HAW HAMBURG

Bachelorarbeit

Timo Quednau

Frontend Architekturen und Patterns für microservices-basierte Backendsysteme

Betreuung durch: Prof. Dr. Lars Hamann / Prof. Dr. Stefan Sarstedt
Eingereicht am: 08.06.2023

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Timo Quednau

Thema der Arbeit

Frontend Architekturen und Patterns für microservices-basierte Backendsysteme

Keywords: Frontend, Architekturen, Microfrontend, Microservice, Qualitätsmodell

Kurzzusammenfassung

Diese Arbeit bewertet mittels Prototypen implementierte Microfrontend-Architekturen. Mithilfe eines Qualitätsmodells werden Unterschiede identifiziert und Architekturmethoden verglichen. Es werden ebenfalls Frameworks und Architekturmethoden für die entwickelten Prototypen verglichen und deren Vor- und Nachteile analysiert. Das erstellte Qualitätsmodell deckt die Hauptbereiche wie Entwicklungszeit, Kosten und Benutzerfreundlichkeit ab. Es werden sieben übergeordnete Qualitätsattribute betrachtet und anhand spezifischer Merkmale bewertet. Dafür wird ein Bewertungsschema entwickelt, um die Attribute zu vergleichen. Das Qualitätsmodell wird anhand der Prototypen angewendet und dokumentiert. Im speziellen werden die Unterschiede zwischen IFrames und Custom-Web-Components untersucht. Am Ende wird eine Bewertung über die Architekturen und Frameworks abgegeben.

Timo Quednau

Title of Thesis

Front-end architectures and patterns for microservices-based back-end systems.

Keywords: Frontend, Architecture, Microfrontend, Microservice, Quality model

Abstract

This thesis evaluates prototype-implemented microfrontend architectures, with using a quality model to identify differences and compare architectural methods. It also compares frameworks and architecture methods for the developed prototypes and analyses their advantages and disadvantages. The created quality model that covers major areas such as development time, cost, and usability. Seven overarching quality attributes are considered and evaluated based on specific characteristics. For this purpose an evaluation schema is developed to measure the attributes. The quality model is applied and documented using the prototypes. In particular the differences between IFrames and custom web components are examined. At the end an evaluation will be made about the architectures and frameworks.

Inhaltsverzeichnis

1 Einleitung **2**

1.1 Motivation . . . 2

1.2 Zielsetzung der Arbeit . . . 2

1.3 Gliederung . . . 3

2 Grundlagen **4**

2.1 Monolith . . . 4

2.2 Microservices . . . 5

2.3 Microfrontends . . . 6

2.3.1 IFrames . . . 7

2.3.2 Custom-Web-Components . . . 7

3 Erstellung des Qualitätsmodells **9**

3.1 Attribute . . . 9

3.2 Bewertungsschema . . . 11

3.3 Qualitätsmodell . . . 11

4 Prototypen **15**

4.1 Architektur . . . 15

4.2 Custom-Web-Component . . . 18

4.2.1 Einbindung in die Shop-Website . . . 18

4.2.2 Export in Angular . . . 21

4.2.3 Export in React . . . 24

4.3 IFrame . . . 28

4.3.1 Angular . . . 28

4.3.2 React . . . 29

4.4 Dokumentation . . . 29

4.4.1 Custom-Components . . . 30

4.4.2 IFrames . . . 31

4.5 Kommunikation . . . 32

4.5.1 IFrames . . . 32

4.5.2 Custom-Web-Components . . . 33

4.5.3 Andere Methoden . . . 33

5 Vergleich der Prototypen **34**
5.1 IFrame Angular/React . . . 34
5.2 Custom-Web-Component Angular/React . . . 35
5.3 IFrame vs Custom-Web-Component . . . 36
5.3.1 Aufsetzen, Deployment . . . 36
5.3.2 Kommunikation . . . 37
5.3.3 Styling . . . 37
5.3.4 Performance . . . 38

6 Anwendung der Qualitätsmodelle **39**
6.1 IFrame . . . 39
6.1.1 Funktionalität . . . 39
6.1.2 Effizienz . . . 39
6.1.3 Zuverlässigkeit . . . 40
6.1.4 Benutzerfreundlichkeit . . . 40
6.1.5 Wartungsfreundlichkeit . . . 41
6.1.6 Übertragbarkeit . . . 41
6.1.7 Marktfähigkeit . . . 41
6.2 Custom-Web-Component . . . 42
6.2.1 Funktionalität . . . 42
6.2.2 Effizienz . . . 42
6.2.3 Zuverlässigkeit . . . 43
6.2.4 Benutzerfreundlichkeit . . . 43
6.2.5 Wartungsfreundlichkeit . . . 44
6.2.6 Übertragbarkeit . . . 44
6.2.7 Marktfähigkeit . . . 44
6.3 Qualitätsmodelle . . . 45
6.3.1 IFrames . . . 45
6.3.2 Custom-Web-Component . . . 46

7 Auswertung **47**

8 Fazit **50**
8.1 Ausblick . . . 51

Literatur **52**

Listings 54

Abbildungsverzeichnis 54

Tabellenverzeichnis 55

Selbstständigkeitserklärung 56

Glossar

- **Angular** Angular ist ein JavaScript-Framework zur Entwicklung von Webanwendungen. Es bietet eine umfangreiche Sammlung von Werkzeugen und Funktionen, um robuste, skalierbare und performante Anwendungen zu erstellen.
- **React** React ist eine JavaScript-Bibliothek zur Entwicklung von Benutzeroberflächen. Es ermöglicht die Komponenten-basierte Entwicklung, bei der UI in wiederverwendbare Bausteine aufgeteilt wird.
- **http/https** HTTP (Hypertext Transfer Protocol) ist ein unverschlüsseltes Kommunikationsprotokoll, während HTTPS (Hypertext Transfer Protocol Secure) eine sichere Version davon ist, bei der die Daten verschlüsselt übertragen werden, um die Vertraulichkeit und Integrität der Informationen zu gewährleisten.
- **API-Gateway** Ein API-Gateway ist ein zentrales Verbindungspunkt, der den Zugriff auf verschiedene APIs bündelt und verwaltet, um die Kommunikation zwischen Frontends und den dahinter liegenden Microservices zu vereinfachen und zu kontrollieren.
- **Parent-Website** Eine Webanwendung, in welches ein Microfrontend intergriert wurde.
- **Pod-Anzahl** Ein Pod ist die kleinste Kubernetes-Einheit, in welche eine oder mehrere Instanzen einer Anwendungen laufen
- **Traffic** Data-Traffic bezieht sich auf den Fluss von digitalen Daten durch ein Netzwerk, die zwischen Geräten oder Systemen ausgetauscht werden.

1 Einleitung

1.1 Motivation

Nachdem die Backendmonolithen bereits in vielen Systemen durch Microservice-Architekturen ersetzt werden, wird dabei auch oft der bestehende Monolith durch neue Microservices ersetzt. Bei dem Wechsel der Architektur wird häufig das Frontend vernachlässigt, welches als Frontendmonolith bestehen bleibt und mit einer Mehrzahl an Microservices im Backend kommuniziert, somit werden die losgelösten Microservices dennoch in einem großen Frontendprojekt wieder verknüpft. Dieses Problem hat auch Billy J. Lando in seiner Arbeit behandelt [Lan].
Um die Frontend-Architektur von der monolithischen Architektur zu lösen, ist der Microfrontend-Ansatz aus der Idee der Microservices entstanden. Hierbei wird wie beim Backend das Frontend in Komponenten unterteilt, bestenfalls gibt es zu einem Microservice ein dazugehöriges Microfrontend. Die Idee von Microfrontends hat mehrere Umsetzungsmöglichkeiten und diese Arbeit setzt sich mit den zwei am häufigsten eingesetzten Architekturen für die Microfrontend-Entwicklung auseinander.
Um die Arbeit in den Entwicklungsteams agiler und flexibler zu gestalten, sind die Auswirkungen der Architekturwahl für die Teams zu beachten. Die Vorteile der Microfrontend-Architektur für die Entwicklungsgeschwindigkeit und einfache Teamkopplung sollen durch diese Arbeit hervorgehoben werden.

1.2 Zielsetzung der Arbeit

Die Arbeit soll durch die entwickelten Qualitätsmodelle die Möglichkeit bieten, bestehende Microfrontend-Architekturen zu bewerten, Lücken für Verbesserungen zu finden und Architekturmethoden zu vergleichen. Zudem soll die Arbeit aufzeigen, welche Frameworks und Architekturmethoden für die im Rahmen dieser Arbeit entwickelten Prototypen zum Einsatz kommen und welche Vor- und Nachteile sie besitzen.
Dabei sollen die Fragen der Einsatzgebiete beantwortet, auf die Unterschiede eingegangen und ebenfalls die Implementierung in den zwei am häufigsten eingesetzten Frameworks nach Sakovich [Sak] verglichen werden.

1.3 Gliederung

Die Arbeit besteht aus acht Kapiteln und beginnt mit der Einleitung, in der die Motivation zu dem Thema und die Zielsetzung erklärt wirds.
In den Grundlagen, dem zweiten Kapitel, werden die beiden Architekturen von Backendsystemen und deren Unterschiede erläutert, damit ein Grundverständnis für die Arbeit und die Ansätze aufgebaut wird. Im Anschluß werden Microfrontends grundlegend erklärt. Was auch die beiden Ansätze IFrame und Custom-Web-Component beinhaltet, um das Verständnis bei den Prototypen aufzubauen.
Das dritte Kapitel beginnt dann mit der Erstellung des Qualitätsmodells, dort werden die sieben übergeordneten Attribute, wie Funktionalität, Effizienz, Zuverlässigkeit usw., detailierter beschrieben und ihre Merkmale einem Bewertungsschema zugeordnet. Mit den Attributen, Merkmalen und dem Bewertungsschemata wird dann das Qualitätsmodell zusammengesetzt.
Die Prototypen werden im vierten Kapitel beschrieben zu Begin mit der Erklärung des Architekturansatz. Da die beiden Ansätze IFrame und Custom-Web-Component parallel zueinander in die Parentwebsite eingebaut wurden, wird die Architektur dort auch in abgewandelter Form, wie sie unter realen Bedingungen sein könnte, dargestellt. Danach wird auf die Prototypen genauer eingegangen, von der Einbindung über die Bereitstellung zur Dokumentation und Kommunikation.
Anschließend, im fünften Kapitel, werden die Prototypen miteinander verglichen und die Unterschiede herausgestellt, dabei werden sowohl die Ansätze IFrame und Custom-Web-Component als auch die Frameworks, in denen die Prototypen implementiert sind, analysiert.
Im sechsten Kapitel wird mit dem im dritten Kapitel erstellten Qualitätsmodell und der Beschreibung der Prototypen nun das Qualitätsmodell auf die Prototypen angewandt. Dafür werden die Attribute und deren Merkmale für beide Ansätze einzeln beschrieben und dann bewertet, woraus sich am Ende zwei ausgefüllte Qualitätsmodelle ergeben.
Die erstellten Qualitätsmodelle und Vergleiche der Prototypen werden im siebten Kapitel ausgewertet und im achten Kapitel wird das Fazit aus der Auswertung gezogen. Zudem wird noch ein Ausblick auf Weiterführungen der Arbeit gegeben.

2 Grundlagen

2.1 Monolith

Die monolithische Architektur in der Softwareentwicklung ist auf den herkömmlichen oder auch traditionellen Ansatz der Softwareanwendungen zurückzuführen, wobei eine Anwendung als einziges, eng gekoppeltes Programm erstellt wird. Diese fasst in der Codebasis alle Funktionen, Methoden und Komponenten zusammen.
Es ist eine einfache und unkomplizierte Art zu entwickeln und die Anwendung zu pflegen. Vergrößert sich jedoch der Umfang/Komplexität eben dieser, so wird es schwieriger einen Überblick zu behalten und diese zu verwalten.
Bei monolithischen Architekturen stellt eine der größten Herausforderungen das Ändern/-Modifizieren des Systems dar, ebenfalls ist die Skalierbarkeit einzelner Teile des Systems nicht möglich, nur das System als Ganzes kann mehr Rechenkapazität zugeteilt bekommen.
Die Fehlerbehebung ist in komplexen Monolithen ebenfalls eine Herausforderung, da auch nur kleine Änderungen in einer großen Codebasis ungewollte Veränderungen hervorrufen können.
Um diese Probleme zu lösen, wenden sich viele Unternehmen jetzt den Microservices-Architekturen zu, bei denen eine Anwendung in eine Reihe von lose gekoppelten Diensten aufgeteilt wird, die unabhängig voneinander entwickelt, bereitgestellt und gewartet werden können. Die Architektur und deren Vor- und Nachteile sind in den Arbeiten von Harris [Har] und Newman [New15] beschrieben.

Die wesentlichen Nachteile von Monolithen sind:

- Unflexibilität: Als eine einzige, eng verbundene Einheit kann eine monolithische Architektur schwer zu ändern oder zu modifizieren sein. Dies kann es schwierig machen, neue Funktionen hinzuzufügen oder Änderungen an bestehenden Funktionen vorzunehmen, ohne das gesamte System zu beeinträchtigen.

- Mangelnde Skalierbarkeit: Monolithische Architekturen sind in der Regel nicht für die Bewältigung großer Verkehrs- oder Datenmengen ausgelegt, was die Skalierung der Anwendung zur Erfüllung wachsender Anforderungen erschwert.

- Lange Bereitstellungszeiten: Da alle Komponenten eng miteinander gekoppelt sind, erfordert die Bereitstellung einer neuen Version der Anwendung häufig eine kom-

plette Neuentwicklung und Bereitstellung des gesamten Systems. Dies kann zu langen Bereitstellungszeiten und erhöhten Ausfallzeiten führen.

- Schwierige Tests: Das Testen einer monolithischen Architektur kann schwierig sein, da Änderungen in einem Teil des Systems unbeabsichtigte Auswirkungen auf andere Teile haben können.
- Schwierigere Diagnose und Behebung von Fehlern: Die Fehlersuche und -behebung in einer monolithischen Architektur kann sich schwierig gestalten, da das gesamte System eng miteinander gekoppelt ist, so dass es aufwändig ist, die Fehlerquelle zu isolieren.

2.2 Microservices

Microservices können gut mit den Worten von Martin Fowler [MF] beschrieben werden: "Kurz gesagt ist die Microservice-Architektur ein Ansatz zur Entwicklung einer einzigen Anwendung als eine Reihe kleiner Dienste, von denen jeder in einem eigenen Prozess läuft und mit leichtgewichtigen Mechanismen, häufig einer HTTP-Ressourcen-API, kommuniziert. Diese Dienste sind um Geschäftsfunktionen herum aufgebaut und können unabhängig voneinander von vollautomatischen Bereitstellungsmaschinen eingesetzt werden. [...]"
Die Microservice-Architektur ist mittlerweile weit verbreitet und wird in neueren Entwicklungen gerne verwendet. Die Vorteile der Microserviceses sind kurz zusammengefasst, dass sie sprachenunabhängig und mit wenig Kopplung untereinander entwickelt werden, zudem kann in kleineren Teams gearbeitet werden, wodurch die Organisation und Synchronisation innerhalb des Teams und der Entwickler vereinfacht wird.
Die kleinere Codebasis von Microservices im Gegensatz zu der großen eines Monolithens sorgt für eine vereinfachte Wartung und schnelle Einarbeitungszeit, zudem ist die Skalierbarkeit einzelner Komponenten genau wie das unabhängige Deployment in einer Microservice-Architektur gegeben.
Natürlich gibt es nicht nur positive Aspekte, einer der Nachteile einer Microservice-Architektur ist das Monitoring der Services. Um ein Monitoring der gesamten Applikation zu gewährleisten, muss jeder Service gemonitort und dies zentral zusammengeführt werden, welches die Komplexität des Systems erhöht. Ebenfalls beim Testing wird es in einer Mircoservice-Architektur komplexer, da die Services einzelnd getestet werden müs-

sen, zum Beispiel mittels End-To-End-Tests. Dies wurde von Newman [New15] ebenfalls erläutert.

2.3 Microfrontends

Bei der Microfrontend-Architektur handelt es sich um einen Ansatz zur Entwicklung einer einzelnen Anwendung in Form einer Reihe von Frontend-Komponenten. Es gibt verschiedene Wege die Mircrofrontend-Architektur umzusetzen, diese Arbeit wird sich mit den Methoden der IFrames, der Angular Components und den Custom Elements auseinandersetzen.
In der Software-Entwicklung änderen sich die Technologien und Anforderungen stetig, somit ist es sehr wichtig, dass Änderungen am Sourcecode schnell und verständlich vorgenommen werden können. Das dies auch im Frontend einen wichtigen Faktor darstellt und dafür eine Lösung gefragt ist, zeigen einige Firmen. Zum Beispiel hat Trivago [Rei] ihre Style-Dateien im Frontend refactort, da immer mehr Inkonsistenzen auftraten.
Die Idee von Microfrontends kommt aus der Entwicklung und Aufteilung von Backendsystemen in Microservices, die in Abschnitt 2.2 beschrieben wurden. Dort wird eine Anwendung auch in Form einer Reihe von Services entwickelt, welche unabhängig voneinander programmiert und genutzt werden können. Die Microfrontend-Architektur hat seinen Grundbaustein durch die Microservice-Architektur, welche sich bereits weit verbreitet und wird auch von großen Unternehmen wie Netflix [Kha] oder Zalando [Kol] verwendet.
Die Idee der Microfrontend-Architektur in Verbindung mit der Microservice-Architektur besteht darin, eine Anwendung komplett in separate Teams aufzuteilen, welche mit so wenig Kopplung wie möglich, unabhängig voneinander arbeiten können. Jedes dieser Teams hat ein spezielles Thema oder Baustein, für das ein Service und ein Frontend entwickelt wird, wodurch die Kommunikation innerhalb eines Teams effizienter und zielgerichteter gestaltet werden soll.

Microfrontend ist ein Begriff, der ein Softwarearchitekturmuster für die Entwicklung moderner Webanwendungen beschreibt. Die Idee hinter Microfrontends ist es, eine monolithische Frontend-Anwendung in kleinere, unabhängige Komponenten zu zerlegen, die unabhängig voneinander entwickelt, bereitgestellt und gewartet werden können. Jedes Microfrontend ist für eine bestimmte Funktion oder einen bestimmten Bereich der Anwendung zuständig und kommuniziert mit anderen Microfrontends und den Backend-Diensten über APIs.

2.3.1 IFrames

Ein IFrame oder auch Inline-Frame, ist ein HTML-Element, welches es ermöglicht, eine Website innerhalb einer anderen zu laden. Meist werden sie verwendet, um externe Inhalte wie Anzeigen, Werbung oder Videos zu zeigen. Die Verwendung und Einbindung ist schnell und unkompliziert durch das HTML-Tag *iframe* und die URL der Website, die angezeigt werden soll.
IFrames haben auch eine Abkapselung für das Styling, wodurch die CSS-Klassen der Parentwebsite nicht auf das IFrame übertragen werden. Zudem sind IFrames über CORS geschützt, dies verhindert, dass Parentwebsites mit einer anderen Top-Level-Domain dies nicht integrieren können. Dieses Verhalten kann über Konfigurationen verändert werden.

2.3.2 Custom-Web-Components

Eine benutzerdefinierte Webkomponente ist eine wiederverwendbare, in sich geschlossene Einheit einer Benutzeroberfläche, die zur Erstellung von Webanwendungen verwendet werden kann. Benutzerdefinierte Webkomponenten werden unter Verwendung von Webstandards wie HTML, CSS und JavaScript erstellt und sind so konzipiert, dass sie auf einer Vielzahl von Plattformen und Frameworks funktionieren.
Bei der Definition einer benutzerdefinierten Webkomponente sind mehrere Schritte erforderlich:

- Definition der Vorlage für die Komponente:
  Die Vorlage definiert die Struktur und das Aussehen der Komponente mit Hilfe von HTML und CSS. Die Vorlage kann Platzhalter für dynamische Daten und Event-Handler für die Benutzerinteraktion enthalten.

- Schreiben des JavaScript-Code für die Komponente:
  Der JavaScript-Code definiert das Verhalten der Komponente, einschließlich der Datenbindung, der Ereignisbehandlung und der Logik für die Aktualisierung des Zustands der Komponente.

- Registrierung der Komponente:
  Um die benutzerdefinierte Webkomponente in einer Anwendung zu verwenden, muss sie mit Hilfe der Custom-Elements-API beim Browser registriert werden. Dazu muss eine benutzerdefinierte Elementklasse erstellt und die Komponente mit

der Methode *customElements.define* beim Browser registriert werden. Um das Element definieren zu können, muss die dazugehörige JS-Datei, meist die *main.js*, vom Server heruntergeladen werden. Dazu wird das *<script/>* Tag verwendet.

- Verwendung der Komponente im HTML:
  Sobald die Komponente registriert ist, kann sie in dem erstellten HTML-Code wie jedes andere HTML-Element verwendet werden. Die Eigenschaften der Komponente können festgelegt und Daten mit Hilfe von HTML-Attributen und JavaScript an die Komponente gebunden werden.

Mit benutzerdefinierten Webkomponenten können wiederverwendbare, modulare Komponenten erstellt werden, die sich leicht zu komplexen Webanwendungen zusammenstellen lassen. Benutzerdefinierte Webkomponenten können auch dazu beitragen, die Wartbarkeit und Skalierbarkeit des Codes zu verbessern, da sie eine klare Trennung zwischen der Struktur, dem Aussehen und dem Verhalten der entwickelten Benutzeroberfläche ermöglichen.

3 Erstellung des Qualitätsmodells

Die Qualitätsmerkmale werden zusammengestellt, um ausgewählte Hauptbereiche abzubilden, welche die Entwicklungszeit, Kosten und Nutzerfreundlichkeit bewerten. Zudem kann mit dem entwickelten Modell die Qualität des Systems, der Infrastruktur oder Architektur bewertet werden. Um diese Merkmale/Attribute zusammen- und ein Bewertungsschema für die Prototypen aufzustellen, wird auf die Arbeit zu *Bewertung von Qualitätsproblemen bei der komponentenbasierten Softwareentwicklung* [Alm] zurückgegriffen. Bei der Bewertung der Softwarekomponenten wird auf das Buch *Measuring the usability of software components* [MB06] verwiesen, welche die ISO 9126 als Basis verwendet. Hierbei wird unterschieden zwischen der Qualität bei der Laufzeit und der beim gesamten Lebenszyklus.

3.1 Attribute

Die Attribute sind zusammengesetze Merkmale aus den Qualitätsmodellen *Quality Attributes for a Component Quality Model* [AA05], *Assessing Quality Issues in Component Based Software Development* [Alm] und *A Software Component Quality Model: A Preliminary Evaluation* [AA06].

Die übergeordneten Qualitätsattribute sind:

- Funktionalität (Functionality)
- Effizienz (Efficiency)
- Zuverlässigkeit (Reliability)
- Benutzerfreundlichkeit (Usability)
- Wartungsfreundlichkeit (Maintainability)
- Übertragbarkeit (Portability)
- Marktfähigkeit (Marketability)

Funktionalität
Das Attribut der Funktionalität wird anhand der folgenden Charakteristiken bewertet:

- Die Sicherheit, bei welcher auf die Datenverschlüsselung und die Zugangskontrolle geachtet wird
- Die Einhaltung der Vorschriften von Standardisierungen und Zertifizierungen
- Die Kompatibilität in Bezug auf Versionierung und Schnittstellen

Effizienz
Die Effizienz gibt an, wie resourcenintensiv das System ist und die Dauer der Antwortzeiten des Servers und der Webseiten. Zudem ist die Skalierbarkeit des Systems und der Komponenten ein wichtiger Bestandteil.

Zuverlässigkeit
Ob ein System zuverlässig ist, wird in den Absturzraten und der benötigten Zeit und Komplexität zur Wiederherstellung gemessen, zudem auch wie mit System-Errorhandling umgegangen wird.

Benutzerfreundlichkeit
Wie benutzerfreundlich ein System ist, kann durch die folgenden Punkte zusammengestellt werden:

- Benötigte Zeit, um Aktionen durchzuführen
- Konfigurationszeit
- Einarbeitungszeit
- Dokumentation
- Verwaltungsaufwand
- Betreibungsaufwand
- Benötigtes Vorwissen
- Flexibilität

Wartungsfreundlichkeit
Unter die Wartungfreundlichkeit fällt neben der Wartung an sich auch die Veränderbarkeit des Systems in Form von Weiterentwicklungen. Diese Eigenschaften messen sich an dem Aufwand und der Komplexität, um das System im Verhalten zu ändern und wie diese Weiterentwicklungen kontrolliert an die Anwender weitergegeben werden. Zudem wird die Wartbarkeit durch die Testbarkeit bestimmt und somit daran bemessen, ob automatisierte Tests möglich sind und der Aufwand dieser Test.

Übertragbarkeit
Bei der Übertragbarkeit wird überprüft, wie einfach Komponenten ersetzt, nach Änderungen deployed und wiederverwendbar sind.

Marktfähigkeit
Für die Marktfähigkeit sind die eigentliche Entwicklungszeit und die Kosten, sowie die Betriebskosten relevant.

3.2 Bewertungsschema

Für die in Abschnitt 3.1 zusammengetragenen Attribute des Qualitätsmodells muss noch ein Schema entworfen werden, mit dem die einzelenen Attribute bewertet werden können. Da die Attribute sich in ihren Eigenschaften unterscheiden, werden auch verschiedene Schemata benötigt.
Manche Attribute können mit einer Standard-Zahlskala von 0-10, andere hingegen können einfach mit einem Boolean wie *Vorhanden* oder *Fehlend* bewertet werden. Um die Latenz von Requests zu messen, wird einfach die Zeit angegeben, welche in verschiedene Level unterteilt werden kann, *gering*, *durchschnittlich* und *hoch*.

3.3 Qualitätsmodell

Mit den in Abschnitt 3.1 beschriebenen Attributen und in Abschnitt 3.2 beschriebenen Schemata können die eigentlichen Qualitätsmodelle mit den Merkmalen und dem dazugehörigen Bewertungsschema erstellt werden.

Funktionalität

Merkmal	Schema
Sicherheit	Boolean
Einhaltung der Vorschriften	Boolean
Kompatibilität	Boolean

Tabelle 1: Qualitätseigenschaften des Funktionalitäts-Attributs

Effizienz

Merkmal	Schema
Ressourcenkosten	Zahlskala
Antwortzeiten	Latenz
Skalierbarkeit	Zahlskala

Tabelle 2: Qualitätseigenschaften des Effizienz-Attributs

Zuverlässigkeit

Merkmal	Schema
Absturzraten/risiko	Zahlskala
Wiederherstellungskomplexität	Zahlskala
Errorhandling	Boolean

Tabelle 3: Qualitätseigenschaften des Zuverlässigkeit-Attributs

Benutzerfreundlichkeit

Merkmal	Schema
Aktionszeit	Zahlskala
Konfigurationszeit	Zahlskala
Einarbeitungszeit	Zahlskala
Dokumentation	Zahlskala
Verwaltungsaufwand	Zahlskala
Betriebsaufwand	Zahlskala
Benötigtes Vorwissen	Zahlskala
Flexibilität	Zahlskala

Tabelle 4: Qualitätseigenschaften des Benutzerfreundlichkeit-Attributs

Wartungsfreundlichkeit

Merkmal	Schema
Veränderbarkeit	Zahlskala
Testbarkeit	Zahlskala

Tabelle 5: Qualitätseigenschaften des Wartungsfreundlichkeit-Attributs

Übertragbarkeit

Merkmal	Schema
Deployment	Zahlskala
Wiederverwertbarkeit	Zahlskala

Tabelle 6: Qualitätseigenschaften des Übertragbarkeit-Attributs

Marktfähigkeit

Merkmal	Schema
Entwicklungszeit	Zahlskala
Entwicklungskosten	Zahlskala
Betriebskosten	Zahlskala

Tabelle 7: Qualitätseigenschaften des Marktfähigkeit-Attributs

4 Prototypen

Zum Testen des in dieser Arbeit konzeptionierten Qualitätsmodells aus Kapitel 3 wurden Microfrontend-Prototypen entwickelt. Es wurden 4 Microfrontend-Prototypen, eine Hauptwebsite, welche die Microfrontends verwendet, ein Node-JS Server, der die Custom-Component Dateien hostet und eine Spring-Rest Applikation entwickelt. Das Routing all dieser Komponenten erfolgt hinter einem Spring API-Gateway.

4.1 Architektur

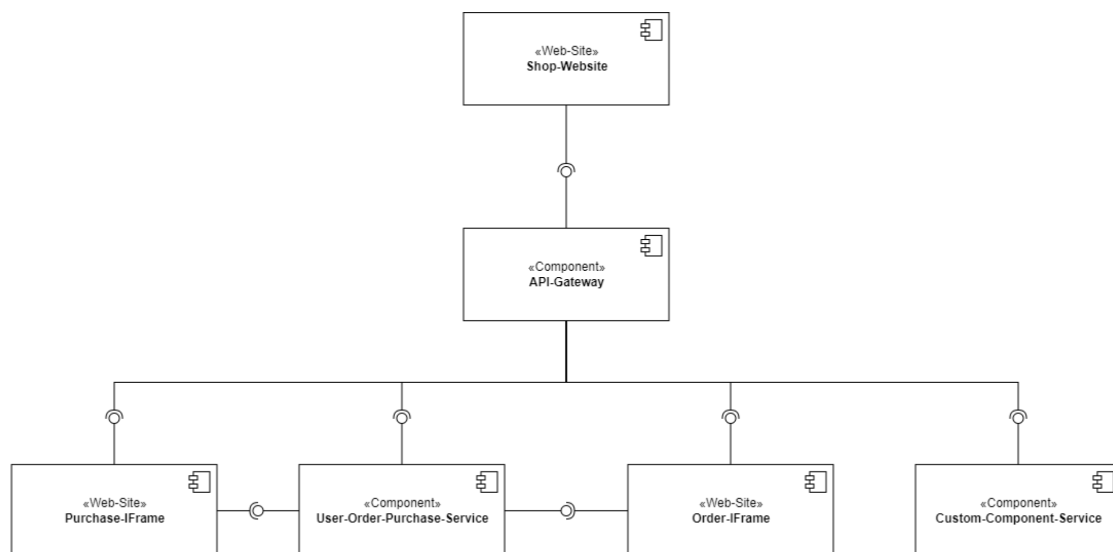

Abbildung 1: Komponenten-Diagramm der Prototypen-Architektur

In Abbildung 1 ist die Architektur der Prototypen dieser Arbeit dargestellt. Wie zu erkennen ist, liegen die Services und die IFrame-Komponenten hinter einem Gateway und nur die eigentliche Shop-Website ist nicht hinter dem Gateway geroutet. Da die Implementierungen der Prototypen dieser Arbeit so simpel wie möglich gehalten werden, aber trotzdem einen realen Verwendungszweck simulieren sollen, findet eine Kommunikation über das HTTPS-Protokoll statt. Dadurch wird im Gateway genau wie im Frontend ein Zertifikat der (Sub-)Domain hinterlegt. Somit werden keine weiteren Zertifikate in den Services und/oder IFrame-Websites benötigt werden.

Die Shop-Website greift über das Gateway auf alle Komponenten, die hinter diesem liegen, zu. Die IFrame-Websites werden als Microfrontends im Shop integriert, wie in Unterabschnitt 2.3.1 erläutert wurde. Um die Custom Web Components ebenfalls in die Shop-Website zu intergrieren, wird über das Gateway auf den *Custom-Component-Service* zugegriffen, auf dem die JS-Files mit dem SourceCode liegen. Diese Vorgehensweise wurde im Unterabschnitt 2.3.2 detailliert erklärt. Der *User-Order-Purchase-Service* wurde für die Implementierung der Prototypen aus 3 Microservices in einer Applikation zusammengefasst, auf die Aufteilung wird in der Beschreibung der nachfolgenden Abbildung 3 eingegangen.

Um die Abhängigkeiten der Prototypen voneinander besser darzustellen, wurden diese in Abbildung 2 ohne das Gateway, welches in der Architektur nur für die Weiterleitung und Zertifikatverwaltung zuständig ist, skizziert.

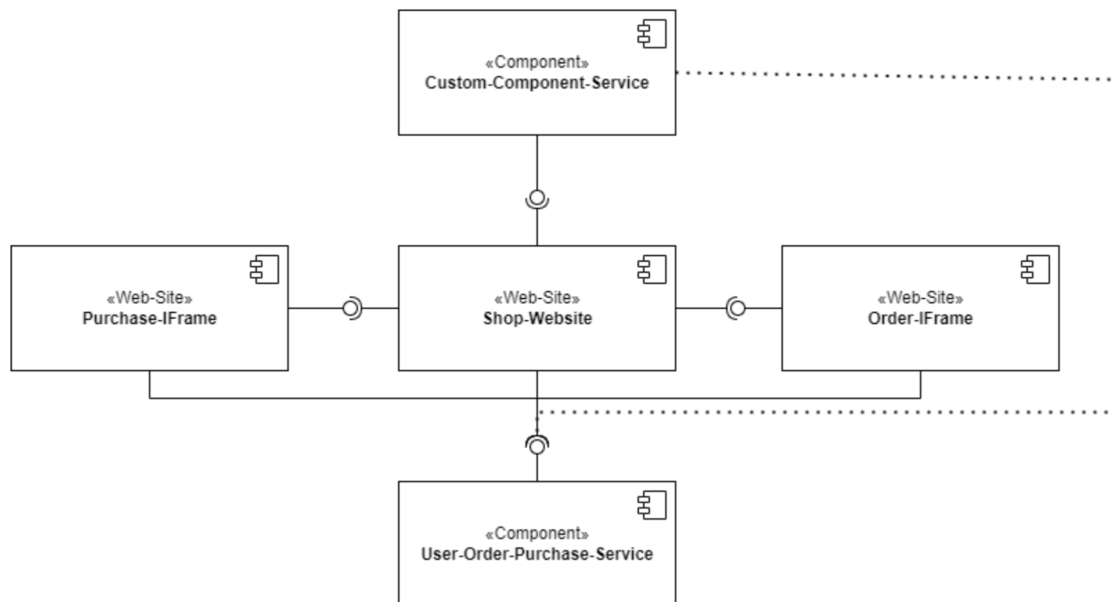

Abbildung 2: Komponenten-Diagramm der Prototypen-Architektur ohne Gateway

Zu erkennen ist, dass die Shop-Website zu allen Komponenten Abhängigkeiten aufweist, die IFrame-Websites nur zum User-Order-Purchase-Service. Dem liegt zugrunde, dass die Microfrontends, hier werden die IFrames als Beispiel verwendet, keine Abhängigkeit von der Hauptwebsite aufzeigen. Dies sorgt dafür, dass die Microfrontends auch in anderen Projekten verwendet werden können, ohne diese zu modifizieren.

Zudem zeigt die Abbildung, dass der Custom-Component-Service keine direkte Abhängigkeit zu dem User-Order-Purchase Service aufweist, dies liegt daran, dass der Service an sich keine Abhängigkeiten besitzt und nur die JS-Dateien für den Download/Request bereitstellt. In dem Fall der hier implementierten Prototypen besitzen eben diese bereitgestellten Microfrontends in den JS-Dateien Abhängigkeiten zum User-Order-Purchase Service, welche in der Abbildung als indirekte Abhängigkeit gekennzeichnet sind.

Der eigentliche Vorteil von Microfrontends besteht darin, die Architektur im Optimalfall mit Microservice pro Microfrontend aufzubauen. Diese Darstellung wird in Abbildung 3 gezeigt.

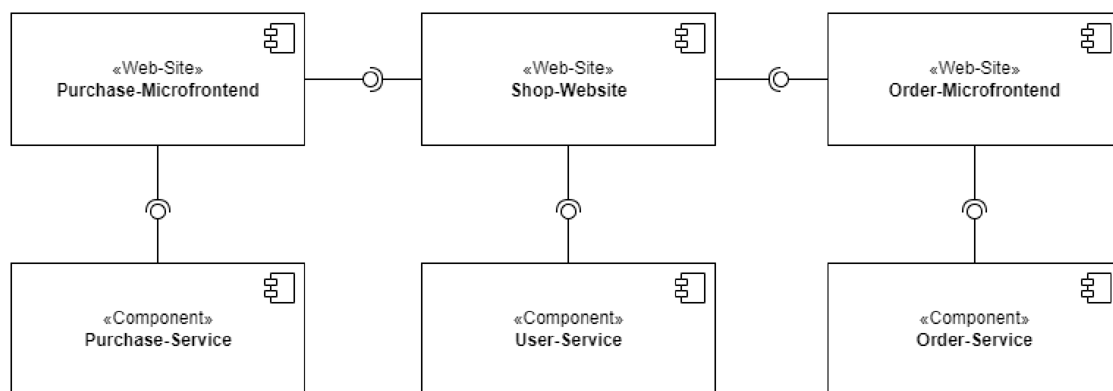

Abbildung 3: Komponenten-Diagramm der Prototypen-Architektur mit geteilten Microservices

Die IFrames und Custom-Web-Components wurden ausgetauscht gegen die eigentliche Web-Site, die diese repräsentieren. Der User-Order-Purchase Service aus den vorherigen Abbildungen wurde in User-Service, Order-Service und Purchase-Service aufgeteilt, wie es unter realen Bedingungen sein würde. Zudem hat die Shop-Website nur noch die Abhängigkeiten zu den Microfrontends, welche über verschiedene Verfahrensweisen integriert werden, und zum Userservice, welcher in den Prototypen zur Authentifikation verwendet wird.

4.2 Custom-Web-Component

Bei Custom-Web-Components ist die Möglichkeit gegeben, dass die Frontends versioniert werden können. Da der Zugriff über den Download erfolgt, wurde für die Prototypen ein Custom-Component-Service aufgesetzt. Dieser Service stellt statisch die JS-Dateien zur Verfügung, wodurch nach dem Deployment einer neuen Version auch die vorherigen Versionen noch zur Verfügung stehen, damit andere Teams nicht nach einem Deployment mögliche Produktionsprobleme bekommen. Dadurch ist die Änderung von Events und Attributen versionsabhängig und die Teams können ihre Microfrontends auf die Version anpassen, welche für sie notwendig ist.
Ein weiterer Vorteil eines Services, welcher die statischen Dateien zur Verfügung stellt, ist, dass nur eine Instanz auf einem Server laufen muss und nicht pro Microfrontend jeweils eine einzelne Instanz.

4.2.1 Einbindung in die Shop-Website

Die Einbindung von Custom-Web-Components erfolgt wie bereits in den Grundlagen von Unterabschnitt 2.3.2 erklärt. Im Folgenden wird auf die zusätzlichen Einbindungsmöglichkeiten der Prototypen in die Shop-Website eingegangen, welcher mittels Angular implementiert wurde und wie die zwei Microfrontend-Prototypen, von denen eins mittels Angular, das andere mittels React implementiert wurde, in die Parentwebsite eingebunden wurden.[Ver]

Damit Angular die Schemata von Custom-Web-Components akzeptiert, muss in der Datei *app.module.ts* noch das Schemata von *angular/core* importiert werden. Dies sieht wie folgt aus:

Listing 1: Schema in app.module

```
1  // Project: Shop
2  // File: app.module.ts
3  import {CUSTOM_ELEMENTS_SCHEMA, NgModule} from '@angular/core';
4  ...
5  schemas: [
6      CUSTOM_ELEMENTS_SCHEMA
7  ]
```

In Angular sind die Komponenten in **.component.html* und **.component.ts* aufgeteilt. Um festzulegen, wo auf der Parentwebsite das Microfrontend angezeigt wird, wird der Custom-HTML-Tag in die HTML-Datei der Komponente eingebunden, dies sieht beispielhaft wie im Folgenden aus:

Listing 2: Beispiel Custom-HTML-Tag

```
1  <!-- Project: Shop -->
2  <!-- File: cusom-component-angular.component.html -->
3  <html>
4  <body>
5      <div>
6          <div>
7              <h1>Überschrift/Anderer Content</h1>
8          </div>
9          <div>
10             <purchase-routing></purchase-routing>
11         </div>
12     </div>
13 </body>
14 </html>
```

Wie in dem Codeausschnitt zu erkennen, wird die Custom-Web-Component, hier das Purchase-Microfrontend, einfach über das Custom-HTML-Tag, hier *purchase-routing*, eingebunden. In dem Prototypen wurde noch eine Überschrift über das Custom-HTML-Tag eingefügt, um zu zeigen, wie die Positionierung des Microfrontends leicht ohne weiterführende Kenntnisse über HTML/CSS hinaus veränderbar ist.
In der **.component.html* Datei können auch initiale Attribute der Custom-Web-Component hinzugefügt werden, um dem Microfrontend Werte/Variablen zu übergeben. Dies sieht wie bei standart HTML-Attributen aus, nur dass diese keine Standart-Attribute sind. Als Beispiel:

Listing 3: Beispiel Custom-HTML-Attributes

```
1 <!-- Project: Shop -->
2 <!-- File: cusom-component-angular.component.html -->
3 <purchase-routing
4     id="purchase-component"
5     product-id="1"
6     product-name="Test"
7 >
8 </purchase-routing>
```

Diese Custom-HTML-Attribute werden von manchen IDEs unter Umständen als Fehler markiert, genau wie die Custom-HTML-Tags. In den meisten Fällen bieten die IDEs Funktionen an, diese sind Custom-HTML-Tag hinzuzufügen, um somit Fehlermeldungen zu negieren.

In den **.component.ts* Dateien wird auf das *DOCUMENT* zugegriffen, um die Werte der Attribute von dem Custom-HTML-Tag zu ändern. Über das Injecten von ng-Variablen wurden diese zwar initial gesetzt, aber Veränderungen wurden von der Custom-Web-Component nicht erkannt. Der folgende Codeabschnitt zeigt eine Beispielimplementierung:

Listing 4: Beispiel von update Custom-HTML-Attributes

```
1 // Project: Shop
2 // File: cusom-component-angular.component.ts
3 constructor(@Inject(DOCUMENT) private document: Document) {}
4 updateCustomAttribute(id: string, name: string, value: string) {
5     document.getElementById(id).setAttribute(name, value);
6 }
```

In Zeile 3 ist zu erkennen, wie das *document* initialisiert wird und die Funktion von Zeile 4 bis 6 ist eine Beispiel-Funktion, mit welcher die Attribute von außerhalb des Microfrontends upgedatet werden können.

Um die Änderungen auch innerhalb der Custom-Web-Component zu verarbeiten, müssen diese dort auch implementiert werden, hierbei ist aber das Framework oder die Lib, mit welcher das Microfrontend geschrieben ist, relevant. Im ersten Schritt wird die Implementierung einer Angular-Custom-Web-Component betrachtet.

4.2.2 Export in Angular

Um eine Custom-Web-Component mit Angular zu erstellen, wird die Angular App konfiguriert. Die *package.json* sollte für einfachere Entwicklung um die folgenden Scripts erweitert werden:

Listing 5: Zusätzliche und überarbeitete Scrips

```
1 "**comment**": "Projekt: cc-purchase",
2 "***comment***": "File: package.json",
3 "start": "npm run build & npm run serve_dist",
4 "build": "ng build --output-hashing none --single-bundle true
      --configuration production",
5 "serve_dist": "http-server ./dist/cc-purchase -p 4300 -c-1 --core -s"
```

Das start-Script wurde soweit überschrieben, dass es das build-Script und das serve-Script ausführt. Bei dem build-Script wird mittels des Flags *–single-bundle* bereits festgelegt, dass die Outputfiles zusammengefasst bzw. in einem Bundle ausgegeben werden sollen. Für die lokale Entwicklung hilft das serve-Script, da es einen http-Server öffnet, der die Dateien aus dem Build-Ordner für Requests bereitstellt.
Zudem sollten folgende Abhängigkeiten hinzugefügt werden, um die Erstellung von einer Custom-Web-Component zu ermöglichen:

Listing 6: Zusätzliche und überarbeitete Dependencies

```
1 "**comment**": "Projekt: cc-purchase",
2 "***comment***": "File: package.json",
3 "@ng-bootstrap/ng-bootstrap": "^13.0.0",
4 "@webcomponents/custom-elements": "^1.5.0",
5 "document-register-element": "^1.14.10",
6 "http-server": "^14.1.1",
```

Nun können die erstellten Komponenten als Custom-HTML-Tags definiert werden. Dies geschieht in der *app.module.ts* und sieht wie folgt aus:

Listing 7: Definieren von Custom-HTML-Tags

```
// Project: cc-purchase
// File: app.module.ts
...
constructor(private injector: Injector) {}
ngDoBootstrap(): void {
    customElements.define(
      'purchase-routing',
      createCustomElement(
        PurchaseComponent,
        {injector: this.injector}
      )
    );
    customElements.define(
      'purchase-overview',
      createCustomElement(
        PurchaseOverviewComponent,
        {injector: this.injector}
      )
    );
}
...
```

Die vierte Zeile zeigt den Constructor, in dem der Injector initialisiert wird.
In der vierten Zeile beginnt die ngDoBootstrap-Funktion, in der die Custom-HTML-Tags definiert werden. Hier wurden zwei definiert, einmal das Routing-Element, welches anhand der übergebenen Attribute auf die jeweilige Seite weiterleitet. Die Zweite ist das Tag *purchase-overview*, bei der nur die Übersichtsseite gezeigt wird. Somit wird ein dynamisches Verwenden des Microfrontents gewährleistet, und beide Möglichkeiten können ohne weitere Modifikation verwendet werden und eine Änderung in der Übersichtsseite wirkt sich auf beide Varianten aus.

Im Folgenden wird einmal die Struktur des Projektes aus Übersichtszwecken dargestellt, damit ein Verständnis für den Unterschied der übergeordneten Routing-Komponente und der direkten Overview-Komponente entsteht:

```
app
├── app.module.ts
└── purchase
    ├── purchase-routing.module.ts
    ├── purchase.component.css
    ├── purchase.component.html
    ├── purchase.component.ts
    ├── purchase-order
    │   ├── purchase-order.component.css
    │   ├── purchase-order.component.html
    │   └── purchase-order.component.ts
    └── purchase-overview
        ├── purchase-overview.component.css
        ├── purchase-overview.component.html
        └── purchase-overview.component.ts
```

Wenn in Custom-Components mehr als eine Seite angezeigt werden soll, wird das Routing benötigt. Da aber innerhalb des Microfrontends sich die Seite ändern soll, kann die URL nicht einfach manipuliert werden, da davon die Haupt-Website betroffen wäre. Da ebenfalls ein flüssiger Übergang gewünscht ist, in dem nur die Routingelemente verändert werden sollten, wird ein RouterModule verwendet. Da für das Routing innerhalb einer anderen Website ein sogenanntes *Outlet* benötigt wird, wird in dem Prototypen testweise das RouterTestingModule verwendet. Hierfür wurde eine neue *...-routing.module.ts* Datei erstellt, um das Routing mit dem jeweiligen Path zu definieren.

Listing 8: Definieren der Routen

```
1  // Project: cc-purchase
2  // File: purchase-routing.module.ts
3  ...
4  const routes = [
5    { path: 'order', component: PurchaseOrderComponent, outlet: '
        purchaseOutlet' },
6    { path: 'overview', component: PurchaseOverviewComponent, outlet: '
        purchaseOutlet' }
7  ];
8  @NgModule({
9    imports: [
10     RouterTestingModule.withRoutes(routes),
11     HttpClientModule
12   ],
13   exports: [RouterTestingModule],
14   providers: []
15 })
16 export class PurchaseRoutingModule {}
17 ...
```

Zu erkennen ist, dass zwei Routen definiert wurden, zum einen die Route für eine Bestellung und die Route zur Bestellübersicht. Beide Routen sind in dem Outlet *purchaseOutlet* festgelegt.
Da der Prototyp HTTPS-Request ausführen soll, wurde in Zeile 8 auch nochmals das *HttpClientModule* importiert. Durch dieses Modul ist nun ein Routing innerhalb des Microfrontends möglich.

4.2.3 Export in React

Um in React eine Custom-Web-Component zu erstellen, wird erstmal ein standart React Projekt initialisiert. Die *package.json* benötigt zusätzlich folgende Abhängigkeiten, damit die Custom-Web-Component als Standalone-Component gebaut werden kann:

Listing 9: Package.json für Custom-Web-Components in React

```
1  "**comment**": "Projekt: cc-products",
2  "***comment***": "File: package.json",
3  ...
4      "esbuild": "^0.15.11",
5      "react-dom": "^18.2.0"
6  },
7  "scripts": {
8      "build-micro": "esbuild src/components/Product-List.tsx
           --outfile=dist/main.js --bundle --minify,
9  ...
```

Die *esbuild* Lib wird für das Bauen der *main.js* benötigt, wie in Unterabschnitt 2.3.2 erklärt wurde. Das Script zum Bauen wurde in Zeile 8 eingefügt und legt fest, welche Komponente verwendet und wo die *main.js* hinterlegt werden soll.
Zum Testen der Komponente, kann ein NodeJs Express Server in dem Projekt angelegt werden, dadurch wird der Ordner *./dist/* statisch über einen Port erreichbar. Sollten CORS-Probleme beim Testen auftreten, kann der folgende Codeausschnitt hinzugefügt werden, damit Webseiten ohne Probleme die JS-Dateien laden können.

Listing 10: Beispiel einer CORS-Erweiterung bei einem NodeJs-Server

```
1  ...
2  // Projekt: cc-products
3  // File: server.js
4  const cors = require('cors');
5  app.use(cors({
6    origin: '*'
7  }));
```

Die eigentliche Custom-Web-Component wurde in der *Product-List.txs* implementiert. Hierfür sollte vorher die Schnittstelle mit der Custom-Web-Component festgelegt werden, außerdem welche Events nach Außen oder Innen ermöglicht werden müssen und welche Attribute/Variablen die Custom-Web-Component benötigt.
Für den Prototypen in React wurden folgende Schnittstellen definiert:

```
Team: products
Tag: "products-list"
Versions: [0.0.1, 0.0.2]
Latest: 0.0.2
ImportUrls: [
             "https://components.your.domain/products/0.0.2-main.js",
             "https://components.your.domain/products/main.js"
            ]
Attributs: [accesstoken: String, userid: String, apiurl: String]
DispatchEvents: [
    "products:purchase":
        {
            detail: {
                id: String,
                name: String
            }
        }
    ]
```

Abbildung 4: Schnittstellendefinition von der Custom-Web-Component Products

Für die Custom-Web-Component Products wurde somit definiert, dass die Attribute *accesstoken, userid, apiurl* im Format String übergeben werden müssen und die Komponente ein Event mit der id *purchase* und dem beschriebenen Informationen im Body an die Parent-Website senden kann.

Wenn die Schnittstelle definiert ist, kann die Komponente implementiert werden. Dafür wird die Standard-React-Komponente mit dem *HTMLElement* erweitert (extendes). Nun werden die Attribute von der Schnittstelle als Variablen mit dem Typ *string | null* angelegt. Die Option, dass die Variablen auch *null* sein können, ist notwendig, falls die Komponente aufgerufen wird und nicht alle Attribute einen Wert/Value besitzen. Die Initialwerte werden in dem Konstruktor über die Methode *this.getAttribute(name);* gesetzt, hierbei ist der Name beispielhaft *accesstoken*. Sollte dieses Attribut keinen Wert besitzen, gibt die Funktion *null* zurück.
Zudem müssen die Attribute, welche auf Veränderungen überwacht werden sollen, definiert werden. Es müssen dafür die folgenden Methoden implementiert werden:

Listing 11: Überschreiben der Methoden zur Überwachung der Attribute

```
...
// Projekt: cc-products
// File: Product-List.js
static get observedAttributes() {
    return ["accesstoken", "userid"];
}

attributeChangedCallback(name: any, oldValue: any, newValue: string) {
    switch (name) {
        case "accesstoken":
            this.accesstoken = newValue;
            this.getProducts();
            return;
            ...
        default:
            return;
    }
}

connectedCallback() {
    this.render();
}
...
```

Die Funktion in Zeile 3 legt die Attribute fest, welche überwacht werden sollen. Diese Liste muss nicht alle möglichen Attribute enthalten, nur die, bei welchen eine Veränderung relevant ist. Die API-URL zum Beispiel verändert ihren Wert nicht und muss daher nicht überwacht werden.
Die Funktion in Zeile 8 *attributeChangedCallback* wird ausgeführt, sobald sich ein Wert eines Attributes verändert. In der Implementierung des Prototyps wurde nach der Änderung des Accesstokens die Funktion *getProducts* erneut aufgerufen, um sicherzustellen, dass die Daten valide sind.
Die Funktion *connectedCallback* wird aufgerufen, sobald die Parent-Website die Custom-Web-Component vollständig geladen hat und dazu wird in Zeile 21 die *render* Funktion aufgerufen.

Um die Komponente am Ende auch als Custom-HTML-Tag verwenden zu können, muss das Tag festgelegt und die Elemente definiert werden. Dies kann in der *Product-List.tsx* implementiert werden, falls das Microfrontend nur eine kleine Komponente wie hier im Beispiel wiederspiegelt oder an gesammelter Stelle, falls mehrere Komponenten bereitgestellt werden sollen. Die Definition sieht wie folgt aus:

Listing 12: Definieren des Custom-HTML-Tags

```
1 ...
2 // Projekt: cc-products
3 // File: Product-List.js
4 ...
5 const tagName = "products-list";
6 if (!window.customElements.get(tagName)) {
7   window.customElements.define(tagName, ProductListComponent);
8 }
```

Damit Fehler bei der Definition verhindert werden können, wird vorher überprüft, ob das Custom-HTML-Tag bereits definiert ist. Dies geschieht in Zeile 6.

4.3 IFrame

Wie ein IFrame eingebunden wird, wird in den Grundlagen Unterabschnitt 2.3.1 erläutert. Die Prototypen wurden hinter dem Gateway gehostet, um den Aufwand für die extra Zertifikate zu sparen, wie in Abschnitt 4.1 erklärt wird.
Mögliche Variablen werden bei IFrames per URL-Attribute übergeben, siehe Unterabschnitt 4.5.1.
Eine Versionierung wie bei Custom-Web-Componets ist nicht so einfach möglich, da dies mehrere laufende Instanzen benötigen würde, der Grund dafür wird in Unterabschnitt 6.1.5 erklärt.

4.3.1 Angular

Worauf bei der Implementierung des Angular-Prototypen geachtet wurde, war die Fixierung der Höhe mittels dem Style-Attribute *height: fit-content*, damit sich die Höhe automatisch an die Höhe des Inhalts des IFrames anpasst.

Die IFrame Prototypen kommunizieren über MessageEvents, hierfür benötigt die ParentWebsite einen Listener:

Listing 13: Eventlistener einer ParentWebsite in Angular

```
1 @HostListener('window:message', ['$event'])
2 onMessage(event: MessageEvent) {
3     const message: IMessage = JSON.parse(event.data);
4     if (message.name !== undefined && message.id !== undefined)
5       ...
6 }
```

Im Angular-IFrame wird eine Funktion mittels einer Annotation zum Eventlistener. Die Daten der Message können mit JSON und einem Interface in ein Objekt umgewandelt und verwendet werden, wobei der Konfigurationsaufwand gering ist. Auf genauere Kommunikation von IFrames in diesem Zusammenhang wird im Unterabschnitt 4.5.1 eingegangen.

4.3.2 React

Ein React-IFrame einzubinden, erfordert, genau wie das Angular-IFrame in Unterabschnitt 4.3.1, keinen großen Konfigurationsaufwand.
Bei dem React-IFrame war eine Fixierung der Höhe auf die Höhe des dynamischen Contents mit Styling-Attributen nicht möglich.
Die Attribute werden aus der URL ausgelesen. Dies sieht bei den Prototypen wie folgt aus:
const accessToken = searchParams.get('access_token');

4.4 Dokumentation

Um die Verwendung der Komponenten für alle Teams zu vereinfachen, ist eine aussagekräftige und universalverständliche Dokumentation sinnvoll. Dies vermeidet Missverständnisse und minimiert technische Fragen bezüglich der Komponenten zwischen den Teams.

Es sollte eine Dokumentationsvorlage erstellt werden, die Dokumentationen sollten für die Teams zugänglich sein und vor jeder Veröffentlichung der Komponenten sollten die Schnittstellen getestet werden.

4.4.1 Custom-Components

Um eine Custom-Web-Component zu dokumentieren, sollte sich innerhalb der Teams auf ein einheitliches Pattern geeinigt werden. Das geht von der Namensgebung, über Styling bis zu Attributen und Events.
Im Folgenden werden Beispiele für eine Dokumentation von Custom-Web-Components genannt.

```
Team: products
Tag: "products-list"
Versions: [0.0.1, 0.0.2]
Latest: 0.0.2
ImportUrls: [
              "https://components.your.domain/products/0.0.2-main.js",
              "https://components.your.domain/products/main.js"
             ]
Attributs: [accesstoken: String, userid: String, apiurl: String]
DispatchEvents: [
    "products:purchase":
        {
            detail: {
                id: String,
                name: String
            }
        }
    ]
```

Abbildung 5: Dokumentationsbeispiel von Custom-Web-Components

Für die Prototypen wurde sich entschieden, bei den Namensgebungen der Events und Tags sich anhand des Teamnames zu orientieren. Diese werden vor dem Bindestrich des Tags geschrieben und bei Events wird der Teamname mit dem Namen des Events durch ein Doppelpunkt getrennt. [Gee20] Die Attribute benötigen keine weitere Teamzugehörigkeit in Form von Namespacing.

Die Dokumentation ist framework- und liberary-unabhängig, somit können die Teams sich frei nach Ihrem Wissen und Vorlieben für das Framework oder die Liberaries entscheiden.

4.4.2 IFrames

Für IFrames eine Dokumentation aufzubauen, muss sich nicht sehr von der von Custom-Web-Components unterscheiden. Bei IFrames sind die relevanten Attribute des IFrame die URL-Paramenter und MessageEvents. Diese sind in der Dokumentation zu beschreiben. Die Versionen fallen bei IFrames weg, wie in Abschnitt 4.3 erläutert.

```
Team: products
Url: "https://iframes.your.domain/products"
Attributs: [access_token: String, user_id: String, product_id: String,
    product_name: String]
MessageEvents: [
        {
            messageId: "products:purchase"
            body: {
                id: String,
                name: String
            }
        }
    ]
```

Abbildung 6: Dokumentationsbeispiel von einem IFrame

Die Attribute werden im String-Format übergeben, von komplexen Objektstrukturen wird von Geers [Gee20] abgeraten.
Bei den MessageEvents wird bei den Prototypen eine JSON-Struktur verwendet, dabei wird eine *messageId* als Identifikator verwendet. Hierbei wird die Namensgebung wie bei allen Prototypen verwendet und dokumentiert.

4.5 Kommunikation

Bei der Kommunikation sollten sich die Teams ebenfalls auf ein Pattern einigen und auch in der Dokumentation festhalten. Als Beispiel kann sich an den MessageEvents und CustomEvents der Prototypen orientiert werden, wie in Abschnitt 4.4 beschrieben.
Es gibt mehrere Möglichkeiten, wie Microfrontends untereinander und mit der Parentwebsite kommunizieren können, welcher Weg der Kommunikation gewählt wird, ist architekturabhängig. Wichtig dabei ist, dass die Cross-Team-Communication der Frontends zu anderen Team-Backends vermieden wird.

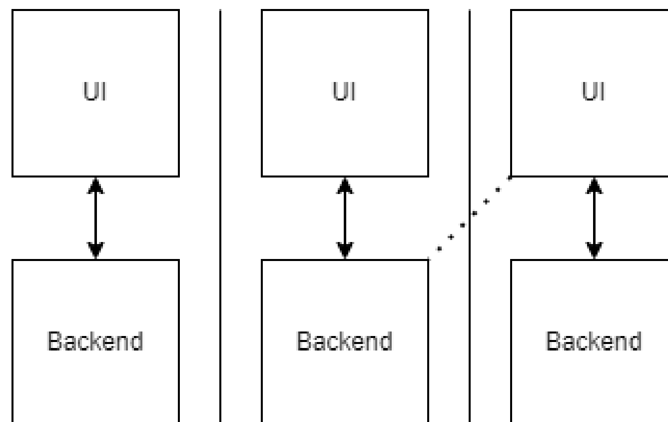

Abbildung 7: Architektur Cross-Team-Communication

Die Abbildung soll darstellen, dass die Frontends eines Teams nicht mit den Backend eines anderen Teams kommunizieren sollten (hier die gepunktete Linie). Um die Architektur und die Team-Vorteile der Microfrontends einzuhalten, ist diese Abtrennung sinnvoll.

4.5.1 IFrames

Die Kommunikations-Methode, welche bei IFrames zur ParentWebsite empfohlen wird, ist die MessageDispatch-Methode, diese wird auch bei den Prototypen dieser Arbeit verwendet. Hierfür senden die IFrames Messages über die *window*-Schnittstelle, diese sollten eine festgelegte Struktur aufweisen, wie in Unterabschnitt 4.4.2 beschrieben.
Um die Nachrichten zu empfangen, wird in der ParentWebsite ein MessageEvent-Listener initialisiert, dieser kann abhängig von dem verwendeten Framework/Libary über ein Interface geparsed werden. Von der ParentWebsite kann der Status des IFrames über die URL angepasst werden.

4.5.2 Custom-Web-Components

Bei den Custom-Web-Components wird die Veränderung der HTML-Custom-Attributes innerhalb der Mircofrontends überwacht. Dies gewährleistet eine dynamischere Verarbeitung der Änderungen ohne Neuladen der Komponente. Um von einer Custom-Web-Component zu anderen Custom-Web-Component oder zur Parentwebsite zu kommunizieren, können CustomEvents erstellt und dispatched werden. Um diese zu verarbeiten, muss ein EventListener auf ein Event mit einem festgelegten Namen definiert werden. In CustomEvents können auch komplexere Objekte versendet werden. Je nach Ebene auf die das Event gesendet werden soll, können bei dem CustomEvent *bubbles* auf true oder false gesetzt werden, wie Geers [Gee20] erklärt.

4.5.3 Andere Methoden

Eine andere Methode der Kommunikation ist die API-basierte Kommunikation, bei der die Frontends ihren State in einem Backendsystem angeben und andere Frontends diesen abrufen und verändern können. Dieser Ansatz erfordert ein sorgfältiges Design, komplexes Testen und einen extra aufgesetzten Backendservice. Dafür ist sicherzustellen, dass die API stabil ist und das Netzwerk den zusätzlichen Traffic verarbeiten kann.

Die Broadcast Channel API ist eine weitere Methode wie Frontends miteinander kommunizieren können. Die Methode ist ähnlich zu der von MessageEvents von IFrames, hierbei wird ein BroadcastChannel mit einem Namen initialisiert, andere Frontends können mit dem Namen dem Channel beitreten. Dort können dann Messages geposted und mittels dem Festlegen einer onMessage-Funktion gelesen werden. Der Hauptunterschied zu der Message-Dispatch-Methode ist die Identifikation des BroadcastChannels per Name und dass diese Methode nicht in allen Browsern unterstützt wird. Diese Ansätze werden von Piana [Pia] und Geers [Gee20] behandelt.

5 Vergleich der Prototypen

In diesem Kapitel werden die entwickelten Prototypen miteinander verglichen: Zuerst die beiden Frameworks im Bezug auf die Implementierungen der Mircofrontend-Technologien, anschließend die Microfrontend-Technologien.

5.1 IFrame Angular/React

Beim Aufsetzen von IFrames in beiden Frameworks wird wie beim normalen Aufsetzen einer Website in dem jeweiligen Framework vorgegangen. Die Websites werden initialisiert, entwickelt und für ein Deployment vorbereitet. Die beiden IFrame-Prototypen werden dann mittels Docker gebaut und im Deployment auf einen virtuellen Server hochgeladen. Sie werden hinter einem API-Gateway mit SSL-Zertifikat gestartet, um die Kommunikation über https laufen zu lassen. Durch das Routing hinter einem API-Gateway werden beim Prototypen die *base-href* gesetzt. Die Übergabe von Attributen erfolgt bei beiden IFrames über URL-Parameter, wie in Unterabschnitt 4.5.1 erklärt wurde.

Bei der IFrame Implementierung gibt es zwischen dem Angular und dem React auf IFrame-Ebene nur die standart Framework-Unterschiede, bei der Integration in die Parentwebsite tritt ein deutlicherer Unterschied auf.
Wenn ein IFrame als Teil einer Website eingebunden werden soll, ist für eine gute User-Experience die richtige Größe des IFrames wichtig. Diese sollte sich je nach Einsatzgebiet automatisch an die Größe des Inhalts anpassen oder festgelegt sein. Bei dem IFrame, welches in Angular geschrieben wurde, konnte mittels dem Styling-Attribut *height: fit-content* die Höhe an den Inhalt des IFrames dynamisch angepasst werden. Dies ist bei dem Prototypen in React auch mittels verschiedener Styling-Methoden nicht möglich gewesen, dort hat sich die Größe des IFrames nicht angepasst.
Die Kommunikation verläuft über die festgelegten Schnittstellen, wie in Unterabschnitt 4.4.2 erläutert. Es ist zu erkennen, dass die Kommunikation ein gemeinsames Pattern aufweißt, womit die Parentwebsite framework-unabhängig ist. Da die IFrames mit der Parentwebsite über Messages kommunizieren und hier mehr als nur ein unbenanntes Attribut übermittelt werden soll, wurde sich für ein JSON-Pattern als String entschieden. Die Parentwebsite kann dann die JSON-Struktur auslesen. Auch die Nachrichtenübermittlung von IFrames ist bei beiden Implementierungen recht gleich und unkompliziert über die *window.postmessage* Funktion gelöst.

5.2 Custom-Web-Component Angular/React

Auch bei den Custom-Web-Component Prototypen gibt es bereits beim Aufsetzen gewisse Unterschiede in den Frameworks. Bei Angular muss die Liberary *ngx-build-plus* hinzugefügt werden, damit die Custom-Web-Component gebaut werden kann.
In React wird die Liberary *esbuild* verwendet. Die Standard Build-Funktionen bauen das Projekt als Website und nicht als Custom-Web-Component.

Um in der Entwicklung das Microfrontend zu testen, kann ein einfaches HTML-Dokument verwendet werden, welches über ein *<script>*-Tag die vom Build generierten JS-Dateien herunterlädt. Dann kann das Custom-HTML-Tag einfach in den *body* des HTML-Dokuments eingefügt und getestet werden. Diese Variante funktioniert frameworkunabhängig für beide Prototypen, nur bei der Bereitstellung der JS-Dateien unterscheiden sich die Frameworks.
In Angular kann über den Befehl *http-server* ein einfacher Fileserver eines angegebenen Ordners über den angegebenen Port geöffnet werden. Während der Entwicklung kann darüber eine Kombination aus Build- und Server-Script verwendet werden. Für das Deployment wurde ein NodeJS Server verwendet, welcher die JS-Dateien bereitstellt, wie in Abschnitt 4.2 beschrieben.
In Angular werden beim Build eine *polyfills.js* Datei erstellt und die Funktionen von Browsern ergänzt, falls dieser die benötigten nicht besitzt. Es ist empfehlenswert, daher beide Dateien von diesem Prototypen zu exportieren.
React hingegen exportiert beim Bau mit *esbuild* nur die *main.js* Datei. Für die Entwicklung wurde bei dem Prototypen ein NodeJS/Express Server aufgesetzt, welcher den Output-Ordner der Build-Datei exportiert.
React bietet bei dem Nutzen von Custom-HTML-Attributen weniger Support wie Angular, wo mittels *@Input* die Attribute automatisch überwacht werden. Bei React müssen diese manuell ausgelesen und festgelegt werden, welche überwacht werden müssen. Wobei die React *attributeChangedCallback*-Funktion eine bessere Verarbeitung von Attributsveränderungen bietet, da dort neben dem Namen des Attributes auch der alte Wert übergeben wird.

Die Lifecycle-Funktionen bei Angular sind z.B.:

- ngOnChanges
- ngOnInit
- ngOnDestroy

während React die Folgenden verwendet:

- componentDidMount
- componentWillUnmount
- componentDidUpdate
- observedAttributes
- attributeChangedCallback

Zudem bietet Angular Build-In das Componenten-Styling, welches in React nicht vorhanden ist. Dies ist auch in der Verwendung als Microfrontend der Fall.

5.3 IFrame vs Custom-Web-Component

Abschließend werden die beiden Ansätze IFrame oder Custom-Web-Component miteinander verglichen.

5.3.1 Aufsetzen, Deployment

Beginnend beim Aufsetzen des Projektes.
IFrames werden wie normale Websites aufgesetzt, es ist keine zusätzliche Konfiguration nötig, mit Ausnahme der Base-Href, deren Mehraufwand sehr gering ausfällt. Auch das Deployment gleicht dem einer normalen Website.
Bei Custom-Web-Components kann das Aufsetzen je nach Verwendung etwas komplexer sein. Sollte nur eine Komponente exportiert werden, kann der einfache http-File-Server in dem Projekt verwendet werden. Sollten aber mehr als nur eine Komponente exportiert werden, ist ein dedizierter Component-Server empfehlenswert.

Für die Implementierung beider Prototypen wurde ein NodeJS-Server verwendet, welcher statische Ordner nach außen öffnet. Das Building und Deplyoment kann über CI/CD automatisiert und auch die Versionierung kann dort automatisiert werden. Dies wurde jedoch im Rahmen der Prototypen nicht implementiert.

5.3.2 Kommunikation

Weitere Unterschiede treten bei der Kommunikation auf. IFrames kommunizieren mit der Parent-Website über Messages, wie in Unterabschnitt 4.5.1 erläutert wurde. Dies erfordert eine ausführliche Dokumentation der Kommunikation und kann zu Performance-Problemen führen, falls komplexe und große Objekte ausgetauscht werden. Auch wenn die Implementierung bei IFrames weniger komplex aussieht, wird mit erhöhter Variation der Message-Types und Objekte erhöhter Entwicklungsaufwand nötig werden.
Bei der Kommunikation von Custom-Web-Components, welche in Unterabschnitt 4.5.2 erläutert, besteht die Möglichkeit mit der Parent-Website über Custom-Events und Observables direkter kommunizieren zu können, wodurch die Integration und Kommunikation effizienter und weniger komplex gestaltet ist.
Einschränkend ist zu erwähnen, sollte die Kommunikation der Komponenten ein komplexeres und variierendes Pattern aufweisen als einfache String-Messages und Parameter, wirkt sich das auf die Effizienz der IFrame-Kommunikation aus.

5.3.3 Styling

Bei dem Styling sind IFrames isoliert von dem Styling der Parent-Website, wodurch in dynamischen IFrames das dynamische Styling selbst implementiert werden muss. Die Isolierung kann vorteilhaft sein, um Style-Konflikte zu vermeiden, erschwert jedoch das einheitliche Styling beider Komponenten, IFrame und Parentwebsite.
Custem-Web-Components können individualisiert von der Parent-Website abgekapselt werden mittels eines *Shadow DOM*, siehee Geers [Gee20], welcher Teile oder die gesamte Komponente bezüglich des Sytlings isolieren kann. Dadurch kann eine IFrame ähnliche Isolierung geschaffen werden, um ebenfalls Styling-Konflikte zu vermeiden. Eine andere Möglichkeit bei Custom-Web-Components ist es, das Sytling mit Namespacing zu gestalten. Dabei können globale CSS-Klassen komponentenübergreifend verwendet werden und die Komponenten können über die Namensgebung der CSS-Klassen ein eigenes Styling verwenden, welches offen für Dynamik in Form von Variablen ist. Sollten Custom-Web-

Components nicht oder nur teilweise isoliert sein, wird das Einhalten des einheitlichen Styles der Website vereinfacht und auch eine dynamische Nutzung der Custom-Web-Component ermöglicht ohne weitere Modifikationen innerhalb des Microfrontends. Ein komponentenübergreifendes Styling wird so unkompliziert ermöglicht, welches durch festes Namespacing organisiert wird. Ein solches Verhalten kann auch bei IFrames hervorgerufen werden, sollten diese auch die Shared-Style-Datei integrieren. Dies könnte aber zu erhöhter Ladezeit führen, sollte die Datei eine komplexe Größe erreichen. Durch die Isolierung der IFrames müsste die Datei auch doppelt geladen werden. Das Cashing ist hierbei browser- und implementierungsabhängig.

5.3.4 Performance

IFrames weisen durch ihre Isolierung einen erhöhten Delay beim Laden der Seite auf, da der Inhalt des IFrames ein eigenes Dokument darstellt und somit extra mit einer HTTP-Anfrage geladen werden muss. Custom-Web-Components im Vergleich dazu sind in das Dokument der Parent-Website eingebunden und das Script kann asynchron zur Darstellung der eigentlichen Seite geladen werden, wodurch die Ladezeit der Gesamtseite verringert werden kann, vergleiche Redmond [Red].
Zudem kann eine komplexe Kommunikation, wie in Unterabschnitt 5.3.2 beschrieben, ebenfalls die Website verlangsamen, sollte das Endgerät nicht die nötige Rechenkapazität aufweisen.

6 Anwendung der Qualitätsmodelle

In Kapitel 3 wurden Qualitätsmodelle für verschiedene Attribute erstellt, diese werden nun auf die Prototypen weitestgehend angewendet, beschrieben und im Abschnitt 6.3 in einer Tabelle dokumentiert.

6.1 IFrame

6.1.1 Funktionalität

Die IFrame-Komponenten wurden hinter einem API-Gateway deployed, welches ein SSL Zertifikat hinterlegt hat. Daher wird die Kommunikation über HTTPS verschlüsselt. Das IFrame als Microfrontend ist für alle Domains verfügbar, dies kann über die *X-Frame-Options* des Web-Servers konfiguriert werden. Die Zugangskontrolle und Verschlüsselung ist damit vorhanden.
Da die Prototypen weder bestimmten Sicherheitsbestimmungen oder Vorschriften entsprechen müssen, wird der Punkt im Qualitätsmodell gestrichen.
Für die IFrames wurden Dokumentationen der Schnittstellen erstellt, jedoch ist eine Versionierung sehr umständlich und nicht gegeben, siehe Unterabschnitt 6.1.6.

6.1.2 Effizienz

Für die beiden IFrame Prototypen musste jeweils eine Instanz auf dem virtuellen Server aufgesetzt werden. Die Instanzenanzahl verläuft linear mit der Anzahl der IFrame-Microfrontends.
Je nach Anforderungen und verfügbarer Infrastruktur können die Instanzen vertikal und horizontal skaliert werden, sollte eine erhöhte Anfrage das System zu sehr fordern. Die Instanzen können aber nur einzelnd skaliert werden.
Die Antwortzeiten bei den IFrame Prototypen wurden mittels *PageSpeed Insights* und eigenen Messungen ermittelt.

Für den React Prototypen liegt die Gesamtladezeit beim ersten Laden bei 1170ms und nach dem Caching bei 640ms. Bis der erste Inhalt auf der Seite angezeigt wird, beträgt die Zeit 300ms und bis der vollständige Inhalt beim ersten Laden angezeigt wird 600ms. Bei Angular liegt die Gesamtladezeit beim ersten Laden bei 1280ms und nach dem Caching bei 650ms. Bis der erste Inhalt auf der Seite angezeigt wird, beträgt die Zeit 700ms und bis der vollständige Inhalt beim ersten Laden angezeigt wird 800ms.

6.1.3 Zuverlässigkeit

Die Prototypen wurden mittels Docker gebaut und auf einem Cloud-Server eines Rechenzentrums gehostet, wodurch die Ausfallrate des Servers minimiert wurde.
Da die Prototypen in einem Docker-Container gebaut und deployed sind, ist auch die Wiederherstellung nach einem Fehler innerhalb des Containers sehr einfach und kann automatisiert werden.
Damit die Endnutzer nicht eine leere Seite angezeigt bekommen, wurde ein HealthCheck-Endpoint eingerichtet. Dieser kann von dem Frontend angefragt werden, um zu überprüfen, ob der Service erreichbar ist. Sollte der Service nicht erreichbar sein, kann eine Fehlermeldung angezeigt werden.

6.1.4 Benutzerfreundlichkeit

Mit der Verwendung von IFrames ist die Einarbeitungszeit recht gering, da Webentwickler eine normale Website programmieren und nur die vorher beschlossenen Kommunikationsschnittstellen via Messages einbinden müssen. Der Konfigurationsaufwand eines IFrame-Microfrontends ist ebenfalls sehr gering, da es wie eine normale Website gebaut und deployed werden kann.
Verwaltungsmäßig kann eine Vielzahl von IFrame-Microfrontends viele Pods/Webservices generieren, welches die Übersichtlichkeit und Verwaltung erschweren könnte, zudem wird der Betriebsaufwand pro weiteres Microfrontend erhöht.
Vorwissen ist für die Implementierung von IFrames nur im Bereich des Messages und der Dokumentation notwendig, das eigentliche Wissen eine Website zu programmieren wird vorausgesetzt. Die Message-Schnittstelle des IFrames bei den Prototypen ist dokumentiert genauso wie die Übergabe der benötigten Parameter. In einer vollständigen Dokumentation für das Produkt sollte aber ebenfalls das Verhalten bei fehlerhafter Übergabe und der Funktionumfang enthalten sein.

Dies wurde im Rahmen der Prototypen nicht umgesetzt. Die Flexibilität bei IFrame Microfrontends ist nicht optimal, da zum Beispiel bei React die Höhe nicht einfach dynamisch eingestellt werden kann und durch die Isolierung von IFrames diese ebenfalls nicht stylistisch flexibel eingesetzt werden können. Das Verhalten von IFrames ist das einer eigenen Website und somit ist die Flexibilität eingeschränkt.

6.1.5 Wartungsfreundlichkeit

Um die Verhaltenweise eines IFrame-Mircofrontends zu verändern, ist eine Abstimmung mit den anderen Teams, welches dieses Microfrontend nutzen, nötig. Sobald die Änderungen auf dem Server deployt werden, spiegeln diese auch gleich das veränderte Verhalten in den Parent-Website wieder. Ohne Teamabstimmung kommt es zu Fehlverhalten.
Das Testen eines IFrames ist ohne Parent-Website möglich. Die Einbindung und Kommunikation muss aber über eine Parent-Website, welche auch nur ein HTML-Dokument zum Testen sein kann, geprüft werden.

6.1.6 Übertragbarkeit

Das Deployment ist durch die Verwendung von Docker, GitLab CI/CD und Cloud-Servern automatisiert, wodurch der Prozess vom Deployment bis zum Aufsetzen in anderen Umgebungen unkompliziert und wenig zeitintensiv ist.
Die Prototypen sind so programmiert, dass diese wenig Abhängigkeiten zur Parent-Website und somit eine hohe Wiederverwertbarkeit für andere Projekte aufweisen. Nur das Design ist isoliert.

6.1.7 Marktfähigkeit

Zu den Kosten werden für die Prototypen wenig Aussagen getroffen. Es werden nur die Betriebskosten erwähnt, da bei einer hohen Anzahl an IFrame-Microfrontends viele Pods/Webservices benötigt werden.

6.2 Custom-Web-Component

6.2.1 Funktionalität

Custom-Web-Components werden im Umfange der Prototyp-Entwicklung in den Projekten gebaut und die zu exportierenden Dateien werden mittels eines NodeJS Servers, welcher hinter dem Gateway geroutet ist, gehostet. Da es sich um Frontend-Komponenten handelt, ist eine Verschlüsselung nicht nötig, aber um die Intigrität der Hauptwebsite zu erhalten, werden die Dateien über eine HTTPS-Verbindung bereitgestellt. Manche Browser akzeptieren keine HTTP-Verbindungen, wenn die Hauptwebsite als HTTPS angegeben ist.
Da die Prototypen weder bestimmten Sicherheitsbestimmungen oder Vorschriften entsprechen müssen, wird der Punkt wie bei Unterabschnitt 6.1.1 gestrichen.
Durch die Eigenschaft, dass Custom-Web-Components als JS-Dateien und die Prototypen mittels eines NodeJS-Servers bereitgestellt werden, können dort mehrere Versionen der Microfrontends hinterlegt werden. Die Dokumentation wurde, wie in Unterabschnitt 4.4.1 beschrieben, erstellt.

6.2.2 Effizienz

Für die Custom-Web-Components wurde im Rahmen der Prototypen ein NodeJS-Server aufgesetzt, welcher die JS-Dateien hostet. Hierfür wurde für beide Prototypen gemeinsam nur ein Server benötigt. Dieser Service kann je nach Infrastruktur horizontal und/oder vertikal skaliert werden.
Um die Dateien herunterzuladen, wurde für den Angular Prototyp, welcher zwei JS-Dateien benötigt, deren Requests asynchron durchgeführt werden, die folgenden Ladezeiten gemessen:
Main.js: 162ms bis maximal 203ms
Polyfills.js: 83ms bis maximal 90ms

Die React JS-Datei hat dabei nur eine Ladezeit von 108ms bis maximal 183ms aufgewiesen.

6.2.3 Zuverlässigkeit

Da die Dateien der Custom-Web-Components mittels eines NodeJS-Servers bereitgestellt werden, welcher zu einem Docker-Container gebaut wurde, ist die Wiederherstellung und das Deployment recht simple und automatisierbar. In den NodeJS-Server wurde ebenfalls ein HealthCheck eingebaut, sodass die Websites, welche die Microfrontends nutzen wollen, auch überprüfen können, ob der Service erreichbar ist und falls nicht eine entsprechende Meldung dem Nutzer anzeigen können.

6.2.4 Benutzerfreundlichkeit

Für die Verwendung von Custom-Web-Components wird über das Standardwissen, welches zur Webentwicklung benötigt wird, nichts vorausgesetzt. Sie werden wie HTML-Tags in den Code eingebunden und es werden auch die Attribute wie bei normalen HTML-Tags gesetzt. Der Unterschied ist bei den Custom-Events, welche noch im Logik-Teil der Website eingebunden werden müssen. Diese setzen aber keine Komplexität voraus und sind bei den Prototypen mittels Namespacing zuordbar und dokumentiert. Diese Attribute und Custom-Events sind in der Dokumentation beschrieben und ebenfalls ist dort die URL zur Integration hinterlegt.
Um Custom-Web-Components zu verwalten, wurde für die Prototypen ein eigener NodeJS-Server gewählt, welcher die Dateien bereitstellt. Bei einer strukturierten Ordnerstruktur, ist weder der Verwaltungsaufwand hoch, noch der Betriebsaufwand, da eine Versionierung möglich ist. Ebenso ist die Konfiguration simpel, da weder ein weiterer Path im Gateway angelegt werden muss, wie beim IFrame, noch viel Aufwand für das Deployment betrieben werden muss, wenn das Buildung automatisiert ist.
Die Einbindung von Custom-Web-Components bietet viel Flexibilität, da diese wie HTML-Tags behandelt und ihnen auch Style-Attribute zugewiesen werden können, um sie im Verhalten zu kontrollieren.

6.2.5 Wartungsfreundlichkeit

Custom-Web-Components sind sehr wartungsfreundlich. Durch die Versionierung können Veränderungen des Systems von verschiedenen Versionen besser verglichen werden und Änderungen können vorgenommen und deployed werden, ohne ein zeitgleiches Deplyoment mit anderen Teams abzusprechen. Tests sind wie bei normalen Websites über UI-Tests, möglich und bestimmen die Testbarkeit. Dies wurde für die Prototypen im Rahmen dieser Arbeit nicht durchgeführt.

6.2.6 Übertragbarkeit

Da das Deplyoment bei den Custom-Web-Components über Docker, GitLab CI/CD und Cloud-Servern durchgeführt wird, ist weder das Aufsetzten in neuen Projekten noch das Übertragen auf eine andere Infrastruktur komplex. Ebenfalls sind die Prototypen mit so wenig Abhängigkeiten zur Parentwebsite entwickelt wie möglich, wodurch diese ebenfalls in andere Parentwebsites integriert werden können.

6.2.7 Marktfähigkeit

Zu den Kosten werden für die Prototypen wenig Aussagen getroffen. Bei der Custom-Web-Components-Implementierung der Prototypen ist die Anzahl der Microfrontends und Versionen von geringer Relevanz, da diese mittels eines Servers, gehostet werden und auch eine erhöhte Anzahl keine lineare Steigerung der Pod-Anzahl benötigt.

6.3 Qualitätsmodelle

6.3.1 IFrames

Bereich	Merkmal	Bewertung
Funktionalität	Sicherheit	True
	Einhaltung der Vorschriften	-
	Kompatibilität	False
Effizienz	Ressourcenkosten	5
	Antwortzeiten	300-1280ms
	Skalierbarkeit	5
Zuverlässigkeit	Absturzraten/risiko	10
	Wiederherstellungskomplexität	10
	Errorhandling	True
Benutzerfreundlichkeit	Aktionszeit	10
	Konfigurationszeit	10
	Einarbeitungszeit	9
	Dokumentation	5
	Verwaltungsaufwand	3
	Betriebsaufwand	3
	Benötigtes Vorwissen	10
	Flexibilität	2
Wartungsfreundlichkeit	Veränderbarkeit	2
	Testbarkeit	10
Übertragbarkeit	Deployment	10
	Wiederverwertbarkeit	7
Marktfähigkeit	Entwicklungszeit	-
	Entwicklungskosten	-
	Betriebskosten	5

Tabelle 8: Qualitätsmodelle für die IFrame-Prototypen

6.3.2 Custom-Web-Component

Bereich	Merkmal	Bewertung
Funktionalität	Sicherheit	True
	Einhaltung der Vorschriften	-
	Kompatibilität	True
Effizienz	Ressourcenkosten	10
	Antwortzeiten	83-203ms
	Skalierbarkeit	10
Zuverlässigkeit	Absturzraten/risiko	10
	Wiederherstellungskomplexität	10
	Errorhandling	True
Benutzerfreundlichkeit	Aktionszeit	10
	Konfigurationszeit	10
	Einarbeitungszeit	9
	Dokumentation	5
	Verwaltungsaufwand	8
	Betriebsaufwand	8
	Benötigtes Vorwissen	10
	Flexibilität	10
Wartungsfreundlichkeit	Veränderbarkeit	10
	Testbarkeit	10
Übertragbarkeit	Deployment	10
	Wiederverwertbarkeit	10
Marktfähigkeit	Entwicklungszeit	-
	Entwicklungskosten	-
	Betriebskosten	10

Tabelle 9: Qualitätsmodelle für die Custom-Web-Component-Prototypen

7 Auswertung

In dem Vergleich der Prototypen in Kapitel 5 wird erst auf die Unterschiede von Angular zu React in Bezug auf die Entwicklung eines IFrame-Microfrontends eingegangen. Hierbei wurden keine nennenswerten Unterschiede, bis auf ein Styling-Problem bei React, der beiden Frameworks festgestellt.
Werden nun die beiden Frameworks in Bezug auf Custom-Web-Components verglichen, wurden mehrere Unterschiede sowohl im Support des Frameworks, als auch im eigentlichen Export/Build des Microfrontends festgestellt. Bei einer einmaligen Konfiguration eines Whitelabel-Projektes ist der Konfigurationsaufwand für weitere Projekte durch ein Duplizieren des Whitelabel-Projektes für beide Frameworks gering. Auch für die Entwicklung bietet zwar Angular mit dem integrierten http-Server einen einfacheren Zugriff auf die gebauten JS-Dateien, dieser ist jedoch nicht nennenswert bezogen auf die geringe Komplexität, einen NodeJS-Server aufzusetzen, der die gleiche Funktionalität bereitstellt. Auch wenn Angular zwei JS-Dateien zum Export bereitstellt, um mögliche Browser-Inkompatiblität auszugleichen, ist festzustellen, dass die beiden Frameworks keine entscheidenden Konfigurationsunterschiede aufweisen, die eine Wahl preferieren würde.
Der wohl größte Unterschied liegt bei den Lifecycle-Funktionen, die bei bestimmten Events der Microfrontends aufgerufen werden. Je nach Einsatzgebiet oder Funktion, welche das Microfrontend abdecken soll, könnte dort ein Framework durch seine Lifecycle-Funktionen preferiert werden.
Beide Frameworks bieten das volle Set zum Erstellen und Deployen von Microfrontends. Welches Framework genutzt werden sollte, ist vom spezifischen Ansatz des Projektes abhängig. Angular ist für komplexere Komponenten und Custom-web-Components, React für dynamische Generierung optimiert.

Um die beiden Ansätze für Microfrontends miteinander besser zu vergleichen, wurden die Unterschiede der Qualitätsmodelle im Folgenden zusammengefasst:

Bereich	Merkmal	IFrame	C-W-C
Funktionalität	Kompatibilität	False	True
Effizienz	Ressourcenkosten	5	10
	Antwortzeiten	300-1280ms	83-203ms
	Skalierbarkeit	5	10
Benutzerfreundlichkeit	Verwaltungsaufwand	3	8
	Betriebsaufwand	3	8
	Flexibilität	2	10
Wartungsfreundlichkeit	Veränderbarkeit	2	10
Übertragbarkeit	Wiederverwertbarkeit	7	10
Marktfähigkeit	Betriebskosten	5	10

Tabelle 10: Vergleich der Qualitätsmodelle von den IFrame- und Custom-Web-Component-Prototypen

Der erste Punkt, worin sich die Ansätze unterscheiden, ist die Kompatibilität. Das Deployment von Custom-Web-Components ist durch Versionierung der JS-Dateien den IFrames etwas voraus, das heißt, auch andere Teams können ältere Custom-Web-Components verwenden, bis sie ihre Website auf neueste Version angepasst und getestet haben, und die Teams müssen das Deployment nicht untereinander abstimmen. Bei IFrames wird das Deployment einer neuen Version sofortige Auswirkungen auf alle Teams zeigen, welche dieses integriert haben, und deshalb müssen Änderungen in beiden Komponenten (Microfrontend und Parentwebsite) und auch das Deployment abgestimmt werden.

Die Ressourcenkosten unterscheiden sich sehr, da für jedes IFrame-Microfrontend eine weitere Instanz auf dem Server aufgesetzt werden muss. Eine hohe Anzahl an IFrame-Microfrontends erzeugt daher eine ebenso hohe Anzahl von laufenden Instanzen. Auch wenn Frontendinstanzen nicht viel Rechenleistung benötigen, kann bei einem Kubernetescluster als Beispiel schnell die maximale Anzahl an Pods erreicht werden.
Die Custom-Web-Components, wenn diese wie bei den Prototypen mit einem NodeJS-Server aufgesetzt werden, benötigen für eine unbegrenzte Anzahl nur eine Instanz. Diese eine Instanz kann je nach Infrastruktur automatisiert skaliert werden. Durch diesen Ansatz werden Ressourcenkosten gespart und die Übersicht von laufenden Instanzen wird nicht überfüllt mit Microfrontends.
Die Antwortzeiten der beiden Ansätze unterscheiden sich auch sehr. Die Custom-Web-Components haben eine schnellere Downloadzeit aller benötigten Scripts, die Custom-

Web-Component-Scripts können auch vor dem Anzeigen der Seite, also wenn der Nutzer noch auf einer anderen Seite der Parentwebsite ist, im Hintergrund heruntergeladen werden, wodurch die Ladezeit beim Anzeigen des Microfrontends noch weiter verringert wird.

Im Bereich der Benutzerfreundlichkeit überwiegt die Custom-Web-Component ebenfalls. Der Betriebs- und Verwaltungsaufwand ist hauptsächlich mit den Organisationsaufwänden der Instanzen verbunden, auf welche im Zusammenhang mit der Effizienz bereits eingegangen wurde. Zudem sind bei den Custom-Web-Componenten die Versionierung und die dazugehörigen Unterschiede in der Dokumentation und den Schnittstellen zu pflegen. Die Flexibilität der IFrames wird durch die Abkapselung und Styling-Probleme beim dynamischen Content eingeschränkt, wobei dies bei Custom-Web-Componenten frei konfigurierbar ist.

Bei der Veränderbarkeit der Ansätze ist zu erkennen, dass die IFrames schlechter abschneiden. Dies ist mit der fehlenden Versionierung zu begründen. Die Versionierung von Microfrontends ermöglicht es, dass die Teams unabhängig voneinander Änderungen und neue Versionen deployen können. Es ist keine Abstimmung der Teams notwendig und keine zeitgleichen Deployments. Durch das Fehlen dieser Eigenschaft bei den IFrames müssen die Teams bezüglich des Deployments in engen Austausch stehen, um fehlerhaftes Verhalten zu vermeiden, und es entstehen engere Abhängigkeiten.

Die Wiederverwertbarkeit ist bei beiden Varianten durch das auf Cloud-Infrastruktur orientierte Deployment und die ausführliche Schnittstellendefinitionen hoch bewertet. Die Abhängigkeit der IFrame-Microfrontends von der Parentwebsite ist nur durch die Domain und die damit verbundene CORS-Abhängigkeit gegeben. Abhängig von dem Umgang mit den Style-Dateien können ebenfalls Abhängigkeiten auftreten.

Da es sich bei der bewerteten Implementierung um Prototypen handelt, welche nicht mittels Projekt und Kostenplan entwickelt wurden, sind nur die Betriebskosten in der Marktfähigkeit bewertet worden.
Durch die bereits bei den Ressourcenkosten erwähnten Unterschiede können diese ebenfalls auf die Betriebskosten angewendet werden.

8 Fazit

In dieser Arbeit wurden verschiedene Architekturen und Frameworks zur Entwicklung von Microfrontends betrachtet, dabei wurden speziell IFrames und Custom-Web-Components verglichen. Im Zuge der Betrachtung wurde ein Qualitätsmodell zur Bewertung der Microfrontentends entworfen. Um die verschiedenen Patterns der Microfrontends mittels des Qualitätsmodells auszuwerten, wurden im Rahmen der Arbeit vier Prototypen entworfen, zwei IFrame Prototypen mit den Frameworks Angular und React und zwei Custom-Web-Components, ebenfalls auf die gleichen Frameworks aufgeteilt. Diese Prototypen wurden miteinander verglichen und mittels des Qualitätsmodells bewertet und im Kapitel 7 die Ergebnisse ausgewertet.
Beide Ansätze haben ihre Einsatzgebiete, wobei die Custom-Web-Componenten für Projekte mit mehreren Microfrontends gegenüber den IFrames überwiegen. Durch den generellen Ansatz von Microfrontends können die Teams verkleinert werden und sich auf ihren Part des Systems konzentrieren. Dadurch würden Kommunikationswege und -aufwand verringert werden. Die Versionierung und die damit entstehende Reduzierung benötigter Abstimmungen unter den Teams durch die Custom-Web-Componenten bieten flexiblere und agilere Teamorganisation, Zielsetzung und Resourcenplanung.
Beide Ansätze sind durch die Kommunikationspattern frameworkunabhängig und flexibel in den Einsatzgebieten.

Sollten die Microfrontends als WebViews in Mobile-Applications ebenfalls genutzt werden, ist zu beachten, wie die Schnittstellen definiert sind und dass bei Custom-Web-Componenten ein HTML-Dokument mit einer Java-Script Brücke/Interface nach der Dokumentation [GA] für die Kommunikation genutzt werden muss.
Auch wenn die Custom-Web-Component im Qualitätsmodell besser abschneidet als die IFrame-Architektur, ist die Verwendung der Microfrontends und der Umfang des Systems entscheidend. Sollten WebViews in Apps und nur wenige Microfrontends mit möglicherweise erhöhter Komplexität verwendet werden, können IFrames von Vorteil sein. Hierbei sollte das Versionierungsproblem und die verbundenen Komplikationen mit versionsunterschiedlicher Kommunikation unbedingt miteinbezogen werden.
In der Entwicklung für Browser Systeme oder bei einer hohen Anzahl an verschiedenen Microfrontends und bei generell flexibler Ziel- und Teamkombination, sowie bei der WebView Verwendung, sind die Custom-Web-Componenten klar im Vorteil.

8.1 Ausblick

In dieser Arbeit wurden die beiden Microfrontend-Architekturen von IFrames und Custom-Web-Components behandelt. Dabei wurde insbesondere auf die Frameworks React und Angular eingegangen. Das erstellte Qualitätsmodell sowie die identifizierten Unterschiede der Frameworks können auf weitere Frameworks erweitert werden, um über die hier getroffene Auswahl hinauszugehen.
Darüber hinaus sollte die Sicherheit der Kommunikation und Integration genauer betrachtet werden, um zu sicherzustellen, dass eigene Microfrontends nur auf auf eigenen WebSites verwendet werden können.

Aufgrund der Verbreitung von WebViews in mobilen Applikationen stellt sich auch die Frage nach der Rolle von Microfrontends und deren Kommunikationsmustern sowie den auftretenden Komplikationen. Weiterhin kann geprüft werden, welcher der Architekturansätze, die in dieser Arbeit behandelt wurden, für ein solches Einssatzgebiet am besten geeignet sind.

Literatur

[AA05] Alexandre Alvaro, S. M.: *Quality Attributes for a Component Quality Model.* Computer Science, 2005

[AA06] Alexandre Alvaro, S. M.: *A Software Component Quality Model: A Preliminary Evaluation.* 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE, 2006

[Alm] Almeida, Fernando: *Assessing Quality Issues in Component Based Software Development.* https://www.academia.edu/19555836/Assessing_Quality_Issues_in_Component_Based_Software_Development

[GA] Google; AOSP: *Dokumentation: Build web apps in WebView.* https://developer.android.com/develop/ui/views/layout/webapps/webview#BindingJavaScript

[Gee20] Geers, Michael: *Mirco Frontends in Action.* Manning, 2020

[Har] Harris, Chandler: *Microservices und monolithische Architektur im Vergleich.* https://www.atlassian.com/de/microservices/microservices-architecture/microservices-vs-monolith

[Kha] Khaku, Ammar: *How Netflix microservices tackle dataset pub-sub.* https://netflixtechblog.com/how-netflix-microservices-tackle-dataset-pub-sub-\a068adcc9a

[Kol] Kolesnikov, Dmitry: *Using Microservices to Power Fashion Search and Discovery.* https://engineering.zalando.com/posts/2017/02/using-microservices-to-power-fashion-search-and-discovery.html

[Lan] Lando, Billy J.: *Extrahieren von Micro-Frontends aus einer monolithischen Frontend Anwendung.* https://oceanrep.geomar.de/id/eprint/55815/1/thesis-lando-2022.pdf

[MB06] M. Bertoa, A. V.: *Measuring the usability of software components.* Elsevier, 2006

[MF] Martin Fowler, James L.: *Microservices*. https://martinfowler.com/articles/microservices.html

[New15] Newman, Sam: *Building Microservices*. O'Reilly Media, Inc., 2015

[Pia] Piana, Alessandro: *Broadcasr Channel API*. https://gist.github.com/alexis89x/041a8e20a9193f3c47fb

[Red] Redmond, Sam: *Will Web Components Replace iFrames?* https://sam-redmond.com/will-web-components-replace-iframes-275a0824377b

[Rei] Reinartz, Christoph: *Large Scale CSS Refactoring at trivago*. https://medium.com/@pistenprinz/large-scale-css-refactoring-at-trivago-4602113c4a26

[Sak] Sakovich, Natallia: *Top Most Popular Frontend Frameworks 2023*. https://www.sam-solutions.com/blog/best-frontend-framework/

[Ver] Verma, Anjali: *Micro-Frontend using Web Components*. https://medium.com/swlh/micro-frontend-using-web-components-e9faacfc101b

Listings

1 Schema in app.module . . . 18
2 Beispiel Custom-HTML-Tag . . . 19
3 Beispiel Custom-HTML-Attributes . . . 20
4 Beispiel von update Custom-HTML-Attributes . . . 20
5 Zusätzliche und überarbeitete Scrips . . . 21
6 Zusätzliche und überarbeitete Dependencies . . . 21
7 Definieren von Custom-HTML-Tags . . . 22
8 Definieren der Routen . . . 24
9 Package.json für Custom-Web-Components in React . . . 25
10 Beispiel einer CORS-Erweiterung bei einem NodeJs-Server . . . 25
11 Überschreiben der Methoden zur Überwachung der Attribute . . . 27
12 Definieren des Custom-HTML-Tags . . . 28
13 Eventlistener einer ParentWebsite in Angular . . . 29

Abbildungsverzeichnis

1 Komponenten-Diagramm der Prototypen-Architektur . . . 15
2 Komponenten-Diagramm der Prototypen-Architektur ohne Gateway . . . 16
3 Komponenten-Diagramm der Prototypen-Architektur mit geteilten Microservices . . . 17
4 Schnittstellendefinition von der Custom-Web-Component Products . . . 26
5 Dokumentationsbeispiel von Custom-Web-Components . . . 30
6 Dokumentationsbeispiel von einem IFrame . . . 31
7 Architektur Cross-Team-Communication . . . 32

Tabellenverzeichnis

1 Qualitätseigenschaften des Funktionalitäts-Attributs 12
2 Qualitätseigenschaften des Effizienz-Attributs 12
3 Qualitätseigenschaften des Zuverlässigkeit-Attributs 12
4 Qualitätseigenschaften des Benutzerfreundlichkeit-Attributs 13
5 Qualitätseigenschaften des Wartungsfreundlichkeit-Attributs 13
6 Qualitätseigenschaften des Übertragbarkeit-Attributs 13
7 Qualitätseigenschaften des Marktfähigkeit-Attributs 14
8 Qualitätsmodelle für die IFrame-Prototypen 45
9 Qualitätsmodelle für die Custom-Web-Component-Prototypen 46
10 Vergleich der Qualitätsmodelle von den IFrame- und Custom-Web-Component-Prototypen 48

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ______________________________

Vorname: ______________________________

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Frontend Architekturen und Patterns für microservices-basierte Backendsysteme

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

______________	______________	______________________
Ort	Datum	Unterschrift im Original