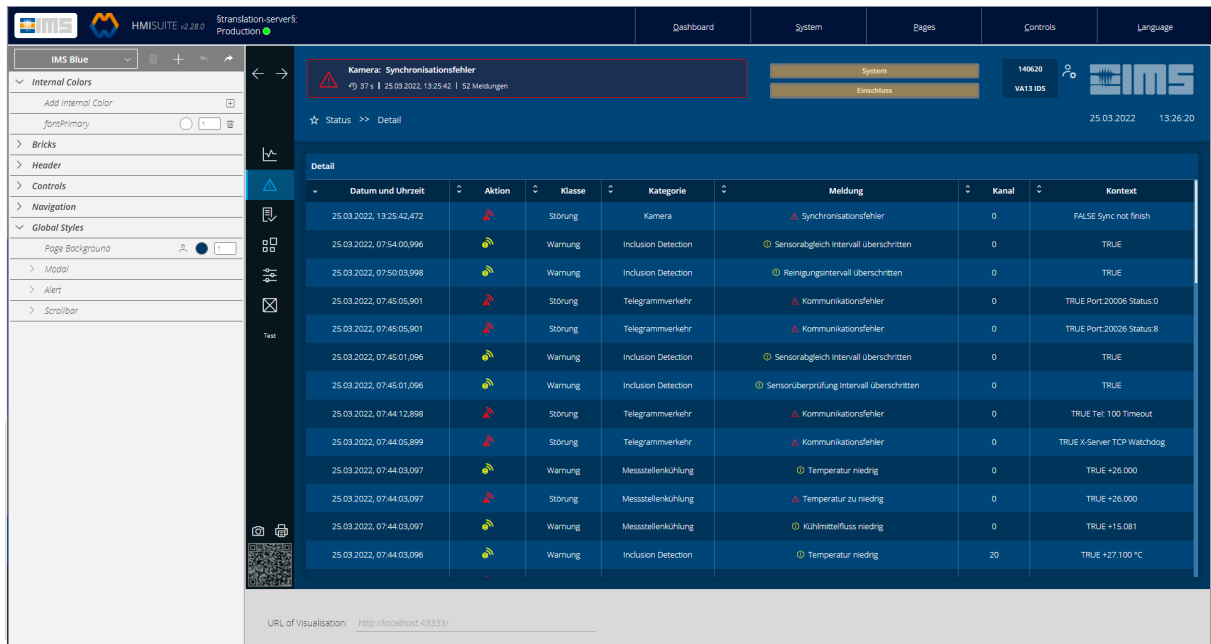


Abbildung 3.6: Erster Prototyp des Theme-Creators

3.3 Ausweitung zum Proof of Concept

In Abschnitt 3.2 wurde ein Auswahlbaum, sowie eine funktionierende Vorschau der Visualisierung als View umgesetzt. Die dort eingestellten Farb- und Alphawerte werden bereits in einem `themes`-Objekt abgelegt, welches vergleichbar zu Abbildung 3.7 ist. Ebenfalls befinden sich in den einzelnen Themes die eingestellten *Internal Colors*, sowie die drei Variablen namens `logo`, `print` und `expert`. Erstere bestimmt, ob das Logo in der Visualisierung im PNG- oder als SVG-Format dargestellt wird. Die Print-Themes sollen im Theme-Creator editierbar, aber in der Visualisierung nicht wählbar sein, da das Print-Theme nur bei der gleichnamigen Funktion genutzt werden soll. Der Expert-Schalter ist bereits aus Abschnitt 3.2 bekannt.

```
1 "imsDark": {
2   "title": "IMS Dark",
3   "logo": "white",
4   "print": false,
5   "expert": true,
6   "internalColors": {
7     "fontPrimary": {
8       "color": "#ffffff",
9       "alpha": "1"
10    },
11  }
12  "cssVariables": {
13    "background-color": {
14      "color": "#000000",
15      "alpha": "1"
16    },
17  }
18 }
19 // ... weitere Themes
```

Abbildung 3.7: Beispielhafter Aufbau des `themes`-Objekts

Dieses Objekt muss nun auf der Festplatte abgespeichert werden, damit die eingestellten Themes beim folgenden Programmstart wieder verfügbar sind. Zum Speichern wird es per POST-Request, vergleichbar zu Abschnitt 2.1.7, an den Server gesandt. Im Anschluss wird der bereits existierende *FileProvider* zum Speichern der JSON-Datei genutzt. Das Laden der so entstandenen Themes.json erfolgt per GET-Request im Frontend und nutzt ebenfalls den *FileProvider* zum Einlesen der JSON-Datei von der Festplatte.

Aus dem `themes`-Objekt wird ebenfalls das `style`-Element für den HTML-Header erstellt. Dafür wurde im shared-Verzeichnis, welches gleichzeitig von der Visualisierung und der Suite genutzt werden kann, eine statische Themes-Klasse angelegt. Diese enthält Attribute wie das aktive Theme, die boolesche Expert-Variable, das aktuelle `themes`-Objekt, sowie ein Objekt für die Basis-Themes. Neben einer Initialisierungsfunktion für die Themes befindet sich in dieser Klasse auch die Funktion `writeThemeStyletag`. In dieser Funktion wird das `themes`-Objekt anhand des aktiven Themes in einen String im HTML-Format umgewandelt. Im Anschluss wird dieser String an das `head`-Element des aufrufenden Microservices angehängen.

Um den Proof of Concept abzuschließen werden jeweils in der HMI-Suite und der Visualisierung mittels der JS-Funktion `setInterval` ein Aktualisierungsintervall für die Themes eingebaut. In diesem wird alle drei Sekunden per REST-GET-Request die aktuelle Themes.json von der Festplatte geladen. Im Anschluss wird das `themes`-Objekt der statischen

Themes-Klasse, sowie der jeweilige Style-Tag aktualisiert. In den beiden Projekten kann nun das CSS für den Hintergrund der View entsprechend Abbildung 2.3 angepasst werden. Somit ist es möglich über den Theme-Creator die Farbe des Hintergrunds anzupassen. Auch ein Wechsel zwischen den einzelnen Themes ist ohne sichtbare Verzögerung oder das Neuladen der gesamten Seite möglich. Somit konnte der Proof of Concept erfolgreich realisiert werden.

3.4 Iterative Implementierung

Nach der Erreichung des Proof of Concept können nach und nach alle existierenden Bricks auf das neue Theming umgebaut werden. Dies soll iterativ passieren, sodass sich der Funktionsumfang des Theme-Creators inkrementell erweitert. Jeder umgebaute Brick muss im Anschluss ausführlich getestet werden. Dabei werden die Attribute, die durch den Theme-Creator farblich angepasst werden können, nach bestem Wissen und Gewissen ausgewählt. Hier kann es sein, dass bei den Reviews in Abschnitt 3.5 manche Attribute weiter aufgeteilt oder zusammengelegt werden, um den Auswahlbaum so kurz wie möglich und so lang wie nötig zu halten. Im Folgenden werden die einzelnen Arbeitsschritte für den Umbau eines Bricks anhand des Button-Bricks erklärt.

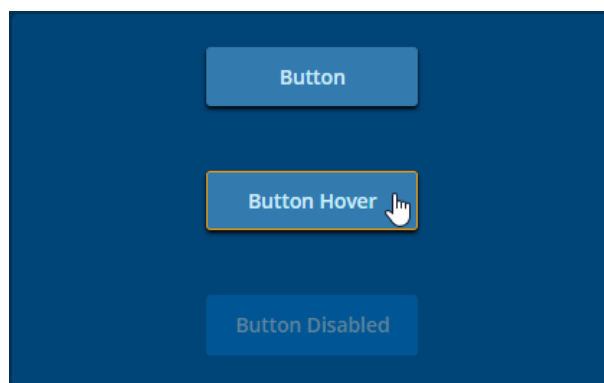


Abbildung 3.8: Der IMS-Button-Brick in der Visualisierung

1. Attribute identifizieren

Anhand der Abbildung 3.8 kann man alle benötigten Farben des Buttons identifizieren. Der Button kann entweder aktiv oder inaktiv sein und je nach Status werden eigene Farben benötigt. Die farbliche Gestaltung des inaktiven Buttons kann durch die Schrift-, Hintergrund- und Umrandungsfarbe beschrieben werden. Im aktiven

Zustand wird der Button um einen Schatten, sowie eine Umrandung im *Hover*-Zustand benötigt. Für die Basis-Themes ergibt sich damit die Tabelle 3.1.

Variablenname	imsBlue	imsDark	imsLight	imsPrint
button-background-color	337BAE	404040	337BAE	FFFFFF
button-border-color	337BAE	404040	337BAE	337BAE
button-border-hover-color	FFA104	FFA104	FFA104	FFA104
button-font-color	CDED9F	BEBEBE	FFFFFF	337BAE
button-shadow-color	000000	000000	000000	FFFFFF
button-background-disabled-color	005694	2E2E2E	B2CDE0	FFFFFF
button-border-disabled-color	005694	2E2E2E	B2CDE0	B2CDE0
button-font-disabled-color	61849E	646464	F0F0F0	B2CDE0

Tabelle 3.1: Auflistung der Farben des Buttons für alle Themes

Die Farben Hex-Codes der einzelnen Themes ergeben sich aus den bisher statisch codierten CSS-Regeln und einer Liste für ein Re-Design, das für den Red Dot Design Award 2021 erstellt wurde. Der Variablenname wird nach folgendem Muster aufgebaut: `[Brick]-...-[BrickN]-[Attribut]-[Status]-[Eigenschaft]`. Das Attribut bezieht sich auf den Teil des Bricks, der geändert werden soll. Häufige Attribute sind `background`, `border`, `font` oder `icon`. Als Status wird entweder nichts oder `active`, `disabled` oder `hover` angegeben. Im aktuellen Projekt wird für die Eigenschaft lediglich `color` verwendet. In der Zukunft sind durch den Einsatz neuer Controls, wie in Abschnitt 3.2 beschrieben, beispielsweise auch `radius` möglich. So können beispielsweise Buttons nicht nur in ihrer Farbe, sondern auch in ihrer Form verändert werden. Sollten Komponenten ineinander verschachtelt sein, wie zum Beispiel der Button im Manage-Station-Brick, dann werden die Bricks nacheinander im Variablennamen aufgelistet.

2. ThemeDict erweitern

Die erarbeiteten Eigenschaften können mit den selben Variablenamen im ThemeDict, wie in Abbildung 3.5 dargestellt, eingetragen werden. Dadurch aktualisiert sich automatisch der Theme-Tree. Die Farbwerte können nun entweder händisch in der Themes.json oder über den funktionierenden Theme-Creator eingepflegt werden. Da das Brick noch nicht aktualisiert wurde, ist noch keine Änderung sichtbar. Über die Entwicklungswerkzeuge kann nachvollzogen werden, ob das `HTML-style`-Element korrekt erstellt wurde.

3. Brick aktualisieren

Damit die eingestellten Variablen durch das Brick genutzt werden kann, muss das CSS angepasst werden. Der Einbau folgt dem technischen Konzept aus Abschnitt 3.1. Eine beispielhafte Implementierung kann der Abbildung 3.9 entnommen werden.

```

1  ims-button-brick {
2      // ... other styles that are not important for the theming
3      ims-button { // correctly use capsulation as shown in fig A.1
4          background: var(--button-background-color, $missingColor);
5          border: 1px solid var(--button-border-color, $missingColor);
6          box-shadow: 0 rem(2) rem(3) 0 var(--button-shadow-color, $missingColor);
7          color: var(--button-font-color, $missingColor);
8
9          &:hover {
10             border: rem(1) solid var(--button-border-hover-color, $missingColor);
11         }
12     }
13     &.disabled {
14         // ... other styles that are not important for the theming
15         ims-button {
16             background: var(--button-background-disabled-color, $missingColor);
17             color: var(--button-font-disabled-color, $missingColor);
18             border: rem(1) solid var(--button-border-disabled-color, $missingColor);
19             box-shadow: none;
20         }
21     }
22 }

```

Abbildung 3.9: Einbau des Themings im Button-Brick

Hier ist zu beachten, dass die Komponenten anhand von Abbildung A.1 korrekt ineinander verschachtelt werden. In diesem Fall überschreiben die Regeln des disabled-Zustandes die normalen Regeln des normalen Zustandes. Dies liegt daran, dass vergleichbar zu Abbildung 2.2 Klassen höher gewichtet werden als Elemente. Sollte dies nicht der Fall sein, muss bei dem disabled-Status auf **!important** zurückgegriffen werden. Der Umbau eines Bricks ist damit abgeschlossen und manche Änderungen bereits sichtbar.

4. Globale Regeln zurückbauen

Damit alle Änderungen sichtbar werden, müssen die bereits vorhandenen, globalen Regeln entfernt werden. Dafür werden die in Abschnitt 3.1 beschriebenen globalen Themes-SASS-Dateien schrittweise zurückgebaut. In den meisten Fällen müssen auch weitere Komponenten bearbeitet werden. In Abbildung A.1 ist sichtbar, dass die meisten Bricks eine Komponente aus *Project-Components* nutzen. Auch hier

können Farben statisch hinterlegt sein, die entfernt werden müssen. Falls Farben durch die *Framework-Components* vorgegeben sind, müssen die Regeln überschrieben werden, da das Framework auch in anderen Microservices verwendet wird. Um alle CSS-Regeln zu finden, die ein Brick beeinflussen werden die Entwicklungswerkzeuge, wie in Abbildung 2.12 dargestellt, genutzt. Sobald alle Altlasten entfernt wurden ist der Umbau auf das neue Theming für ein Brick fertig gestellt.

5. Testen und Dokumentieren

Zuletzt muss jedes einzelne Brick ausführlich getestet werden. Dabei sollte nicht nur der Theme-Creator zum Ein- bzw. Umstellen der Farben verwendet werden. Auch die Visualisierung und der Page-Composer müssen in verschiedenen Szenarios in allen Themes getestet werden. Im Anschluss wird die Arbeit dokumentiert und die tatsächlich verwendeten Farben aus Tabelle 3.1 werden in einer gemeinsamen Tabelle eingetragen.

Nach diesem Schema wurden sukzessiv alle Bricks umgebaut. Zum Abschluss muss das Gesamtsystem durch unabhängige Personen getestet und bei Bedarf verbessert werden.

3.5 Reviews und inkrementelle Verbesserungen

Der Theme-Creator wurde durch einen Designer in acht Zyklen ausführlich getestet und rezensiert. Dabei sollen nicht nur mögliche Bugs gefunden werden, sondern auch *Quality of Life (QoL)*-Aspekte beachtet. Die meisten gefundenen Punkte können sich in folgende Kategorien einteilen lassen:

- **Aktualisierung des ThemeDicts**

Beim Erstellen des PrintThemes und dem Prüfen der einzelnen Bricks ist aufgefallen, dass Flüchtigkeitsfehler wie doppelte Einträge, ein falscher Titel oder eine fehlerhafte Zuordnung des Parents passiert sind. Hier reicht es den jeweiligen Eintrag im ThemeDict zu aktualisieren.

- **Fehlende oder zu viele Farben**

Da das Theming in allen Bricks nach bestem Wissen und Gewissen umgesetzt wurde kann es sein, dass bei manchen Bricks weitere Attribute benötigt werden. So wurden anfangs Checkboxes mit nur einer Farbe umgesetzt, obwohl der Rahmen und das Check-Icon unabhängig voneinander einstellbar sein sollten. Ebenso kann es passieren dass Eigenschaften zusammengelegt werden können. Ein gutes Beispiel ist das

Message-Line-Brick aus Abbildung XX, in dem das Uhr-Icon, die vertikalen Striche, sowie die Schriftfarbe der zweiten Textzeile unabhängig voneinander eingestellt werden konnten.

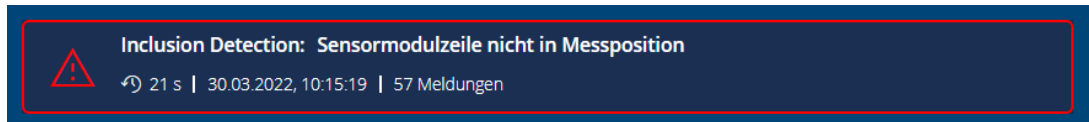


Abbildung 3.10: Das Message-Line-Brick

An dieser Stelle ist eine gemeinsame Schriftfarbe vollkommen ausreichend. Somit können die Einträge im Auswahlbaum und das CSS der Komponente reduziert werden. In diesem Fall muss nicht nur das ThemeDict, sondern auch das CSS der Komponente bearbeitet werden. Falls sich Variablennamen verändert haben oder neu entstanden sind, muss die jeweilige Farbe über den Theme-Creator für alle Themes eingestellt und getestet werden. Eine entfernte Farbe muss händisch aus der Themes.json entfernt werden.

- **Fehler durch den Rückbau globaler Regeln**

In manchen Fällen sind fehlerhafte Darstellungen dadurch aufgetreten, dass globale regeln zu großzügig entfernt wurden. Hier wurden Abhängigkeiten zu anderen Komponenten nicht ordnungsgemäß beachtet. In diesen Fällen mussten die CSS-Regeln wieder eingebaut werden. Im Anschluss wurden diese innerhalb der Bricks überschrieben. Dieses Problem ist vor allem bei den Tabellen aufgetreten.

- **QoL-Aspekte und Fehler unabhängig vom Theming**

Durch das ausführliche Testen aller Bricks wurden Fehler, Altlasten und QoL-Aspekte gefunden. Obwohl diese im Rahmen des Themings entdeckt wurden, sind sie nicht durch den Einbau des neuen Themings entstanden. Diese Punkte wurden gesondert dokumentiert und von anderen Entwickler*innen bearbeitet.

- **QoL-Aspekte abhängig vom Theming** Diese Punkte beziehen sich rein auf die Implementierung des Theme-Creators aus den Abschnitten 3.2 und 3.3. Alle Controls wurden überarbeitet, um intuitiver bedienbar zu sein. Zudem wurde der für einen besseren Überblick Auswahlbaum angepasst. Ein wichtiger Punkt ist das Verbessern der Leistung, sowie der Aktualisierung. Nach dem Proof of Concept musste nach dem Ändern einer Farbe mehrere Sekunden gewartet werden, bis die Änderung in der Vorschau sichtbar wurde. Somit ist das Erstellen neuer Themes zeitaufwändig

und die anwendende Person kann sich nicht sicher sein, ob die Änderung im Theme-Creator tatsächlich übernommen wurde. Zuerst wurde erfolglos das in Abschnitt 3.3 eingebaute Aktualisierungsintervall angepasst. Das eigentliche Problem besteht darin, dass der genutzte FileProvider nur im Intervall die Festplatte beschreibt, um diese bei vielen kleinen Änderungen nicht komplett auszulasten. Somit ist es sinnvoller das aktualisierte **Themes**-Objekt eventgesteuert und schnellstmöglich an die statische **Themes**-Klasse der Visualisierung und der HMI-Suite zu geben. Da sich der Theme-Creator ebenfalls in der HMI-Suite befindet der Umbau hier unproblematisch. Sobald eine Änderung im Theme-Creator vollzogen wird, wird die Funktion zum Austauschen des **style**-Elementes aufgerufen. Schwieriger ist es das Themes-Objekt und damit das **style**-Element in der Visualisierung zu überschreiben. Wie in Abschnitt 2.1.7 beschrieben, sieht das Sicherheitskonzept des Browser ein *Cross-Origin Resource Sharing (CORS)* nicht vor. Deshalb muss der Client der HMI-Suite das aktualisierte Themes-Objekt an den Server übergeben. Dieser reicht es an den Server der Visualisierung weiter, welcher das Themes-Objekt an alle möglichen Visu-Clients per Broadcast verteilt. Für diese eventgesteuerte Aktualisierung wird eine Kette aus Websocket-Verbindungen aufgebaut.

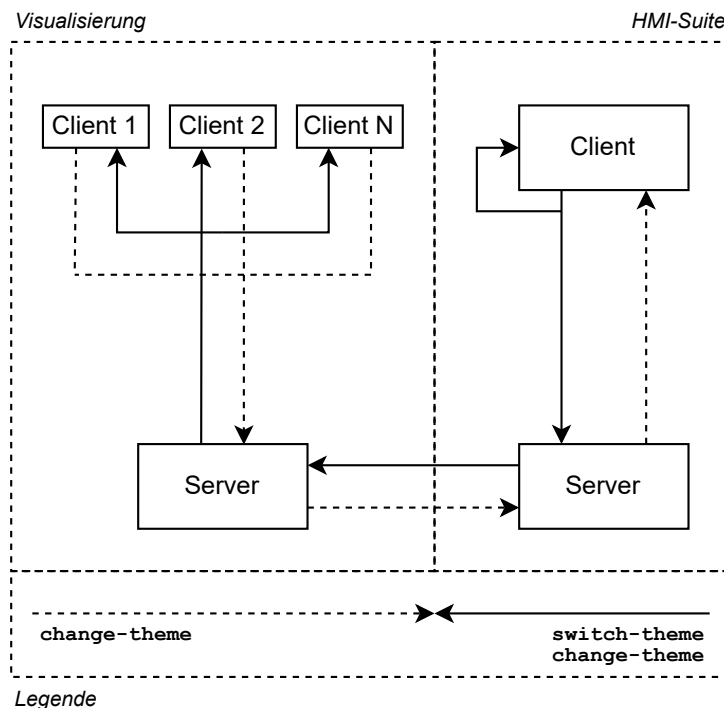


Abbildung 3.11: Die Kommunikation bei Farbänderungen bzw. beim Theme-Wechsel

Diese ist in Abbildung 3.11 zu sehen. Die durchgezogenen Pfeile stellen die Kommunikation vom Client der HMI-Suite zu den Client(s) der Visualisierung(en) dar. Die gestrichelte Linie zeigt die entgegengesetzte Richtung. Wird das Theme farblich geändert, müssen die **style**-Elemente beider Services getauscht werden. Deshalb heißt das ausgelöste Event **switch-theme** und kann nur in der HMI-Suite entstehen. Ein Theme-Wechsel kann hingegen in einem Client der HMI-Suite oder der Visualisierung passieren. Hier wird jeweils das Event **change-theme** ausgelöst und das eingestellte Theme wird in allen Clients synchronisiert. Dadurch wird verhindert, dass aus Versehen Änderungen in einem Theme vollzogen werden, die in der Vorschau nicht sichtbar sind.

Durch den Aufbau der Kommunikationskette per Websockets kann das Themes-Objekt schnellstmöglich übertragen werden. Änderungen im Theme-Creator werden sofort sichtbar.

Nach diesen acht Reviewzyklen und inkrementellen Verbesserungen wurde ein Stand des Themings erreicht, welcher zum weltweiten Ausrollen geeignet ist. Dadurch, dass es nun möglich ist sich eigene Themes anzulegen, kann die Visualisierung problemlos an eine Rot-Grün- oder Kontrastschwäche angepasst werden.

4 Fazit und Ausblick

Die erste Implementierung des Theme-Creators war ein voller Erfolg für das zukünftige Kernprodukt der IMS. Mit dessen Hilfe konnte der abteilungsinterne Designer schnell und einfach seine chromatische Vision der bisherigen drei Basis-Themes umsetzen. Zudem konnten die Basis-Themes durch ein viertes Print-Theme ergänzt werden, welches rein durch den Theme-Creator erstellt wurde. Mit dieser Theme-Vision hat die IMS 2021 den Red Dot Design Award gewonnen QUELLE RED DOT. Normalerweise mussten farbliche Änderungswünsche vom Designer an einen Entwickler kommuniziert werden. Dieser setzte diese dann um und ließ die Änderungen vom Designer prüfen. Mithilfe des Theme-Creators können die Änderungen direkt umgesetzt werden, was Ressourcen freigibt. Auch eine CI-spezifische Umsetzung für einzelne Firmen kann so schneller durch eine Einzelperson vollzogen werden, da nun der Code nicht verändert werden muss.

Ein weltweiter Roll-Out in die Produktivstätten der Stahlindustrie hat trotz eines umfassendem Refactorings jedes Bricks und der meisten Komponenten fast komplett reibungslos funktioniert. Es mussten nur einige visuelle Fehler behoben werden, welche den Arbeitssalltag nicht behindert haben.

Zuletzt wurde durch das Theming ein wichtiger Schritt unter dem Aspekt der Barrierefreiheit vollzogen. Mit diesem lassen sich kompromisslos kontrastreiche oder dichromatische Themes erzeugen. Somit ist sichergestellt, dass auch Personen mit Sehschwäche Warnungen auf der Visualisierung erfolgreich erkennen können.

4.1 Persönliches Fazit

Wie bereits in Abschnitt 1.5 erwähnt war dieses Projekt aus eigenen Erfahrungen ein persönliches Anliegen. Ich bin sehr froh, dass ich mit meiner Arbeit einen potentiellen Beitrag zur Barrierefreiheit in der Softwaretechnik geschaffen habe.

Dennoch hat mich dieses Projekt auch vor große Herausforderungen gestellt, da ein Großteil der verwendeten Komponenten im Projekt visuell angepasst wurde. Hier galt es das gesamte System ausführlich zu testen und potentiellen Fehlern entgegenzuwirken. Dies war vor allem bei solchen Komponenten ein Problem, die nicht häufig genutzt werden und dementsprechend auch nicht in den Standardprojekten zu finden sind. Tatsächlich

hat das ausführliche Testen durch andere Entwickler, dem Designer und den automatisierten Frontend-Tests dazu beigetragen, dass ein reibungsloser Übergang auf das neue Theming möglich war.

Zuletzt hat dieses Projekt mir dabei geholfen mich weiter in unsere Kern-Software einzuarbeiten und ein besseres Verständnis unseres hauseigenen View-Component-Framework zu erlangen. Ebenso habe ich mir durch das Erarbeiten von Abbildung 2.8 einen Überblick über das Zusammenspiel der einzelnen Komponenten verschafft und habe mein Wissen in CSS vertieft.

4.2 Ausblick

Der aktuelle Theme-Creator wurde so entwickelt, dass er in Zukunft erweiterbar ist. Durch die Implementierung generischer Controls wie dem Color- und dem Alphapicker können hier weitere mögliche CSS-Attribute durch den Theme-Creator editiert werden. Beispielsweise können Strichstärken für Umrandungen, Umrandungsradien, Schatten, Schriftgrößen und Schriftgewichtungen eingepflegt werden.

Zudem ist die Erweiterung des Theme-Creators, sowie des Page-Composers durch kontextspezifische Farbpaletten geplant. Aktuell kann im Page-Composer bei der Wahl einer Farbe für eine Brickinstanz nur eine Internal Color gewählt werden. Manche Bricks, wie zum Beispiel das Chart-Brick, sollen eigene Farbpaletten bekommen, mit dem die einzelnen Kurven besser dargestellt werden können. Diese sollen, wie auch die Themes, aus uneditierbaren Basis-Farben bestehen und durch eigene Farben erweiterbar sein. Die Paletten sollen im Theme-Creator eingestellt werden und dann im Page-Composer auswählbar sein.

Zuletzt soll der gesamte Theme-Creator ein konsistentes und intuitives Design bekommen, welches die Bedienung weiter vereinfacht.

Abbildungsverzeichnis

2.1	Beispiel CSS	6
2.2	Prioritätsordnung CSS [7]	7
2.3	Nutzung von Variablen in CSS	8
2.4	Beschreibung einer Navigation: SASS vs. CSS	9
2.5	klassisches JS vs. jQuery	10
2.6	Beispielhafte Components in der View Page-Composer	11
2.7	Beispielhafte Components in der View-Component-Sicht	12
2.8	Übersicht der Bricks und der von ihnen verwendeten Komponenten	13
2.9	Beispiel JSON [json]	15
2.10	Beispielhafter Ablauf von REST	16
2.11	Kommunikation via Websockets [15] WEBSOCKET PROTOCOL	17
2.12	Tab <i>Elemente</i> der DevTools	18
2.13	Barrierefreiheit in den DevTools	19
2.14	Speicherfunktion des Browsers	19
2.15	Vereinfachte Darstellung der Netzhaut bei verschiedenen Sehschwächen	23
2.16	Verschiedene Effekte der Farbenblindheit [20]	24
3.1	Umsetzung des alten Themings	27
3.2	HMTL-Aufbau im neuen Theming	29
3.3	Zuweisung der Theme-Variablen in CSS	30
3.4	Mock-Up des Theme-Creators	31
3.5	Beispielhafter Aufbau des ThemeDict-Objekts	32
3.6	Erster Prototyp des Theme-Creators	34
3.7	Beispielhafter Aufbau des themes-Objekts	35
3.8	Der IMS-Button-Brick in der Visualisierung	36
3.9	Einbau des Themings im Button-Brick	38
3.10	Das Message-Line-Brick	40
3.11	Die Kommunikation bei Farbänderungen bzw. beim Theme-Wechsel	41
A.1	Übersicht der Bricks und der von ihnen verwendeten Komponenten	VIII
A.2	Custom Colors mit Farbcode	IX
A.3	Custom Colors mit Color Picker	X

A.4	Komponenten-Baum mit zukünftigen Möglichkeiten	X
A.5	Bedienung der einzelnen Controls im Komponentenbaum	XI
A.6	Auswahl der Custom Color in einer Komponente	XI
A.7	Erstellng eines neuen Themes	XII
A.8	Auswahl eines Themes	XII

Tabellenverzeichnis

3.1	Auflistung der Farben des Buttons für alle Themes	37
-----	---	----

Literatur

- [1] *Red Dot Design Award: MEVIweb*. URL: <https://www.red-dot.org/project/meviweb-55296> (besucht am 12.04.2022).
- [2] *Change color contrast in Windows*. URL: <https://support.microsoft.com/en-us/windows/change-color-contrast-in-windows-fedc744c-90ac-69df-aed5-c8a90125e696> (besucht am 12.04.2022).
- [3] *Jugendzentrum GG - Pädagogisches Zocken - Spieleratgeber NRW*. URL: <https://www.spieleratgeber-nrw.de/Jugendzentrum-GG--Padagogisches-Zocken.5948.de.1.html> (besucht am 12.04.2022).
- [4] *Stack Overflow Developer Survey 2021*. en. URL: https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021 (besucht am 12.04.2022).
- [5] *Node.js. About*. en. URL: <https://nodejs.org/en/about/> (besucht am 12.04.2022).
- [6] *HTML Standard*. URL: <https://html.spec.whatwg.org/multipage/> (besucht am 12.04.2022).
- [7] Thomas Yip. *The definitive guide to CSS styling order*. en. URL: <https://vecta.io/blog/definitive-guide-to-css-styling-order> (besucht am 12.04.2022).
- [8] *Pseudo-elements - CSS: Cascading Style Sheets — MDN*. en-US. URL: <https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements> (besucht am 12.04.2022).
- [9] *CSS Functional Notation - CSS: Cascading Style Sheets — MDN*. en-US. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Functions (besucht am 12.04.2022).
- [10] *Introducing asynchronous JavaScript - Learn web development — MDN*. en-US. URL: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing> (besucht am 12.04.2022).
- [11] *JavaScript With Syntax For Types*. en. URL: <https://www.typescriptlang.org/> (besucht am 12.04.2022).
- [12] *W3Techs - extensive and reliable web technology surveys*. URL: <https://w3techs.com/> (besucht am 12.04.2022).

- [13] *Components Basics* — *Vue.js*. en-US. URL: <https://vuejs.org/guide/essentials/component-basics.html> (besucht am 12.04.2022).
- [14] *gulp.js*. en. URL: <https://gulpjs.com/> (besucht am 12.04.2022).
- [15] Alexey Melnikov und Ian Fette. *The WebSocket Protocol*. Request for Comments RFC 6455. Internet Engineering Task Force, Dez. 2011. URL: <https://datatracker.ietf.org/doc/rfc6455/> (besucht am 12.04.2022).
- [16] *DIN EN ISO 9241-220:2020-07, Ergonomie der Mensch-System-Interaktion.- Teil 220: Prozesse zur Ermöglichung, Durchführung und Bewertung menschenzentrierter Gestaltung für interaktive Systeme in Hersteller- und Betreiberorganisationen (ISO 9241-220:2019); Deutsche Fassung EN ISO 9241-220:2019*. Techn. Ber. Beuth Verlag GmbH. DOI: 10.31030/2851768. URL: <https://www.beuth.de/de/-/-/289443385> (besucht am 12.04.2022).
- [17] § 4 BGG - Einzelnorm. URL: https://www.gesetze-im-internet.de/bgg/_4.html (besucht am 12.04.2022).
- [18] *Web Content Accessibility Guidelines (WCAG) 2.1*. URL: <https://www.w3.org/TR/WCAG21/> (besucht am 12.04.2022).
- [19] *Achromatopsia* - *American Association for Pediatric Ophthalmology and Strabismus*. en. URL: <https://aapos.org/glossary/achromatopsia> (besucht am 12.04.2022).
- [20] *Rot-Grün-Sehschwäche*. de. Page Version ID: 221417255. März 2022. URL: <https://de.wikipedia.org/w/index.php?title=Rot-Gr%C3%BCn-Sehschw%C3%A4che&oldid=221417255> (besucht am 12.04.2022).
- [21] *The Rods and Cones of the Human Eye*. URL: <http://hyperphysics.phy-astr.gsu.edu/hbase/vision/rodcone.html> (besucht am 12.04.2022).
- [22] *Nachtblindheit – Ursachen, Anzeichen & Behandlung • Kuratorium Gutes Sehen e.V.* de. URL: <https://www.sehen.de/sehen/sehswaeche/nachtblindheit/> (besucht am 12.04.2022).
- [23] *Luminance Contrast*. URL: https://colorusage.arc.nasa.gov/luminance_cont.php (besucht am 12.04.2022).
- [24] *Demonstration: Color tutorial*. URL: <https://colortutorial.design/tutor.html> (besucht am 12.04.2022).
- [25] Jonathan Snook. *Scalable and Modular Architecture for CSS*. eng. 2nd ed. Ottawa: Selbstvlg, 2012. ISBN: 9780985632106.

- [26] *Organizing Multiple Theme Styles with Sass* — Zell Liew. en. URL: <https://zellwk.com/blog/organizing-multiple-theme-styles-with-sass/> (besucht am 12.04.2022).
- [27] *Using CSS custom properties (variables) - CSS: Cascading Style Sheets* — MDN. en-US. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties (besucht am 12.04.2022).

A Anhang

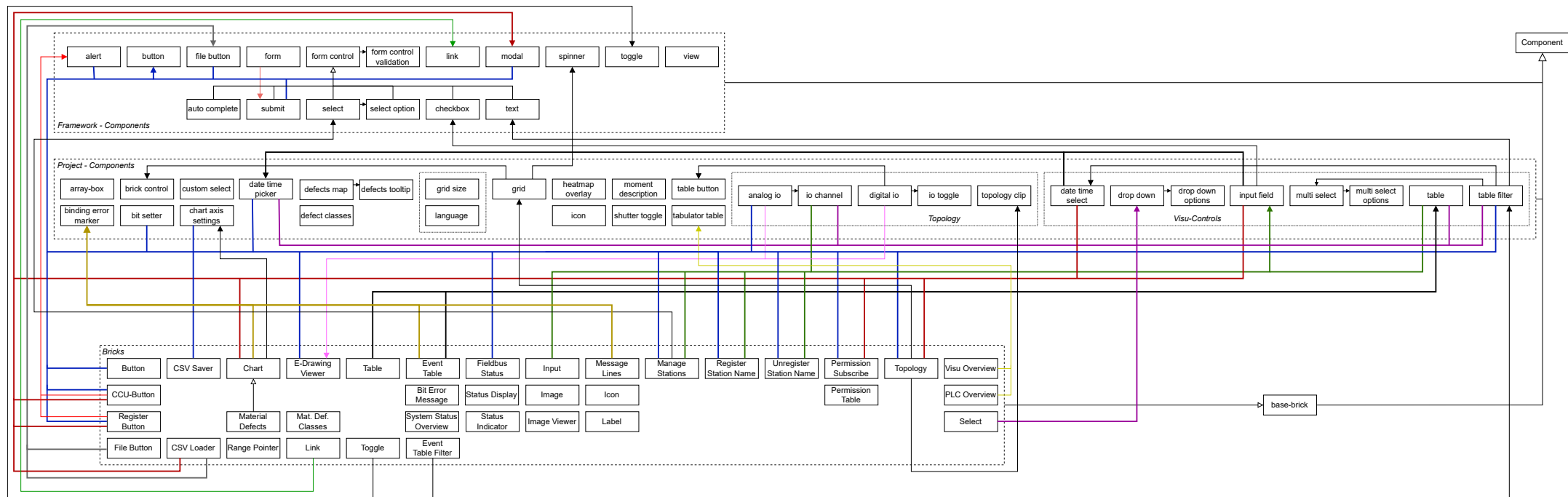


Abbildung A.1: Übersicht der Bricks und der von ihnen verwendeten Komponenten

A.1 Design-Visionen des Theme Creators

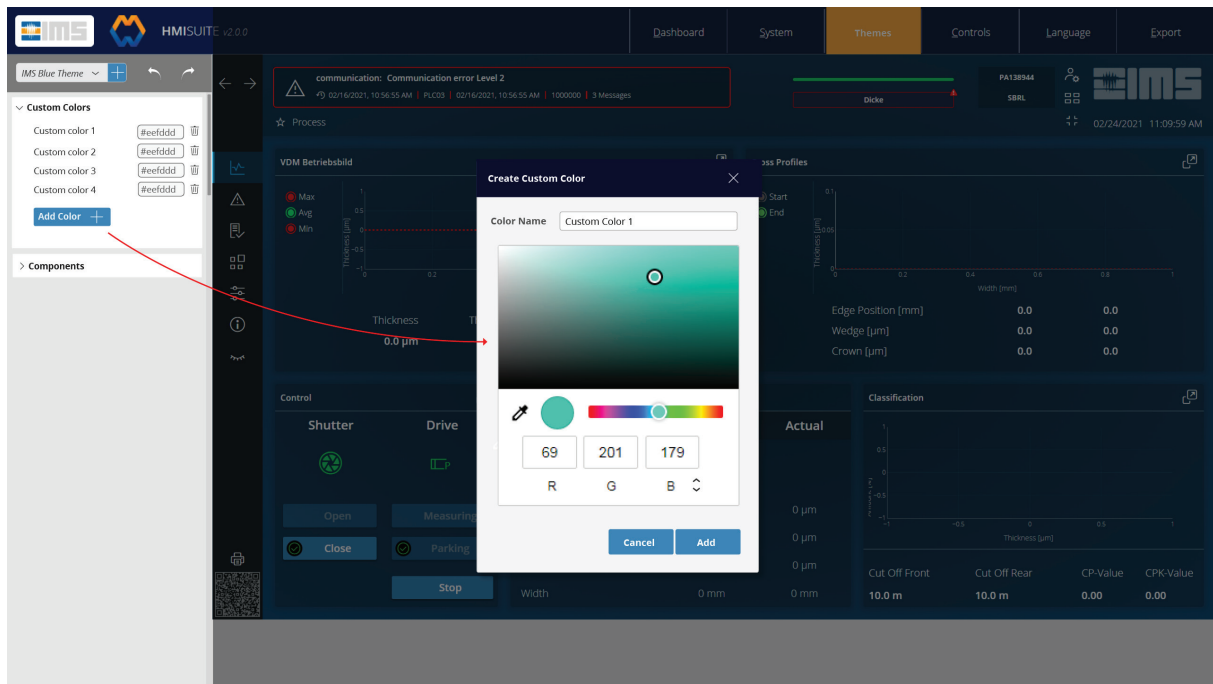


Abbildung A.2: Custom Colors mit Farbcode

A Anhang

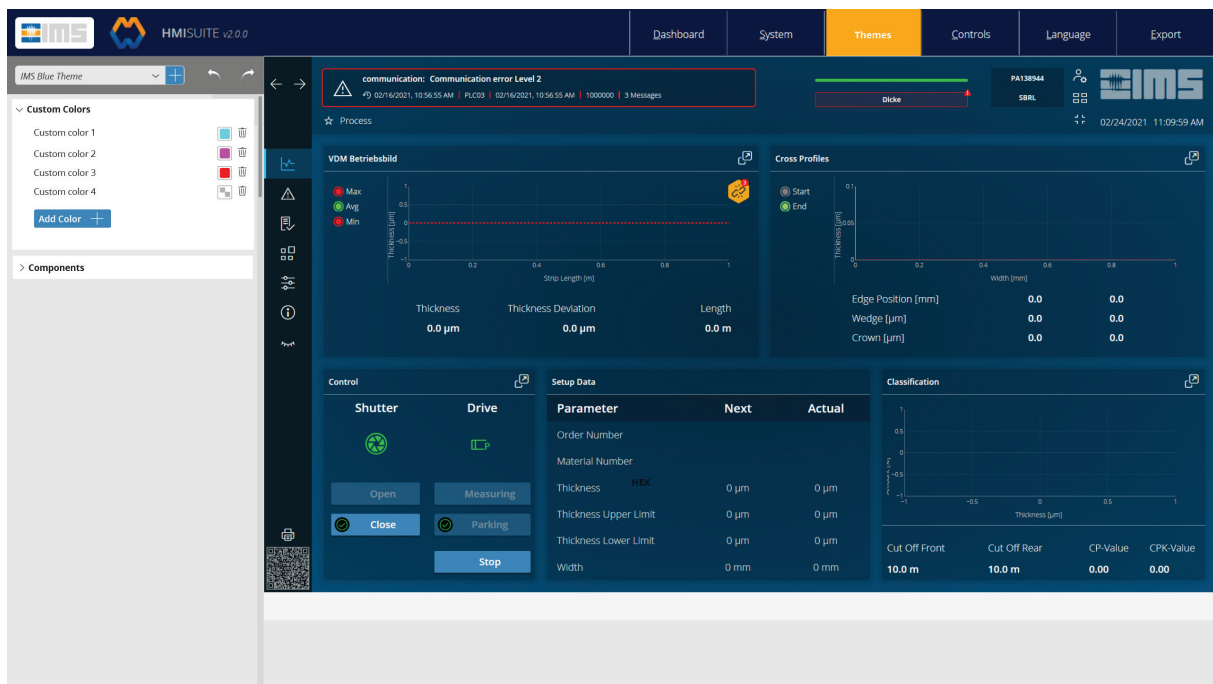


Abbildung A.3: Custom Colors mit Color Picker

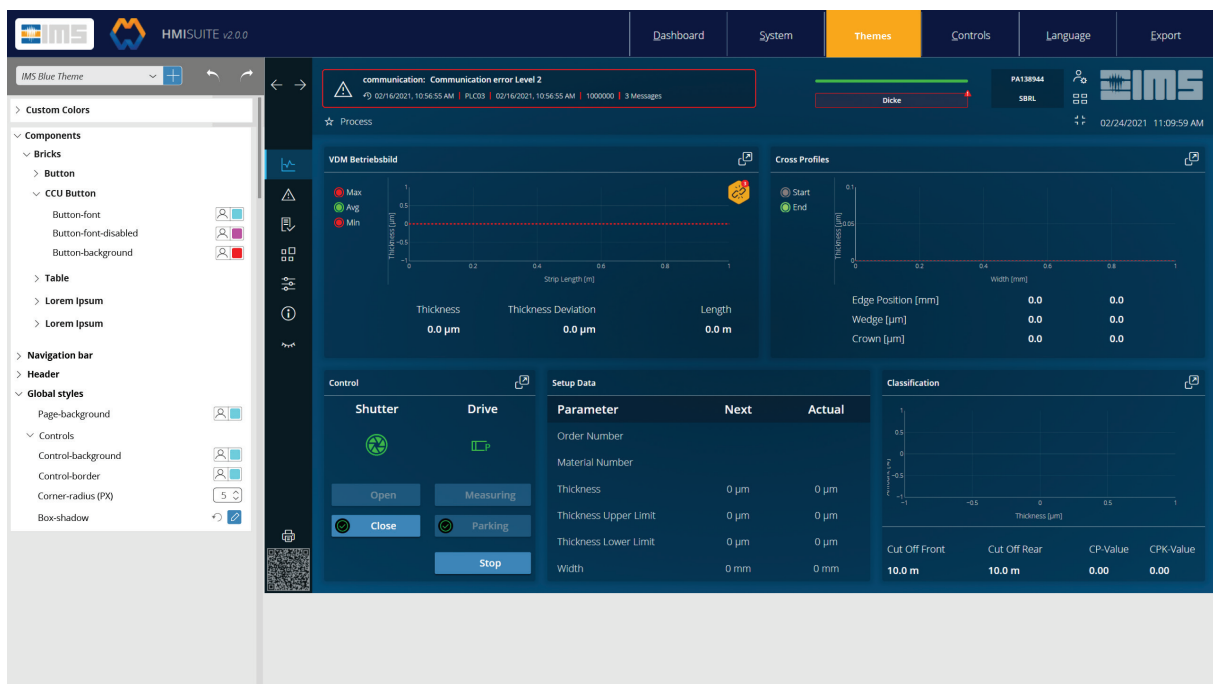


Abbildung A.4: Komponenten-Baum mit zukünftigen Möglichkeiten

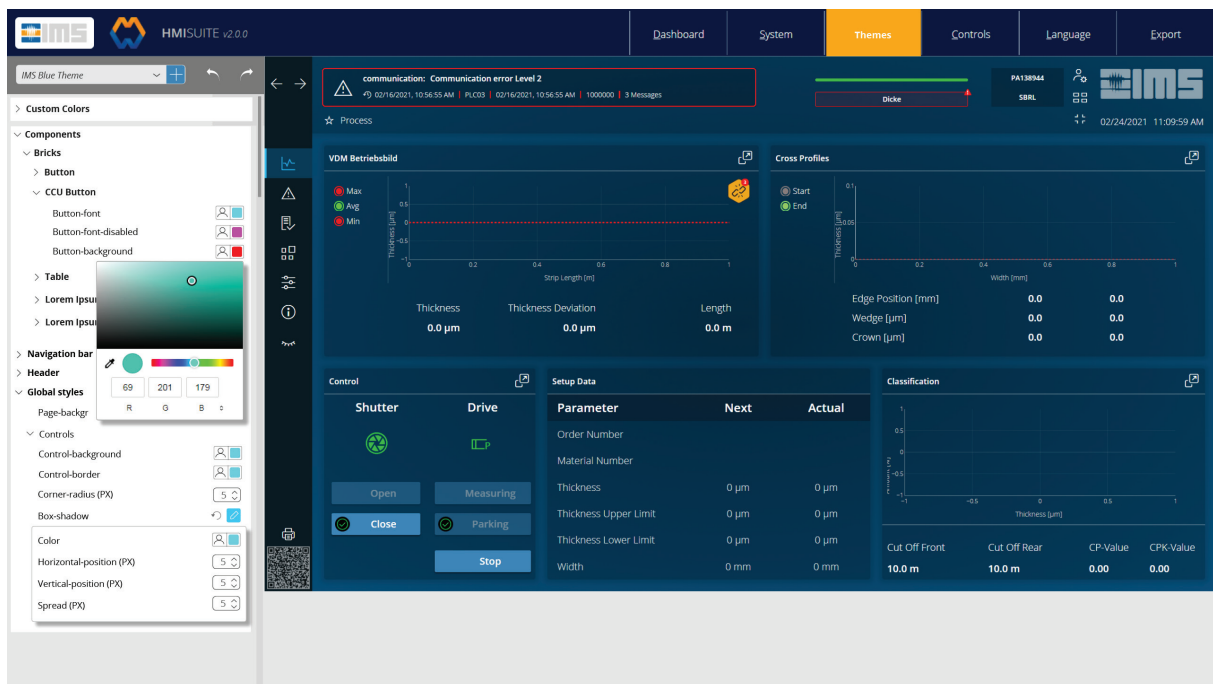


Abbildung A.5: Bedienung der einzelnen Controls im Komponentenbaum

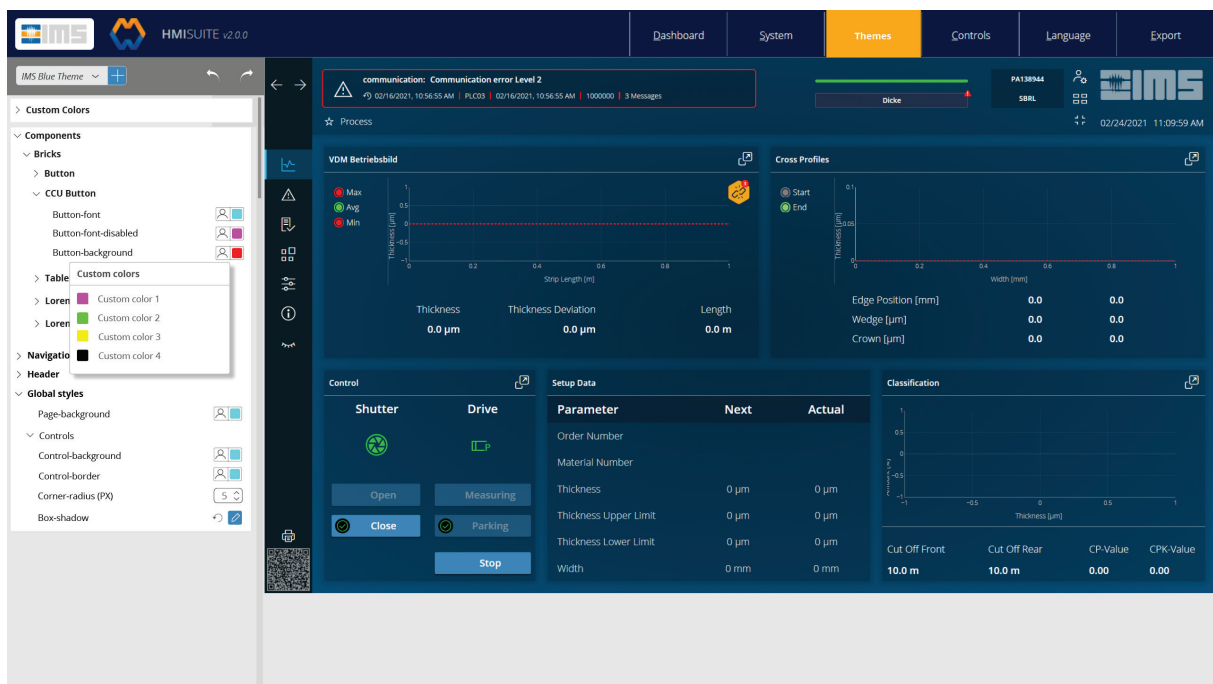


Abbildung A.6: Auswahl der Custom Color in einer Komponente

A Anhang

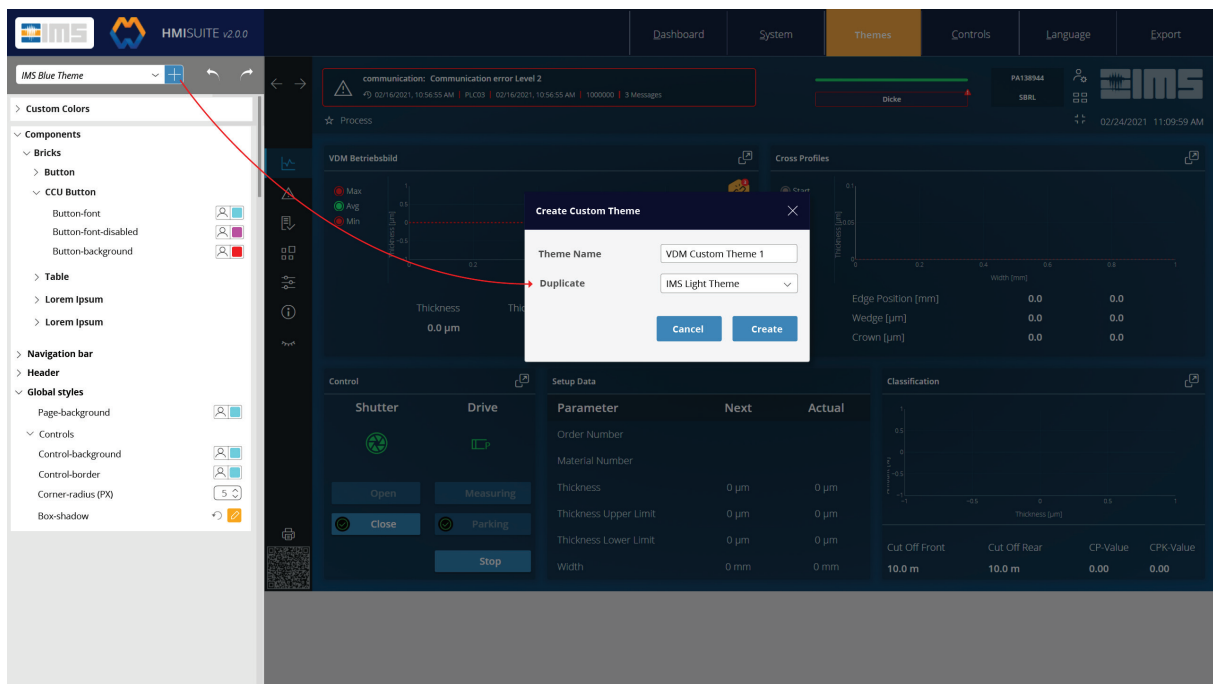


Abbildung A.7: Erstellung eines neuen Themes

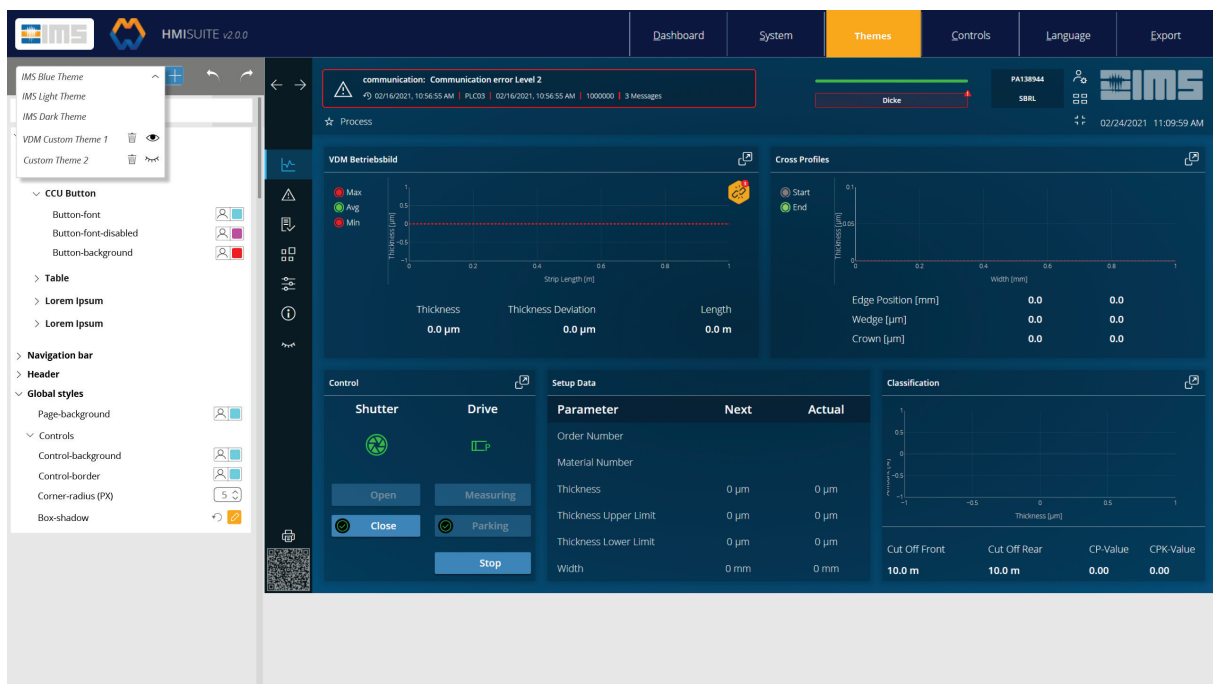


Abbildung A.8: Auswahl eines Themes