h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Informatik –

Die Überführung eines monolithischen Web-Frontends in eine Micro-Frontend-Architektur

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Matthias Feyll

Matrikelnummer: 764775

Referent	:	Prof. Dr. Andreas Heinemann
Korreferent	:	Prof. Dr. Urs Andelfinger

ERKLÄRUNG

Ich versichere hiermit, dass ich den vorliegenden Bericht selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 13. September 2023

Matthias Feyll

ZUSAMMENFASSUNG

Webseiten und Webapplikationen haben in der heutigen Zeit eine große Bedeutung. Große Webseiten besitzen komplexe Frontends, in denen HTML, CSS und JavaScript im Browser ausgeführt werden. Frontends wachsen mit der Entwicklung stetig und werden in einem monolithischen Aufbau zunehmend größer. Damit werden Aspekte wie die Wartbarkeit gefährdet. 2016 wurde erstmals eine Alternative zur monolithischen Architektur im Frontend vorgestellt. Diese wird Micro-Frontend-Architektur genannt. Es handelt sich um eine dezentrale Architektur, bei der das Frontend in kleinere Module, die sogenannten Micro-Frontends, aufgeteilt wird. Mit der Dezentralisierung soll die Komplexität in der stetigen Weiterentwicklung beherrschbar bleiben. Zusätzlich bietet die Architektur viele Möglichkeiten, für einzelne Micro-Frontends spezifische Technologien und Frameworks einzusetzen, um die Entwicklung zu vereinfachen.

Die Arbeit präsentiert ein Konzept zur Überführung eines monolithischen Web-Frontends in eine Micro-Frontend-Architektur. Darin wird beschrieben, wie eine Micro-Frontend-Architektur beschrieben wird und wie ein Monolith überführt werden kann. Als Abgrenzung beschränkt sich bei den Strategien zum Wiederzusammensetzen von Micro-Frontends lediglich auf das sogenannte Client-Side-Composing.

Darüber hinaus präsentiert die Arbeit ein weiteres Konzept, welches ein Vorgehen zur Identifizierung, Analyse und Auswertung von Fehlern beschreibt, die in einer Micro-Frontend-Architektur auftreten können. Neben der Übersicht soll das Konzept den Aufwand zur Implementierung einer vollumfänglichen Fehlerbehandlung reduzieren.

Die zwei Konzepte werden jeweils anhand einer Fallstudie und anschließender Diskussion evaluiert. An einem monolithischen Anwendungsfall wird zunächst die Überführung durchgeführt. Anschließend werden die Fehler, die in diesem Anwendungsfall auftreten können, herausgearbeitet und anschließend deren Behandlung implementiert.

Beide Konzepte konnten erfolgreich an dem Anwendungsfall durchgeführt werden. Die Diskussion eröffnet Aspekte, an denen die Konzepte weiter optimiert werden können.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

API Application Programming Interface
CDN Content Delivery Network
CLI Command Line Interface
CSS Cascading Style Sheet
DOM Document Object Model
ESI Edge-Side-Includes
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
JSON JavaScript Object Notation
JWT JSON Web Token
MF Micro-Frontend
NPM Node Package Manager
PWA Progressive Web App
RFC Request for Comments
SASS Syntactically Awesome Style Sheets
SEO Search Engine Optimization
SPA Single Page Applications
URI Uniform Resource Identifier
URL Uniform Resource Locator
UX User Experience

Teil I

THESIS

1 EINLEITUNG

1.1 MOTIVATION

Die Zahl der Webseiten ist über die letzten Jahrzehnte gestiegen [Loh23]. Ursprünglich bestehen diese aus den drei essenziellen Bestandteilen HTML, Cascading Style Sheet (CSS) und JavaScript. Mit den Jahren sind durch neue Technologien und Frameworks Webseiten deutlich größer und komplexer geworden.[Den23; Riv23; HYL23; TM22; Vai23]. Aspekte der Softwarequalität werden zunehmend schwieriger auf hohem Niveau zu halten, dazu zählt die Wartbarkeit und Skalierbarkeit [TM22]. Um die Komplexität zu beherrschen, existiert seit 2011 die Idee von Microservices [LF14]. Dies ist eine dezentrale Architektur, bei der Code, der auf der Serverseite ausgeführt wird (Backend), in Module aufgeteilt wird. Diese Module können dann vollständig voneinander entwickelt werden [Dra+17][RS21, Kapitel 1]. Microservices sind heute sehr beliebt und tragen dazu bei, Webseiten wartbar und erweiterbar zu halten [Dra+17].

Die MF-Architektur überträgt die Idee der dezentralen Architektur vom serverseitigem Backend in das Frontend [TM22]. Im Frontend wird das vom Backend gesendete HTML, CSS und JavaScript in einem Browser interpretiert. Die Idee der MF-Architektur ist die Unterteilung des Codes in kleinere Module, die sogenannten Micro-Frontends MFs [PMT21]. Dadurch kann die Entwicklung im Frontend von den vielen Vorteilen der dezentralen Architektur profitieren, ähnlich wie bei Microservices im Backend. Darunter zählen vor allem Aspekte der Wartbarkeit, Unabhängigkeit und Skalierbarkeit [PMT21]. Neben den Aspekten der Softwarequalität wird die Entwicklung im laufenden Betrieb effizienter. Jedes MF wird von einem Entwicklerteam betreut. Diese können vollständig auf einen kleineren Bereich fokussieren [Jac19].

Die Idee der MF-Architektur wurde erstmals 2016 von dem Unternehmen Thoughtworks[1] vorgestellt [Jac19; Rad20]. In den folgenden Jahren haben große Firmen wie Amazon, SAP, IKEA, Spotify, DAZN, Thalia, Zalando und viele weitere begonnen, die Idee der MF-Architektur in ihre Webseiten zu integrieren [Kro21; PMT21; TM22].

Es existiert keine einheitliche Definition wie eine MF-Architektur umgesetzt wird [Gee20, Kapitel 2]. Der Autor Geers [Gee20] betont, dass dies immer vom jeweiligen Anwendungsfall abhängt. Wissenschaftliche Publikationen stellen Best Practices, Empfehlungen und Lessons Learned zu allgemeinen Aspekten der MF-Architektur vor. Sie thematisieren wiederkehrende Probleme bei der Umsetzung.

In dieser Arbeit werden die wissenschaftlichen Empfehlungen und Vorgehensweisen aus der Literatur in ein Konzept überführt. Das Konzept beantwor-

1 https://www.thoughtworks.com/ zuletzt eingesehen am 20.07.2023

tet die Forschungsfrage: *"Wie kann ein monolithisches Web-Frontend in eine MF-Architektur überführt werden?"*

Neben den Vorteilen der MF-Architektur gibt es auch Herausforderungen, die berücksichtigt werden müssen. Durch die MF-Architektur entstehen zusätzliche Prozesse, die potentiell fehlschlagen können. Zum Beispiel kann ein Netzwerkfehler das Nachladen eines MFs verhindern. Werden Fehler nicht behandelt, kann dies zu einem inkonsistenten Zustand der Webseite und möglicherweise sogar zu Sicherheitslücken führen [WN04]. Daher müssen Fehler abgefangen und behandelt werden [PMT21]. Als Erweiterung der Forschungsfrage umfasst diese Arbeit ein Konzept zur Behandlung von Fehlern, die beim Nachladen und Einbinden von MFs auftreten können.

1.2 ZIEL DER ARBEIT

Um die Forschungsfrage zu beantworten, wird ein Konzept zur Überführung eines monolithischen Web-Frontends in eine MF-Architektur präsentiert. Dieses Konzept wird anschließend in einer Fallstudie praktisch umgesetzt und anschließend diskutiert.

Das zweite Konzept beschreibt ein Vorgehen zur Identifizierung, Analyse und Auswertung von Fehlern, die beim Nachladen und Einbinden von MFs auftreten können. Das Ziel des Konzeptes ist es, eine Übersicht über alle Fehler zu erhalten. Anschließend soll mithilfe dieser Übersicht der Aufwand zur Implementierung der Fehlerbehandlung reduziert werden. Das zweite Konzept wird ebenfalls in einer Fallstudie praktisch umgesetzt und diskutiert.

1.3 ABGRENZUNG

Die MF-Architektur ergibt sich aus einer Reihe von Entscheidungen, die getroffen werden (siehe Abschnitt 3.1). Um den Umfang einzuschränken, wird sich die Arbeit bei den Zusammensetzungsstrategien (Composing) auf das Client-Side-Composing (siehe Unterabschnitt 2.1.2) beschränken.

In den Büchern *The Art of Micro-Frontends* und *Micro-Frontends in Action* wird jeweils die Umsetzung der MF-Architektur mit Server-Side-Composing beschrieben [RS21, Kapitel 7][Gee20, Kapitel 4]. Die Autoren zeigen die Umsetzung an einem Open-Source-Projekt anhand eines Beispiels. Dies ist auf GitHub[2] einsehbar.

Das Buch *The Art of MFs* [RS21, Kapitel 8] beschreibt außerdem am selben Projekt das Edge-Side-Composing, welches die dritte Composing-Strategie ist.

1.4 GLIEDERUNG

Im zweiten Kapitel werden zunächst technische Grundlagen zu allgemeinen Aspekten der Webentwicklung sowie zu Technologien für die Umsetzung von MF-Architekturen zusammengefasst. Der darauf folgende Abschnitt erläutert

2 https://github.com/neuland/micro-frontends zuletzt eingesehen am 06.08.2023

den Stand der Forschung anhand wissenschaftlicher Publikationen und zeigt die Forschungslücke auf, die in dieser Arbeit behandelt wird.

Kapitel drei beschreibt die Konzepte zur Überführung in eine MF-Architektur sowie deren Fehlerbehandlung beim Nachladen, Einbinden und Aushängen der MFs. Im vierten Kapitel werden die Konzepte praktisch umgesetzt. Auf dieser Grundlage findet in Kapitel fünf die Diskussion und Evaluation der beiden Konzepte statt. Den Abschluss bilden die Zusammenfassung und das Fazit in Kapitel sechs.

2 GRUNDLAGEN UND STAND DER WISSENSCHAFT

Das Kapitel führt in die Fachterminologie ein. Es werden zunächst allgemeine Technologien in der Webentwicklung und anschließend solche, die in MF-Architekturen verwendet werden, vorgestellt. Der zweite Abschnitt beschreibt den Stand der Wissenschaft zum Thema MF-Architekturen und zeigt ein Themenfeld auf, das bisher in der Wissenschaft weniger Beachtung gefunden hat.

2.1 TECHNISCHE GRUNDLAGEN

Die ersten Aspekte befassen sich mit für die Arbeit relevanten Techniken der Webentwicklung. Anschließend werden Technologien, die bei einer MF-Architektur eingesetzt werden können und für das Konzept relevant sind, erklärt.

2.1.1 *Grundlegende Aspekte der Webentwicklung*

Neben den essenziellen Bestandteilen einer Webseite (HTML, CSS und JavaScript) werden heutzutage auch weitere Sprachen in der Webentwicklung eingesetzt, wie zum Beispiel TypeScript und Syntactically Awesome Style Sheets (SASS). Diese werden oft in Kombination mit verschiedenen Frameworks verwendet [TM22]. Dieser Abschnitt gibt einen Einblick in weitere Technologien und Frameworks, die im Frontend bei der Entwicklung von Webseiten verwendet werden.

In der modernen Webentwicklung kommen Frameworks wie React[1], Angular[2] oder Vue[3] zum Einsatz [Vai23]. Diese werden im Folgenden als JavaScript-Frameworks bezeichnet, um sie von anderen Frameworks, die zur Implementierung von MF-Architekturen verwendet werden, zu unterscheiden. Sie abstrahieren viele wiederkehrende Programmieraufgaben und erleichtern somit das effiziente Entwickeln von Webseiten im Frontend.

Die JavaScript-Frameworks zwingen Entwickler dazu, die Webseite in Komponenten zu entwickeln. Die Komponenten bestehen aus HTML, CSS und JavaScript. Sie sind isoliert, sodass sie unabhängig voneinander sind und sich nicht gegenseitig beeinflussen können. Darüber hinaus hat jedes JavaScript-Framework spezifische Bezeichnungen und Vorgehensweisen, die die Entwicklung von Webseiten weiter vereinfachen [Gee+22].

In Angular werden beispielsweise logisch zusammenhängende Komponenten in einem Modul zusammengefasst. Module können weitere Angular-Klassen wie Services enthalten. Services enthalten die Geschäftslogik des Frontends und sind von den darstellenden Elementen einer Komponente (Präsentations-

1 *https://react.dev/* zuletzt eingesehen am 01.08.2023
2 *https://angular.io/* zuletzt eingesehen am 01.08.2023
3 *https://vuejs.org/* zuletzt eingesehen am 01.08.2023

schicht) gekapselt. Durch Dependency Injection werden Instanzen der Service-Klassen in den Komponenten zur Verfügung gestellt. Die Dependency Injection ist ebenfalls Teil des Angular-Frameworks [Gee+22].

Mit diesen und weiteren Ansätzen gewährleisten JavaScript-Frameworks wie Angular viele Faktoren der Softwarequalität. Dazu gehören Wiederverwendbarkeit, Portabilität, Wartbarkeit und weitere Faktoren (siehe ISO/IEC 25010[4]).

Viele JavaScript-Frameworks bieten optional weitere Funktionalitäten zur Vereinfachung einzelner Programmieraufgaben an. Hierzu zählt beispielsweise das clientseitige Routing innerhalb eines JavaScript-Frameworks. Routing beschreibt den Wechsel zu einer Unterseite der Webseite. In klassischen Webseiten wird der Code für die Unterseite vom Server angefordert. Dies hat zur Folge, dass der Client auf die Anfrage und Antwort des Servers warten muss, bis der neue Inhalt angezeigt wird. In JavaScript-Frameworks ist der Code für die neu angeforderte Unterseite bereits als Komponente vorhanden, wenn die Webseite initial geladen wird. Die Komponente wird lediglich ausgetauscht, sodass kein zusätzlicher Code vom Server nachgeladen werden muss. Das Routing mittels JavaScript-Frameworks wird standardmäßig bei Single Page Applications (SPA)s angewendet.

Ein weiterer Ansatz in der modernen Webentwicklung ist das sogenannte State Management. State Management bezieht sich auf die Verwaltung von Daten beziehungsweise Zuständen der Webseite. Ein Beispiel ist ein Textfeld zur Eingabe eines neuen Passworts. Die Bedingungen, die für das Passwort erfüllt sein müssen, werden direkt im State Management abgebildet. Wenn das eingegebene Passwort diese Bedingungen erfüllt, ändert sich entsprechend der Zustand im State Management. Somit abstrahiert das State Management den Zustand des Systems und bietet gleichzeitig einen sogenannten Single Point of Truth[5]. JavaScript-Frameworks bieten hierfür häufig vorgefertigte Lösungen wie Redux[6], NgRx[7] oder Pinia[8] an. Diese werden als State Management Systems bezeichnet.

In der Webentwicklung verbessern Sprachen wie TypeScript oder SASS die Programmierung von komplexen Anwendungen. Diese Sprachen müssen in JavaScript und CSS übersetzt werden, sodass sie von den Browsern interpretiert werden können. Dieser Vorgang wird Transpilieren genannt. Das Transpilieren wird von sogenannten Bundlern durchgeführt. Bundler bündeln, optimieren und transpilieren Code zur Kompilierungszeit. Der aufbereitete Code wird dann von einem Webserver ausgeliefert, sodass dieser ihn bei Bedarf an die Clients verteilen kann. Bundler vereinfachen die Entwicklung erheblich und verbessern gleichzeitig die Performance einer Webseite.

4 *https://www.iso.org/standard/81913.html* zuletzt eingesehen am 01.08.2023
5 Dies beschreibt eine verlässliche Datenquelle im System. Mehr dazu *https://de.wikipedia.org/wiki/Single_Point_of_Truth* zuletzt eingesehen am 29.08.2023
6 https://redux.js.org/ zuletzt eingesehen am 29.08.2023
7 https://ngrx.io/ zuletzt eingesehen am 29.08.2023
8 https://pinia.vuejs.org/ zuletzt eingesehen am 29.08.2023

Ein beliebter Bundler, der in vielen JavaScript-Frameworks verwendet wird, ist Webpack[9]. Webpack nutzt Plugins, um weitere Funktionalitäten wie zum Beispiel weitere Programmiersprachen zu unterstützen. Mithilfe des Plugins "Module Federation"[10] können MFs einfach nachgeladen werden.

User Experience (UX)

Die User Experience (UX) beschreibt die Interaktion des Clients mit einem Programm und sollte so benutzerfreundlich wie möglich gestaltet werden. Dies beinhaltet beispielsweise die Bereitstellung barrierefreier Inhalte, um eine gute Nutzerfreundlichkeit zu gewährleisten. Eine umfassende Liste aller Aspekte einer nutzerfreundlichen Oberfläche ist auf der Website ux-ui-design.de[11] verfügbar.

Ein wichtiger Aspekt für diese Arbeit ist das einheitliche Design der Webseite. Elemente wie beispielsweise Buttons, die dieselbe Funktion haben, sollten überall gleich aussehen, um den Client nicht zu verwirren. Ein weiterer Aspekt ist die Fehlerbehandlung. Eine gute User Experience (UX) zeichnet sich dadurch aus, dass unvermeidbare Fehler gut behandelt werden. Dazu gehört eine benutzerfreundliche Fehlerbeschreibung sowie Hinweise zur Behebung. Automatische Fehlerkorrekturmechanismen sollten implementiert werden, wenn möglich. Ein Beispiel hierfür ist die automatische Wiederherstellung einer unterbrochenen Verbindung zum Server.

Lazy und Eager Loading

Die Ladezeit einer Webseite nach einer Client-Anfrage ist entscheidend für die UX. Webseiten sollten schnell geladen werden, da längere Ladezeiten die Zufriedenheit der Benutzer*innen beeinträchtigen. Laut Amazon verringert sich der Umsatz um ein Prozent für jede zusätzliche 100 Millisekunden Latenz [RS21, Kapitel 2]. Die Ladezeit einer Webseite hängt hauptsächlich von der Qualität der Internetverbindung ab. Daher sollten die Ladezeiten so gut wie möglich optimiert werden, um auch Geräten mit geringer Netzwerkgeschwindigkeit angemessene Ladezeiten zu bieten. Um dies zu erreichen, gibt es unter anderem das Konzept des Lazy Loadings. Beim Lazy Loading werden Ressourcen erst geladen, wenn sie benötigt werden. Das Gegenstück dazu ist das Eager Loading, bei dem bewusst sofort alle Ressourcen auf der Clientseite zur Verfügung gestellt werden. In der Webentwicklung wird Lazy Loading in vielen Bereichen eingesetzt. Mit dieser Technik können JavaScript-Code beziehungsweise MFs auf Anfrage nachgeladen werden.

9 *https://webpack.js.org/* zuletzt eingesehen am 14.08.2023
10 *https://webpack.js.org/concepts/module-federation/* zuletzt eingesehen am 29.08.2023
11 https://ux-ui-design.de/10-prinzipien-fuer-usability-und-user-interface-design/ zuletzt eingesehen am 02.08.2023

2.1.2 *Technologien in der Micro-Frontend-Architektur*

Eine MF-Architektur ist in mehrere Teilaspekte unterteilt. Jeder Teilaspekt erfüllt eine bestimmte Aufgabe oder Herausforderung, die in der Architektur auftritt. Nach [TM22] sind die Aufgaben wie folgt unterteilt:

- Zerlegung
- Zusammensetzungsstrategien (Composing)
- Routing
- Kommunikation
- Implementierung

Um diese Aufgaben zu erfüllen, gibt es unterschiedliche Technologien. Jede Technologie hat Vor- und Nachteile, die gegeneinander abgewogen werden müssen. Im Folgenden werden diese Technologien vorgestellt.

Zerlegung

Wie bereits in Abschnitt 1 beschrieben, besteht eine MF-Architektur aus vielen einzelnen MFs. Diese sind vollständig isoliert und unabhängig voneinander. Die MFs dürfen daher keine Abhängigkeiten zu anderen MFs besitzen (keine Kopplung) [Gee20, Kapitel 6]. Gleichzeitig sollte innerhalb eines MFs jeweils ein logisches Modul abgebildet werden. Der Zusammenhalt innerhalb des Moduls sollte hoch sein (hohe Kohäsion). Diese Anforderungen an Module und deren Grenzen werden als Domänen bezeichnet [Mez21, Kapitel 10].

Die Problemstellung bei der Zerlegung besteht darin, Optionen zu finden, die Domänen beschreiben. Eine Domäne bildet ein MF [PMT21]. Die zwei Optionen zur Lösung des Problems sind die vertikale und horizontale Zerlegung (siehe Abbildung 2.1).

Vertikale Zerlegung
Bei der vertikalen Zerlegung werden die Domänen anhand der visuellen Struktur der Webseite definiert. Im einfachsten Fall bedeutet dies, dass jede sichtbare Unterseite einer Webseite ein eigenes MF besitzt [PMT21].

Bei einer vertikalen Zerlegung ist nur ein MF gleichzeitig aktiv. Somit müssen weniger Informationen als bei einer horizontalen Zerlegung zwischen den MFs ausgetauscht werden. Allerdings kann es aufgrund der strikten Isolierung dazu kommen, dass Code durch die mehrfache Verwendung von Komponenten doppelt geschrieben wird [Kro21; Jac19].

Horizontale Zerlegung
Die horizontale Zerlegung definiert Domänen anhand einzelner Komponenten. Klassische Komponenten sind hier beispielsweise die Navigationsleiste oder der Footer[12]. Bei dieser Art der Aufteilung sind mehrere Domänen beziehungsweise MFs gleichzeitig aktiv [PMT21].

Eine horizontale Zerlegung bietet sich für große Webseiten mit viel statischem Inhalt sowie für Webseiten mit wenigen Unterseiten, aber dafür mehr Funktionalität innerhalb einer Unterseite an [RS21; TM22, Kapitel 5]. Allerdings müssen die MFs mehr als bei der vertikalen Zerlegung miteinander kommunizieren, um Informationen und Ereignisse auszutauschen [PMT21].

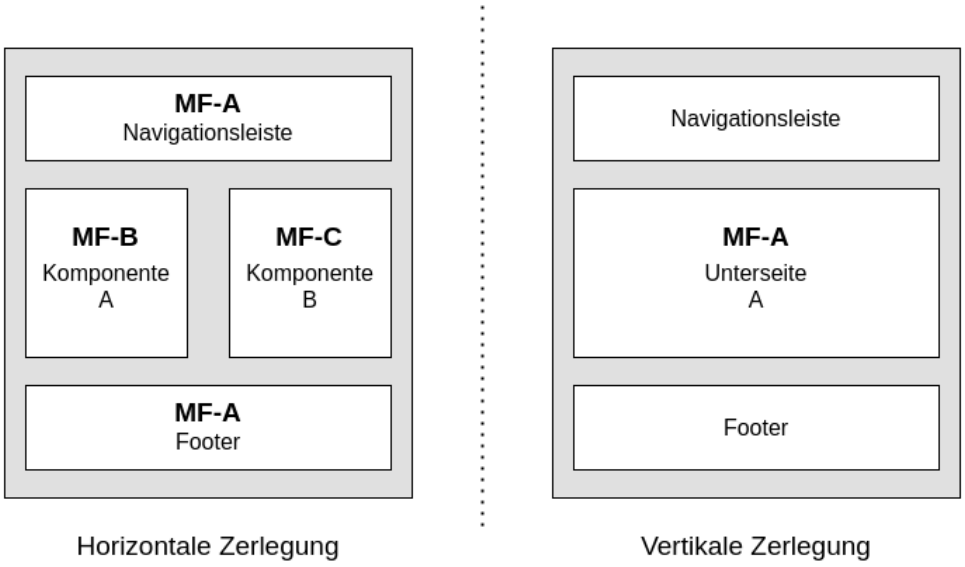


Abbildung 2.1: Die zwei Optionen zur Zerlegung einer Website in MFs (MF)

Zusammensetzungsstrategien (Composing)

Zerlegte MFs müssen wieder durch sogenanntes Composing zusammengesetzt werden. Das Composing beschreibt den Ort, an dem die Zusammensetzung erfolgt. Die Orte sind auf dem Server (Server-Side), in einem Content Delivery Network (CDN) (Edge-Side) oder auf dem Client (Client-Side) (siehe Abbildung 2.2).

Server-Side-Composing
Das Composing wird zur Laufzeit oder Kompilierungszeit auf dem Server durchgeführt. Der gerenderte Code besteht aus HTML, CSS und JavaScript. Fragt ein Client ein MF an, werden genau diese drei Inhalte vom Server zurückgesendet.

In der ursprünglichen Form wurden zur Umsetzung Server-Side-Includes[13] verwendet. Beide Technologien rendern auf Anfrage eines Clients Code und

12 Fußzeile einer Webseite. Sie ist typischerweise über alle Unterseiten gleich.
13 *https://www.w3.org/Jigsaw/Doc/User/SSI.html* zuletzt eingesehen am 27.07.2023

schicken diesen dem anfragenden Client wieder zurück. Diese sind heute abgelöst durch moderne Server-Side-Rendering Frameworks wie Gatsby[14], Nuxt[15] oder Angular Universal[16]. Diese helfen bei der Entwicklung von komplexen Webseiten, die auf der Serverseite gerendert werden. Ein großer Vorteil von Server-Side-Rendering ist die kurze Ladezeit, da der Server über mehr Rechenkapazität verfügt [PMT21]. Darüber hinaus kann der gerenderte Inhalt auf dem Server zwischengespeichert werden [RS21, Kapitel 7].

Des Weiteren verbessert sich die Rangfolge in Suchmaschinen (Search Engine Optimization (SEO)-Index) [PMT21]. Sogenannte Crawler analysieren Webseiten, die öffentlich zugänglich und für die Bewertung zugelassen sind. Bereits gerenderten Inhalt kann der Crawler deutlich besser analysieren, wodurch die Wahrscheinlichkeit für eine gute Bewertung steigt [RS21, Kapitel 4].

Server-Side-Composing eignet sich nicht, wenn viele Inhalte der Webseite nur für einen Client oder eine kleine Gruppe an Clients zutreffen. Das Zwischenspeichern ist nur für Inhalte möglich, die für alle Clients gleich sind. Clientspezifischer Inhalt muss bei jeder Anfrage neu gerendert werden. Des Weiteren muss aufgrund steigender Komplexität und/oder zunehmender Zahl an Clients langfristig eine Strategie zur Skalierung des Servers überlegt werden. Eine mögliche Lösung ist das Zwischenspeichern von Ergebnissen in einem CDN, siehe Abbildung 2.2 [PMT21].

Edge-Side-Composing

Eine Zwischenlösung zwischen Client- und Server-Side-Composing ist das Zusammensetzen der MFs in einem CDN. Ein CDN ist ein verteiltes Netzwerk, welches Inhalte auf verschiedenen Servern zwischenspeichert und an anfragende Clients sendet. Sowohl durch das Zwischenspeichern als auch durch die geografische Verteilung der Server wird die Antwortzeit deutlich reduziert [Pen04]. Die MFs werden mithilfe von Edge-Side-Includes (ESI)[17] zusammengesetzt [TM22]. ESI nutzt eine Auszeichnungssprache, um in HTML die jeweiligen MFs durch ESI-Tags einzubinden. Die ESI-Tags werden im CDNs interpretiert und laden das jeweilige MF [Tsi+01].

Edge-Side-Composing verfolgt das Ziel, ähnliche Vorteile wie beim Server-Side-Composing zu nutzen, und gleichzeitig von der Lastverteilung eines verteilten Netzwerks CDN zu profitieren [Kro21; Tsi+01]. Es bleibt aber weiterhin das Problem, dass Inhalte, die nur für einen Client oder eine kleine Gruppe an Clients bestimmt sind, nicht zwischengespeichert werden können.

Client-Side-Composing

Enthält eine Webseite viele für einen Client spezifische Inhalte oder Inhalte für eine kleine Gruppe, eignet sich das Client-Side-Composing. Bei diesem Composing wird zunächst eine übergeordnete Hauptapplikation (Shell) geladen. Die Shell verwaltet alle MFs, indem sie diese nachlädt, einbindet und wieder aushängt. Nachdem die Shell geladen ist, wird das zu der Route passende MF

14 *https://www.gatsbyjs.com/* zuletzt eingesehen am 26.07.2023
15 *https://v2.nuxt.com/* zuletzt eingesehen am 27.07.2023
16 *https://angular.io/guide/universal* zuletzt eingesehen am 27.07.2023
17 *https://www.w3.org/TR/esi-lang/* zuletzt eingesehen am 26.07.2023

nachgeladen. Ein Vorteil dieses Composings ist, dass auch für einen Client spezifische Inhalte zwischengespeichert werden können. Die Shell kann mithilfe eines JavaScript-Frameworks entwickelt werden, wodurch der Programmieraufwand deutlich reduziert wird. Darüber hinaus gibt es einige Frameworks und Technologien, die die Aufgaben der Shell abstrahieren und vereinfachen (siehe Tabelle A.1).

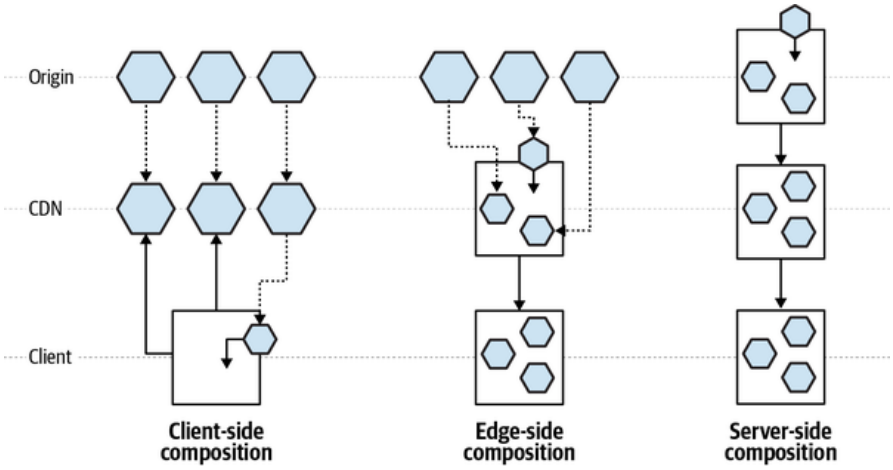


Abbildung 2.2: Composing-Optionen (nach [Mez21, Kapitel 3])

Routing

Das Routing entscheidet wann welches MF angezeigt wird. Es wird zwischen Hard- und Softlinks unterschieden [Gee20, Kapitel 7]. Hardlinks laden eine Seite mit dem gesamten HTML-Code komplett neu (Server- und Edge-Side-Composing), wohingegen Softlinks nur Teile des HTML-Codes austauschen (Client-Side-Composing) [Gee20, Kapitel 7]. Da die MFs verschiedene Uniform Resource Identifier (URI)s[18] besitzen, müssen diese im Routing beachtet und implementiert werden. Aufgrund der Abgrenzung in dieser Arbeit (siehe Abschnitt 1.3), wird das Routing für Server- und Edge-Side-Composing nur oberflächlich als eine allgemeine Option zusammengefasst.

Clientseitiges Routing

Bei einem Client-Side-Composing wird das Routing auf der Clientseite durchgeführt. Das Ziel von clientseitigem Routing in einer MF-Architektur ist es, ausschließlich Softlinks zu verwenden. Durch das unbemerkte Austauschen des HTML-Codes wird dem Client eine zusammenhängende Webseite suggeriert. Er bemerkt nicht, dass es sich hierbei um MFs handelt.

Verantwortlich für das globale Routing ist die Shell. Das globale Routing beschreibt das Navigieren zwischen verschiedenen MFs (Top-Level-Routing). Diese Routen werden in der Shell statisch eingetragen. Wie in Abbildung 2.3 dargestellt ist, können innerhalb eines MFs ebenfalls mehrere Routen existieren.

18 Uniform Resource Locator (URL) beschreibt die Adresse einer Webseite wohingegen URI die URL inklusive der angefragten Ressource (Route) meint. Die URL ist somit ein Teil der URI.

Für dieses Routing ist das jeweilige MF selbst verantwortlich (Second-Level-Routing). Um doppelte Routen zu vermeiden, sollten alle Routen als Präfix das Top-Level-Routing enthalten. Somit ergibt sich folgender Syntax: *[URL]/<top-level-routing>/<second-level-routing>* [Gee20, Kapitel 7]. Für den in Abbildung 2.3 gezeigten Aufbau ergibt sich folgende URI:

```
1 https://my-server-address.de/dashboard/monitoring
```

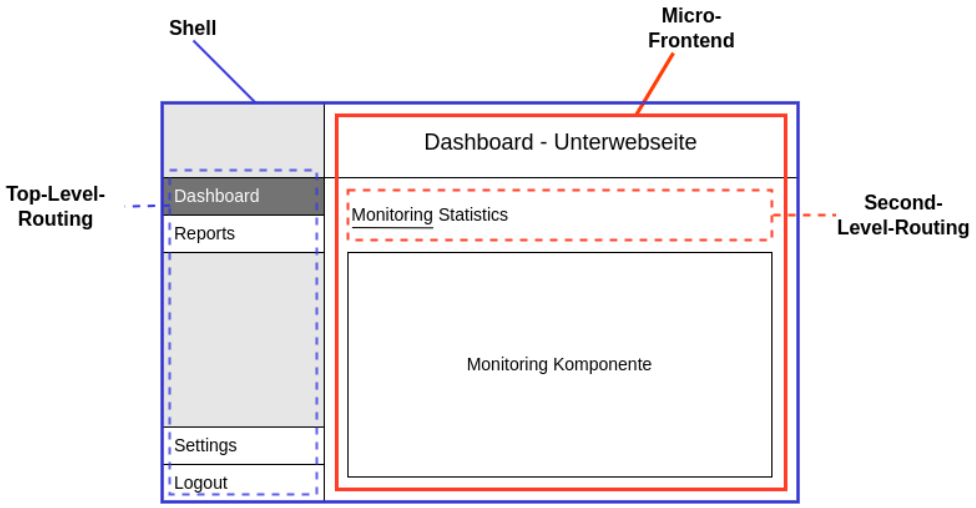


Abbildung 2.3: Website mit Shell verantwortlichen Routen (Top-Level-Routing) und MF verantwortlichen Routen (Second-Level-Routing)

Serverseitiges Routing

Serverseitiges Routing in einer MF-Architektur unterscheidet sich kaum von herkömmlichem serverseitigem Routing. Der Server bekommt die Anfrage zu einem bestimmten MF. Serverseitig wird das MF durch Server-Side-Rendering gerendert und an den anfragenden Client zurückgegeben. Anders als bei clientseitigem Routing werden nur Hardlinks verwendet [TM22].

Kommunikation

Egal, ob bei horizontaler oder vertikaler Zerlegung, sind MFs immer isolierte Komponenten. Nichtsdestotrotz müssen diese miteinander kommunizieren, um Ereignisse und Informationen auszutauschen. Beispielsweise muss auf einer E-Commerce-Webseite, auf der in einer Unterseite Details zu einem Produkt angezeigt werden, diese Unterseite wissen, um welches Produkt es sich handelt. Um den Informationsaustausch zur Detailseite umzusetzen, gibt es mehrere Optionen [TM22] [Gee20, Kapitel 6].

Query Strings

Mithilfe von Query Strings können Informationen an die URI angehängt werden. Sie sind eine einfache und gängige Methode, um eine angefragte Ressource auf dem angefragten Server genauer zu beschreiben. Die Beschreibung erfolgt über eine eindeutige Referenz (zum Beispiel eine ID oder Hash) [Mez21, Kapitel 3]. In dem oben genannten Beispiel kann die URI wie folgt aussehen:

`https://my-server-address.de/product/42`. Das MF, welches für die Details des Produktes zuständig ist, fragt die Informationen zu dem Produkt mit der ID 42 im Backend an.

Mit Query Strings ist der Informationsaustausch NUR während dem Seitenaufruf durch die URI möglich. Sollen beispielsweise bei der vertikalen Zerlegung, bei der mehrere MF gleichzeitig aktiv sind, kontinuierlich Informationen ausgetauscht werden, eignen sich Query Strings nicht. Sie sind allerdings eine gute und leicht zu implementierende Lösung, wenn es sich um die Beschreibung einer Ressource handelt, die den Hauptteil des MF beziehungsweise der Unterseite ausmacht.

Storage

Größere Datenmengen können mithilfe des Storage in ein anderes MF übertragen werden. Der Storage ist ein lokaler Schlüssel-Werte-Speicher im Browser. Typischerweise werden dort Zugangstoken (JSON Web Token (JWT)) gespeichert. Der Storage ist eine einfache Möglichkeit, Daten auch über die Lebenszeit der Webseite hinaus auf dem Client zu persistieren. Durch die Same-Origin-Policy[19] eines Browsers besitzt jede URL einen eigenen, isolierten Storage. Alle MFs können gleichermaßen auf den isolierten Bereich zugreifen. Wiederum haben andere Webseiten mit einer anderen URL einen eigenen isolierten Bereich. Somit gibt es keine Seiteneffekte zu anderen Webseiten über den Storage.

In dem genannten Beispiel wird die Produkt ID 42 in dem Storage mit einem entsprechenden Schlüssel gespeichert:

```
1 {
2     "productId": 42,
3 }
```

Die Kommunikation über den Storage sollte mit Bedacht gewählt werden. Jedes MF und der Client können den Storage manipulieren, sodass unbedingt die Anwesenheit und Integrität der Daten überprüft werden müssen. Des Weiteren beschreibt Geers [Gee20] in Kapitel 6, dass der Storage nicht benutzt werden soll, um große Informationen auszutauschen, die über eine API gesendet wurden, um die Anzahl der API-Aufrufe zu reduzieren. Stattdessen soll jedes MF einen separaten API-Aufruf an das Backend senden.

Custom Events

Während die zwei vorherigen Optionen für den Austausch von Informationen zuständig sind, sind Custom Events für die Übertragung von Ereignissen zuständig. Die Abbildungen 2.1 und 2.2 zeigen ein Beispiel, wie ein Custom Event ausgelöst und abgefangen wird. Wichtig ist, dass der EventHandler beziehungsweise die Funktion `setEventHandler()` zuerst aufgerufen werden muss, bevor das Event ausgelöst wird. Entsteht in einem MF ein Ereignis, löst das MF dieses Ereignis durch ein Custom Event aus. Alle MFs, die sich auf dieses

19 *https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy* zuletzt eingesehen am 27.07.2023

Event registriert haben, reagieren entsprechend und führen den Code aus, der in Zeile 3 des JavaScript-Codes 2.1 enthalten ist.

Custom Events sind vor allem für die horizontale Zerlegung wichtig, da dort mehrere MFs gleichzeitig aktiv sind, in denen Ereignisse auftreten können [TM22]. Bei einer vertikalen Zerlegung können ebenfalls Ereignisse zwischen Shell und MF auftreten.

```
1 function setEventHandler() {
2     document.addEventListener("event_name", (event) => {
3         /** do something */
4     });
5 }
```

Listing 2.1: MF A: registriert EventHandler, der auf das Event “event_name” hört.

```
1 function triggerEvent() {
2     const event = new CustomEvent("event_name", { details: "foo" });
3     document.dispatchEvent(event);
4 }
```

Listing 2.2: MF B: löst das Event “event_name” aus und überträgt “foo” als Information.

Vor- und Nachteile der Micro-Frontend-Architektur

Der größte Vorteil ist nach den Ergebnissen der Meta-Analyse von [PMT21] die Verwendung unterschiedlicher JavaScript-Frameworks. Dadurch kann jedes MF spezifische Technologien für seine Anforderungen verwenden.

Die Isolierung bietet weitere Vorteile. Sie garantiert, dass keine Kopplung zwischen MFs entstehen kann. Dies steigert Aspekte wie die Wartbarkeit des Codes. Die Entwicklung ist ebenfalls effizienter, da Entwickler sich nur auf ihr MF fokussieren können und nur bedingt über Kommunikationsschnittstellen mit Seiteneffekten beschäftigen müssen [TM22].

Durch die Modularisierung lassen sich die einzelnen MFs effizient laden. Dadurch können sich Metriken wie der First Contentful Paint (Ladezeit bis der Renderingprozess beginnt) und der Largest Contentful Paint (Ladezeit des größten sichtbaren Inhalts im sichtbaren Bereich) verbessern. Dies hat einen positiven Einfluss auf die UX als auch auf die SEO. Die SEO ist für Suchmaschinen wichtig. Bei einer guten SEO wird die Webseite weiter oben in der Rangfolge der Treffer angezeigt. Jedoch ist die SEO nur relevant, falls die Webseite öffentlich über Suchmaschinen auffindbar sein soll. Dies ist besonders für E-Commerce-Webseiten wichtig [Gee20, Kapitel 9].

Ein weiterer relevanter Aspekt ist die Analyse der Webseite durch A/B-Tests. Damit werden zwei verschiedene Versionen der Webseite parallel an unterschiedliche Clients ausgeliefert. Durch die Analyse dieser Clients können Rückschlüsse gezogen werden, wie sich die verschiedenen Versionen der Webseite verhalten. A/B-Tests lassen sich mit MFs sehr gut durchführen, da einzelne MFs einfach ausgetauscht werden können [PMT21].

Neben den Vorteilen der MF-Architektur gibt es auch Nachteile, die im Folgenden beschrieben werden. Die Implementierung der MF-Architektur ist im Vergleich zu einem Monolithen aufwendiger. Mechanismen zum Nachladen und Einbinden sowie die Fehlerbehandlung (siehe vorherigen Abschnitt 2.3) müssen zusätzlich entwickelt werden. Dadurch steigt die Abstraktion in der gesamten Anwendung. Dies kann zu einer langsameren Entwicklung und potenziellen Programmierfehlern führen. Auch müssen die Entwickler sich in diese Thematik neu einarbeiten [TM22; PMT21; Jac19].

Ein weiterer Nachteil entsteht aus der Eigenschaft, dass MFs isoliert sind. Dadurch müssen weitere zusätzliche Mechanismen entwickelt werden, wie diese untereinander Informationen und Ereignisse austauschen können. Dies erzeugt weiteren Code mit potentiellen Fehlern [TM22].

Durch die Isolierung muss Code, der von mehreren MFs verwendet wird, ausgelagert werden. Es müssen Prozesse geschaffen werden, um sicherzustellen, dass Änderungen in dem ausgelagerten Code zu keinen unerwarteten Verhalten in den MF führt. Diese Prozesse können je nach Umfang der Webseite und des Entwicklerteams langwierig sein, wodurch die Entwicklung verlangsamt werden kann.

Des Weiteren kann es bei manchen JavaScript-Frameworks in Kombination mit der MF-Architektur zu Problemen kommen. In der Publikation von Lando [Lan22] wird gezeigt, dass die Umsetzung eines Client-Side-Composings mit dem JavaScript-Frameworks Ember.js[20] nur mit deutlichen Schwierigkeiten umsetzbar ist [Lan22]. Das Problem bei diesem Anwendungsfall ist, dass die Micro-Frontends nicht mit der Shell zusammen geladen werden können, da die Routen nicht ordnungsgemäß aufgelöst werden. Als Lösung wurde der Vorteil der Technologieunabhängigkeit genutzt und Angular als alternatives JavaScript-Framework in der Shell verwendet [Lan22]. Allerdings wird bei dieser Lösung zusätzlicher Code geladen, wodurch die Latenz der Webseite steigt. Zusätzlich wird eine weitere Technologie in die Webanwendung hinzugefügt, wodurch die Anforderung an ein breitgefächertes Wissen bei den Entwicklern steigt.

20 https://emberjs.com/ zuletzt eingesehen am 04.09.2023

Implementierung

Auf Basis der ersten vier Entscheidungen muss nachfolgend die Entscheidung getroffen werden, wie die MF-Architektur konkret implementiert werden soll. Während sich das Composing aus Unterabschnitt 2.1.2 mit der Frage beschäftigt, wo MFs zusammengesetzt werden, beschäftigt sich die Implementierung damit, wie sie zusammengesetzt werden können. Es geht hierbei darum, eine technische Lösung zu finden, wie MFs geladen und eingebunden werden können. Diese Aufgaben werden durch Technologien beziehungsweise Frameworks abstrahiert und vereinfacht. Kroiß [Kro21] zeigt in seiner Dissertation eine Tabelle, in der die Frameworks aufgelistet sind (siehe Anhang A.1). Die Tabelle enthält folgende relevante Spalten:

- Für welches Composing ist das Framework geeignet?
- Wie umfangreich, zugänglich und vollständig ist die Dokumentation des Frameworks?
- Ist das Framework auf ein JavaScript-Framework beschränkt?

Bei der Auswahl der Frameworks werden diejenigen ausgesucht, die technisch ein weites Spektrum von Anwendungsfällen abdecken. Wie in Abschnitt 1.3 beschrieben, thematisiert diese Arbeit nur clientseitiges Composing. Dementsprechend werden auch nur Frameworks für clientseitige Implementierungen behandelt. Ausgehend von der Tabelle werden einige Frameworks im Folgenden näher beschrieben:

Web Components

Web Components[21] ist eine Technologie, die von allen gängigen Browsern nativ unterstützt wird. Der Kerngedanke von Web Components besteht darin, die HTML-Standardelemente um weitere, sogenannte Custom Elements[22] zu erweitern. Die Erweiterung erfolgt gemäß den Vererbungsregeln der objektorientierten Programmierung.

```
1 class MyCustomElement extends HTMLElement {
2     /** eigener Code */
3 }
```

Ein Custom Element kann als ein MF definiert werden. Die Custom Elements werden durch JavaScript dem DOM hinzugefügt.

Das DOM ist eine Datenstruktur, die baumartig aufgebaut ist. Darin enthalten sind sogenannte Nodes, die HTML-Elemente beinhalten. Die Nodes können durch Attribute wie zum Beispiel CSS-Selektoren beschrieben werden. JavaScript kann den Nodes und deren Inhalt manipulieren.

Um die Manipulationen eines Custom Elements durch JavaScript anderer Custom Elements zu verhindern, kann ein sogenanntes Shadow-DOM[23] ver-

21 *https://developer.mozilla.org/en-US/docs/Web/API/Web_components* zuletzt eingesehen am 03.08.2023

22 *https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_custom_elements* zuletzt eingesehen am 03.08.2023

23 *https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM* zuletzt eingesehen am 03.08.2023

wendet werden. Hierzu wird das Custom Element nicht direkt in das DOM, sondern über ein Shadow-DOM eingebunden. Ein Shadow-DOM kapselt das darin befindliche HTML, CSS und JavaScript vom Rest des DOMs. Damit wird sichergestellt, dass der Code, der sich innerhalb einer Web Component befindet, nicht von Seiteneffekten in seiner Funktionsweise gestört wird. Durch die Verwendung von Shadow-DOMs wird das MF isoliert. Abbildung 2.3 zeigt eine beispielhafte Web Component, die isoliert in einem Shadow-DOM in das DOM eingehängt wird.

```
1  export default class FooWebComponent extends HTMLElement {
2    constructor() {
3      super();
4
5      // closed mode to forbid external javascript access
6      const shadow = this.attachShadow({ mode: 'closed'});
7
8      // create node and append it to the shadow-dom
9      const paragraph = document.createElement("p");
10     paragraph.appendChild(document.createTextNode("foo"));
11     shadow.appendChild(paragraph);
12   }
13 }
14
15 // append class to global customElements
16 customElements.define('web-component', MyCustomElement);
```

Listing 2.3: Beispielcode für eine durch Shadow-DOM isolierte Web Component

Das Nachladen, Einbinden und Aushängen dieser Custom Elements wird von der Shell übernommen. Das Einbinden und Aushängen wird durch entsprechende Manipulationen des DOMs durchgeführt. Ein ausführliches Beispiel ist in Anhang A.2 zu finden.

Web Components sind in Kombination mit Shadow-DOMs eine einfache Möglichkeit, MFs zu erstellen. Allerdings ist Web Components lediglich eine Technologie und definiert kein Vorgehen, wie MFs nachgeladen werden sollen. Für eine vollständige MF-Architektur müssen daher weitere Verfahren eingesetzt werden. In dem Beispiel in Anhang A.2 werden beispielsweise dynamische Imports[24] verwendet. Dynamische Imports laden JavaScript-Code zur Laufzeit nach. Es ist eine einfache, nativ in JavaScript umsetzbare Lösung. Allerdings ist der Umfang beschränkt. So besteht die Gefahr, dass Code, der in beiden MFs benötigt wird, doppelt geladen wird. Für dieses Problem bietet Module Federation eine betriebsbereite Lösung. Des Weiteren muss der nachgeladene Code auch eingebunden und wieder ausgehängt werden. Hierzu kann single-spa[25] eingesetzt werden.

Web Components eignen sich gut in einer horizontalen Zerlegung, bei der mehrere MFs gleichzeitig aktiv sind. Durch die native Unterstützung im Browser entsteht keine zusätzliche Nutzlast durch weiteren Code der Technologien.

24 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import zuletzt eingesehen am 17.08.2023

25 https://single-spa.js.org/ zuletzt eingesehen am 05.09.2023

Webpack Module Federation

Moduel Federation[26] ist eine Erweiterung (Plug-in) für den Bundler Webpack (nähere Informationen zu Bundlern siehe Unterabschnitt 2.1). Im Folgenden wird das Problem beschrieben, welches das Plug-in behebt.

Eine große Herausforderung der clientseitigen Implementierung von MF-Architekturen ist das mehrfache Nachladen von gleichem Code. MFs sind durch ihre Anforderung gekapselt und können auch unabhängig voneinander existieren. Durch diese Autonomie verwaltet jedes MF seine eigenen Abhängigkeiten zu Bibliotheken. Dadurch werden Bibliotheken, die in mehreren MFs verwendet werden, doppelt geladen und in jedem MF isoliert ausgeführt. Durch das mehrfache Nachladen und Ausführen der Bibliotheken entsteht zusätzliche Latenz, die zu einer deutlichen Beeinträchtigung der UX führen kann.

Der Kernaspekt von Module Federation ist die optimale Zerlegung der MFs zur Kompilierungszeit in sogenannte Chunks. Chunks können alle Ressourcen (Bilder, CSS, HTML, Sprachdateien, Fontdateien etc.) sein, sind aber typischerweise JavaScript-Code. Die Chunks teilen den Code so auf, dass dieser optimal nachgeladen werden kann. Die Informationen, welche Codeteile in welche Chunks aufgeteilt wurden und wie diese heißen, werden in eine Datei geschrieben. Diese heißt standardmäßig *remoteEntry.js* und wird im Folgenden als Manifest bezeichnet.

Um den Ablauf bildlich zu verdeutlichen, wird ein Szenario in Abbildung 2.4 beschrieben. In diesem Szenario ist bereits ein MF geladen (MF A). MF A benötigt die Ressourcen R1 und R2, die ebenfalls nachgeladen sind. Der Client fordert dann MF B an. Hierfür wird zunächst das Manifest von dem MF geladen. Dieses wird vom Client interpretiert und überprüft, welche weiteren Ressourcen benötigt werden und noch nicht dem Client zur Verfügung stehen. Da R2 schon geladen ist, muss lediglich R3 vom MF B geladen werden. Gleichzeitig wird der Code für das eigentliche MF B angefragt.

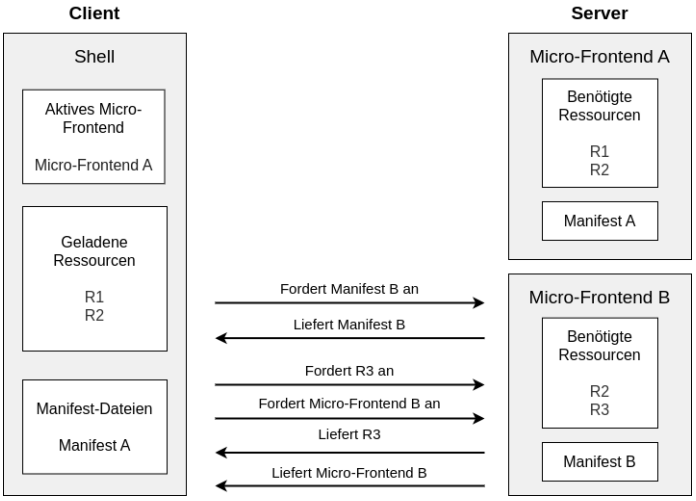


Abbildung 2.4: Szenario mit Module Federation Implementation: MF A sowie die Ressourcen R1 und R2 und das Manifest wurden bereits geladen. Für MF B muss nur noch der Code selbst sowie R3 geladen werden

26 *https://webpack.js.org/concepts/module-federation/* zuletzt eingesehen am 03.08.2023

Im Folgenden wird die Funktionsweise von Module Federation anhand der Konfigurationsdatei 2.4 erklärt.

Jedes MF besitzt eine Konfigurationsdatei. Diese wird zur Kompilierungszeit von Webpack interpretiert, um den Code des MFs optimal in Chunks aufzuteilen. Zunächst muss ein eindeutiger Name für das MF definiert werden, sodass die MFs unterschieden werden können. Anschließend wird der Dateiname des Manifests definiert. Die zwei wichtigen Attribute sind `exposes` und `shared`.

Im Attribut `exposes` werden alle Dateien und Pfade angegeben, die das MF veröffentlicht. Andere MFs können auf diese veröffentlichten Ressourcen zugreifen. Für die MF-Architektur ist es wichtig, einen Zugangspunkt zu veröffentlichen, über den das MF aufgerufen werden kann. In dem gezeigten Beispielcode wird ein Angular-Modul veröffentlicht.

Das zweite Attribut `shared` listet alle Ressourcen auf, die das MF selbst benötigt, um ausgeführt zu werden. Diese Ressourcen sind hauptsächlich externe NPM-Pakete, die in der `package.json` als “dependencies“ aufgelistet sind. Des Weiteren wird die Liste um Ressourcen ergänzt, die von anderen Quellen als der typischen `package.json` stammen. Für die MF-Architektur ist damit geteilter Code gemeint, der in mehreren MFs verwendet wird und daher ausgelagert wird. Dieses Verfahren wird im Konzept in Kapitel **??** als Shared Libraries beschrieben.

```
1  const deps = require('../../package.json').dependencies;
2
3  module.exports = {
4    plugins: [
5      new ModuleFederationPlugin({
6        name: 'home',
7        filename: 'remoteEntry.js',
8        exposes: {
9          HomeModule: './projects/mf-home/src/app/home/home.module.ts',
10       },
11       shared: {
12         ...deps,
13         'shared-lib-global': {
14             import: 'shared-lib-global',
15             requiredVersion: '1.2.0',
16         },
17         'shared-lib-statistics': {
18             import: 'shared-lib-statistics',
19             requiredVersion: '1.0.0',
20         },
21       },
22     }),
23   ],
24 };
```

Listing 2.4: Webpack Konfigurationsdatei mit Informationen über das Modul

Die optimierte gemeinsame Nutzung von Code ist ein Alleinstellungsmerkmal von Module Federation. Es eignet sich vor allem für große Webseiten, die viele Abhängigkeiten zu Bibliotheken besitzen. Durch das optimierte Laden der Abhängigkeiten werden Ladezeiten verringert, und die Performance der Webseite steigt. Es muss allerdings bedacht werden, dass dieses Plug-in nur in

Kombination mit Webpack 5.0 (veröffentlicht im Oktober 2020[KC20]) als Bundler funktioniert. Die restlichen Bibliotheken beziehungsweise JavaScript-Frameworks der Webseite müssen entsprechend mit dieser Webpack-Version kompatibel sein.

Single-spa

Während Web Components ein simpler Ansatz ist, Code in MFs zu kapseln, und Module Federation das Nachladen und Verwalten von Ressourcen optimiert, fokussiert sich single-spa[27] darauf, die MFs zu verwalten. Die Hauptaufgabe des Frameworks besteht darin, die jeweiligen MFs automatisch einzubinden beziehungsweise wieder auszuhängen. Single-spa ermöglicht einen einfachen Einsatz von mehreren JavaScript-Frameworks. Eine Liste aller unterstützten JavaScript-Frameworks und wie sie eingebunden werden, ist in der Dokumentation[28] zu finden. Die Registrierung neuer MFs erfolgt wie im Codebeispiel 2.5 gezeigt. Es ist zu sehen, dass der Code `./\ac{MF}s/foo.js` lazy geladen wird. Hier tritt das Problem auf, dass bereits geladene Ressourcen redundant geladen werden, was Module Federation löst. Es ist möglich, beziehungsweise von den Entwickler*innen empfohlen, diesen Prozess zu optimieren. Als Optimierung wird SystemJS[29] vorgeschlagen. Dem Einsatz von Module Federation sind sie auch nicht abgeneigt [DM20].

Ein Beispielcode, bei dem ein MF mithilfe von Single-spa implementiert wird, ist in Anhang A.3 zu finden.

```
1 singleSpa.registerApplication({
2   name: 'foo',
3   app: () => import('./\ac{MF}s/foo.js'),
4   activeWhen: '/foo',
5 })
```

Listing 2.5: Registrierung eines MFs in single-spa

2.2 STAND DER WISSENSCHAFT

Seit 2016 erscheinen stetig wissenschaftliche Artikel zum Thema MF-Architekturen [PMT21]. Die seitdem erschienenen Publikationen befassen sich einerseits mit allgemeinen Konzepten, andererseits mit konkreten Anwendungsfällen. Der erste Teil dieses Abschnitts befasst sich mit Publikationen, die ausschließlich allgemeine Konzepte beschreiben. Darauf folgt die Zusammenfassung von zwei Publikationen, die einen Anwendungsfall in eine MF-Architektur überführen.

27 https://single-spa.js.org/ zuletzt eingesehen am 17.08.2023
28 https://single-spa.js.org/docs/ecosystem zuletzt eingesehen am 18.08.2023
29 https://github.com/systemjs/systemjs zuletzt eingesehen am 18.08.2023

2.2.1 *Wissenschaftliche Literatur*

Eines der ersten wissenschaftlichen Publikationen wurde 2019 von Yang, Liu und Su [YLS19] veröffentlicht. Die Autoren beschrieben die Ideen von MFs und stellen erste Ansätze zum Composing, Routing und der Implementation von MFs vor. Diese Publikation ist eine von anderen [YLS19; TM22; Kro21; Büh+22] häufig zitierte Quelle.

Diese wissenschaftlichen Publikationen stellen weitere Technologien vor, die bei der Umsetzung einer MF-Architektur helfen (IFrames, Web Components, Module Federation). Ein Beispiel, das die Autoren Taibi und Mezzalira [TM22] beschreiben, ist das Plug-in Module Federation, welches im Mai 2020 zusammen mit Webpack Version 5.0 veröffentlicht wurde [TM22; Jac20]. Die Autoren stellen ebenfalls ein kurzes Vorgehen zur Umsetzung einer MF-Architektur vor. Sie gehen dabei auf die Punkte Zerlegung, Composing, Routing, Kommunikation und Implementierung ein.

Die Publikation [PMT21] präsentiert Ergebnisse einer Meta-Analyse über wissenschaftliche Artikel sowie graue Literatur, die sich mit dem Thema MF-Architektur beschäftigen. Die Analyse fasst Gründe zusammen, warum Artikel eine Überführung zu einer MF-Architektur empfehlen oder umgesetzt haben. Außerdem zeigt sie, was nach Ansicht der analysierten Population die größten Herausforderungen sowie der größte Nutzen der Überführung ist. Die Analyse wurde auf den folgenden Suchplattformen für wissenschaftliche Literatur durchgeführt: ACM digital Library, IEEEExplore Digital Library, Science Direct, Scopus, Google Scholar, Citeseer library, Inspec, Springer Link. Nach einer Sortierung blieben 43 Artikel für die Analyse übrig. Aus dieser Population geben 37 % eine steigende Komplexität als Grund für die Entscheidung der Überführung an. 51 % der Artikel sehen Vorteile durch die gewonnene Technologieunabhängigkeit. Nachteile sind laut 23 % der Population die Umsetzung einer konsistenten UX (Näheres siehe 2.1.2). Die Meta-Analyse wurde im April 2020 durchgeführt. Interessant ist zu erwähnen, dass von 43 Artikeln nur drei Artikel wissenschaftliche Publikationen sind [PMT21].

Allgemein sind die in den wissenschaftlichen Publikationen vorgestellten Ansätze Vorschläge beziehungsweise Empfehlungen, wie eine MF-Architektur umgesetzt werden kann. Allerdings ist die konkrete Umsetzung immer von vielen individuellen Faktoren eines Anwendungsfalls abhängig. Daher ist eine Beschreibung eines standardisierten Vorgehens zur Umsetzung nicht möglich [Gee20, Kapitel 2].

Daher befassen sich wissenschaftliche Publikationen mit konkreten Anwendungsfällen, bei denen eine MF-Architektur umgesetzt worden ist [Men+19; Kro21; NK22; Büh+22]. Diese Anwendungsfälle sind sehr komplexe Webanwendungen, bei denen individuelle MF-Architekturen umgesetzt werden. Ein Beispiel ist die Publikation von Bühler u. a. [Büh+22], bei der mithilfe von sogenannten Vereinbarungen die MFs auf definierte API-Endpunkte des Backends zugreifen, um Ressourcen zu laden. Das Besondere an diesem Anwendungsfall ist der Fokus auf mobile Endgeräte. Um diesen Fokus zu erfüllen, entschieden

sie sich für eine Progressive Web App (PWA)[30]. Eine PWA ist eine Webseite beziehungsweise Webapplikation, die vom Browser auf dem System des Clients installiert wird. Sie besteht ebenso wie herkömmliche Webseiten aus HTML, CSS und JavaScript. Durch die Installation der Webapplikation ist diese auch offline verfügbar. PWAs bieten zudem weitere Vorteile, wie den verbesserten Zugriff auf hardwarenahe Ressourcen. In der Publikation von Bühler u. a. [Büh+22] wird eine PWA in Kombination mit der MF-Architektur umgesetzt, sodass nur einige MFs als PWA implementiert werden. Dieser Ansatz vereint zwei moderne Lösungen.

Die Publikation von "Experiences on a Frameworkless Micro-Frontend Architecture in a Small Organization" [MTR23] beschreibt einen Anwendungsfall, der in eine MF-Architektur überführt wird. Der Anwendungsfall ist eine Webseite zur Verwaltung von Immobilien. Sie wird ohne die Verwendung eines JavaScript-Frameworks in eine MF-Architektur umgebaut. Zur Umsetzung werden Web Components verwendet. Die Autoren betonen, dass ohne den Einsatz von Frameworks Aspekte wie das Routing deutlich komplexer umzusetzen sind [MTR23].

Neben wissenschaftlichen Publikationen greifen auch Bücher das Thema auf. Mezzalira [Mez21] beschreibt in seinem Buch *Building Micro-Frontends* weitere technische und organisatorische Aspekte bei der Umsetzung und Überführung einer MF-Architektur. Dazu zählen Tests, Konzepte zur kontinuierlichen Integration und Auslieferung von Programmcode, ein Überführungskonzept sowie Details zur Zerlegung und dem Composing (siehe Abschnitt 3.1). Weiterhin werden in dem Buch weitere Technologien und Architekturen wie GraphQL[31], Backend for Frontends und API Gateways beschrieben. Diese Technologien sollen die Effizienz der API-Aufrufe verbessern, was wiederum die Performance der Webseite steigert [Mez21, Kapitel 8]. Diese Technologien betreffen hauptsächlich das Backend. Allerdings beeinflusst das Backend durch die eingesetzte Technologie das Verhalten des Frontends [Gee20].

In dem Buch von Geers [Gee20] beschreibt der Autor in Kapitel sechs detailliert die Kommunikation zwischen MFs. Er zeigt Lösungen auf, wie Informationen und Ereignisse nativ in JavaScript an andere MFs übertragen werden können. Der Autor weist darauf hin, dass jedes MF sein eigenes State Management System haben sollte, sofern eines eingesetzt wird. Um keine Kopplung zwischen MFs zu erzeugen, sollen diese lediglich mit ihrem zuständigen Backend kommunizieren, um von dort aus Daten anzufordern (siehe Anhang A.1). Mit diesem Wissen und dem aus dem vorherigen Buch [Mez21] ergibt sich folgender Schluss: Die Optimierung der Informationsverwaltung für Daten aus dem Backend erfolgt nicht in den MFs durch den Austausch von Daten, sondern durch das Ansprechen des jeweiligen Backends, das für das MF verantwortlich ist. Die Optimierung erfolgt durch die Bereitstellung optimierter API-Schnittstellen durch entsprechende Technologien wie zum Beispiel GraphQL oder API-Gateways.

30 https://web.dev/what-are-pwas/ zuletzt eingesehen am 24.07.2023
31 https://graphql.org/ zuletzt eingesehen am 29.08.2023

Ein weiterer für die Arbeit relevanter Punkt des Buches ist die Thematisierung von isomorphen Composing-Lösungen. Geers [Gee20] widmet diesem Thema ein ganzes Kapitel [Gee20, Kapitel 8]. Isomorphes Composing beschreibt die Vereinigung mehrerer Composing-Lösungen in einer MF-Architektur. Die einzelnen Composing-Lösungen werden im Abschnitt 2.1.2 ausführlich beschrieben. Das Ziel bei isomorphen Lösungen ist es, die Vorteile der einzelnen Composing-Lösungen zu vereinen. Es ist für komplexe Webseiten gedacht, bei denen verschiedene MFs unterschiedliche Composing-Lösungen bevorzugen und kein einheitlicher Kompromiss gefunden werden kann. Allerdings steigt durch die Verwendung weiterer Technologien zur Implementierung isomorpher Lösungen das Abstraktionsniveau, sodass das Einarbeiten neuer Mitarbeiter mehr Zeit in Anspruch nimmt [Gee20, Kapitel 8].

2.2.2 *Graue Literatur*

Ergänzend zu wissenschaftlichen Publikationen und Büchern gibt es viel graue Literatur, in der Erfahrungen, Umsetzungen und Best Practices beschrieben werden[32]. Hierzu zählen Tech-Blogs von Unternehmen wie Otto [Ste16], Zalando [Bro21], DAZN [Tab21; Mez19] und viele andere. Diese berichten über ihre umgesetzten MF-Architekturen. Teilweise werden komplexe Lösungen wie isomorphes Composing oder die bereits erwähnten Optimierungen mittels GraphQL oder API Gateways vorgestellt.

Des Weiteren gibt es unabhängige Artikel, wie beispielsweise von Jackson [Jac20], einem der Hauptentwickler des Webpack Module Federation Plug-ins. Er stellt dieses Plug-in in seinem Blog vor [Jac20] (Näheres siehe Unterabschnitt 2.1.2). Ein weiterer Artikel ist von Jackson auf [Jac19]. Der Autor beschreibt viele Aspekte, die bei der MF-Architektur berücksichtigt werden müssen. Ein Aspekt ist die Verwaltung von geteiltem Code, der in mehreren MFs verwendet wird. Dieser Code wird in sogenannte Shared Libraries ausgelagert. Jackson [Jac19] empfiehlt, bei der Entwicklung einer neuen Webseite auf Basis einer MF-Architektur, zunächst auf Shared Component Libraries zu verzichten. Sobald redundante Codeteile klar ersichtlich werden, können diese zu einem späteren Zeitpunkt in eine Shared Component Library ausgelagert werden.

Darüber hinaus werden Aspekte betrachtet, wie Testing, die Vereinigung verschiedener Architekturen im Backend mit der MF-Architektur sowie Nachteile der MF-Architektur. Ein von Jackson [Jac19] genannter Nachteil ist, dass Code, der in mehreren MFs verwendet wird, mehrfach auf der Clientseite geladen wird. Dadurch erhöht sich die Menge der über das Netzwerk an den Client zu sendenden Daten. Eine weitere Herausforderung laut Jackson [Jac19] ist, dass das CSS einheitlich für alle MFs sein muss um eine konsistente UX zu gewährleisten.

32 Die Autoren der Tech-Blogs sind teilweise die gleichen wie die der wissenschaftlichen Publikationen. Mezzalira für [Mez21; Mez19; TM22], sowie Geers für [Gee20; Gee23]

2.2.3 *Zusammenfassung*

Zusammenfassend werden in der Wissenschaft einige Technologien und Ansätze zur Umsetzung einer MF-Architektur beschrieben ([YLS19; PMT21; TM22; Pav+20]). Diese werden durch Beispiele an konkreten Anwendungsfällen praktisch verdeutlicht ([Men+19; Kro21; NK22; Büh+22]). Dadurch werden auch konkrete Probleme und Herausforderungen sichtbar (siehe Abschnitt 2.1.2). Durch diese individuellen Anforderungen in diesen einzelnen Anwendungsfällen entstehen neue Herangehensweisen beziehungsweise spezifische Konzepte, wie beispielsweise PWAs in Kombination mit einer MF-Architektur.

In der grauen Literatur von großen Unternehmen sowie Entwicklern werden weitere Lösungen präsentiert, wie die MF-Architekturen umgesetzt werden können ([Gee23; Mez19; Jac20; Jac19; Ste16; Bro21; Tab21]).

Bücher beschreiben umfassend viele Aspekte der MF-Architektur sowie praktische Anwendungsfälle ([RS21; Gee20; Mez21]). Allerdings legen die Bücher keine wissenschaftlichen Belege für ihr Vorgehen vor.

2.3 FORSCHUNGSLÜCKE

Wie im vorherigen Abschnitt beschrieben, ist das allgemeine Vorgehen zur Umsetzung und Implementierung einer MF-Architektur bereits gut beschrieben. Allerdings wird dabei die Fehlerbehandlung beim Laden von MFs vernachlässigt. Ein Beispiel hierfür ist der Code der Webseite, den die Autoren Geers und Rappl und Schottner [Gee20; RS21] in ihrem Anwendungsfall als Vorlage verwenden [Gee23]. Abbildung 2.6 zeigt die relevante Funktion der Web Component aus dem Beispiel, die mittels eines asynchronen Funktionsaufrufs Code nachlädt. Die Funktion `fetch()` behandelt potentiell auftretende Fehler, die durch den Promise-Rückgabewert nicht mit `catch(()=> {...})` abgefangen werden.

Gleiches gilt für das GitHub-Repository des Webpack Module Federation Plug-ins[33]. Dort gibt es Beispiele, wie eine MF-Architektur mithilfe des Module Federation Plug-ins umgesetzt werden kann. Abbildung 2.7 zeigt ein nicht vorhandenes `catch()` im Funktionsaufruf `loadRemoteModule()`.

```
1  render() {
2      const sku = this.getAttribute('sku');
3      // immediately render skeleton view (no data)
4      this.innerHTML = render();
5      // load data asynchronously and rerender with actual data
6      fetch(sku).then((data) => { this.innerHTML = render(data); });
7  }
```

Listing 2.6: Beispiel aus dem GitHub-Repository[34] von [Gee23]: In Zeile 6 wird keine Fehlerbehandlung durch ein *catch()* durchgeführt.

33 *https://github.com/module-federation/module-federation-examples/tree/master* zuletzt eingesehen am 02.08.2023

34 https://github.com/neuland/micro-frontends/blob/master/3-data-fetching/team-green/src/green-recos/custom-element.js zuletzt eingesehen am 03.09.2023

```
1  path: 'profile',
2  loadChildren: () =>
3      loadRemoteModule({
4      remoteName: 'profile',
5      remoteEntry: 'http://localhost:4201/remoteEntry.js',
6      exposedModule: 'ProfileModule',
7  }).then(m => m.ProfileModule),
```

Listing 2.7: Beispielcode von Module Federation GitHub[35]: In Zeile 7 wird keine Fehlerbehandlung durch ein *catch()* durchgeführt.

Bei den gezeigten Beispielen wird im Fehlerfall von JavaScript der Fehler in der Konsole ausgegeben. Der anzuzeigende Inhalt wird stattdessen ohne weitere Warn- oder Fehlermeldungen auf der Webseite mit einer leeren Seite gefüllt. Dies ist eine klare Verletzung der User Experience (UX), da der Client nicht erfährt, dass ein Fehler aufgetreten ist oder warum eine leere Seite angezeigt wird [PMT21].

Des Weiteren wird die Fehlerbehandlung in den wissenschaftlichen Publikationen nicht thematisiert [TM22; NK22; Büh+22; Men+19; YLS19]. Peltonen, Mezzalira und Taibi [PMT21] weist darauf hin, dass eine Fehlerbehandlung in einer MF-Architektur notwendig ist und sogar eine Herausforderung darstellt, da ansonsten die UX beeinträchtigt wird.

Als Forschungslücke stellt die Arbeit ein weiteres Konzept zur Behandlung von Fehlern vor, die beim Nachladen und Einbinden von MFs auftreten können.

35 https://github.com/module-federation/module-federation-examples/blob/master/angular15-microfrontends-lazy-components/projects/mdmf-shell/src/app/app-routing.module.ts zuletzt eingesehen am 03.09.2023

3 KONZEPT DER ÜBERFÜHRUNG UND FEHLERBEHANDLUNG

Dieses Kapitel beschreibt sowohl das Konzept zur Überführung eines Monolithen in eine MF-Architektur als auch die anschließende Fehlerbehandlung. Zu den beiden Konzepten wird jeweils auf die Methodik eingegangen. Anschließend werden die Vorgehensweisen zur Umsetzung der Ziele beschrieben.

3.1 ÜBERFÜHRUNGSKONZEPT

Das Ziel des Überführungskonzepts ist die Beantwortung der Forschungsfrage, wie ein monolithisches Web-Frontend in eine MF-Architektur überführt werden kann. Das Konzept beinhaltet ein Vorgehen, welches Teilentscheidungen beschreibt, wie eine MF-Architektur für einen Anwendungsfall definiert ist. Es werden allgemeine Herausforderungen und deren Lösungen in der Architektur erläutert. Anschließend wird im Vorgehen die definierte MF-Architektur überführt.

3.1.1 *Methodik*

Als Grundlage des Konzeptes wird sich an den bereits beschriebenen Vorgehen aus den wissenschaftlichen Publikationen (siehe Abschnitt 2.2) orientiert. Dabei wird vor allem die Struktur der Publikation “Micro-Frontends” [TM22]. Der Grund dafür ist, dass diese Publikation im September 2022 erschienen ist und damit aktuelle Technologien wie Module Federation betrachtet. Sie fokussiert sich außerdem auf Technologien zur clientseitigen Implementierung, was mit der Abgrenzung dieser Arbeit harmonisiert (siehe Abschnitt 1.3).

Des Weiteren ähnelt die in der Publikation beschriebene Struktur früheren wissenschaftlichen Publikationen ([PMT21; YLS19]) und Büchern ([Mez21; Gee20]).

Das Konzept wird durch eine Fallstudie evaluiert. In dieser wird ein monolithisches Web-Frontend durch das Konzept in eine MF-Architektur überführt. Eine Fallstudie eignet sich in der Arbeit gut, da eine praktische Überführung direkt mit der Forschungsfrage verknüpft ist. Des Weiteren kann die Fallstudie gut mit anderen bereits durchgeführten Überführungen aus der Wissenschaft verglichen werden (siehe Abschnitt 2.2).

Anhand der Ergebnisse wird evaluiert, wie gut das Konzept umsetzbar ist und ob bei der Umsetzung Probleme auftreten. Die Wahl der Webseite erfolgt in Kapitel 4.

3.1.2 *Vorgehen*

Das Konzept ist in mehrere Teilentscheidungen unterteilt. Jede Teilentscheidung beschreibt ein Problem, für das eine Lösung gefunden werden muss. Um dieses Problem zu lösen, gibt es jeweils mehrere Optionen. Eine Option besteht aus einer Technologie, die eine potenzielle Lösung für das Problem präsentiert. In manchen Teilentscheidungen können auch mehrere Optionen zur Lösung des Problems ausgewählt werden.

Es gibt keine einheitliche Lösung für die Architektur [Gee20, Kapitel 2]. Die Entscheidung, welche Option ausgewählt wird, muss abhängig vom Anwendungsfall getroffen werden. Gegebenenfalls muss eine gewählte Option auch individuell angepasst oder erweitert werden. Die Aggregation der Teilentscheidungen ergibt die MF-Architektur [TM22]. Diese Teilentscheidungen werden im Folgenden aufgelistet.

1. Zerlegung
2. Zusammensetzungsstrategien (Composing)
3. Routing
4. Kommunikation
5. Implementation

Sobald die MF-Architektur definiert ist, muss diese überführt beziehungsweise neu implementiert werden. Die Überführung wird im Anschluss an die Implementierung beschrieben.

Zerlegung

Wie im Kapitel 2.1 beschrieben, gibt es zwei Herangehensweisen, ein monolithisches Web-Frontend in MF zu zerlegen. Häufig wird die Identifizierung der Domänen auch durch das eingesetzte JavaScript-Framework beeinflusst. Beispielsweise wird in den von Angular eingesetzten Routing-Mechanismen Module aufgerufen und geladen. Diese Module sind Domänen, die einfach in MFs zerlegt werden können. Allerdings bleibt die Bedingung, dass diese Domänen keine Abhängigkeiten zur restlichen Applikation besitzen.

In diesem Konzept wird eine vertikale Zerlegung für die meisten Webseiten empfohlen. Diese kann direkt an der visuellen Struktur abgeleitet werden. Sind die einzelnen MF zu groß beziehungsweise eignen sich dazu, weiter zerlegt zu werden, kann dies im weiteren Verlauf der Entwicklung durchgeführt werden. Dieser Ansatz harmonisiert gut mit den Empfehlungen aus [Jac19] für das Auslagern von geteiltem Code. Der geteilte Code soll nach den Empfehlungen auch zu einem späteren Zeitpunkt in eine Shared Component überführt werden. Nach der initialen Überführung kann dann in einem weiteren Schritt die horizontale Zerlegung eingeführt und geteilter Code in Shared Libraries überführt werden.

Entspricht allerdings die Webseite deutlich den Empfehlungen für eine horizontale Zerlegung, ist diese direkt durchzuführen.

Zusammensetzungsstrategien (Composing)

Entsprechend der Abgrenzung in dieser Arbeit (siehe Abschnitt 1.3), beschränken sich das Konzept auf die Verwendung von Client-Side-Composing.

Routing

Da das Routing stark von dem Composing abhängig ist, wird hierfür in dem Konzept ebenfalls sich auf das clientseitige Routing beschränkt.

Das Routing auf der Clientseite erfolgt durch die Shell. Dort werden in der Implementierung Technologien verwendet, um die Abbildung der URI auf die Routen zu abstrahieren. Dies minimiert den Implementierungsaufwand und senkt dabei die Chance auf Programmierfehler durch eigens programmierte Prozesse.

Kommunikation

Es können mehrere Technologien für die Kommunikation zwischen MFs eingesetzt werden. Diese sollten entsprechend ihrem Anwendungsbereich eingesetzt werden. Beispielsweise wird der Storage für das Persistieren des Zugangstokens (JWT) und weiterer Daten, die über die Laufzeit der Webseite hinaus bestehen bleiben sollen, verwendet.

Es muss klar dokumentiert werden, welche Daten, über welche Technologie, wann und von welchem MF verschickt werden. Dies soll helfen, den Überblick über die Kommunikationsflüsse zu behalten. Andernfalls besteht die Gefahr, dass speziell bei mehreren Entwicklern der Überblick verloren geht. Dies erhöht die Gefahr von Programmierfehlern. Implementation Das Konzept zur Implementierung beschreibt ein Verfahren, welche Technologien beziehungsweise Frameworks eingesetzt werden. In der Tabelle im Anhang A.1 von [Kro21] werden viele Frameworks aufgezählt. Im ersten Schritt ist das Ziel, die Anzahl an möglichen Frameworks, die zur Implementierung genutzt werden können, zu reduzieren.

Die Tabelle wird nach dem verwendeten Composing gefiltert. Anschließend muss für den jeweiligen Anwendungsfall analysiert werden, ob es durch die Verwendung eines oder mehrerer JavaScript-Frameworks zu weiteren Beschränkungen in der Auswahl kommt. Zum Beispiel ist Piral lediglich mit dem JavaScript-Framework React einsetzbar. Nach diesen Bedingungen wird die Tabelle ebenfalls gefiltert. Für eine verlässliche Informationsquelle sollte außerdem die Dokumentation mit "gut"bewertet sein.

Die übrig gebliebene Menge an Frameworks muss dann für den jeweiligen Anwendungsfall auf Anwendbarkeit geprüft und gegeneinander abgewogen werden. Hierbei können Aspekte wie beispielsweise bereits vorhandenes Wissen der Entwickler*innen in den Frameworks oder die Größe und Komplexität der definierten MFs aus der Zerlegung mit einfließen.

Eine Kombination von mehreren Frameworks, die für unterschiedliche Aufgaben zuständig sind, ist ebenfalls denkbar. Als Beispiel ist es technisch möglich, Web Components zur Implementierung der MFs zu verwenden.

Diese werden durch Module Federation nachgeladen und über single-spa im DOM ein- und ausgebunden.

Eine weitere Aufgabe der Implementierung ist es, Code, der in mehreren MFs verwendet wird, auszulagern. Ein simples Beispiel hierfür ist CSS, welches in jedem MF ein einheitliches Design garantieren soll (siehe Unterabschnitt 2.1.1). Damit das CSS in jedem MF vorhanden ist, muss eine Lösung gefunden werden, wie dieser Code in den isolierten MFs gemeinsam genutzt werden kann [PMT21].

Hierfür werden sogenannte Shared Libraries erstellt [PMT21]. In ihnen ist der gemeinsam genutzte Code enthalten. Eine einfache Lösung zur Implementierung der Shared Libraries ist die Verwendung von NPM. Mithilfe von NPM werden die Shared Libraries vor der Kompilierung der eigentlichen MFs kompiliert und als externe Bibliothek den MFs zur Kompilierungszeit zur Verfügung gestellt. Dies ist eine simple und effektive Möglichkeit, den Code allen MFs zugänglich zu machen. Allerdings wird dieser Code nach der Kompilierung der MFs in jedem dieser enthalten. Dadurch wird beim Laden der MFs der Code der Shared Libraries mehrfach geladen. Module Federation löst dieses Problem, indem es die Shared Libraries als Chunks nachlädt (siehe Abschnitt 2.1.2). Die Verwendung dieses Plug-ins ist daher, sofern dies technisch möglich ist, empfohlen.

Überführung Nachdem im Konzept die MF-Architektur für einen Anwendungsfall definiert ist, muss das bestehende monolithische Web-Frontend des Anwendungsfalls in diese überführt werden. Hierzu werden fünf Schritte beschrieben, wie die Überführung erfolgt.

1. Neues Projekt erstellen und Implementierung des Grundgerüsts (Shell)
2. Auslagern von gemeinsam genutztem Code in Shared Libraries
3. Implementierung der einzelnen MFs
4. Aufruf der MFs durch Composing und Routing
5. Implementierung der Kommunikation zwischen den MFs

Grundgerüst Zunächst muss ein neues Projekt erstellt werden. Das alte Projekt mit der monolithischen Architektur bleibt bestehen und wird zur Entwicklungszeit des neuen Projektes weiter betrieben und gegebenenfalls weiterentwickelt.

Das Grundgerüst besteht aus der Shell sowie weiteren initial angelegten Dateien. Dies sind Konfigurationsdateien für die jeweiligen Anwendungsfälle wie beispielsweise Prozesse zur Qualitätssicherung und dem kontinuierlichen Ausliefern des Codes.

Es empfiehlt sich, die Command Line Interface (CLI) des jeweiligen JavaScript-Frameworks zu benutzen, welches verwendet wird. Die CLI ist ein Programm, über das Entwickler Befehle ausführen können, um beispielsweise neue Projekte oder Module mit dem spezifischen Aufbau des JavaScript-Frameworks anzulegen oder um einen betriebsfertigen Webserver zu starten. Über die CLI wird das Projekt angelegt. Anschließend wird ein Ordner für alle MFs erstellt.

In diesen Ordner wird die Shell als weiteres neues Projekt beziehungsweise Unterprojekt angelegt.

Abschließend sollte die Shell mit Standardelementen durch entsprechende Befehle der CLI aufrufbar sein. Die Standardelemente werden durch globale Elemente wie die Navigationsleiste für das Top-Level-Routing und den Footer ersetzt.

Shared Libraries

Die Shared Libraries beinhalten den Code, der von mehreren MFs verwendet wird. Das Konzept beschreibt, welcher Code von den MFs ausgelagert wird und wann eine neue Shared Library erstellt wird (siehe Abbildung 3.1 und 3.2).

Sobald Code, der eine geringe Kopplung besitzt, in allen MFs gefunden wird, bietet sich dieser zum Auslagern an. Ein einfaches Beispiel ist globales CSS. Dies kann keine Kopplung besitzen und ist leicht auszulagern. Globales CSS sollte unbedingt ausgelagert werden, damit ein einheitliches Design auf der Webseite garantiert wird.

Schwieriger wird es für JavaScript. In großen und komplexen MFs kann es sich als schwierig erweisen, Code zu finden, der in mehreren MFs existiert. Wie in Abbildung 3.1 und 3.2 gezeigt, können auch Codes mit hoher Kopplung zusammen in eine Shared Library überführt werden, solange diese zum restlichen MF keine Kopplung besitzen.

Jede Shared Library beinhaltet Code für bestimmte MFs. Abbildung 3.2 zeigt, dass der Code aus MF-A und MF-B in eine Shared Library überführt wurde. Der globale CSS-Code, der von allen MFs verwendet wird, ist in einer eigenen Shared Library.

Mit diesem Vorgehen wird der Code effizient aufgeteilt. Die Aufteilung verhindert, dass der gesamte Code aus den Shared Libraries eager geladen wird, obwohl dieser nicht gebraucht wird. Wird wie in der Abbildung 3.2 gezeigt, nur MF-C geladen, wird der Code von Shared Library A nicht geladen.

Darüber hinaus verbessert sich die Übersicht, welcher Code in welchen MFs verwendet wird. Bei einer Vielzahl von unterschiedlichen Shared Libraries bietet sich gegebenenfalls eine separate Dokumentation zur Übersicht an.

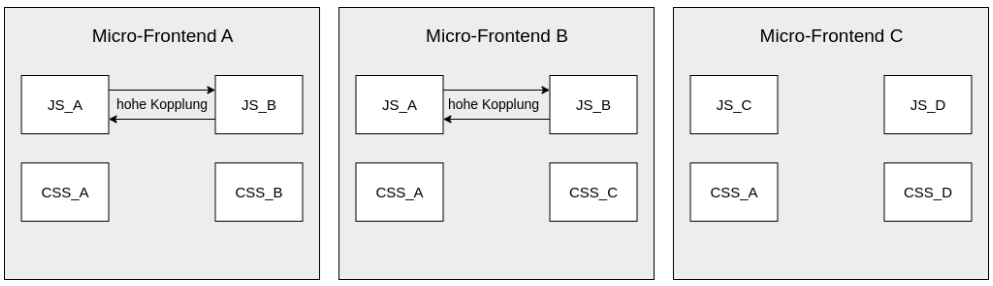


Abbildung 3.1: Redundanter Code in mehreren MFs ohne den Einsatz von Shared Libraries

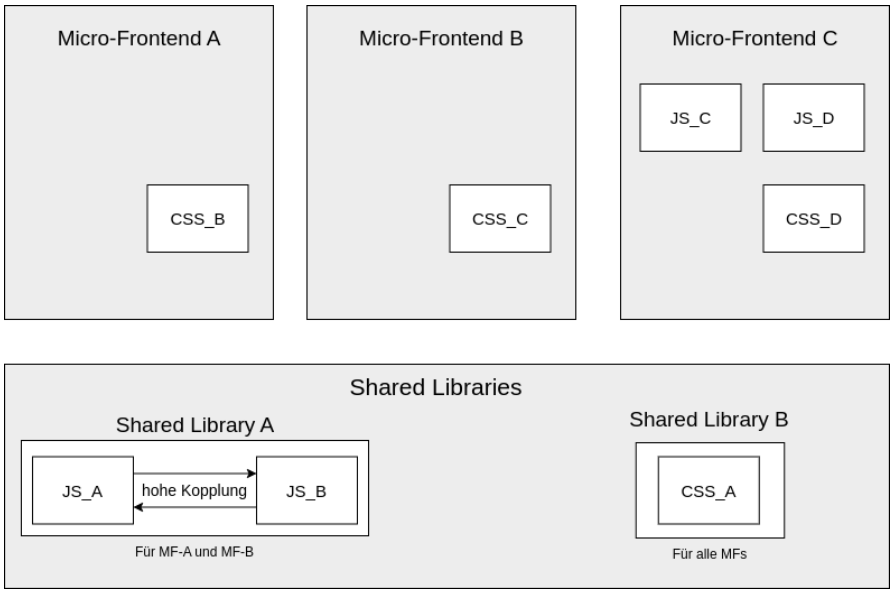


Abbildung 3.2: Redundanter Code in mehreren MFs ohne den Einsatz von Shared Libraries

Die Verlinkung für global verwendete Shared Libraries kann bereits in diesem Schritt durchgeführt werden. Hierzu müssen jeweils die Shared Libraries kompiliert werden. Anschließend erfolgt die Verlinkung aus dem Ordner, in dem die kompilierten Dateien abgelegt wurden.

```
npm link #in dem ordner in dem die kompilierte shared-library-a liegt
cd <Wurzelverzeichnis des Projektes>
npm link shared-library-a
```

Listing 3.1: Verlinkung der *shared-library-a* zu allen MFs (globale Verlinkung)

Implementierung Micro-Frontends

Bei der Implementierung werden die MFs sukzessive überführt. Die folgenden Schritte in der Implementierung wiederholen sich für jedes MF.

Zunächst wird mithilfe der CLI ein weiteres Unterprojekt angelegt. Das Unterprojekt sollte über den Webserver der CLI aufrufbar sein. Anschließend wird der Code aus dem monolithischen Projekt, der zu dem definierten MF gehört, in das Unterprojekt kopiert. Der Code, der in die Shared Libraries überführt wurde, wird entfernt, sofern dies noch nicht geschehen ist. Der entfernte Code wird durch die Shared Libraries verlinkt und ersetzt.

```
npm link # in dem ordner in dem die kompilierte shared-library-a liegt
cd <Wurzelverzeichnis des Unterprojektes>
npm link shared-library-a
```

Listing 3.2: Verlinkung der *shared-library-a* zu dem MF mf-1

Das MF ist nach diesen Schritten durch die CLI ausführbar. Gegebenenfalls bestehen Fehler durch die fehlende Implementierung der Kommunikation.

Composing und Routing

Bislang sind alle MFs isoliert überführt. Die Shell beinhaltet die Navigationsleiste und den Footer. Die Routing-Funktionalität bei der Verwendung wird in diesem Schritt in der Shell implementiert.

Das Routing ist abhängig von dem gewählten JavaScript-Framework sowie zusätzlichen Technologien zur Implementierung der MF-Architektur. Zwei einfache Beispiele zur Implementierung eines Routings sind im Anhang zu finden (Web Components A.2, single-spa A.3). Die Abbildung im Anhang A.2 zeigt das Beispiel von single-spa. Das Routing wird durch die zwei Buttons angestoßen, die eine URI aufrufen.

Nach der Implementierung des Routings in der Shell muss es möglich sein, über die Shell die jeweiligen MFs aufzurufen. Der Aufruf kann gegebenenfalls durch die fehlende Kommunikation zu Fehlern führen.

Kommunikation

Der letzte Schritt implementiert die gewählten Technologien zur Kommunikation zwischen den MFs und der Shell.

Bei der Verwendung von QueryStrings müssen die Routen angepasst werden. In Code 3.3 wird anhand des JavaScript-Frameworks Angular gezeigt, wie eine Route mit QueryStrings aussehen kann. In dem gezeigten Beispielcode wird bei dem Aufruf der Route *localhost:4200/products/details/42* ein MF geladen. Der *ProductResolver* liest den Wert aus der URI (42) aus und übergibt diesen dem MF, sobald es geladen ist.

```
1 {
2     path: 'products/details/:slug',
3     resolve: {
4         article: ProductResolver
5     },
6     loadChildren: () => {/** load remote mf */},
7 }
```

Listing 3.3: Beispielcode für eine Route in Angular mit QueryStrings

Eine weitere einfache Methode zur Kommunikation ist das Speichern von Daten im Storage. Es muss unbedingt die Anwesenheit der Daten beim Auslesen aus dem Storage überprüft werden. Eine Integritätsprüfung ist in diesem Beispiel im Frontend nicht durchführbar, da keine Informationen über existierende Produkt-IDs vorhanden sind. Die Integritätsprüfung muss im Backend erfolgen. Ist eine Integritätsprüfung möglich, ist diese durchzuführen.

```
localStorage.setItem("productId", 42);
```

Listing 3.4: Speichern der Produkt-ID im mf-1 über den Storage

```
1 const productId = localStorage.getItem("productId");
2 if (!productId) {
3     /** error handling process */
4 }
```

Listing 3.5: Auslesen der Produkt-ID im mf-2 über den Storage

Nach der Implementierung der Kommunikation funktionieren die MFs fehlerfrei. Die gesamte Webseite kann ordnungsgemäß verwendet werden und ist aus der Perspektive des Clients identisch zur monolithischen Webseite.

3.1.3 *Zusammenfassung*

Der Aufbau der MF-Architektur erfordert vielschichtige Entscheidungen. Es ist wichtig zu betonen, dass diese Entscheidungen immer an den jeweiligen Anwendungsfall angepasst werden können und somit Eigeninterpretationen möglich sind.

Bei der Überführung eines Monolithen in eine MF-Architektur müssen einige Aspekte berücksichtigt werden. Der Monolith sollte so aufgebaut sein, dass eine vertikale oder horizontale Zerlegung möglich ist. Das Konzept empfiehlt zunächst eine vertikale Zerlegung durchzuführen. Zu einem späteren Zeitpunkt wird geteilter Code in Shared Libraries ausgelagert und gegebenenfalls einzelne MFs horizontal zerlegt.

Das Composing und damit verbundene Routing beschränkt sich in dem Konzept auf clientseitige Lösungen.

Das Konzept empfiehlt eine Dokumentation der Kommunikationsflüsse zwischen MFs sowie bei der Auslagerung des Codes in Shared Libraries anzulegen.

Die Überführung wird in fünf Schritten durchgeführt. Mit diesen Schritten ist es möglich, ein monolithisches Web-Frontend in eine MF-Architektur zu überführen. Dabei wird ein neues Projekt erstellt, wodurch das Konzept auch auf noch nicht bestehende Projekte angewendet werden kann, die von Grund auf eine MF-Architektur verwenden.

3.2 FEHLERBEHANDLUNGSKONZEPT

Fehler treten in jedem Programm auf, auch in einer MF-Architektur. Ein klassisches Beispiel für einen Fehler in einer solchen Architektur ist eine instabile Netzwerkverbindung. MFs können dann nicht mehr ordnungsgemäß nachgeladen werden, wodurch die Funktionalität der Webseite gestört wird. Für diese Fehler sollte es Verfahren geben, um sie abzufangen und zu behandeln. Andernfalls können Fehler nicht nur zu inkonsistenten Webseiten führen, sondern auch zusätzlich durch Sicherheitslücken gefährden [WN04]. Die Fehlerbehandlung sollte einheitlich und benutzerfreundlich gestaltet sein (UX).

Als Ergänzung zu dieser Arbeit ist das Ziel dieses Konzepts, eine Übersicht über die Fehler zu erhalten, die während des Nachladens, Einbindens und Aushängens von MFs auftreten können. Diese Übersicht soll den Aufwand für die Entwicklung der Behandlung der einzelnen Fehler reduzieren. Für die Übersicht wird eine Tabelle erstellt, in die die Fehler eingetragen und um weitere Informationen ergänzt werden. Aus dieser Tabelle werden die Fehler gruppiert. Die Fehlerbehandlung wird dann anhand der Gruppen durchgeführt. Dadurch reduziert sich die Anzahl der zu entwickelnden Prozesse zur Behandlung der Fehler.

3.2.1 *Methodik*

Die in diesem Konzept vorgestellte Fehlerbehandlung wird durch eine Fallstudie evaluiert. Diese wird am Anwendungsfall von Conduit, die bereits in eine MF-Architektur überführt wurde, umgesetzt. Auch für das zweite Konzept eignet sich eine Fallstudie gut, da dies eine schnelle und effektive Methode ist, das Konzept zu evaluieren. Anhand der Ergebnisse wird bewertet, wie gut das Konzept umsetzbar ist und ob bei der Umsetzung Probleme auftreten.

Die Fehler, die auftreten können, hängen vom jeweiligen Backend ab, das verwendet wird. Beispielsweise kann eine Firewall dafür verantwortlich sein, Anfragen an den Server von einem Client zu limitieren, um eine Überlastung zu verhindern. Die Firewall kann die Anfrage dann mitteilen, dass das sogenannte Rate Limiting erreicht ist. In der Fallstudie wird davon ausgegangen, dass lediglich ein einfacher Webserver verwendet wird, um die MFs auszuliefern. Die verschiedenen MFs sowie die Shell sind über definierte Ports erreichbar. Es werden keine weiteren Technologien wie Load Balancer, Firewalls, Reverse Proxy und so weiter verwendet.

Das Konzept beschreibt den Aufbau der Tabelle. Diese wird in drei Schritten erstellt:

1. **Identifizierung**: Alle Fehler die möglicherweise auftreten können werden identifiziert und in die Tabelle geschrieben.
2. **Analyse**: Zu jedem Fehler muss ein Szenario gefunden werden. Anhand dieses Szenarios werden die Fehler auf “Auswirkung“, “Eintrittswahrscheinlichkeit“ und “Lösbarkeit“ bewertet.
3. **Auswertung**: Für jedes Bewertungskriterium werden Vorgehensweisen definiert, wie die Webseite reagieren sollte.

3.2.2 *Identifizierung*

Im ersten Schritt wird eine Tabelle erstellt, um die Fehler einzutragen. Diese Tabelle wird zunächst mit den Spalten “Kategorie“ und “Fehler“ erstellt.

Tabelle 3.1: Tabelle ohne Inhalt mit den Spalten nach der Identifizierung

Kategorie	Fehler

Für die Identifizierung der Fehler gibt es verschiedene Herangehensweisen. Zunächst muss der Code des Anwendungsfalls nach den verwendeten Verfahren zum Nachladen der MFs durchsucht werden. Typischerweise gibt es zwei Verfahren: Das erste Verfahren sind Script-Tags. Diese werden direkt über das HTML interpretiert und bieten eine einfache Möglichkeit, JavaScript nachzuladen. Allerdings werden im Fehlerfall keine Statuscodes mitgegeben. Somit kann der Fehler nicht identifiziert werden, wodurch eine spezifizierte Fehlerbehandlung nicht möglich ist. Möglich ist lediglich eine allgemeine Fehlerbehandlung [Lev05].

XML HttpRequest[1] ist das zweite Verfahren, wie MFs nachgeladen werden können. Bei diesen Requests hat der Entwickler*innen mehr Möglichkeiten, zusätzliche Informationen wie HTTP-Header an die Anfrage anzuhängen. Bei XML HttpRequests wird auch der Statuscode im Falle eines Fehlers mitgeliefert. Somit ist eine spezifizierte Fehlerbehandlung möglich. Es gibt verschiedene Bibliotheken, die das Laden mithilfe von XML HttpRequest vereinfachen. Eine beliebte Bibliothek ist Axios[2].

Für eine detaillierte Identifizierung des Fehlers ist der Einsatz von XML HttpRequest aufgrund genauerer Fehlermeldungen zu empfehlen. Anderenfalls werden UX-Aspekte durch ungenaue Fehlermeldungen verletzt [Lev05]. Sollte der Einsatz von XML HttpRequest nicht möglich sein, dann kann bei der Verwendung von Script-Tags nur ein Eintrag vorgenommen werden.

Wenn XML HttpRequest verwendet wird, kann der Fehler über den HTTP-Statuscode identifiziert werden. Die Statuscodes werden in den Request for Comments (RFC)s 6585, 7725, 7231 und folgenden, 7807 sowie 8470 beschrieben[3]. Im Fehlerfall wird der Statuscode von XML HttpRequest zurückgegeben und kann dadurch identifiziert werden. Alle 400er- und 500er-Statuscodes werden in die Tabelle überführt. Diese Statuscodes beschreiben Fehler, die entweder aufgrund einer fehlerhaften Anfrage (400er) oder eines fehlerhaften Verhaltens des Servers (500er) entstehen. Darüber hinaus entstehen weitere Fehler, wenn keine valide HTTP-Anfrage oder Antwort gesendet wird. Ein klassisches Beispiel ist das Ablaufen der Wartezeit auf eine HTTP-Antwort. Die Art, wie diese Fehler identifiziert werden müssen, kann über die jeweils eingesetzte Technologie in der Dokumentation nachgelesen werden. In Axios gibt es zu den Fehlercodes keine Dokumentation. Diese können jedoch direkt im Code eingesehen werden. Siehe dazu den Code auf GitHub[4]. Alle gefundenen Fehler werden in die Tabelle überführt.

Nachdem alle Netzwerkfehler, die beim Laden eines MFs auftreten, identifiziert und in die Tabelle überführt wurden, müssen Fehler, die beim Einbinden des MFs auftreten können, identifiziert werden. Hierzu gibt es zwei typische Fehlerquellen. Die erste betrifft die Manipulation des DOMs. Dabei können Fehler entstehen, bei denen das DOM inkorrekt manipuliert wird, sodass die Integrität der Baumstruktur verletzt wird [Moz23]. In der Dokumentation zum Einbinden[5] und Aushängen[6] werden die Fehler jeweils beschrieben.
Des Weiteren können abhängig von der gewählten Technologie beziehungsweise dem gewählten Framework weitere Fehler während des Einbindens entstehen. Diese sind in den jeweiligen Dokumentationen nachzulesen.

1 https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest zei. am 02.09.2023
2 https://axios-http.com/docs/intro zei am 02.09.2023
3 Alle Links zuletzt eingesehen am 09.08.2023
4 https://github.com/axios/axios/blob/v1.x/lib/helpers/HttpStatusCode.js zei. am 02.09.2023
5 https://developer.mozilla.org/docs/Web/API/Node/appendChild zei. am 02.09.2023
6 https://developer.mozilla.org/docs/Web/API/Node/removeChild zei. am 02.09.2023

3.2.3 *Analyse*

Nachdem alle Fehler in der Tabelle aufgelistet sind, werden sie analysiert. Bei der Analyse wird im ersten Schritt ein Szenario für den Fehler gesucht. Alle Fehler, bei denen kein Szenario gefunden werden kann, werden aussortiert, und nur die verbleibenden Fehler werden bewertet. Die Bewertung ist unterteilt in drei Punkte: Eintrittswahrscheinlichkeit, Auswirkung und Lösbarkeit. Die Tabelle wird um die Spalten “Szenario“, “Eintrittswahrscheinlichkeit“, “Auswirkung“, “Lösungsversuchünd “Lösbarkeit“ ergänzt.

Tabelle 3.2: Tabelle ohne Inhalt mit den Spalten nach der Analyse

Kategorie	**Fehler**	**Szenario**	**Eintrittswahrscheinlichkeit**

Auswirkung	**Lösungsversuch**	**Lösbarkeit**

Szenario

Um ein Szenario für den jeweiligen Fehler in der Tabelle zu finden, muss die Webseite analysiert werden. Dabei wird nicht nur das Frontend, sondern auch das Backend und dessen Infrastruktur betrachtet. Für die HTTP-Statuscodes muss herausgefunden werden, ob der Webserver beziehungsweise die eingesetzte Infrastruktur im Backend solche Statuscodes überhaupt zurücksendet. Benutzt der Webserver beispielsweise keine Limitierung der eingehenden Anfragen (sogenanntes Rate Limiting), kann kein 429 HTTP-Statuscode zurückgegeben werden, da dieser nach der Definition des RFC 6585[7] auf ein Rate Limiting zurückzuführen ist [FN12]. Kann der Webserver den Statuscode zurücksenden, muss hierfür ein Szenario beschrieben werden. Die RFCs dienen dabei als Dokumentation für zusätzliche Informationen.

Für andere Fehler, speziell zum Einbinden des MFs, muss, wie bei der Identifizierung der Fehler, auf die Dokumentation der verwendeten Technologie zur Implementation zurückgegriffen werden. Gegebenenfalls ist es auch nötig, den Code manuell zu analysieren, um herauszufinden, ob und wann der Fehler in dem Anwendungsfall auftreten kann. Dies ist beispielsweise bei den geworfenen Fehlern bei der Bibliothek Axios der Fall. Bei einem bereits bestehenden System kann alternativ durch das bewusste Herbeiführen der Fehler getestet werden, wann dieser auftritt.

Alle Fehler, für die kein Szenario gefunden werden kann, werden im weiteren Konzept ignoriert. Sie werden nur der Vollständigkeit halber aufgeführt und können der Tabelle schnell hinzugefügt werden, wenn sie in der weiteren Entwicklung doch auftreten beziehungsweise ein Szenario gefunden wird.

7 https://datatracker.ietf.org/doc/html/rfc6585 zuletzt eingesehen am 24.08.2023

Eintrittswahrscheinlichkeit

Die Wahrscheinlichkeit, dass ein Fehler auftritt, wird in die Bewertungskriterien “gering“, “mittel“ und “hoch“ eingeteilt. Jeder Fehler wird hierauf analysiert und das Ergebnis wird in die Spalte “Eintrittswahrscheinlichkeit“ eingetragen.

- **gering**: Der Fehler kann nur durch aktives Eingreifen des Clients herbeigeführt werden. Hierzu zählt beispielsweise das Manipulieren des Storages. Des Weiteren fallen hierunter Bugs, die hohe Auswirkungen auf die Funktionalität der Webseite haben und bei allen Clients auftreten.
- **mittel**: Der Fehler tritt ohne aktives Eingreifen des Clients auf. Derartige Fehler gibt es selten und nur bei einzelnen Clients (zum Beispiel durch inkompatible Browser). Des Weiteren fallen hierunter Bugs mit geringen oder mittleren Auswirkungen. Außerdem zählen dazu auch Ausfälle der Server, die zu jeglichen 500er HTTP-Statuscodes führen.
- **hoch**: Der Fehler tritt ohne aktives Eingreifen des Clients auf. Dies kann durch variable, äußere Einflüsse des Systems entstehen, wie beispielsweise eine Unterbrechung der Verbindung zum Server.

Auswirkung

Die Auswirkung beschreibt die Folgen des Fehlers für die Webseite. Wie auch bei der Eintrittswahrscheinlichkeit werden drei Bewertungskriterien vorgestellt, in die der Fehler eingeordnet wird. Das Ergebnis wird in die Spalte Äuswirkungëingetragen.

- **gering**: Das zu ladende MF kann trotz des Fehlers geladen werden und ist in seinen Funktionen beschränkt ausführbar. Oder der Fehler ist sofort oder in absehbarer Zeit lösbar. Ein Beispiel hierfür ist der HTTP-Statuscode 503. Dieser besitzt ein Retry-After-Feld, über das der Server mitteilen kann, wann die Anfrage erneut geschickt werden kann [FR14].
- **mittel**: Die Funktionalität des zu ladenden MF ist infolge des Fehlers zu einem großen Teil oder vollständig gestört. Die restliche Webseite und alle anderen MFs funktionieren weiterhin ordnungsgemäß.
- **hoch**: Der Fehler wirkt sich nicht nur auf das nachladende MF aus, sondern auf die gesamte Website. Dadurch ist die Funktionalität der restlichen Website in Teilen oder ganz gestört.

Lösbarkeit

Die Lösbarkeit hängt von der Art des Fehlers und dem bereits definierten Szenario ab. Zu unterscheiden ist außerdem zwischen Problemlösungen, die mit der Interaktion des Clients oder ohne dessen Zutun erfolgen. Die Lösbarkeit wird in die Spalte “Lösbarkeit“ der Tabelle durch folgende Bewertungskriterien definiert:

- **gering**: Es wird erwartet, dass der Fehler weiterhin existiert. Dies sind beispielsweise Serverfehler. Solche Fehler können nicht auf der Clientseite gelöst werden.
- **mittel**: Der Fehler kann potenziell gelöst werden. Der Erfolg ist allerdings ungewiss. Dies betrifft zum Beispiel Netzwerkfehler. Ein erneuter Versuch beziehungsweise eine wiederhergestellte Verbindung kann die Webseite wieder in einen regulären Zustand bringen. Allerdings kann der Grund für den Verbindungsverlust weiterhin bestehen und das Wiederherstellen scheitern.
- **hoch**: Es wird erwartet, dass mit der Problemlösung der Fehler behoben wird. Die Webseite ist danach wieder in ihrem regulären Zustand.

3.2.4 *Auswertung*

Entsprechend der Bewertungskriterien werden Anforderungen an die Applikation gestellt. In der Auswertung werden zu den Bewertungskriterien der Eintrittswahrscheinlichkeit, Auswirkung und Lösbarkeit jeweils Vorgehensweisen definiert. Das Vorgehen beschreibt eine Reaktion der Webseite. Jeder Anwendungsfall besitzt andere Anforderungen, wie mit Fehlern umgegangen werden soll beziehungsweise wie hoch die Anforderungen an die Robustheit einer Webseite sind. Daher müssen die Vorgehensweisen für jeden Anwendungsfall individuell erstellt werden.

Ein Beispiel ist, Fehler, die mit einer geringen Eintrittswahrscheinlichkeit bewertet wurden, nicht zu behandeln. Es sind auch Kombinationen denkbar wie beispielsweise: “Fehler, die mit einer geringen Eintrittswahrscheinlichkeit bewertet werden und höchstens eine mittlere Auswirkung haben, werden nicht behandelt“.

3.2.5 *Vorgehen*

Dieser Unterabschnitt beschreibt das Vorgehen, um die Fehlerbehandlung mit möglichst reduziertem Entwicklungsaufwand zu implementieren. Dazu werden zunächst die Fehler aus der Tabelle gruppiert. Die Gruppierung erfolgt anhand der logischen Zusammengehörigkeit der Fehler. Die logische Zusammengehörigkeit basiert darauf, dass das Verhalten der Webseite bei den Fehlern das Gleiche ist. Ein Beispiel für die logische Zusammengehörigkeit sind alle Fehler, die aufgrund einer Störung des Servers entstehen (alle > 500 HTTP-Statuscodes). Hierfür kann dem Client mitgeteilt werden, dass die Funktionalität des Servers gestört ist.

Für jede Gruppe wird im Code eine Fehlerbehandlungskomponente erstellt. Die Fehlerbehandlungskomponente hat die Aufgabe, den aufgetretenen Fehler dem Client zu beschreiben und ihm Hinweise zu geben, wie er das Problem lösen kann.

Soll für die Behebung des Fehlers ein Prozess erstellt werden, wird hierfür eine separate Komponente erstellt, die sogenannte Fehlerbehebungskomponente. Diese Komponente führt den in der Tabelle beschriebenen Lösungsversuch zu dem Fehler durch. Für jeden unterschiedlichen Lösungsversuch gibt es eine separate Komponente. Beispielsweise wird bei dem Lösungsversuch “erneut versuchen“ der Prozess zum Nachladen des MFs erneut angestoßen. Dieser Prozess ist in einer Fehlerbehebungskomponente implementiert.

Für jeden Fehler kann eine Fehlerbehandlungs- und eine Fehlerbehebungskomponente zugewiesen werden. Die Fehlerbehebungskomponente kann optional sein, wenn es für den Fehler keinen Lösungsversuch gibt oder dieser keine Aussichten auf Erfolg hat. Die Zuweisung erfolgt statisch im Code. Die Abbildung 3.3 zeigt schematisch, wie die Zuweisung und der Zusammenbau der Komponenten implementiert wird. Treten Fehler auf, die nicht zugewiesen worden sind, werden diese von einer Standardfehlerkomponente behandelt. Diese Komponente gibt eine generische Fehlermeldung aus.

Die in der Abbildung 3.3 dargestellte *RootErrorHandlingComponent* verwaltet die Fehlerbehebungs- und -behandlungskomponente. Sie stellt die Kommunikation beider Komponenten und weitere Informationen zur Verfügung. Darüber hinaus liegen dort allgemeingültige Prozesse, die für alle Fehler ausgeführt werden.

Durch diese Art der Implementierung können leicht weitere Fehler, die durch zukünftige Weiterentwicklung hinzukommen, Gruppen zugewiesen werden. Auch das Hinzufügen von neuen Gruppen beziehungsweise Komponenten ist einfach umsetzbar. Durch die Modularisierung werden Verantwortlichkeiten getrennt, wodurch viele Aspekte der Softwarequalität berücksichtigt werden (Wiederverwendbarkeit, Wartbarkeit, Testbarkeit, Portabilität etc.).

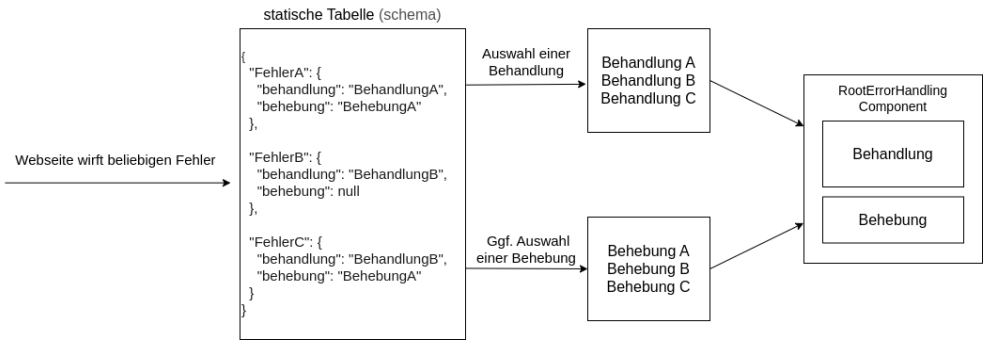


Abbildung 3.3: Implementierung einer Fehlerbehandlung. Fehler A, B und C werden aus den zur Verfügung stehenden Optionen zur Behandlung und Behebung zusammengesetzt.

4 IMPLEMENTIERUNG

Die Evaluation erfolgt durch eine Umsetzung der Konzepte an einem praktischen Anwendungsfall in einer Fallstudie. Im Folgenden wird eine monolithische Webseite in eine MF-Architektur überführt und anschließend das Fehlerbehandlungskonzept darauf angewendet.

4.1 DIE WEBSEITE FÜR DIE FALLSTUDIE

Um die Suche nach einem passenden Anwendungsfall einzuschränken, werden im folgenden Abschnitt zunächst Kriterien definiert und begründet, welche Eigenschaften dieser haben muss. Anschließend wird beschrieben, wie die Suche und die Auswahl des Anwendungsfall erfolgt ist.

4.1.1 *Auswahlkriterien*

Die Webseite muss einige Kriterien erfüllen. Im Folgenden werden die Kriterien mit jeweils einer Begründung aufgelistet.

- Die Webseite muss eine monolithische Architektur besitzen. Dieses Kriterium wird durch die Forschungsfrage definiert.
- Das Projekt muss in englischer oder deutscher Sprache entwickelt worden sein. Dadurch soll keine sprachliche Barriere entstehen, die die Überführung deutlich schwieriger gestalten würde.
- Die Webseite muss ein JavaScript-Framework besitzen. Dies muss entweder React oder Angular sein. Diese Eingrenzung ergibt sich aus den genannten zeitlichen Aspekten. Des Weiteren ist nicht jedes JavaScript-Framework einfach überführbar. In der Arbeit [Lan22] wird gezeigt, dass die Bibliotheken des JavaScript-Frameworks Ember.js[1] nicht kompatibel mit clientseitigen Implementationen sind.
- Die Webseite muss für ein Client-Side Composing (siehe Abschnitt 2.1.2) geeignet sein. Dieses Kriterium ergibt sich aus der Abgrenzung in Abschnitt 1.3.
- Die Webseite sollte mindestens drei Unterseiten beinhalten. Des Weiteren sollte sie mindestens drei augenscheinlich gut voneinander zu unterscheidende Komponenten besitzen. Mit diesen zwei Einschränkungen soll gewährleistet sein, dass eine vertikale sowie horizontale Zerlegung gut vorstellbar ist. Dies soll zusätzlich gewährleisten, dass die Webseite in mehrere MFs zerlegt werden kann.

1 https://emberjs.com/ zuletzt eingesehen am 15.08.2023

- Für eine weitere Eingrenzung muss das Projekt Webpack 5 unterstützen. Somit besteht bei der Entscheidung für die Implementierung der MF-Architektur die Möglichkeit, Module Federation zu verwenden (Siehe Unterabschnitt 2.1.2).
- Die Webseite muss ein Backend für eine API besitzen. Wie auch im letzten Kriterium geht es hierbei darum, die Überführung an einem praxisnahen Beispiel durchzuführen.
- Das Backend muss ohne größere Vorbereitungen sofort ausführbar sein. Dies kann beispielsweise durch eine Docker[2]-Umgebung oder ein im Internet öffentliches Backend (API) gelöst werden.
- Der Code des Frontend sollte muss für die Überführung entsprechend überführbar sein. Wie im Unterabschnitt 3.1.3 beschrieben, sollte die Webseite ein geringes Maß an Kopplung besitzen.

4.1.2 *Vorgehensweise bei der Suche und Auswahl*

Ausgehend von den Auswahlkriterien wurden folgende Ausdrücke erstellt, um öffentliche Projekte auf GitHub[3] zu finden. Das Datum ergibt sich aus dem Veröffentlichungsdatum der Framework-Version, die zum ersten Mal Webpack 5.0 unterstützt[4].

- *topic:react topic:website language:TypeScript created:>2020-10-20*
- *topic:angular topic:website language:TypeScript created:>2021-05-13*

Die Projekte wurden entsprechend der Kriterien gefiltert. Zur Bewertung der Auswahlkriterien wird der Code der Webseiten analysiert. Dabei sticht das Projekt "Angular-realworld-example-app"[5] hervor. Es gehört zu einem Verbund von Projekten, die unter dem "Realworld"-Projekt[6] stehen. Dies ist eine große Open-Source-Community, die für einen Anwendungsfall eine Vielzahl an unterschiedlichen Lösungen mit verschiedenen Frameworks präsentiert. Die Community hat viele Tausend Mitglieder, darunter auch den Gründer des State-Management-Systems Redux. Der Anwendungsfall für die Vielzahl an Projekten soll so nah wie möglich an einer realen Webseite sein [Ani+23]. Der Anwendungsfall wird im nächsten Unterabschnitt 4.2.1 beschrieben.

Das genannte Projekt aus der Community ist nach den in der Angular-Dokumentation vorgelegten Standards entwickelt. Es erfüllt alle Auswahlkriterien und bietet darüber hinaus Authentizität der Realworld-Community. Es ist der finale Kandidat, an dem beide Konzepte aus Kapitel 3 praktisch umgesetzt werden.

2 https://www.docker.com/ zuletzt eingesehen am 15.08.2023
3 https://github.com/ zuletzt eingesehen am 15.08.2023
4 Angular: 13.05.2021 [Tho21] | React: 20.10.2020 [AN20]
5 https://github.com/khaledosman/angular-realworld-example-app zei. am 15.08.2023
6 https://www.realworld.how/ zuletzt eingesehen am 15.08.2023

4.2 BESCHREIBUNG DER WEBSEITE

Der Anwendungsfall für die Fallstudie wird in dem folgendem Abschnitt beschrieben. Es wird zunächst allgemein die Verwendung und den Nutzen der Webseite zusammengefasst. Anschließend werden für die Überführung technische relevante Inhalte kurz beschrieben.

4.2.1 *Allgemeine Informationen*

Die Realworld-Community orientiert sich bei ihrem Anwendungsfall an der Webseite medium.com[7]. Dies ist eine Blogger-Webseite, auf der jeder registrierte Client einen eigenen Blog-Artikel schreiben kann. Inhaltlich möchte die Webseite Wissen mit diesen Blog-Artikeln austauschen und verbreiten. Durch die Möglichkeit zu Kommentieren soll eine Diskussion möglich sein. Nicht zuletzt wird Medium.com auch von vielen Entwickler*innen als Plattform genutzt, um Tech-Blog-Artikel zu verfassen. Der Projektname des Anwendungsfalls den die Realworld-Community entwickelt hat nennt sich Conduit. In Abbildung 4.1 sind die beiden Webseiten medium (links) und conduit (rechts) nebeneinander zu sehen. Die Webseite Conduit bietet eine Vielzahl an öffent-

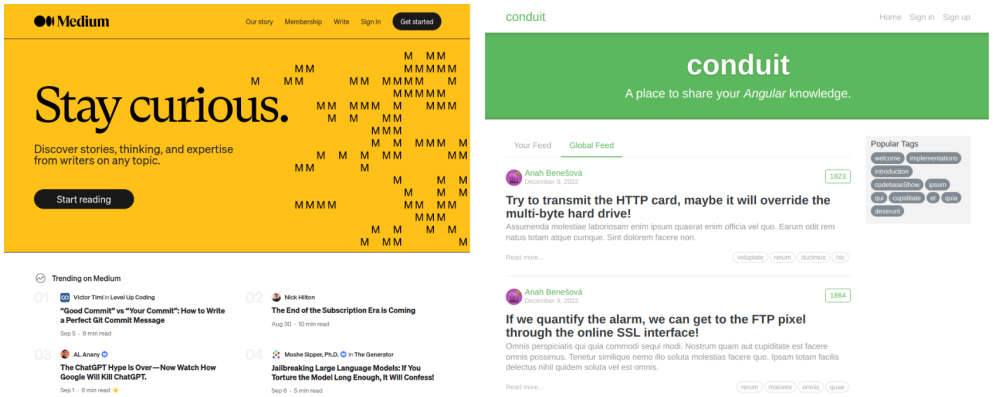


Abbildung 4.1: Der Anwendungsfall der Realworld-Community (rechts) ist an die Webseite Medium.com (links) angelehnt

lich einsehbaren Artikeln. Da es sich nur um eine Beispielwebseite handelt, ist der Inhalt lediglich ein Lorem ipsum Text. Des Weiteren kann öffentlich das Profil des Autors und seine weiteren geschriebenen Artikel eingesehen werden. Um in den geschützten Bereich der Webseite zu gelangen, gibt es eine Anmelde- und Registrierungsseite. Ist der Client angemeldet, gelangt dieser in den geschützten Bereich. Dort kann der Client eigene Blog-Artikel verfassen und bearbeiten, anderen Clients folgen, um Neuigkeiten von ihnen zu erhalten, sowie Kommentare unter Blog-Artikeln schreiben. Des Weiteren kann der Client seine persönlichen Daten auf der Profil-Seite einsehen und ändern.

7 https://medium.com zuletzt eingesehen am 12.09.2023

Insgesamt gibt es folgende sechs Unterseiten auf der Webseite Conduit:

- Hauptseite mit einer Übersicht über alle Artikel (öffentlich)
- Artikelseite mit dem Inhalt eines Artikels (öffentlich)
- Anmelde- und Registrierungsseite (öffentlich)
- Profilseite (geschützt)
- Einstellungsseite (geschützt)
- Artikel-Editorseite (geschützt)

Die Navigation erfolgt über die Reiter der Navigationsleiste. Diese ist auf jeder Seite sichtbar. Es werden weitere Reiter zu geschützten Unterseiten angezeigt, sobald der Client sich angemeldet hat.

4.2.2 *Technische Beschreibung*

Die technische Beschreibung zeigt die Webseite aus Sicht des Entwickler*innens. Dabei wird sich auf den für die Konzepte relevanteren Inhalt konzentriert. Das Projekt bezieht sich dabei auf den Stand vom 28.06.2023. Die Webseite wird mit dem JavaScript-Framework Angular 15 entwickelt. Als State-Management wird Redux verwendet. Als CSS-Framework wird Bootstrap[8] verwendet. Das gesamte CSS wird zur Laufzeit von einem Server geladen[9]. Dies garantiert, dass bei allen Realworld-Community-Projekten das gleiche CSS verwendet wird.

Die Webseite kommuniziert mit der `https://api.realworld.io/api` API. Diese ist öffentlich über das Internet erreichbar und liefert Informationen über die Artikel und deren Inhalt sowie über Profile. Um auf geschützte Seiten und API-Endpunkte zugreifen zu können, benötigt der Client ein Authentifizierungstoken (JWT). Ist die Anmeldung erfolgreich, sendet die API diesen Token an den Client zurück. Der JWT wird anschließend im Storage gespeichert.

Des Weiteren wird bei einem erfolgreichen Anmeldevorgang ein Ereignis über ein *ReplaySubject*[10] an die Angular-Komponenten gemeldet. *ReplaySubjects* sind erweiterte Subjects des Observer-Patterns. Diese speichern zusätzlich den letzten emittierten Wert und geben diesen sofort an neue Observer, die sich am Subject anmelden. Die Angular-Komponenten, die sich auf das *ReplaySubject* registriert haben, reagieren darauf und zeigen entsprechende Inhalte an, wie beispielsweise den Abmelde-Button (siehe Abbildung 4.5).

8 https://getbootstrap.com/ zuletzt eingesehen am 16.08.2023
9 *demo.productionready.io/main.css* zuletzt eingesehen am 22.08.2023
10 *https://www.learnrxjs.io/learn-rxjs/subjects/ReplaySubject* zuletzt eingesehen am 17.08.2023

4.3 DIE ÜBERFÜHRUNG DER WEBSEITE

Die Überführung der Webseite von einem monolithischen Web-Frontend in eine MF-Architektur erfolgt in zwei großen Abschnitten. Im ersten Abschnitt wird die MF-Architektur anhand der verschiedenen Entscheidungen aus dem Konzept (siehe Kapitel 3) definiert. Der Monolith wird anschließend durch die definierte MF-Architektur ersetzt.

4.3.1 *Entscheidungen aus dem Konzept*

Vor der eigentlichen praktischen Überführung wird zunächst zu jeder Entscheidung aus dem Konzept der MF-Architektur (Abschnitt 3.1) eine Option ausgewählt. Im Folgenden werden Entscheidungen zu den Punkten Zerlegung, Composing, Routing, Kommunikation und Implementierung getroffen.

Zerlegung

Für diese Entscheidung müssen die Domänen identifiziert werden. Conduit besitzt sechs klar voneinander getrennte Unterseiten, an denen die Domänen definiert werden. Eine weitere Zerlegung ist nur für die Navigationsleiste anwendbar, da sie die einzige Komponente ist, die über mehrere Unterseiten verwendet wird ist. Da sie allerdings auf jeder Unterseite ohne weitere Bedingungen sichtbar ist, kann sie ebenfalls in die Shell hinzugefügt werden. Somit fällt die Entscheidung, wie auch von der Empfehlung des Konzeptes auf eine vertikale Zerlegung.

Das Projekt wird durch die Eingrenzung der Domänen anhand der Unterseiten in sechs MFs unterteilt.

Composing

Das Composing wurde bereits mit den Auswahlkriterien aus Abschnitt 4.1.1 auf das Client-Side Composing festgelegt. Nichtsdestotrotz werden im Folgenden die Merkmale der Webseite für die Entscheidung aufgezeigt.

Da die Webseite persönliche Inhalte in geschützten Bereichen besitzt, ist dies ein großes Indiz für ein Client-Side Composing. Die persönlichen Inhalte wie die E-Mail-Adresse ändern sich nicht häufig, wodurch sie einfach clientseitig zwischengespeichert werden können. Des Weiteren sind bereits Prozesse in der Webseite implementiert, über die mit Hilfe der API Informationen clientseitig geladen werden können. Für eine Überführung bietet sich daher ein Client-Side Composing an.

Routing

Die Entscheidung, welches Routing gewählt wird, ergibt sich aus dem bereits gewählten Composing. Angular bietet mit dem NPM-Paket *@angular/router*[11] eine einfache Möglichkeit, clientseitig zwischen Komponenten zu routen.

11 https://www.npmjs.com/package/@angular/router zuletzt eingesehen am 17.08.2023

Kommunikation

Die Webseite verwendet bereits den Storage, um den JWT zu speichern. Der Storage wird weiterhin hierfür verwendet. Ebenfalls werden bereits Query Strings eingesetzt, um einen Artikel auf der Artikelseite zu beschreiben. Im Folgenden ist ein Beispiel für eine URI eines Artikels:

```
http://localhost:4200/article/My\_First\_article\-194554
```

Des Weiteren kommuniziert der Monolith über ein *ReplaySubject* in einem Service mit allen Angular-Komponenten. Diese Art der Kommunikation soll weiterhin verwendet werden. Hierfür gibt es in Angular spezielle Funktionen, um über mehrere MFs hinweg mit einem Service zu kommunizieren.

Diese Erkenntnisse werden in die Tabelle 4.1 überführt. Sie werden mit Informationen ergänzt aus welchen MFs diese Informationen stammen und wann diese sich ändern.

Tabelle 4.1: Dokumentation der Daten, die MFs über die Technologien kommunizieren

Micro-Frontend	**Technologie**	**Daten**	**Szenario**
Authentication	Storage	JWT (string)	An- und Abmeldung
Authentication	ReplaySubject	authenticated (boolean)	An- und Abmeldung
Article	QueryString	article-id (integer)	slug / Routenaufruf

Implementierung

Die Auswahl einer Technologie zur Implementierung eines Client-Side Composings wird mit Hilfe der Tabelle A.1 durchgeführt. Folgende Kriterien zur Eingrenzung der Auswahl wurden ausgewählt:

- Das Framework muss für ein Client-Side Composing geeignet sein (siehe Abgrenzung aus Abschnitt 1.3).
- Dokumentation: Die Dokumentation muss mit “good“ bewertet sein. Eine gute Dokumentation verspricht schnelle Erfolge bei der Umsetzung. Dies ist aufgrund des zeitlichen Rahmens unabdingbar.
- Angular Support: Die Technologie muss Angular unterstützen. Dieses Kriterium geht aus dem Ist-Zustand der Webseite hervor.

Aus der Liste bleiben drei Technologien übrig: Web Components, Single-spa und Module Federation. Wie in Abschnitt 2.1.2 beschrieben, können diese Technologien auch zusammen eingesetzt werden. Für den potenziellen Einsatz wird der jeweilige Nutzen der Technologie zusammengefasst.

- Web Components: Der Vorteil der nativ unterstützten Isolierung von Komponenten wird in Conduit nicht benötigt, da bereits Angular als zusätzliches JavaScript-Framework zur Isolierung der Komponenten eingesetzt wird.

- Single-spa: In Conduit wird nur ein JavaScript-Framework eingesetzt. Die Funktionalität von Single-spa, mehrere JavaScript-Frameworks einfach zu verwalten, wird somit nicht benötigt. Das Routen wird ebenfalls von Angular übernommen.
- Module Federation: Die Webseite nutzt verschiedene Shared Libraries. Darüber hinaus werden einige externe Bibliotheken verwendet, die in vielen MFs verwendet werden. Um zu verhindern, dass diese Bibliotheken mehrfach auf dem Client geladen werden, ist der Einsatz von Module Federation sinnvoll.

Anhand der Filterung der Technologien wird sich für den Einsatz von Module Federation entschieden. Weitere Technologien zur Umsetzung der MF-Architektur werden nicht eingesetzt.

Zusammenfassend wird die MF-Architektur für die Webseite Conduit wie folgt aufgebaut:

- Zerlegung: vertikal
- Composing: Client-Side-Composing
- Routing: Clientseitiges Routing
- Kommunikation: QueryStrings, Custom Events und Storage
- Implementierung: Module Federation

4.3.2 *Umsetzung*

Die Umsetzung der beschriebenen MF-Architektur erfolgt durch das Vorgehen im Konzept (siehe Unterabschnitt 3.1.2). Die folgende Beschreibung führt alle für die MF-Architektur wichtigen Schritte zur Umsetzung auf. Für die Arbeit nicht relevanten Aspekte wie beispielsweise das Aufräumen von Code und Ordnern oder die Konfiguration von Angular für die Entwicklung werden nicht explizit beschrieben. Die Änderungen werden dennoch durchgeführt und können schrittweise in den Commits Commits[12] auf dem GitHub-Repository nachvollzogen werden. Der vollständige Code zum gesamten Projekt ist auf GitHub[13] zu finden.

Projekterstellung und Bau des Grundgerüst

Angular stellt eine CLI zur Verfügung, die für die Überführung verwendet wird. Sie wird mit dem Befehl `ng` aufgerufen. Das Projekt wird mit folgendem Befehl erstellt:

```
ng new Conduit-mf --defaults
```

12 https://github.com/Xerathron/Conduit-mf/commits/master zuletzt eingesehen am 06.09.2023

13 https://github.com/Xerathron/Conduit-mf zuletzt eingesehen am 06.09.2023

Dieser Befehl erstellt viele Dateien, die für die Entwicklung und das JavaScript-Framework, jedoch nicht für die MF-Architektur relevant sind. Im Folgenden wird die Funktionsweise der für die MF-Architektur relevanten Dateien erklärt.

- package.json: Diese JavaScript Object Notation (JSON)-Datei enthält viele Informationen zum Projekt. Für die Arbeit sind zwei Attribute wichtig. `scripts` beinhaltet Kommandozeilen-Befehle, mit denen beispielsweise Webserver zum Entwickeln lokal auf dem Rechner gestartet werden. `dependencies` beinhaltet eine Liste von Abhängigkeiten zu Drittsoftware. Dabei gibt es eine Unterscheidung zwischen normalen Abhängigkeiten und solchen, die nur zur Entwicklung benötigt werden (zum Beispiel Linter, der mittels statischer Codeanalyse für die Softwarequalität verantwortlich ist).
- angular.json: Zum Starten einer Angular-Applikation verwendet die CLI die angular.json-Datei. In ihr werden viele Informationen angegeben, wo bestimmte Dateien wie die webpack-Konfigurationsdatei abgelegt sind. Die angular.json-Datei enthält außerdem Informationen über den verwendeten Webserver und dessen Konfiguration (Port und Adresse etc.).

Nachdem das Projekt angelegt ist, müssen die benötigten Abhängigkeiten hinzugefügt werden. Dafür werden einmalig manuell alle Abhängigkeiten, die in der `package.json`-Datei der Conduit-Webseite vorhanden sind, kopiert und in die jeweilige Liste (“dependencies“ beziehungsweise “devDependencies“) eingetragen. Anschließend werden alle Abhängigkeiten installiert.

```
yarn install
```

Bis hierhin ist eine einfache monolithische Webseite aufgesetzt. Für die Implementierung der Shell wird diese als ein neues Unterprojekt angelegt. In dem Shell-Unterprojekt muss anschließend noch eine webpack.config.js-Datei erstellt werden. Dies ist die Konfigurationsdatei für Module Federation, welche im Unterabschnitt 2.1.2 erläutert wurde. Der vollständige Inhalt der webpack.config.js-Datei der Shell ist im Anhang A.5 zu finden.

```
ng generate application shell --defaults
touch ./projects/shell/webpack.config.js
```

Listing 4.1: Kommandozeilenbefehle zum Erstellen der Shell-Applikation

Die Ordnerstruktur des Projekts ist nun wie folgt:

```
/Conduit-mf
├── projects
│   └── shell
│       ├── src
│       └── webpack.config.js
├── angular.json
└── package.json
```

Abbildung 4.2: Ordnerstruktur der MF-Architektur des Anwendungsfalls (vereinfacht)

Der Code nach diesem Überführungsschritt ist in GitHub zu finden. Nach diesen Änderungen kann die Webseite das erste mal ausgeführt werden. Der Webserver wird durch den Befehl `ng serve shell` in der Kommandozeile gestartet. Es ist lediglich die Standardansicht zu sehen, die die CLI erstellt hat.

Geteilte Ressourcen

Zunächst muss die Infrastruktur für die Shared Libraries geschaffen werden. Mithilfe der CLI kann einfach die Shared-Komponente erstellt werden.

```
ng generate library shared
```

Listing 4.2: Kommandozeilenbefehle zum Erstellen einer Shared-Komponente

Anschließend ist in dem `projects`-Ordner ein weiterer Ordner mit dem Namen "shared" zu finden. Ressourcen, die von mehreren MFs verwendet werden, können hier abgelegt werden.

Im Conduit-Projekt ist sämtlicher Code, der unter den Ordnern "shared" und "core" liegt, geteilter Code, der in mehreren MFs verwendet wird. Unter diesen "components" fallen zum Beispiel die Navigationsleiste und der Footer. Dieser Code wird als eine Shared Library ausgelagert. Der Code kann nicht weiter aufgeteilt werden, da die Abhängigkeiten innerhalb der Shared Library zu groß sind.

Des Weiteren wird das CSS von dem CDN heruntergeladen und in die gleiche Shared Library überführt. Damit das CSS beim Bauen der Shared Libraries mit berücksichtigt wird, muss der Pfad zu der CSS-Datei in der Datei `ng-package.json` unter dem Attribut *assets* angegeben werden. Den gesamten Code, der in die Shared Libraries überführt worden ist, ist auf GitHub zu finden.

Abschließend wird die Shared Library durch folgenden Befehl in Code 4.3 transpiliert.

```
ng build shared
```

Listing 4.3: Kommandozeilenbefehle zum Transpilieren der Shared-Komponente

Implementierung Micro-Frontends

Conduit-mf besitzt aktuell nur die Shell und alle Shared Libraries. Aufbauend darauf werden die MFs sukzessive eingebunden. Der Prozess wird an dem MF, welches die Hauptseite darstellt, vorgestellt. Anschließend wiederholt sich der Prozess für die weiteren fünf MFs.

Wie bei der Shell muss im ersten Schritt ein neues Unterprojekt angelegt und anschließend eine webpack.config.js-Datei erstellt werden.

```
ng generate application mf-home --defaults
touch ./projects/mf-home/webpack.config.js
```

Listing 4.4: Kommandozeilenbefehle zum Erstellen des Hauptseiten MFs

Anschließend wird der Code der Unterseite “Home“ aus dem originalen Conduit-Projekt überführt. Anschließend muss die Konfigurationsdatei von Webpack angepasst werden. Details zu den einzelnen Eigenschaften sind in Unterabschnitt 2.1.2 beschrieben. Die zwei wichtigsten Eigenschaften sind `exposes` und `shared`.

In `exposes` wird das Home-Modul des eigenen MFs angegeben. Dies ist ein Angular-Modul, über das die Unterseite angezeigt wird. In diesem Modul werden zwei weitere Module importiert. Das erste Modul ist ein Routing-Modul. Dort können Einstellungen zum Routing getroffen werden. Das zweite Modul lädt die von dem Home-MF benötigten Shared Library.

```
@NgModule({
  imports: [
    SharedModule,  // Shared-Komponente
    HomeRoutingModule, // Routing Modul
  ],
  declarations: [
    HomeComponent // Deklaration der Komponente, die die Unterseite anzeigt
  ],
})
export class HomeModule {}
```

Listing 4.5: Angular home.module.ts Code, welcher zwei weitere Module importiert

In dem Attribut `shared` der Webpack-Konfigurationsdatei werden die für das Home-MF benötigten Ressourcen angegeben. Hierunter fallen einerseits die Abhängigkeiten zu externen NPM-Paketen, die über `...deps` hinzugefügt werden. Diese werden mithilfe des Codes `const deps = require('../../package.json').dependencies;` aus der `package.json` geladen. Andererseits die Shared Libraries, die das Home-MF benötigt.

```
exposes: {
    HomeModule: './projects/mdmf-home/src/app/home/home.module.ts',
},
shared: {
    ...deps,
    'shared': {
        import: 'shared',
        requiredVersion: require('../shared/package.json').version,
    },
},
```

Listing 4.6: webpack.config.js: Ausschnitt der Module Federation Konfigurationsdatei, welche den Code von 4.5 veröffentlicht und Ressourcen anfordert

Als letzten Schritt wird die Standard-Komponente (app.component.html) ausgetauscht. Dies ist eine von der CLI automatisch generierte Angular-Komponente. Das HTML sollte durch folgenden Code ausgetauscht werden:

```
<div style="background-color: #666;"> <!-- optional -->
    <app-layout-header></app-layout-header>
    <router-outlet></router-outlet>
    <app-layout-footer></app-layout-footer>
</div>
```

Listing 4.7: HomeAppModule HTML für den direkten Zugriff auf das MF

Im Normalbetrieb greift die Shell über das `exposed` Modul auf das MF zu (siehe Abbildung 4.3). Wird aber direkt der Webserver des MFs aufgerufen, nimmt er als ersten Zugangspunkt die *HomeAppModule* Komponente. Damit können Entwickler*innen zur Entwicklungszeit das MF auch ohne die darum liegende Shell im Browser aufrufen. Der Vorteil ist eine vollständig voneinander getrennte Entwicklung, sodass keine ungewollte Kopplung entsteht. Des Weiteren können Programmierfehler zielgerichteter identifiziert und behoben werden, da die Einwirkung von Seiteneffekten anderer MF über die Kommunikationsschnittstellen oder der Shell ausgeschlossen sind. Es garantiert auch, dass das MF ohne die Abhängigkeit anderer MFs fehlerfrei funktioniert. Um dem/der

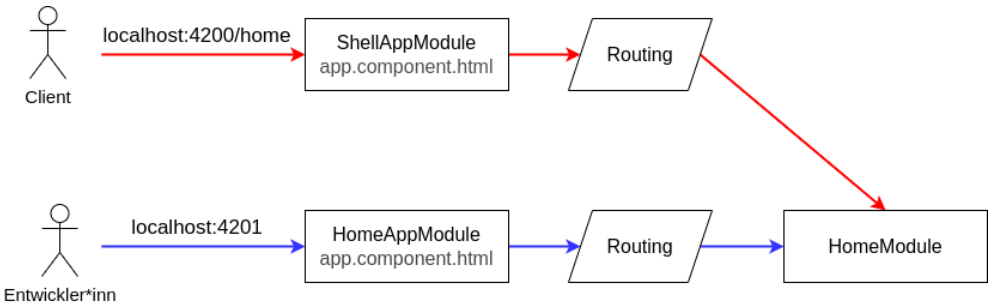


Abbildung 4.3: Regulärer Programmfluss (rot) und der Programmfluss mit direktem Zugriff auf das Home-MF während der Entwicklung (blau)

Entwickler*in zu verdeutlichen, dass er/sie direkt auf das MF zugreift, kann im *HomeAppModule* zusätzlich ein HTML-Div mit einer abweichenden Hintergrundfarbe implementiert werden. Die Abbildung 4.4 zeigt die beiden unterschiedlichen Ansichten. Links greift der/die Entwickler*in über die Shell auf das MF zu. Es wird der Header aus der Shell genommen und nur das *HomeModule* aus dem Code 4.5 in die Shell eingebunden. Im rechten Bild wird über das *HomeAppModule* das MF direkt aufgerufen. Dadurch, dass das *HomeAppModule* den Hintergrund dunkler einfärbt (siehe Code 4.7), bemerkt der Entwickler direkt, dass er das MF direkt aufruft, ohne umhüllende Shell. Der gesamte

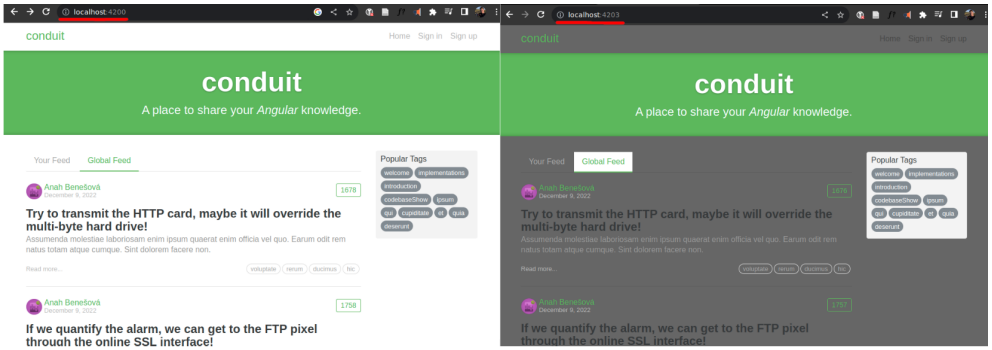


Abbildung 4.4: Links Zugriff über die Shell (localhost:4200). Rechts Zugriff über das Home-MF (localhost:4203)

Code für die Umsetzung des Home-MFs ist auf GitHub zu finden. Dieser Vorgang wird für jedes MF wiederholt, sodass anschließend sieben Unterprojekte vorhanden sind. Davon sind sechs MFs plus die Shell als weiteres Unterprojekt.

Composing und Routing

Bislang sind die MFs und die Shell einzelne, isolierte Projekte. Das Zusammenfügen der MFs erfolgt, wie Abschnitt 4.3.1 erläutert, auf der Clientseite. Das Zusammenfügen erfolgt in den Routern. Router entscheiden auf Basis der angeforderten URI, welche Komponente als nächstes aufgerufen werden soll. Dies stößt wiederum Prozesse zum Nachladen und Einbinden von Code an. Das Ziel ist es, diese Prozesse zu implementieren.

Zunächst benötigt die Shell die Möglichkeit, die MFs nachzuladen, einzubinden und wieder auszubinden. Module Federation bietet in seiner Dokumentation bereits Code an[14]. Der Code ist ebenfalls in Anhang A.6 enthalten. Mit der im Code bereitgestellten Funktion `loadRemoteModule(options)` ist es möglich, MFs nachzuladen und automatisch einzubinden. Das Nachladen und Einbinden erfolgt über Script-Tags.

Die Entscheidung, welches MF bei welcher URI aufgerufen wird, erfolgt über die von Angular zur Verfügung gestellten Funktionalitäten zum Routen. Dort werden wie in Code 4.8 gezeigt die Routen definiert. Trifft der Pfad der Route auf die URI zu, wird die Funktion `loadRemoteModule` durch den Prozess `loadChildren` aufgerufen.

Für alle sechs Unterseiten muss entsprechend ein Routingeintrag in der Routing-Tabelle 4.8 existieren. Dies sind die Top-Level-Routen, die zu den Routen die dazugehörigen MFs nachladen und einbinden. In den MFs sind ebenfalls Router, die die Second-Level-Routen übernehmen.

```
1  {
2      // Leerer Pfad wird unter localhost:4200/ aufgerufen.
3      path: '',
4      pathMatch: 'full',
5      loadChildren: () => loadRemoteModule({
6          remoteEntry: "http://localhost:4203/remoteEntry.js",
7          remoteName: "home",
8          exposedModule: "HomeModule",
9      }).then(m => m.HomeModule)
10 },
```

Listing 4.8: (Shell) app-routing.module.ts) Routing der Hauptseite beziehungsweise des MFs “Home“

Nachdem die Routingeinträge vollständig sind, muss die Module Federation Konfigurationsdatei der Shell angepasst werden. Module Federation muss bekannt gemacht werden, über welche URI die einzelnen MFs erreichbar sind. Hierzu müssen im `remote`-Attribut der Konfigurationsdatei (webpack.config.js) alle sechs MFs angegeben werden (siehe Code 4.9). Der Name des Schlüssels des Objekts muss mit dem Wert des `remoteName`-Feldes aus dem Code 4.8 übereinstimmen.

14 https://webpack.js.org/concepts/module-federation/ zuletzt eingesehen am 22.08.2023

```
remotes: {
    profile: 'profile@http://localhost:4201/remoteEntry.js',
    authentication: 'authentication@http://localhost:4202/remoteEntry.js',
    home: 'home@http://localhost:4203/remoteEntry.js',
    article: 'article@http://localhost:4204/remoteEntry.js',
    settings: 'settings@http://localhost:4205/remoteEntry.js',
    editor: 'editor@http://localhost:4206/remoteEntry.js',
},
```

Listing 4.9: ((Shell) webpack.config.js) Auszug des remote Attributs

Des Weiteren müssen, wie auch in allen MFs, die Abhängigkeiten der Shell in das Attribut `shared` eingetragen werden. Aufgrund eines Bugs muss jedes NPM-Paket, das verschachtelt weitere NPM-Pakete implementiert (das heißt mehrere `package.json`-Dateien besitzt), explizit in die Liste eingetragen werden. Andernfalls kann Module Federation diese verschachtelten Abhängigkeiten nicht auflösen. Abbildung 4.10 zeigt, wie sich der Fehler äußert.

```
main.js:604 Unhandled Promise rejection: Shared module is not available for
    eager consumption: 733
    at __webpack_require__.m.<computed> (http://localhost:4200/main.js:480:54)
    at __webpack_require__ (http://localhost:4200/main.js:34:42)
    at 4987 (http://localhost:4200/vendor.js:6238:63)
    at __webpack_require__ (http://localhost:4200/main.js:34:42)
    at 1353 (http://localhost:4200/vendor.js:6873:79)
    at __webpack_require__ (http://localhost:4200/main.js:34:42)
    at 1308 (http://localhost:4200/vendor.js:6780:67)
    at __webpack_require__ (http://localhost:4200/main.js:34:42)
```

Listing 4.10: Navigationsleiste (Shell) und Unterseite (Home-MF) zeigen andere Inhalte an, wenn der Client angemeldet ist

Um den Fehler zu beheben, müssen die NPM-Pakete *@angular/common* und *@angular/common/http* explizit in die Liste des `shared`-Attributes eingetragen werden. Für weitere Informationen zu dem Fehler siehe GitHub[15].

```
shared: {
    ...deps,
    // bugfix for https://github.com/webpack/webpack/issues/13457
    "@angular/common": {singleton: true, eager: true},
    "@angular/common/http": {singleton: true, eager: true},
}
```

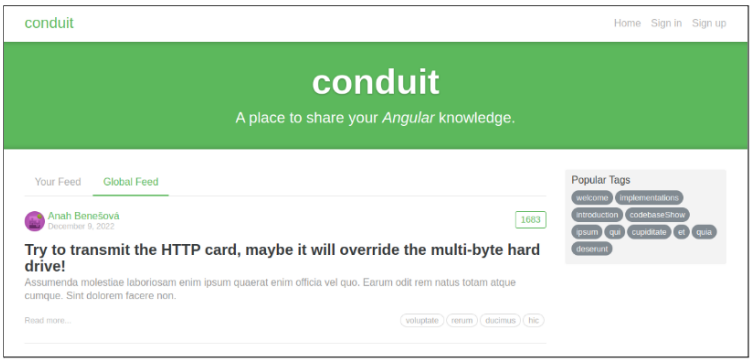
Listing 4.11: ((Shell) webback.config.js) Auszug des remote Attributs

Für die Shell muss ebenfalls die Standardkomponente, wie in Code 4.7 gezeigt, ausgetauscht werden. Anschließend ist es möglich, von der Shell aus (http://localhost:4200) alle Unterseiten aufzurufen. Die MFs werden lazy nachgeladen und eingebunden. Der Code nach diesen Schritten ist auf GitHub[16] zu finden.

15 https://github.com/webpack/webpack/issues/13457 zuletzt eingesehen am 21.08.2023
16 https://github.com/Xerathron/Conduit-mf/commit/a2258e920345e7f1926db37cbd09006082698280 zuletzt eingesehen am 07.09.2023

Kommunikation

Wenn sich ein Client anmeldet oder abmeldet, wird dies bislang durch ein *ReplaySubject*, welches in einem Service gespeichert ist, an alle Komponenten bekannt gemacht. Die Abbildung 4.5 zeigt, wie die Shell und das Home-MF andere Inhalte anzeigen, abhängig davon, ob ein Client angemeldet ist oder nicht. Damit sowohl die Shell als auch das Home-MF entsprechend auf eine Anmeldung beziehungsweise Abmeldung eines Clients reagieren können, müssen beide auf das gleiche *ReplaySubject* hören. Der *UserService* speichert und



Nicht angemeldet

angemeldet

Abbildung 4.5: Navigationsleiste (Shell) und Unterseite (Home-MF) zeigen andere Inhalte an, wenn der Client angemeldet ist

verwaltet den *ReplaySubject*. Ein Service kann als Instanz in andere Angular-Komponenten mittels Dependency Injection eingebunden werden. Das An- und Abmelden erfolgt über Funktionen im *UserService*. Nachdem das An- oder Abmelden erfolgreich ist, emittiert der im *UserService* gespeicherte *ReplaySubject* den aktuellen Zustand (angemeldet oder abgemeldet) an alle registrierten Subjekte. Die Abbildung 4.6 zeigt bildlich, wie beide Unterprojekte auf die gleiche Instanz des *UserService* zugreifen. Es ist zu erkennen, dass der *UserService* nur eine Instanz in der gesamten Webseite besitzen darf, da ansonsten nicht beide Subjekte im Falle einer Zustandsänderung benachrichtigt werden.

Um Dependency Injection in Angular zu verwalten, gibt es sogenannte Injectables (Code 4.12). In ihnen können Eigenschaften definiert werden, wie zum Beispiel Instanzen verwaltet werden sollen. Laut der Dokumentation von Angular gibt es dazu ein InjectableType, mit dem genau eine Instanz des Services für die gesamte Webseite verfügbar gemacht wird [Ric23].

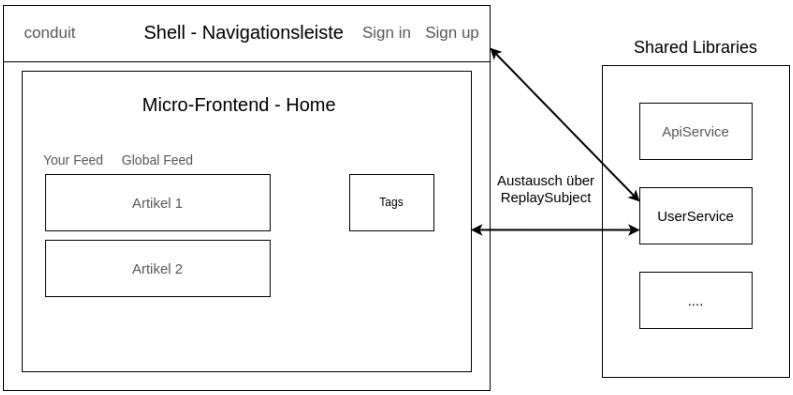


Abbildung 4.6: Beide Unterprojekte (Shell und MF) greifen auf die gleiche Instanz des *UserService* zu

“ ’platform’: A special singleton platform injector shared by all applications on the page.”

```
1 @Injectable({
2     providedIn: 'platform',
3 })
4 export class UserService {
5 ...
6 }
```

Listing 4.12: ((Shared-Komponente) user.service.ts) Mithilfe von Dependency Injection wird eine Instanz der Klasse *UserService* allen Komponenten verfügbar gemacht.

Allerdings führt die Verwendung des *platform*-Injectors im *UserService* nicht zum gewünschten Ergebnis. Es wird sowohl für die Shell als auch für das aktive MF eine eigene Instanz des *UserService* angelegt. Dies führt dazu, dass sobald sich ein Client an- oder abmeldet, dies entweder nur von der Shell oder dem aktiven MF bemerkt wird. So reagiert nicht die gesamte Webseite auf die Zustandsänderung des Clients. Dadurch werden entsprechende Inhalte für einen angemeldeten oder abgemeldeten Client (siehe Abbildung 4.5) erst nach einem manuellen Neuladen der Webseite angezeigt. Dies ist eine erhebliche Verletzung der UX, da dem Client durch die falsch angezeigten Inhalte ein falscher Zustand der Seite suggeriert wird.

Als Reaktion auf diese Erkenntnis wurde ein bislang ungelöster Bug[17] im Github-Repository von Module Federation eröffnet. Der Fehler ist dort detailliert mit einem gesamten Beispielprojekt zum Reproduzieren beschrieben.

Als aktuellen Workaround werden Custom Events im *UserService* verwendet (Code 4.13). Mit diesem Code benachrichtigen sich die einzelnen *UserService*-Instanzen gegenseitig über Zustandsänderungen durch das An- oder Abmelden eines Clients. Der Code für den Workaround ist auf GitHub[18] zu finden.

17 https://github.com/module-federation/module-federation-examples/issues/3100 zuletzt eingesehen am 22.08.2023

18 https://github.com/Xerathron/Conduit-mf/commit/68ec23aa19eea2eb98076e4f63dfe9b1430d7266 zuletzt eingesehen am 07.09.2023

```
registerListener() {
    document.addEventListener(USER_EVENT_TOKEN, (event: CustomEvent) => {
        const user = event.detail.user;
        // emit information to subjects
        this.currentUserSubject.next(user);
        this.isAuthenticatedSubject.next(!!user);
    });
}

dispatchEvent(user: User) {
    const event = new CustomEvent(USER_EVENT_TOKEN, { detail: { user } });
    document.dispatchEvent(event);
}
```

Listing 4.13: ((Shared-Komponente) user.service.ts) Die Custom Events übermitteln Informationen zwischen den *UserService*-Instanzen

Eine weitere Implementierung der Kommunikation über den Storage sowie über QueryStrings ist nicht notwendig, da diese bereits im Monolithen implementiert worden sind. Diese Implementierung wird in gleicher Weise in der MF-Architektur verwendet.

Nach der Implementation der Kommunikation ist die Überführung abgeschlossen. Parallel zum monolithischen Web-Frontend existiert die für den Client gleich aussehende Webseite implementiert als MF-Architektur.

4.4 UMSETZUNG DER FEHLERBEHANDLUNG

Mit der Fertigstellung der Überführung des Monolithen in eine MF-Architektur sind die beiden Webseiten hinsichtlich ihrer Funktionalitäten identisch. Allerdings entstehen durch die MF-Architektur einige neue potenzielle Fehler, die die Funktionalität der Webseite gefährden könnten. Um im Falle eines Fehlers die Webseite in einem regulären Zustand zu halten, wird im Folgenden die Fehlerbehandlung anhand des Konzepts im Abschnitt 3.2 durchgeführt.

In den ersten drei Schritten (Identifizierung, Analyse, Auswertung) wird die Tabelle erstellt. Anschließend erfolgt mithilfe der Tabelle die Implementation der Fehlerbehandlung im Code.

Identifizierung

Im ersten Schritt werden alle potentiellen Fehler, die während des Nachladens, Ein- und Aushängens von MFs entstehen können, in eine Tabelle hinzugefügt.

Module Federation verwendet zum Nachladen der MFs Script-Tags (siehe Code A.6). Für eine bessere Differenzierung der Fehler wird die Funktion `loadRemoteEntry()` optimiert, sodass anstelle von Script-Tags das NPM-Paket Axios[19] eingesetzt wird. Die Umsetzung im Code ist in Anhang A.7 beschrieben. Anhand der Dokumentation von Webpack, Axios und JavaScript sowie der Analyse des Codes werden Fehler, die entstehen können, identifiziert und in eine Tabelle übertragen (siehe Anhang A.2).

19 Axios ist eine auf XML HTTP-Requests aufbauende Bibliothek. *https://axios-http.com/docs/intro* zuletzt eingesehen am 24.08.2023

Analyse

Die Analyse beginnt mit der Frage, ob der Fehler auf der Webseite Conduit auftreten kann. Um dies zu verstehen, wird die Tabelle zunächst um die Spalte "Szenario" ergänzt. In dieser wird in kurzen Worten ein Szenario beschrieben, in dem der Fehler auf der Webseite auftreten kann. Wird kein Szenario für den Fehler gefunden, wird dieser im Verlauf nicht weiter betrachtet.

Alle anderen Fehler werden nach den Bewertungskriterien der Eintrittswahrscheinlichkeit, Auswirkung und Lösbarkeit bewertet. Die Bewertungskriterien sind in Abschnitt 3.2.3 vorgestellt. Die Bewertung erfolgt durch die Analyse des Codes sowie durch Ausprobieren, wie die Webseite im Fehlerfall reagiert.Die Tabelle nach der durchgeführten Bewertung ist in Anhang A.3 dargestellt.

Auswertung

Die Auswertung definiert das Vorgehen für die Bewertungskriterien der Eintrittswahrscheinlichkeit, Auswirkung und Lösbarkeit. Diese werden speziell für den Anwendungsfall Conduit definiert.

Eintrittswahrscheinlichkeit

- **gering**: Der Fehler wird nicht explizit behandelt. Es gibt lediglich eine allgemeine Fehlerbehandlung und einen allgemeinen Hinweis "Ein Fehler ist aufgetreten. Versuchen Sie es später erneut". Diese Art der Fehlerbehandlung ist gleichzeitig auch das Fallback für alle Fehler, die nicht explizit behandelt werden beziehungsweise Fehler, die nicht spezifiziert sind.
- **mittel und hoch**: Der Fehler wird explizit behandelt.

Auswirkung

- **gering**: Eingeschränkte Funktionen werden für den Client sichtbar deaktiviert. Zusätzlich bekommt der Client einen Warnhinweis über die temporäre Einschränkung der Funktionalitäten.
- **mittel**: Der Client wird zu einer Fehlerseite weitergeleitet. Dort hat dieser die Möglichkeit, durch einen Zurück-Button wieder auf die restlich funktionierende Webseite zurückzunavigieren.
- **hoch**: Der Client wird zu einer Fehlerseite weitergeleitet. Da die Funktionalität der Website vollständig gestört ist, gibt es keine Möglichkeit, zurückzunavigieren.

Lösbarkeit

- **gering**: Es wird keine Fehlerbehandlung durchgeführt.
- **mittel**: Es gibt eine Fehlerbehandlung, die manuell durch den Client oder durch einen automatischen Prozess angestoßen wird. Dies ist abhängig von dem jeweiligen Fehler. Ein Beispiel für einen automatischen Prozess kann eine Wartezeit sein, die bei Ablauf einen erneuten Verbindungsversuch automatisch startet.
- **hoch**: Die Fehlerbehandlung wird sofort nach dem Auftreten und ohne Interaktion mit dem Client durchgeführt. Ist der Fehler dadurch nicht behoben, wird der Client auf eine Fehlerseite weitergeleitet.

4.4.1 *Umsetzung*

Bei der Umsetzung werden die Fehler, wie im Konzept beschrieben, gruppiert, sodass diese anschließend in Komponenten implementiert werden können (siehe dazu die Implementierung zum Fehlerbehandlungskonzept in Abschnitt 3.2.5).

Gruppierung

Für die Übersicht wird die Tabelle in Anhang A.3 nach Fehlern gefiltert, die ein Szenario und eine mittlere bis hohe Eintrittswahrscheinlichkeit besitzen. Wie in der Auswertung definiert wurde, werden Fehler mit einer geringen Eintrittswahrscheinlichkeit nicht explizit behandelt.

Tabelle 4.2: Gekürzte Tabelle mit Fehlern, die ein Szenario besitzen und eine mittlere bis hohe Eintrittswahrscheinlichkeit besitzen

Fehler	**Auswirkung**	**Lösungsversuch**	**Lösbar**
Fehler bei Chunk nachladen	mittel	erneut versuchen	mittel
ERR_NETWORK	hoch	erneut versuchen	mittel
ETIMEDOUT	mittel	erneut versuchen	mittel
ERR_TOO_MANY_REDIRECTS	mittel	erneut versuchen	mittel
ERR_BAD_OPTION_VALUE	hoch	ab- und wieder anmelden	hoch
ECONNABORTED	mittel	erneut versuchen	mittel
Unauthorized (401)	hoch	ab- und wieder anmelden	hoch
Conflict (409)	mittel	erneut versuchen	hoch
TooEarly (425)	mittel	erneut versuchen	hoch
InternalServerError (500)	mittel	erneut versuchen	mittel
BadGateway (502)	mittel	erneut versuchen	mittel
ServiceUnavailable (503)	mittel	erneut versuchen	mittel
GatewayTimeout (504)	mittel	erneut versuchen	mittel
VariantAlsoNegotiates (506)	mittel	erneut versuchen	mittel
InsufficientStorage (507)	mittel	erneut versuchen	mittel

Die Fehler werden anhand der logischen Zusammengehörigkeit gruppiert (siehe Abschnitt 3.2.5). Aus den Gruppen werden dann die Fehlerbehandlungskomponenten entwickelt. Die Gruppen sind wie folgt:

- Serverfehler: Alle Fehler, die durch einen HTTP-Statuscode verursacht werden, der größer oder gleich 500 ist. Für den Client wird sinngemäß eine Nachricht angezeigt, dass ein Fehler auf dem Server aufgetreten ist.
- Netzwerkfehler: Alle Fehler, bei denen keine Antwort vom Server zurückkommt. Hierzu zählen ERR_NETWORK, ETIMEDOUT, ERR_TOO_MANY_REDIRECTS, ECONNABORTED. Der Client bekommt die Nachricht, dass keine Netzwerkverbindung besteht. Er soll dies prüfen und es erneut versuchen.
- Zustandsfehler: Fehler, die durch einen inkonsistenten Zustand des Systems entstehen. Dazu zählt der Chunkfehler von Webpack sowie der Fehler Unauthorized (401). Der Client erhält die Meldung, dass die Webseite eventuell durch einen Programmierfehler in einem ungültigen Zustand ist. Er soll die Webseite neu laden und sich ab- und wieder anmelden. Ein Beispiel für die Implementierung der Fehlerbehandlungskomponente zu dieser Gruppe ist im Anhang A.3.
- Sonstige Fehler: Unter sonstige Fehler sind alle Fehler gruppiert, die nicht in der Liste vorhanden sind. Die Gruppe dient daher als Fallback. Dem Client wird eine allgemeingültige Fehlermeldung angezeigt. Er soll die Webseite neu laden.
  Unter dieser Gruppe fallen auch die Fehler Conflict (409) und TooEarly (425), da aufgrund ihrer hohen Lösbarkeit davon ausgegangen wird, dass der Fehler automatisch behoben werden kann. Ist das nicht der Fall, ist dies genauso unerwartet wie ein nicht aufgelisteter Fehler.

Anschließend wird die Auswertung der oben aufgeführten Tabelle 4.2 erneut für die Komponenten der Fehlerbehebung durchgeführt. Dabei sind die unterschiedlichen Lösungsversuche relevant. Für jeden einzigartigen Lösungsversuch wird eine Komponente entwickelt. Somit ergeben sich aus der Tabelle eine Fehlerbehebungskomponente (erneut versuchen).

4.4.2 *Implementierung*

Nachdem die Komponenten zur Fehlerbehebung und -behandlung definiert sind, werden diese im Code durch fünf Schritte implementiert.

1. Abfangen des Fehlers
2. Fehlertabelle und Routing
3. Implementierung der *RootErrorHandlingComponent*
4. Übertragen und Speichern von Fehlerinformationen
5. Implementieren der Fehlerbehandlungs- und -behebungskomponenten

Wie auch bei der Implementierung der MFs, ist die Durchführung der einzelnen Schritte durch die Commit-Historie in GitHub[20] nachvollziehbar. Ein Diagramm bietet eine Übersicht über die technischen Veränderungen der Webseite in jedem Schritt. Das Diagramm A.4 zeigt den bisherigen Stand. Es werden nur relevante Dateien zum Laden und Verwalten der MFs dargestellt.

Schritt 1: Abfangen des Fehlers

Bislang werden die Routen, wie im Code 4.8 zu sehen ist, statisch in einer Variable gespeichert. Für den Funktionsaufruf `loadRemoteModule()` wird bislang keine Fehlerbehandlung durchgeführt. Damit die folgenden Fehlerbehandlungen nicht für jeden Funktionsaufruf erneut durchgeführt werden müssen, werden die Routen zunächst zur Laufzeit erstellt (siehe Anhang A.8).

Um den Fehler abzufangen, bietet Angular einen *ErrorHandler*[21] an. Mit diesem werden alle Fehler, die nicht vorher durch anderen Code behandelt wurden, abgefangen (siehe Code im Anhang A.9). Neben dem Abfangen der Fehler ist die Aufgabe des *ErrorHandler*s Fehler mit unterschiedlichen Strukturen in eine einheitliche Struktur umzuwandeln. Conduit verwendet für die API-Aufrufe eine Angular eigene Technologie (HttpClient[22]). Diese wird neben Axios, welches zum Laden der MFs verwendet wird, betrieben. Beide Technologien haben unterschiedliche Strukturen, um Fehler zu beschreiben. Im Folgenden wird zunächst die Struktur von HttpClient und anschließend die von Axios gezeigt:

```
1 class HttpErrorResponse extends HttpResponseBase implements Error {
2     name: 'HttpErrorResponse'
3     message: string
4     error: any | null
5     ok: false
6 }
```

Listing 4.14: TypeScript-Code-Ausschnit aus der Angular Dokumentation [Com23] zu *HttpClient*. Struktur der Fehler die bei API-Aufrufen enstehen

```
1 function AxiosError(message, code, config, request, response) {
2     this.message = message;
3     this.name = 'AxiosError';
4     code && (this.code = code);
5     config && (this.config = config);
6     request && (this.request = request);
7     response && (this.response = response);
8 }
```

Listing 4.15: JavaScript-Code-Ausschnit aus dem GitHub-Repository[23] von Axios. Struktur der Fehler die beim Laden von MFs enstehen

20 https://github.com/Xerathron/Conduit-mf/commits/master zei. am 25.08.2023
21 *https://angular.io/api/core/ErrorHandler* zuletzt eingesehen am 25.08.2023
22 https://angular.io/api/common/http/HttpClient zuletzt eingesehen am 10.09.2023
23 https://github.com/axios/axios/blob/v1.x/lib/core/AxiosError.js zei. am 10.09.2023

Die beiden Code-Ausschnitte zeigen, dass die relevante Information zur Identifizierung eines Fehlers bei der Technologie *HttpClient* im Attribut “error“ und bei Axios im Attribut “code“ enthalten ist. Um die Unstimmigkeiten zu bereinigen, werden im *ErrorHandler* die Informationen in folgendes Format vereinheitlicht:

```
export interface Reason extends Error {
    code: string,
}
```

Listing 4.16: TypeScript-Interface. Einheitliches Format, in dem die Information über den Grund des Fehlers enthalten ist.

Nachdem die Fehler in das einheitliche Format überführt sind, wird die richtige Fehlergruppe beziehungsweise Komponente für den Fehler ermittelt. Dies wird benötigt, um die Route zu ermitteln, auf die der *ErrorHandler* abschließend weiterleitet. Diese Funktion wird im nächsten Schritt implementiert.

Das Diagramm zur Übersicht A.5 erweitert sich um die ErrorHandling-Komponente und ein dazugehöriges Modul, in dem diese und weitere zur Fehlerbehandlung gehörende Komponenten registriert werden. Der Code für den ersten Schritt ist auf GitHub[24] zu finden.

Schritt 2: Fehlertabelle und Routing

Im zweiten Schritt werden die auftretenden Fehler zu den jeweiligen behebenden Komponenten abgebildet (siehe Code 4.17). Eine Funktion ermittelt über diese Datenstruktur die jeweilige Komponente für den entstandenen Fehler.

```
const networkErrorGroup: ErrorComponent = {
    errorComponent: NetworkErrorComponent,
    recovery: {
        component: RetryRecoveryComponent,
        button_name: 'Retry now',
    }
}
const AxiosErrorComponentMap: ErrorComponentMapType<AxiosErrorKeys> = {
    ERR_NETWORK: networkErrorGroup,
    ETIMEDOUT: networkErrorGroup,
    ECONNABORTED: networkErrorGroup,
    ERR_FR_TOO_MANY_REDIRECTS: networkErrorGroup,
};
```

Listing 4.17: (error-2-component.map.ts) Ausschnitt der Datenstruktur, bei der Axios-Fehler auf Komponenten abgebildet werden.

Um die ermittelte Komponente aufzurufen, muss auf eine URI navigiert werden. Dafür müssen die Komponenten eine Route erhalten. Im folgenden Code 4.18 ist für die Fehlerbehandlungskomponente “Netzwerkfehler“ die Route definiert. Alle Fehler, die in dieser Gruppe liegen (siehe Code 4.17), werden auf die URI *errornetwork_error* weitergeleitet. Dort wird die *NetworkErrorComponent* ausgeführt.

24 github.com/Xerathron/Conduit-mf/commit/cf361a6a8042950af0b32f00ba968975088e1b44 zuletzt eingesehen am 09.09.2023

```
export const ErrorRoutes: Routes = [
    {
    path: "error",
    component: ErrorRootComponent,
    canActivate: [ErrorGuard],
    resolve: {
        recovery: ErrorRecoveryResolver,
    },

    children: [
        {
        path: "network_error",
        component: NetworkErrorComponent,
        },
    ],
    },
];
```

Listing 4.18: Routing für Fehlerrouten (vereinfacht)

Für die drei weiteren Gruppen (Serverfehler, Zustandsfehler und Sonstige Fehler) muss ebenfalls eine Route angelegt werden. Des Weiteren ist im Code 4.18 zu erkennen, dass alle Routen von einem *ErrorGuard*[25] geschützt werden. Dieser überwacht, dass nur auf die Fehlerseiten zugegriffen werden kann, wenn ein Fehler geworfen wird. Greift der Client manuell auf diese URIs zu, wird er zur Hauptseite weitergeleitet.

Die Abbildung A.6 zeigt das Übersichtsdiagramm nach Abschluss des zweiten Schrittes. Zur Übersicht werden die Module für die reguläre Verwaltung der MFs (blau markiert) ausgeblendet. Der hinzugefügte Code ist auf GitHub[26] zu finden.

Schritt 3: Implementierung der RootErrorHandlingComponent

Nachdem in Schritt eins die Fehler abgefangen und in Schritt zwei die Route zu dem entsprechenden Fehler ermittelt wurde, wird in diesem Schritt die Struktur zur Anzeige der Fehler implementiert (siehe Abbildung 4.7). Die primäre Komponente in dieser Struktur ist die *RootErrorHandlingComponent*. Wie auch schon im Code zum Routing in Schritt 2 (Code 4.18) zu sehen ist, wird diese *RootErrorHandlingComponent* von der Top-Level-Route */error* aufgerufen.

Die *RootErrorHandlingComponent* verwaltet die Fehlerbehandlungskomponente (*ErrorHandlingComponent*) und die optionale Fehlerbehebungskomponente (*RecoveryComponent*). Das Klassendiagramm 4.7 beschreibt den Zusammenhang zwischen diesen Komponenten. *ErrorHandlingComponent* und *RecoveryComponent* sind generische Klassen, sodass diese durch Polymorphie zur Laufzeit durch die konkreten Fehlerbehandlungs- beziehungsweise Fehlerbehebungskomponenten ersetzt werden.

25 *https://angular.io/guide/router#preventing-unauthorized-access* zei. am 28.08.2023
26 github.com/Xerathron/Conduit-mf/commit/165ca55b5dd5bb5f127d1a7d4015113ccec53488, zuletzt eingesehen am 09.09.2023

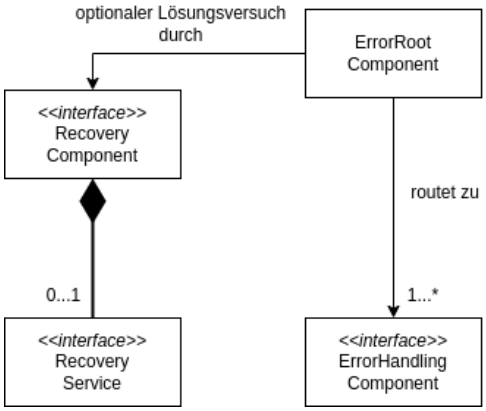


Abbildung 4.7: Klassendiagramm zur Struktur der Fehlerbehandlungs

Abbildung 4.8 zeigt, wie die Fehlerseite unterteilt ist. Dort ist zu erkennen, dass die beiden Buttons zum erneuten Versuchen und zurücknavigieren ebenfalls über die *RootErrorHandlingComponent* dargestellt werden. Des Weiteren werden alle Ereignisse, die auftreten können, über die *RootErrorHandlingComponent* verwaltet. Diese Ereignisse werden entweder automatisch ausgelöst (wie in der Abbildung nach Ablauf der Zeit) oder durch das Klicken des Buttons. Diese Ereignisse lösen den Prozess im *RecoveryServices* aus, wodurch versucht wird, den Fehler zu beheben. Weitere Ergebnisse ergeben sich durch die Darstellung, dass der Prozess angestoßen ist und ob der Prozess zur Fehlerbehandlung erfolgreich war oder nicht.

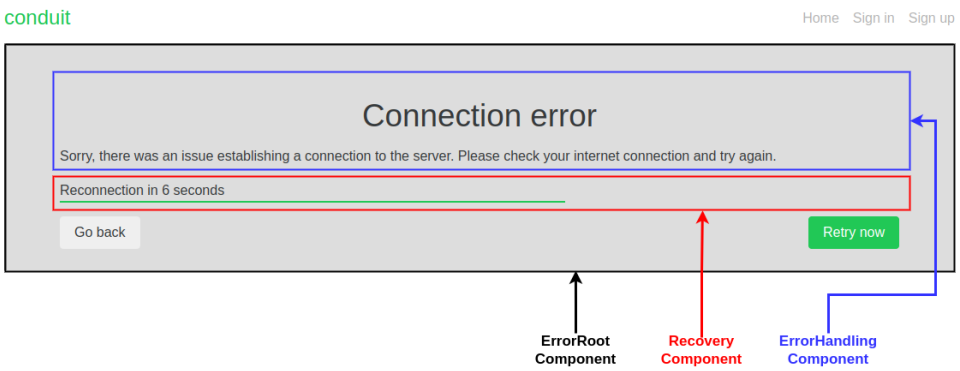


Abbildung 4.8: Fehlerseite unterteilt in RootErrorHandlingComponent, *ErrorHandlingComponent* und *RecoveryComponent* am Beispiel der Netzwerkfehlerseite

Nach Abschluss des dritten Schrittes ist das Übersichtsdiagramm A.7 erneut um zwei Komponenten erweitert worden. Der Code ist auf GitHub[27] zu finden.

27 github.com/Xerathron/conduit-mf/commit/4f352126c19ffac8bfb14277c2403d85ee4ad490, zuletzt eingesehen am 09.09.2023

Schritt 4: Übertragen und Speichern von Fehlerinformationen

Mit der bestehenden Struktur können Fehler abgefangen, zu einer Komponente geroutet und diese durch die *RootErrorHandlingComponent* verwaltet werden. Allerdings ist durch das Navigieren aus der *ErrorHandler*-Klasse zu einer *error* URI die Information verloren gegangen, um welchen Fehler es sich handelt. Damit diese nicht verloren geht und auch nach einem Neuladen der Seite weiterhin bestehen bleibt, wird sie im Storage zwischengespeichert. Dies ist in Zeile 27 im Code A.9 der *ErrorHandler*-Klasse zu erkennen.

Für den Zurück-Button müssen Informationen über die zuletzt regulär besuchte URI-Webseite vorhanden sein. Wenn ein erneuter Versuch unternommen wird, das MF zu laden, muss außerdem die Information über die angeforderte URI vorhanden sein. Das Speichern der angeforderten und zuletzt besuchten Seite im Storage wird durch den Code 4.19 übernommen.

```
1  return () => new Promise<void>((resolve) => {
2      router.events // angular router object
3      .pipe(filter((event) => event instanceof NavigationStart))
4      .subscribe((event: NavigationStart) => {
5          if (event.url.includes("/error")) {
6              return;
7          }
8
9          let routes: Array<string> = storage.get(ROUTE_TOKEN) || [];
10         // True if the user reloads the page manually or
11         // a process triggered a page reload without success
12         if (routes[LAST_ROUTE_INDEX] === event.url) {
13             return;
14         }
15
16         routes = addRoute(routes, event.url);
17         storage.set(ROUTE_TOKEN, routes);
18     });
19     resolve();
20 });
```

Listing 4.19: Schreibt aufgerufene reguläre Seiten (keine Fehlerseiten) in den Storage

Das Übersichtsdiagramm A.8 ist erneut um zwei Komponenten erweitert. Der Code zum Speichern der Informationen im Storage ist auf GitHub[28] einsehbar.

Schritt 5: Implementieren der Fehlerkomponenten

Im letzten Schritt werden die einzelnen Fehlerbehandlungs- und Behebungskomponenten implementiert. Diese sind bereits durch die generischen Klassen *ErrorHandlingComponent* und *RecoveryComponent* beschrieben. Das Übersichtsdiagramm A.9 zeigt die komplette Struktur des Fehlermoduls. Der Code zu den implementierten Komponenten ist auf GitHub[29] einsehbar.

28 github.com/Xerathron/Conduit-mf/commit/a045f980a5b4a08bd7cad4d213d26e46452d6bf7 zuletzt eingesehen am 09.09.2023

29 github.com/Xerathron/Conduit-mf/commit/810bf5fc556eea06e8b46914235b55ed123f0758 zuletzt eingesehen am 09.09.2023

5 BEWERTUNG

5.1 DISKUSSION

In der Diskussion werden die Konzepte anhand der durchgeführten Evaluation zusammengefasst und bewertet. Zur besseren Übersicht wird das Konzept der Überführung von dem Konzept der Fehlerbehandlung getrennt bewertet.

5.1.1 *Überführung*

Das erste Konzept beschreibt die wichtigen Entscheidungen, die bei der Überführung eines Monolithen zu einer MF-Architektur im Frontend getroffen werden müssen. Diese Entscheidungen betreffen die Zerlegung, das Composing, das Routing und die Kommunikation. Dies bildet das technische Fundament, auf dem die Art der anschließenden Implementierung basiert.

Die Implementation führt diese Entscheidungen am monolithischen Web-Frontend von Conduit durch. Durch die praktische Umsetzung wurden folgende Entscheidungen getroffen:

- Zerlegung: vertikal
- Composing: Client-Side-Composing
- Routing: Clientseitiges Routing
- Kommunikation: QueryStrings, Custom Events und Storage
- Implementierung: Module Federation

Mit diesen Entscheidungen ist die MF-Architektur für Conduit definiert. Die Architektur ist nach dem Konzept in den fünf Schritten durchgeführt worden.

1. Neues Projekt erstellen und Implementierung des Grundgerüsts (Shell).
2. Auslagern von gemeinsam genutzten Ressourcen in Shared Libraries.
3. Implementierung der einzelnen MFs.
4. Aufruf der MFs durch Composing und Routing.
5. Implementierung der Kommunikation zwischen den MFs.

Nach diesen Schritten ist die Webseite Conduit vollständig in eine MF-Architektur überführt. Die durchgeführte Überführung in eine MF-Architektur ist erfolgreich mithilfe der Entscheidungen aus dem Konzept erfolgt. Daher beantwortet das Konzept die Forschungsfrage, wie ein monolithisches Web-Frontend in eine MF-Architektur überführt werden kann. Die Überführung evaluierte dies anhand eines Anwendungsfalls. Allerdings gibt es zu dem Konzept ein paar Kritikpunkte beziehungsweise Änderungsvorschläge.

Methodik

In der Arbeit wird das Konzept mit einer Fallstudie durchgeführt. Hieraus ergeben sich direkt mehrere Kritikpunkte. Erstens kann durch das Wissen über das Konzept die Wahl des Anwendungsfalls für die Fallstudie so angepasst werden, dass die Chancen, dass das Konzept erfolgreich durchgeführt werden kann, verbessert werden. Zweitens besteht die Möglichkeit, dass das Konzept nur für spezielle Anwendungsfälle durchführbar ist. Mit der Durchführung einer Fallstudie besteht keine Evidenz, dass das Konzept ansatzweise allgemeingültig für eine Vielzahl an Anwendungsfällen durchführbar ist.

Um dies zu belegen, müssen weitere Fallstudien durchgeführt werden. Diese müssen sich in Eigenschaften des Anwendungsfalls unterscheiden. Einige Eigenschaften sind beispielsweise das eingesetzte JavaScript-Framework oder die Verwendung eines State Management Systems.

Aufbau des Konzeptes

Die Entscheidung, welches Routing gewählt wird, ist stark abhängig von dem gewählten Composing. Ein Client-Side-Composing, bei dem die MFs auf der Clientseite zusammengefügt werden, ist nur dann effektiv, wenn auch auf der Clientseite das Routing durchgeführt wird. Zwar könnten diese Entscheidungen in Kombination mit einer horizontalen Zerlegung auch von den Vorteilen einer MF-Architektur profitieren. Allerdings überwiegen dann die Nachteile beziehungsweise sind die aufkommenden Herausforderungen einer dezentralen Architektur deutlich größer als die Vorteile. Mezzalira [Mez21] geben in ihrer Publikation ebenfalls zu dieser Entscheidung definierte Bedingungen, bei denen das Routing an das Composing geknüpft ist.

Es ist denkbar, dass das Routing nicht als Entscheidung, sondern als Bedingung für die Optionen des Composings behandelt werden sollte.

Die Auswahl von Technologien zur Kommunikation zwischen den MFs kann ebenfalls diskutiert werden. Zunächst ist es, wie auch in der Durchführung gezeigt, durchaus realistisch, dass verschiedene Technologien zum Einsatz kommen. Daher handelt es sich eher um eine Auswahl als um eine Entscheidung.

Des Weiteren gibt es weitere Kommunikationstechnologien, die von der konkreten Implementierung abhängen. Eine gute Alternative zum Storage, der einfach manipuliert und ausgelesen werden kann, ist der Einsatz von State Management Systems. Diese als Option in die Auswahl der Kommunikationstechnologien aufzunehmen, ist sinnvoll. Männistö, Tuovinen und Raatikainen [MTR23] beschreibt ebenfalls, dass die Umsetzung einer MF-Architektur ohne State Management Systeme herausfordernd sein kann. Allerdings ist zu beachten, dass einige State Management Systeme exklusiv für bestimmte JavaScript-Frameworks entwickelt wurden. Die Verwendung solcher Systeme schränkt den Einsatz unterschiedlicher JavaScript-Frameworks ein. Redux hingegen ist ein JavaScript-Framework-unabhängiges State Management System. Geers [Gee20] beschreibt in seinem Buch den Einsatz von Redux (siehe Abschnitt 2.2).

Als Verbesserung für das Konzept ist aus diesen Gründen zu überlegen, den Entscheidungspunkt “Kommunikation“ zu streichen. Stattdessen können die Optionen und Technologien zur Kommunikation als Unterpunkt der Implementierung ausgewählt werden.

Geers [Gee20] erläutert auch den Einsatz mehrerer Composing-Strategien als isomorphes Composing (siehe Abschnitt 2.2). Im Anwendungsfall Conduit ist es durchaus denkbar, anstelle eines Client-Side-Composings auch ein Server-Side-Composing einzusetzen. Die Haupt- und Artikel-Unterseiten enthalten nur Inhalte, die beim Client-Side-Composing nachgeladen werden müssen. Zudem sind die Inhalte für alle Clients gleich, sodass sie gut im Backend zwischengespeichert werden können. Aus diesen Gründen eignet sich für die beiden Unterseiten ein Server-Side-Composing. Ein isomorphes Composing ist eine Lösung, um gezielt die Vorteile der Composing-Strategien zu nutzen.

Allerdings erhöht sich durch den isomorphen Ansatz die gesamte Komplexität der Anwendung. Bei kontinuierlicher Entwicklung bedeutet dies, dass die Einarbeitung neuer Entwickler*innen in den Code sowie das Warten der eingesetzten Technologien aufwändiger wird. Im konkreten Anwendungsfall ist zwar keine kontinuierliche Entwicklung geplant, allerdings ist die Umsetzung für den Zweck der Webseite nicht angemessen.

Der verwendete Anwendungsfall Conduit ist ebenfalls zu hinterfragen. Die Open-Source-Community “Realworld“ entwickelt das Front- und Backend für diesen Anwendungsfall mit unterschiedlichen Frameworks. Sie wollen damit nach eigenen Angaben Wissen in Form eines echten Anwendungsfalls weitergeben [Ani+23]. Die Front- und Backends werden unter optimalen Bedingungen entwickelt (klare Anforderungen, überschaubarer Umfang, kein Zeitdruck, gute Reviews und Feedback der Community). Da solche Bedingungen in realen Projekten nicht immer gegeben sind, besteht die Möglichkeit, dass der gewählte Anwendungsfall von einem realen Projekt abweicht.

Beispielsweise sind im Projekt Conduit die Domänen für die Zerlegung klar definierbar, da die einzelnen Komponenten geringe Kopplung und hohe Kohäsion aufweisen. In realen Projekten ist dies nicht immer gegeben, wodurch die Zerlegung unklar wird. Möglicherweise ist es sogar nötig, vorab eine Überarbeitung des Monolithen durchzuführen (Refactoring), bevor dieser überführt werden kann. Als Erweiterung des bestehenden Konzepts könnte ein Abschnitt mit Vorbedingungen aufgenommen werden, die erfüllt sein müssen, bevor der Monolith überführt werden kann.

Technische Überführung

Trotz der erfolgreichen Überführung ist die Kommunikation über die Angular-Services nicht gelungen (siehe Abschnitt 4.3.2). Der Beitrag zu diesem unerwarteten Verhalten[1] hat bislang keine Lösungsvorschläge zur Behebung des Problems. Als erfolgreicher Workaround werden Custom Events eingesetzt. Allerdings ist dies nur eine Zwischenlösung. Es wäre interessant, das Problem genauer zu analysieren, um in der Zukunft die Kommunikation über Angular-Services zu ermöglichen. Als erster Ansatz könnten verschiedene Angular-Versionen getestet werden, um festzustellen, ob das fehlerhafte Verhalten in allen Versionen auftritt.

Des Weiteren ist die im Konzept beschriebene Idee zum Auslagern von geteiltem Code in Shared Libraries zu betrachten. Die Überführung hat gezeigt, dass das Aufteilen des geteilten Codes in mehrere kleinere Shared Libraries nicht einfach möglich ist, sodass nur eine große Shared Library entstanden ist. In dieser ist sämtlicher geteilter Code vorhanden. Der Grund dafür sind zu viele Abhängigkeiten innerhalb der Shared Library zueinander (hohe Kopplung). Mit einer großen Shared Library konnte zwar immer noch von dem Vorteil der reduzierten Codebasis profitiert werden, allerdings nicht mehr in dem Umfang, wie im Konzept vorgesehen ist.

Im Konzept werden die externen NPM-Pakete alle in die Konfigurationsdatei von Webpack eingetragen. Dies ist ineffizient, da nicht in jedem MF alle NPM-Pakete benötigt werden. Um die Performance der MF-Architektur weiterhin zu steigern, sollten nur Abhängigkeiten in die Konfigurationsdatei eingetragen werden, die auch von dem jeweiligen MF gebraucht werden. In Kombination mit der Optimierung durch die Shared Libraries kann dadurch speziell in großen Projekten mit vielen NPM-Paketen die Ladezeiten erheblich gesenkt werden.

Weitere Aspekte

Das Konzept und die Überführung beziehen sich lediglich auf den technischen Aspekt der Architektur im Frontend. Die MF-Architektur besitzt jedoch viele weitere Aspekte, die im Rahmen der Arbeit nicht behandelt werden, aber dennoch für eine Überführung interessant sind.

Mezzalira [Mez21] beschreibt in seinem Buch verschiedene Technologien, wie die MF-Architektur im Frontend auch Einfluss auf das Backend hat und umgekehrt (siehe Abschnitt 2) [Mez21]. So ist es beispielsweise möglich, dass jedes MF mit einem äquivalenten Backend-Teil kommuniziert (Backend for Frontends). Daher kann die Zerlegung des Monolithen durch die Architektur im Backend beeinflusst werden.

1 https://github.com/module-federation/module-federation-examples/issues/3100 zuletzt eingesehen am 30.08.2023

Darüber hinaus muss für die Umsetzung der MF-Architektur die Infrastruktur der Server angepasst oder sogar neu aufgebaut werden. Damit die MFs geladen werden können, müssen sie anhand der URI unterschieden werden. Dies kann mithilfe eines Reverse Proxys geschehen, der die Anfragen intern an verschiedene Server weiterleitet. Eine andere Option ist, dass alle MFs auf demselben Server existieren und lediglich durch Ports unterschieden werden.

Ein weiterer Aspekt, der in dem Konzept nicht behandelt wird, ist die Infrastruktur des Codes. In der Umsetzung sind alle Unterprojekte in einem großen Ordner oder Projekt enthalten, was als Mono-Repository bezeichnet wird. Die Alternative dazu sind Multi-Repositories, bei denen jedes MF ein eigenes Repository besitzt. Beide Varianten haben Vor- und Nachteile und müssen bei der Überführung abgewogen werden. Multi-Repositories sind in sich isoliert. Jedes Repository besitzt seinen eigenen Prozess zur Integration und Bereitstellung des Codes auf den Server. Das ermöglicht hohe Individualität im Entwicklungsprozess. Allerdings steigt dadurch der Aufwand, die Repositories zu warten, in der Summe an [Pav+20]. Auf der anderen Seite sind Mono-Repositories leichter umzusetzen [Pav+20]. Das Konzept greift den Aspekt der unterschiedlichen Repositories nicht auf. Die Schritte beschreiben implizit die Umsetzung eines Mono-Repositories. Um die MF-Architektur optimal an den Anwendungsfall und seine Umgebung anzupassen, sollte dieser Aspekt in den fünf Schritten zur Überführung erwähnt werden.

In diesem Zusammenhang spielt auch das Entwicklungsteam eine wichtige Rolle. Jedem MF sollte ein klar zugeordnetes Team zugeordnet sein, das dafür verantwortlich ist. Abhängig von der Expertise der Teams können die Anforderungen an Technologieunabhängigkeit wichtiger oder weniger wichtig sein, was die Entscheidungen bei der Auswahl der Technologien beeinflussen kann. Beispielsweise sind einige State Management Systeme an bestimmte JavaScript-Frameworks gebunden.

5.1.2 *Fehlerbehandlung*

Es gibt einige Beispielcode- und Projekte zu MF-Architekturen. Diese beschränken sich auf die konkrete Funktionsweise bestimmter Frameworks oder Ansätze. Dabei wird die Fehlerbehandlung außer Acht gelassen. Dennoch ist die Fehlerbehandlung ein wichtiger Bestandteil der Softwareentwicklung und -Qualität. In dem zweiten Konzept wird ein Vorgehen beschrieben, wie Fehler, die während dem Nachladen und Einbinden sowie Aushängen von MFs auftreten können, behandelt werden.

Das Konzept beschreibt ein Vorgehen zum Erstellen einer Tabelle, in der die Fehler aufgelistet und bewertet werden. Das Ziel ist es, eine Übersicht über die Fehler zu bekommen. In der Implementierung sollen dann durch die Gruppierung der Fehler der Entwicklungsaufwand reduziert werden. Das Konzept wird an einer Fallstudie praktisch durchgeführt. Als Anwendungsfall wird die Webseite Conduit verwendet, die bereits in eine MF-Architektur überführt wurde. Die Durchführung des Konzepts hat ergeben, dass für den Anwendungsfall

vier Fehlerbehandlungskomponenten und eine Fehlerbehebungskomponente benötigt werden. Diese Komponenten sind anschließend implementiert worden. Die Implementierung erfolgt in fünf Schritten:

1. Abfangen des Fehlers
2. Fehlertabelle und Routing
3. Implementierung der RootErrorHandlingComponent
4. Übertragen und Speichern von Fehlerinformationen
5. Implementieren der Fehlerkomponenten

Nach den fünf Schritten ist in dem Anwendungsfall eine Fehlerbehandlung erfolgreich implementiert. Allerdings kann das Konzept und die Umsetzung diskutiert werden.

Methodik

Das Konzept ist durch die praktischen und theoretischen Erfahrungen des Autors der Arbeit gestützt. Für eine weitere Stütze ist es sinnvoll, wissenschaftliche Herangehensweisen zur Umsetzung einer tabellarischen Übersicht mit einzubeziehen. Des Weiteren ist zu prüfen, ob es weitere wissenschaftliche Konzepte zur Fehlerbehandlung zu anderen Themengebieten in der Softwareentwicklung gibt. Von Konzepten können dann Erkenntnisse für das Konzept in dieser Arbeit abgeleitet werden.

Konzept

Das Konzept bietet bei der Analyse der Fehler hinsichtlich der Bewertungskriterien Spielraum zur Eigeninterpretation. In der Durchführung ist es ebenfalls nicht immer einfach gewesen, Fehler in die Kategorien “gering“, “mittel“ und “hoch“ einzustufen. Daher können die Fehler von unterschiedlichen Entwickler*innen unterschiedlich bewertet werden. Werden die Vorgehensweisen zu den einzelnen Stufen der Bewertungskriterien von anderen Entwickler*innen definiert als diese, die die Fehler bewerten, kann dies zu Unstimmigkeiten beziehungsweise Misskommunikation führen.

Ein Beispiel hierfür ist die Bewertung der Lösbarkeit. Schätzt ein/e Entwickler*in die Lösbarkeit eines 429 HTTP-Statuscodes Fehlers (Rate Limiting) als “hoch“ ein, weil er/sie davon ausgeht, dass das Wartefenster des Servers sehr klein ist, wird die Behandlung zu dem nach dem definierten Vorgehen in der Durchführung sofort durchgeführt. Dies führt dazu, dass sofort eine erneute Anfrage an den Server geschickt wird. Durch das Rate Limiting wird allerdings die Anfrage erneut vom Server abgelehnt. Derartiger Interpretationsspielraum kann durch spezifischere Beschreibungen der einzelnen Bewertungsstufen vermieden werden. Auch kann ein allgemeines Vorgehen definiert werden, wenn ein Fehler genau zwischen zwei Bewertungsstufen steht. Zur genaueren Analyse der Annahme zu dem Fehler ist eine Fallstudie mit mehreren Entwickler*innen interessant.

Des Weiteren können die Bewertungsstufen der einzelnen Bewertungskriterien diskutiert werden. Für einen anderen, deutlich komplexeren Anwendungsfall ist es durchaus denkbar, dass es mehr als drei Stufen zur Einordnung der Eintrittswahrscheinlichkeit, Lösbarkeit oder Auswirkung gibt.

Im Konzept werden Fehler nach Gruppen sortiert, die ein gleiches Verhalten der Webseite aufweisen. Das Verhalten einer Webseite ist in unterschiedliche Aspekte aufgeteilt:

- Beschreibung des Fehlers
- Handlungsempfehlungen des Clients
- Behandlung des Fehlers durch die Fehlerbehandlungskomponente

Ein Fehler, der sich der Beschreibung des Fehlers passt, allerdings die Handlungsempfehlungen für den Fehler keinen Erfolg haben, muss in eine neue Gruppe hinzugefügt werden. Hier kann es dann zu Dopplungen in der Beschreibung kommen. Für weitere Fehler ist es schwieriger zu entscheiden, in welche Gruppe dieser hinzugefügt werden soll.

Beispielsweise kann ein Fehler mit einem HTTP-Statuscode 503 (temporäre Überlastung des Servers) durch eine kurze Wartezeit potentiell wieder behoben werden. Ein Fehler mit einem HTTP-Statuscode 500 (allgemeiner Serverfehler) kann durch Programmierfehler auf dem Server entstehen. Bei diesem Fehler ist eine längere Wartezeit durch den Ausfall des Servers möglich. Die Wartedauer in der Handlungsempfehlung ist in beiden Fehlern unterschiedlich.

Gleiches kann auch in den Fehlerbehandlungskomponenten auftreten. Der HTTP-Statuscode 429 kann beispielsweise in der Antwort des Servers ein Retry-After Feld mitgeben [FN12]. Dieses gibt eine Zeit vor bis der Client die Anfrage erneut stellen soll. Dieses sollte von der Fehlerbehandlungskomponente für diesen HTTP-Statuscode berücksichtigt werden. Hierzu muss in der Fehlerbehandlungskomponente eine Fallunterscheidung eingebaut werden. Gibt es viele derartiger Fallunterscheidungen können dadurch Aspekte der Softwarequalität wie die Wartbarkeit darunter leiden.

Das Konzept beschränkt sich auf Fehler, die während dem Nachladen und Einbinden sowie Aushängen von MFs entstehen können. Allerdings ist es möglich, dass weitere Fehler auf der Webseite entstehen, wodurch eine Fehlerseite angezeigt werden soll. Diese Fehler können die gleichen Prozesse verwenden. Eine Einschränkung beziehungsweise ein primärer Fokus auf Ladefehler bei MFs ist durchaus wichtig, jedoch kann im Konzept auch die Möglichkeit beschrieben werden, dass allgemeine Fehler, die eine gleiche Struktur wie die Ladefehler aufweisen, mit in die Tabelle hinzugefügt werden können.

Implementierung

Die Architektur zur Fehlerbehandlung ist sehr modular aufgebaut (siehe Abbildung A.9). Es können schnell Komponenten ausgetauscht, neu entwickelt und umstrukturiert werden. Jede einzelne Komponente in der Architektur besitzt eine kleine, überschaubare Aufgabe. Dadurch ist die Testbarkeit, Wartbarkeit sowie weitere Aspekte der Softwarequalität auf einem hohen Niveau. Gleichzeitig ist allerdings die gesamte Architektur abstrakter und bedarf mehr Einarbeitungszeit. Für einen Anwendungsfall, der sich zukünftig nicht weiter ändert beziehungsweise bei dem abzusehen ist, dass sich die Tabelle hinsichtlich der resultierenden Fehlerkomponenten nicht ändern wird, führt diese Architektur zu mehr zusätzlicher Abstraktion als sie Nutzen bringt. In der Softwarequalität gibt es neben klaren Anforderungen der ISO 25010[2] auch weitere Prinzipien wie das KISS-Prinzip (Keep it simple, stupid). Es besagt, dass für eine Anforderung Code, der weniger abstrakt und komplex ist, einfacher zu verstehen ist. Dieses Prinzip ist mit den Anforderungen des jeweiligen Anwendungsfalls gegenüber Aspekten der Softwarequalität individuell zu diskutieren.

Eine gute, allgemeine Lösung, um die Komplexität zu reduzieren, ist das Auslagern des Codes zur Fehlerbehandlung. Der Code kann als NPM-Paket gebündelt und öffentlich als Bibliothek verfügbar gemacht werden. Dadurch wird das Wissen ähnlich wie bei Frameworks extrahiert und abstrahiert. Die Bibliothek kann als weitere Technologie im Entscheidungsprozess zur Vorgehensweise der Implementierung eingesetzt werden. Die Entwickler*innen müssen dann lediglich die konkreten Fehlerbehandlungs- und Behebungskomponenten, die in Schritt fünf beschrieben sind, durchführen. Der Einsatz des NPM-Pakets könnte wie folgt aussehen:

```
1  import { ErrorModule, ErrorComponents } from 'external_npm_package';
2
3  const ERROR_COMPONENTS: ErrorComponents = {
4    ERR_NETWORK: {errorComponent: NetworkErrorComponent},
5    ETIMEDOUT: {errorComponent: NetworkErrorComponent},
6    "500": {errorComponent: ServerErrorComponent},
7    "501": {errorComponent: ServerErrorComponent},
8  };
9
10 @NgModule({
11   imports: [ErrorModule.create(ERROR_COMPONENTS)],
12   exports: [ErrorModule],
13 })
14 export class ErrorRoutingModule {}
```

Listing 5.1: Beispielscode bei dem die Architektur zur Fehlerbehanldung in einem NPM-Paket (*external_npm_package*) ausgelagert wurde.

Durch diesen Ansatz wird der Aufwand zur Implementierung der Fehlerbehandlung für einen Anwendungsfall weiter reduziert. Mithilfe dieser Bibliothek könnten kleinere Projekte mit weniger Ressourcen eine qualitativ hochwertige Fehlerbehandlung durchführen.

2 https://www.iso.org/standard/35733.html zuletzt eingesehen am 01.09.2023

Da mit dieser Idee einige Anforderungen an das NPM-Paket hinzukommen, müsste dieses erweitert werden. Der für den Anwendungsfall existierende Code ist eine gute Basis. Dieser muss um weitere Aspekte wie das Definieren, Erstellen und Dokumentieren von Schnittstellen erweitert werden, damit die Bibliothek von außen entsprechend benutzbar ist.

In der umgesetzten Implementierung wird der Prozess zur Behandlung der Fehler durch die jeweilige Fehlerbehandlungskomponente gestartet. Ein Prozess zum Starten der Behandlung ist durch einen automatisierten Vorgang, der ebenfalls in der Fehlerbehandlungskomponente implementiert ist, definiert. Diese Prozesse werden im Folgenden als Trigger bezeichnet. Das Problem ist, dass ein Trigger in mehreren Fehlerbehandlungskomponenten vorkommen kann. Ein denkbares Beispiel ist, dass der Trigger Timer, der bereits in der Fehlerbehandlungskomponente “erneut versuchen“ vorgekommen ist, möglicherweise auch in der Fehlerbehandlungskomponente “Webseite neu laden“ vorkommen kann. Damit die Trigger in den Behandlungskomponenten nicht immer neu geschrieben werden müssen, ist es denkbar, diese als weitere Entität zu definieren. Sie werden entsprechend einer Komposition an die jeweilige Behandlungskomponente angehängt. Dies ist eine Erweiterung, um noch mehr Flexibilität im Prozess der Fehlerbehandlung zu erhalten. Andererseits verschärft sich damit das Problem aus dem vorherigen Absatz, dass die Komplexität durch die Abstraktion steigt.

In der Implementierung wurden für das Nachladen von MFs Axios verwendet. Reguläre API-Aufrufe wurden weiterhin mit einer Angular-eigenen Technologie (HttpClient) umgesetzt. Die beiden Technologien besitzen unterschiedliche Strukturen von Fehlern (siehe Code aus der Implementierung 4.14 und 4.15). Daher mussten in der Implementierung des *ErrorHandler* diese unterschiedlichen Strukturen in ein einheitliches Format überführt werden.

Dieses Problem wird in jedem Anwendungsfall auftreten, der mehr als eine Technologie zum Laden von Ressourcen verwendet. Daher sollte der Prozess zur Vereinheitlichung der Struktur von Fehlern in das Konzept aufgenommen werden. In dem Prozess sollte berücksichtigt werden, welche Informationen zur Identifizierung von Fehlern relevant sind. Als weiterer Schritt kann geprüft werden, ob es gegebenenfalls zu Schwierigkeiten durch fehlende Informationen kommen kann. Abbildung 5.1 zeigt einen ersten Ansatz, wie eine Vereinheitlichung der Struktur von Informationen über Fehler aussehen kann. Darüber hinaus kann auch die Alternative in Betracht gezogen werden, nur noch eine Technologie zum Laden von Ressourcen zu verwenden.

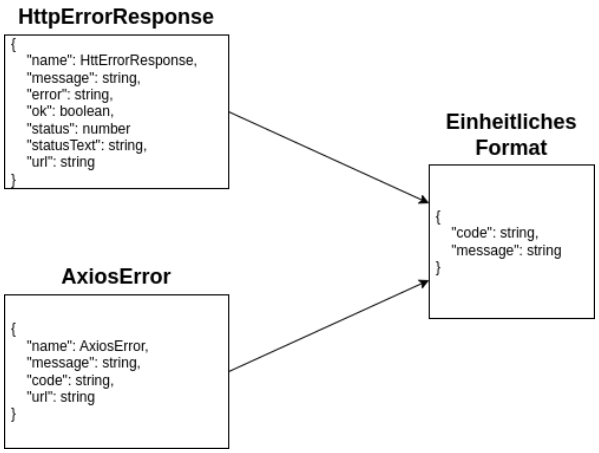


Abbildung 5.1: Struktur der Fehler von HttpClient und Axios in ein vereinheitlichtes Format.

5.2 ZUSAMMENFASSUNG

Das Konzept der Überführung eines Monolithen in die MF-Architektur fasst viele Optionen zusammen, die in einzelnen wissenschaftlichen Publikationen beschrieben werden. Die Umsetzung zeigt erfolgreich anhand eines idealen Anwendungsfalls das Konzept. Nichtsdestotrotz wurden durch die Überführung Punkte zur Verbesserung des Konzepts gefunden.

Geers [Gee20] betont, dass es wichtig ist, dass die Überführung auf den Anwendungsfall individuell angepasst werden muss [Gee20, Kapitel 2]. Neben technischen Aspekten sind auch weitere Aspekte wie Backend-Architekturen, die die MF-Architektur beeinflussen, oder die Infrastruktur des Codes zu berücksichtigen.

Das Konzept zur Fehlerbehandlung ist strukturiert und konnte erfolgreich an dem Anwendungsfall umgesetzt werden. Die resultierende Tabelle bietet eine gute Übersicht über die auftretenden Fehler. Dies bietet dem Entwickler*innenteam eine gute Übersicht, um den Aufwand der Implementierung abzuschätzen. Die durchgeführte Implementierung ist umfangreich und für manche Anwendungsfälle bezüglich der Komplexität zu hinterfragen. Große, stetig entwickelnde Webseiten, für die die MF-Architektur viele Vorteile bringt, ist eine Implementierung wie sie in dieser Arbeit durchgeführt wurde ein geringer Aufwand im Vergleich zum Nutzen, den sie bringt.

Daher ist die durch das Konzept durchgeführte Implementierung ebenfalls eine gute umgesetzte Lösung zur Fehlerbehandlung beim Nachladen und Einbinden sowie Aushängen von MFs.

6 FAZIT UND AUSBLICK

Die Arbeit befasst sich mit zwei Konzepten im Themenbereich der MF-Architektur. Der Kerngedanke der Konzepte wird jeweils zusammengefasst, und es wird auf Punkte der Diskussion eingegangen. Anschließend erfolgt ein Ausblick, in dem weitere interessante Themenfelder und Erweiterungen des Konzepts beschrieben werden.

6.1 FAZIT

Im ersten Konzept wird die Überführung einer monolithischen Webanwendung in eine MF-Architektur beschrieben. Anhand einer praktischen Durchführung und der anschließenden Diskussion wird das Konzept evaluiert. Die Überführung konnte erfolgreich mit dem Konzept durchgeführt werden.

Das zweite Konzept beschreibt die Fehlerbehandlung beim Nachladen, Einbinden sowie Aushängen von MFs. Das Ziel des Konzepts ist die Reduzierung des Aufwandes zur Implementierung der Fehlerbehandlung. Wie auch im ersten Konzept erfolgt eine praktische Durchführung und eine Diskussion des Konzepts.

Überführungskonzept

Zunächst wird im Konzept zur Überführung des Monolithen anhand der Entscheidungen zu den Aspekten Zerlegung, Composing, Routing, Kommunikation und Implementation entschieden, welche Technologien und Vorgehensweisen verwendet werden, um die MF-Architektur für einen konkreten Anwendungsfall zu beschreiben. Anschließend stellt das Konzept fünf Schritte vor, wie der monolithische Anwendungsfall in die MF-Architektur überführt werden kann. Der Kerngedanke dabei ist, ein neues Projekt anzulegen, in dem die MF-Architektur von Grund auf aufgebaut wird. Die Codelogik der Webseite wird dann in dieses neue Projekt übernommen und angepasst.

Mithilfe des Konzepts konnte das Web-Frontend des monolithischen Anwendungsfalls erfolgreich in eine MF-Architektur überführt werden. In manchen Aspekten, wie der Entscheidung des Routings, kann das Konzept weiter optimiert werden. Beispielsweise ist zu überlegen, das Routing als Unterpunkt des Composings anzuhängen, da das Routing stark vom Composing abhängig ist.

Das Konzept betrachtet ausschließlich technische Aspekte. Für eine passgenaue Überführung und Anpassung der MF-Architektur müssen weitere Aspekte im Anwendungsfall betrachtet werden. Darunter zählt beispielsweise der Aufbau des Backends und die Infrastruktur des Codes.

Fehlerbehandlungskonzept

Die Fehlerbehandlung ist ein essenzieller Bestandteil in der Softwareentwicklung. In Beispielen zur Umsetzung der MF-Architektur wird dieser Aspekt außer Acht gelassen. Ebenso wird er zwar in wissenschaftlichen Publikationen als wichtiger Punkt erwähnt, aber nicht behandelt. Als Erweiterung zur Forschungsfrage wird in dieser Arbeit ein Konzept zur Fehlerbehandlung von Fehlern, die beim Nachladen, Einbinden und Aushängen von MFs entstehen können, vorgestellt.

Der Kerngedanke des Konzepts ist, eine Tabelle zu erstellen, die eine Übersicht über die Fehler gibt, die während dem Nachladen, Einbinden und Aushängen von MF entstehen können. Des Weiteren wird analysiert, wie hoch die Wahrscheinlichkeit, Auswirkungen und Lösbarkeit dieser Fehler ist. Anschließend werden die Fehler gruppiert, um den Implementierungsaufwand zu reduzieren. Um die Fehlergruppen wird dann eine Struktur aufgebaut, um die Fehlerbehandlung zu implementieren.

Wie auch für das Überführungskonzept gilt auch bei diesem Konzept, dass das Vorgehen immer an den jeweiligen Anwendungsfall angepasst werden kann. Die Bewertungsstufen können angepasst beziehungsweise neu interpretiert werden. Es ist auch möglich, weitere Fehler, die nicht im Zusammenhang mit der MF-Architektur stehen, aber dennoch durch die Tabelle bewertet und gruppiert werden können, mit zu betrachten. Damit kann dieses Konzept zu einem Fehlerbehandlungskonzept für die gesamte Webseite ausgeweitet werden.

Die Umsetzung des Konzepts hat gezeigt, dass sich mithilfe der Tabelle die Anzahl an Fehlern deutlich reduzieren lässt. Aus einer Tabelle mit über 53 Fehlern sind am Ende vier Komponenten zur Behandlung der Fehler entstanden. Eine weitere Komponente führt Maßnahmen zur Behebung von Fehlern durch. Diese Implementation ist komplex aufgebaut und nicht für jeden Anwendungsfall geeignet. Nichtsdestotrotz zeigt sie eine gute Herangehensweise, wie eine Fehlerbehandlung unter Berücksichtigung vieler Aspekte der Softwarequalität umgesetzt werden kann.

6.2 AUSBLICK

Beide Konzepte können noch weiter optimiert werden. Dies gilt sowohl für die Methodik als auch inhaltlich. Ein Aspekt der Diskussion ist die Durchführung der Konzepte an nur einem Anwendungsfall. Es ist interessant, weitere Erkenntnisse über die Konzepte anhand weiterer Fallstudien durchzuführen. Dabei sollten auch die Anwendungsfälle unterschiedlich sein. Beispielsweise sollten unterschiedliche JavaScript-Frameworks oder andere Entscheidungen in der MF-Architektur getroffen werden. Des Weiteren ist es interessant, nicht nur rein technische Anwendungsfälle zu betrachten, sondern auch weitere Aspekte wie Teamstrukturen oder Projekte mit komplexen Backends in eine MF-Architektur zu überführen und anschließend eine Fehlerbehandlung nach dem Konzept durchzuführen.

In dem Überführungskonzept ist das Web-Frontend ganzheitlich in fünf Schritten überführt worden. Es gibt weitere Ansätze, bei denen nur einzelne MFs sukzessive überführt werden. Diese werden während der Überführungsphase bereits in der Produktivumgebung durch die Shell angesprochen, während die restliche Webseite weiterhin als Monolith (beziehungsweise monolithisches MF) integriert wird. Das Vorgehen eignet sich für große Projekte, die kontinuierlich weiterentwickelt werden. Das Konzept dahingehend zu erweitern ist daher zukünftig interessant.

Ein weiterer moderner Ansatz ist der Einsatz einer PWA. Eine MF-Architektur kann teilweise oder vollständig als PWA aufgebaut werden. Somit können für spezielle Anforderungen in bestimmten Bereichen der Webseite die Vorteile der PWAs genutzt werden. Das Buch [RS21] beschreibt am Rande den Einsatz einer PWA in einer MF-Architektur. Die Verbindung der Architektur mit PWAs ist ein spannender Ansatz, der zwei moderne Lösungen in der Webentwicklung kombiniert. Dieses Themenfeld ist für die Zukunft interessant zu verfolgen.

[AN20] Dan Abramov und Rachel Nabors. *React V17.0 – React Blog*. https://legacy.reactjs.org/blog/2020/10/20/react-v17.html. Okt. 2020. (Besucht am 15. 08. 2023).

[Ani+23] Karandikar Anish, Cameron Chapman, Eric Simon, Albert Pai, James Brewer und Sandeesh S. *Introduction | RealWorld*. https://realworld-docs.netlify.app/docs/intro. Aug. 2023. (Besucht am 15. 08. 2023).

[Bro21] Jan Brockmeyer. *Zalando Engineering Blog - Micro Frontends: From Fragments to Renderers (Part 1)*. https://engineering.zalando.com/posts/2021/03/micro-frontends-part1.html. März 2021. (Besucht am 25. 02. 2023).

[Büh+22] Fabian Bühler, Johanna Barzen, Lukas Harzenetter, Frank Leymann und Philipp Wundrack. "Combining the Best of Two Worlds: Microservices and Micro Frontends as Basis for a New Plugin Architecture". In: *Service-Oriented Computing*. Hrsg. von Johanna Barzen, Frank Leymann und Schahram Dustdar. Communications in Computer and Information Science. Cham: Springer International Publishing, 2022, S. 3–23. ISBN: 978-3-031-18304-1. DOI: *10.1007/978-3-031-18304-1_1*.

[Com23] Angular Community. *Angular - HttpErrorResponse*. https://angular.io/api/common/http/HttpErrorResponse. Feb. 2023. (Besucht am 09. 09. 2023).

[Den23] Domenic Denicola. *HTML Standard*. https://html.spec.whatwg.org/multipage/. Sep. 2023. (Besucht am 02. 09. 2023).

[DM20] Joel Denning und Justin McMurdie. *The Recommended Setup | Single-Spa*. https://single-spa.js.org/docs/recommended-setup/. Feb. 2020. (Besucht am 18. 08. 2023).

[Dra+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin und Larisa Safina. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Hrsg. von Manuel Mazzara und Bertrand Meyer. Cham: Springer International Publishing, Sep. 2017, S. 195–216. ISBN: 978-3-319-67425-4. DOI: *10.1007/978-3-319-67425-4_12*. (Besucht am 20. 07. 2023).

[FN12] Roy T. Fielding und Mark Nottingham. *RFC 6585 Additional HTTP Status Codes*. Request for Comments RFC 6585. Internet Engineering Task Force, Apr. 2012. DOI: *10.17487/RFC6585*. (Besucht am 14. 07. 2023).

[FR14] Roy T. Fielding und Julian Reschke. *RFC 7231 Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. Request for Comments RFC 7231. Internet Engineering Task Force, Juni 2014. DOI: *10.17487/RFC7231*. (Besucht am 13. 07. 2023).

[Gee20] Michael Geers. *Micro Frontends in Action*. Sep. 2020. ISBN: 978-1-61729-687-1. (Besucht am 26. 06. 2023).

[Gee23] Michael Geers. *Micro Frontends - Extending the Microservice Idea to Frontend Development*. https://micro-frontends.org/. Aug. 2023. (Besucht am 02. 09. 2023).

[Gee+22] G Geetha, Mittal Monisha, Prasad Mohana und Ponsam Godwin. *Interpretation and Analysis of Angular Framework*. https://ieeexplore.ieee.org/abstract/document/10047474. Dez. 2022. (Besucht am 11. 09. 2023).

[HYL23] Taku Hoshizawa, Kei Yamashita und Patrick Luthi. *Standards*. https://www.ecma-international.org/publications-and-standards/standards/. Juni 2023. (Besucht am 01. 08. 2023).

[Jac19] Cam Jackson. *Micro Frontends*. https://martinfowler.com/articles/micro-frontends.html. Juni 2019. (Besucht am 09. 05. 2023).

[Jac20] Zack Jackson. *Webpack 5 Federation: A Game Changer in JavaScript Architecture*. Mai 2020. (Besucht am 02. 08. 2023).

[KC20] Tobias Koppers und Sam Chen. *Webpack 5 Release (2020-10-10)*. https://webpack.js.org/blog/2020-10-10-webpack-5-release/. Okt. 2020. (Besucht am 07. 08. 2023).

[Kro21] Manuel Kroiß. “From Backend to Frontend : Case Study on Adopting Mmcro Frontends from a Single Page ERP Application Monolith”. Thesis. Wien, 2021. DOI: *10.34726/hss.2021.85306*. (Besucht am 02. 09. 2023).

[Lan22] Billy J. Lando. “Extrahieren von Micro-Frontends aus einer monolithischen Frontend Anwendung”. Magisterarb. Universität Bremen, Apr. 2022. (Besucht am 09. 05. 2023).

[Lev05] Jason Levitt. *JSON and the Dynamic Script Tag: Easy, XML-less Web Services for JavaScript*. https://www.xml.com/pub/a/2005/12/21/json-dynamic-script-tag.html. Dez. 2005. (Besucht am 13. 07. 2023).

[LF14] James Lewis und Martin Fowler. *Microservices*. https://martinfowler.com/articles/microservices.html. Mai 2014. (Besucht am 09. 05. 2023).

[Loh23] L Lohmeier. *Webseiten - Anzahl weltweit 2023*. Mai 2023. URL: *https://de.statista.com/statistik/daten/studie/290274/umfrage/anzahl-der-webseiten-weltweit/* (besucht am 02. 06. 2023).

[MTR23] Jouni Männistö, Antti-Pekka Tuovinen und Mikko Raatikainen. “Experiences on a Frameworkless Micro-Frontend Architecture in a Small Organization”. In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. März 2023, S. 61–67. DOI: *10.1109/ICSA-C57050.2023.00025*.

[Men+19] Manel Mena, Antonio Corral, Luis Iribarne und Javier Criado. “A Progressive Web Application Based on Microservices Combining Geospatial Data and the Internet of Things”. In: *IEEE Access* 7 (2019), S. 104577–104590. ISSN: 2169-3536. DOI: *10.1109/ACCESS.2019.2932196*.

[Mez19] Luca Mezzalira. *Orchestrating Micro-Frontends*. Apr. 2019. (Besucht am 02. 08. 2023).

[Mez21] Luca Mezzalira. *Building Micro-Frontends*. Nov. 2021. ISBN: 978-1-4920-8298-9. (Besucht am 02. 09. 2023).

[Moz23] Mozilla. *Node: appendChild() Method - Web APIs | MDN*. https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild. Apr. 2023. (Besucht am 09. 08. 2023).

[NK22] Yuma Nishizu und Tetsuo Kamina. “Implementing Micro Frontends Using Signal-based Web Components”. In: *Journal of Information Processing* 30.0 (2022), S. 505–512. ISSN: 1882-6652. DOI: *10.2197/ipsjjip.30.505*. (Besucht am 21. 07. 2023).

[Pav+20] Andrey Pavlenko, Nursultan Askarbekuly, Swati Megha und Manuel Mazzara. “Micro-Frontends: Application of Microservices to Web Front-Ends”. In: *Journal of Internet Services and Information Security* 10.2 (Mai 2020), S. 49–66. DOI: *10.22667/JISIS.2020.05.31.049*. (Besucht am 08. 09. 2023).

[PMT21] Severi Peltonen, Luca Mezzalira und Davide Taibi. “Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review”. In: *Information and Software Technology* 136 (Aug. 2021), S. 106571. ISSN: 0950-5849. DOI: *10.1016/j.infsof.2021.106571*. (Besucht am 28. 03. 2023).

[Pen04] Gang Peng. *CDN: Content Distribution Network*. Nov. 2004. DOI: *10.48550/arXiv.cs/0411069*. arXiv: *cs/0411069*. (Besucht am 26. 07. 2023).

[Rad20] Technology Radar. *Micro Frontends | Technology Radar*. https://www.thoughtworks.com/radar/techniques/micro-frontends. Mai 2020. (Besucht am 20. 07. 2023).

[RS21] Florian Rappl und Lothar Schottner. *The Art of Micro Frontends*. Juni 2021. ISBN: 978-1-80056-356-8. (Besucht am 20. 07. 2023).

[Ric23] Alex Rickabaugh. *Angular - Injectable*. https://angular.io/api/core/Injectable. Mai 2023. (Besucht am 22. 08. 2023).

[Riv23] Florian Rivoal. *CSS SPECIFICATIONS*. https://www.w3.org/Style/CSS/specs.en.html. Apr. 2023. (Besucht am 01. 08. 2023).

[Ste16] Guido Steinacker. *Why Microservices? | OTTO Tech | Blog*. Mai 2016. URL: *https : / / www . otto . de / jobs / en / technology / techblog/blogpost/frontends-with-microservices.php* (besucht am 25. 02. 2023).

[Tab21] Simon Tabor. *How DAZN Manages Micro-Frontend Infrastructure*. Aug. 2021. (Besucht am 02. 08. 2023).

[TM22] Davide Taibi und Luca Mezzalira. "Micro-Frontends: Principles, Implementations, and Pitfalls". In: *ACM SIGSOFT Software Engineering Notes* 47.4 (Sep. 2022), S. 25–29. ISSN: 0163-5948. DOI: *10.1145/3561846.3561853*. (Besucht am 03. 04. 2023).

[Tho21] Mark Thompson. *Angular V12 Is Now Available*. https://blog.angular.io/angular-v12-is-now-available-32ed51fbfd49. Juni 2021. (Besucht am 15. 08. 2023).

[Tsi+01] Mark Tsimelzon, Bill Weihl, Joseph Chung, Dan Frantz, John Basso, Chris Newton, Mark Hale, Larry Jacobs und Conleth O'Connell. *ESI Language Specification 1.0*. https://www.w3.org/TR/esi-lang/. Aug. 2001. (Besucht am 26. 07. 2023).

[Vai23] Lionel Sujay Vailshery. *Most Used Web Frameworks among Developers 2023*. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/. Juli 2023. (Besucht am 20. 07. 2023).

[WN04] Westley Weimer und George C. Necula. "Finding and Preventing Run-Time Error Handling Mistakes". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. Vancouver BC Canada: ACM, Okt. 2004, S. 419–431. ISBN: 978-1-58113-831-3. DOI: *10.1145/1028976.1029011*. (Besucht am 08. 08. 2023).

[YLS19] Caifang Yang, Chuanchang Liu und Zhiyuan Su. "Research and Application of Micro Frontends". In: *IOP Conference Series: Materials Science and Engineering* 490.6 (Apr. 2019), S. 062082. ISSN: 1757-899X. DOI: *10.1088/1757-899X/490/6/062082*. (Besucht am 30. 03. 2023).

Teil II

APPENDIX

A

APPENDIX

Der Anhang beinhaltet weitere Inhalte der Arbeit. Im Folgenden sind größere Code-Ausschnitte, die aber dennoch wichtig für den Kontext der Arbeit sind. In der Arbeit sind vier GitHub-Repositories[1] entstanden:

1. **conduit-mf**[2] ist das wichtigste Repository. Dieses beinhaltet den Anwendungsfall für beide Fallstudien.
2. **Bug-Report-Projekt**[3]. Dieses Projekt ist für das Issue 3100[4] in dem module-federation-examples GitHub-Repository erstellt worden. Es dient zum nachvollziehen und reproduzieren des Fehlers, welcher in Abschnitt 4.3.2 beschrieben wird.
3. **single-spa-example**[5] ist ein Beispiel-Projekt zur Demonstration der Funktionsweise von single-spa. Essenzielle Code-Bestandteile sind ebenfalls im Anhang zu finden.
4. **web-components-example**[6]: Eine Demonstration der Funktionsweise von Web Components ist in diesem Repository zu finden.

Alle Bilder, Tabellen sowie Repositories sind ebenfalls auf GitHub[7] einsehbar.

A.1 KOMMUNIKATION MIT EINEM STATE MANAGEMENT SYSTEM

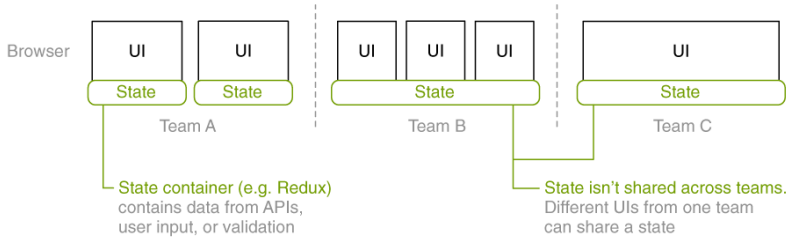


Abbildung A.1: Schaubild aus dem Buch [Gee20]: Um die Kopplung gering zu halten besitzt jedes MF (Team-Grenzen) ein State-Management-System. Diese können aber auch noch feiner aufgeteilt werden

1 https://github.com/Xerathron?tab=repositories zuletzt eingesehen am 12.09.2023
2 https://github.com/Xerathron/conduit-mf zuletzt eingesehen am 12.09.2023
3 *https://github.com/Xerathron/ModuleFederationExample_Angular_MF_Service_Problem* zuletzt eingesehen am 12.09.2023
4 *https://github.com/module-federation/module-federation-examples/issues/3100* zuletzt eingesehen am 12.09.2023
5 https://github.com/Xerathron/single-spa-example zuletzt eingesehen am 12.09.2023
6 https://github.com/Xerathron/web-components-example zuletzt eingesehen am 12.09.2023
7 https://github.com/Xerathron/bachelor-thesis zuletzt eingesehen am 12.09.2023

A.2 CODE-BEISPIEL FÜR WEB COMPONENTS

A.2 zeigt ein einfaches Beispiel für den Aufbau einer MF-Architektur mit Web Components. Es werden zwei MFs clientseitig lazy geladen. Es ist wichtig zu betonen, dass dies lediglich die Funktionsweise von MFs mittels Web Components demonstrieren soll. Aspekte wie Fehlervermeidung und -behandlung sowie Anforderungen an die Softwarequalität werden nicht berücksichtigt. Der hier gezeigte Code ist auf das für die Demonstration von Web Components Relevante beschränkt. Der vollständige Code ist bei GitHub[8], zu finden.

```
1  export default class FooWebComponent extends HTMLElement {
2    constructor() {
3      super();
4
5      // closed mode to forbid external javascript access
6      const shadow = this.attachShadow({ mode: 'closed'});
7
8      // create node and append it to the shadow-dom
9      const paragraph = document.createElement("p");
10     paragraph.appendChild(document.createTextNode("foo"));
11     shadow.appendChild(paragraph);
12   }
13 }
```

Listing A.1: Web Components Beispiel (FooCustomElement.js): Foo Micro Frontend Custom Element

8 https://github.com/Xerathron/web-components-example zuletzt eingesehen am 10.08.2023

```
/** List containing all loaded micro frontend web components */
const registeredMf = [];

/** global root element to which the loaded web components are attached */
const wrapper = document.getElementById("component-wrapper");

/** Mounts the Foo Micro Frontend */
async function mountFoo() {
  const templateName = await loadMicroFrontend("foo", "./FooCustomElement.js")

  unmountAllMfs();
  mountMf(templateName);
}

/** Mounts the Bar Micro Frontend */
async function mountBar() {
  const templateName = await loadMicroFrontend("bar", "./BarCustomElement.js")

  unmountAllMfs();
  mountMf(templateName);
}

/**
 * Generic function to lazy load a micro frontend.
 * Return immediately if the component is already loaded
 *
 * @returns the name of the template which the \ac{MF} can be loaded
 */
async function loadMicroFrontend(name, urlpath) {
  const templateName = `${name}-component`;

  if (registeredMf.includes(templateName)) {
    return templateName;
  }

  const MicroFrontend = await import(urlpath);
  // append class to global customElements
  customElements.define(templateName, MicroFrontend.default);
  registeredMf.push(templateName);

  return templateName;
}

function mountMf(templateName) {
    const template = document.createElement(templateName);
    wrapper.appendChild(template);
}

/** Unmount all micro frontends */
function unmountAllMfs() {
  while (wrapper.firstChild) {
    wrapper.removeChild(wrapper.firstChild);
  }
}
```

Listing A.2: Web Components Beispiel (index.js) Shell JavaScript; Es stellt Funktionen zum Laden Einbinden und Entfernen von MFs zu Verfügung.

A.3 CODE-BEISPIEL FÜR SINGLE-SPA

Der folgende Code ist ebenfalls ein einfaches Beispiel für den Aufbau einer MF-Architektur mit zwei MFs, die clientseitig lazy geladen werden. Abbildung A.2 zeigt, wie die Webseite beim Aufruf aussieht. Zwischen den beiden MFs (Foo und Bar) kann mithilfe der Buttons navigiert werden. Es ist wichtig zu betonen, dass das Projekt lediglich die Funktionsweise von MFs mittels single-spa demonstrieren soll. Aspekte wie Fehlervermeidung und -behandlung sowie Anforderungen an die Softwarequalität werden nicht berücksichtigt. Der hier gezeigte Code ist auf das für die Demonstration von single-spa Relevante beschränkt. Der vollständige Code ist auf GitHub[9] zu finden.

```
1 import singleSpaHtml from 'single-spa-html';
2
3 const htmlLifecycles = singleSpaHtml({
4   template: '<p>foo</p>',
5 })
6
7 export const bootstrap = htmlLifecycles.bootstrap;
8 export const mount = htmlLifecycles.mount;
9 export const unmount = htmlLifecycles.unmount;
```

Listing A.3: Single-spa Beispiel (foo.js) MF “foo“

9 https://github.com/Xerathron/single-spa-example, zuletzt eingesehen am 12.09.2023

```
1  import * as singleSpa from 'single-spa';
2
3  singleSpa.registerApplication({
4      name: 'foo',
5      app: () => import('./\ac{MF}s/foo.js'),
6      activeWhen: '/foo',
7  })
8
9  singleSpa.registerApplication({
10     name: 'bar',
11     app: () => import('./\ac{MF}s/bar.js'),
12     activeWhen: '/bar',
13 })
14
15 window.openFoo = () => {
16     singleSpa.navigateToUrl("/foo");
17 }
18
19 window.openBar = () => {
20     singleSpa.navigateToUrl("/bar");
21 }
22
23 singleSpa.start();
```

Listing A.4: Single-spa Beispiel (index.js) Registrieren der MFs mit Hilfe von single-spa

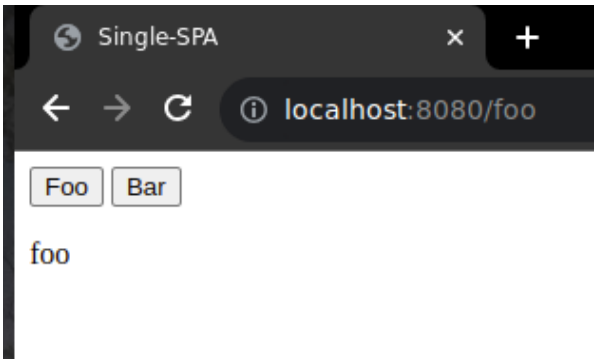


Abbildung A.2: Beispielwebseite single-spa: Routing anhand von zwei Buttons

A.4 ÜBERFÜHRUNG IN EINE MICRO-FRONTEND-ARCHITEKTUR

```
1  // Filters all npm packages that causes the issue
2  // https://github.com/webpack/webpack/issues/13457
3  const deps = Object.fromEntries(
4    Object.entries(require("../../package.json").dependencies).filter(
5      ([key]) =>
6        !(
7          key.includes("@angular/common") ||
8          key.includes("@angular/platform") ||
9          key.includes("rxjs") ||
10         key.includes("tslib") ||
11         key.includes("zone.js")
12       )
13   )
14 );
15 module.exports = {
16   output: {
17     publicPath: "http://localhost:4200/",
18     uniqueName: "shell",
19     scriptType: "text/javascript",
20   },
21   optimization: {
22     runtimeChunk: false,
23   },
24   plugins: [
25     new ModuleFederationPlugin({
26       remotes: {
27         profile: "profile@http://localhost:4201/remoteEntry.js",
28         authentication: "authentication@http://localhost:4202/remoteEntry.js",
29         home: "home@http://localhost:4203/remoteEntry.js",
30         article: "article@http://localhost:4204/remoteEntry.js",
31         settings: "settings@http://localhost:4205/remoteEntry.js",
32         editor: "editor@http://localhost:4206/remoteEntry.js",
33       },
34       shared: {
35         ...deps,
36         // bugfix for https://github.com/webpack/webpack/issues/13457
37         "@angular/common": {singleton: true, eager: true, strictVersion: true,
               requiredVersion: "16.0.1",},
38         "@angular/common/http": {singleton: true, eager: true, strictVersion:
              true, requiredVersion: "16.0.1",},
39         shared: {
40           import: "shared",
41           requiredVersion: require("../shared/package.json").version,
42         },
43       },
44     }),
45   ],
46 };
```

Listing A.5: webpack.config.js-Datei der Shell. im remotes-Attribut sind alle MFs aufgelistet, die die Shell aufrufen kann. Das Shared-Attribut beinhaltet alle Ressourcen, die die Shell selber benötigt.

```
1  // Holds a list with all loaded remotes
2  const moduleMap = {};
3
4  /**
5    * Loads a micro frontend by the remote entry point.
6    *
7    * Rejects if something went wrong
8    *
9    * @param remoteEntry Webpack's entry point to retrieve code from remote
10   * @returns
11   */
12 const loadRemoteEntry = async(remoteEntry: string): Promise<void> => {
13   if (moduleMap[remoteEntry]) {
14     resolve();
15     return;
16   }
17
18   const script = document.createElement("script");
19   script.type = "text/javascript";
20   script.src = remoteEntry;
21   document.body.appendChild(script);
22
23   moduleMap[remoteEntry] = true;
24
25   resolve();
26 }
27
28 async function lookupExposedModule<T>(remoteName: string, exposedModule:
       string): Promise<T> {
29   /**
30    * Code von der Dokumentation:
31    * https://webpack.js.org/concepts/module-federation/
32    * Es behandelt das Nachladen von fehlenden Ressourcen, die
33    * im Manifest eingetragen sind.
34    */
35 }
36
37 export async function loadRemoteModule(options: LoadRemoteModuleOptions):
       Promise<any> {
38   await loadRemoteEntry(options.remoteEntry);
39   return await lookupExposedModule<any>(options.remoteName, options.
         exposedModule);
40 }
```

Listing A.6: MF-loader.utils.ts Funktionen zum Nachladen und Einbinden von MFs durch module federation

```
/**
 * Inserts code in a new <script> tag on the bottom of the document
 *
 * The code will be execute immediately after resolve
 */
const loadScript = (code: any): Promise<void> => {
  return new Promise((resolve) => {
    const script = document.createElement('script');
    script.type = "text/javascript";
    script.appendChild(document.createTextNode(code));
    document.body.appendChild(script);
    resolve();
  });
}

// Holds a list with all loaded remotes
const moduleMap = {};

/**
 * Loads a micro frontend by the remote entry point.
 *
 * Rejects if something went wrong
 *
 * @param remoteEntry Webpack's entry point to retrieve code from remote
 * @returns
 */
const loadRemoteEntry = async(remoteEntry: string): Promise<void> => {
  if (moduleMap[remoteEntry]) {
    return;
  }

  const { data } = await axios.get<any>(remoteEntry, AXIOS_CONFIG);
  moduleMap[remoteEntry] = true;
  await loadScript(data);
}
```

Listing A.7: MF-loader.utils.ts Optimierung der Funktion loadRemoteEntry() und loadScript() durch den alternativen Einsatz von XML HTTPRequests (Axios)

Tabelle A.1: Tabelle aus [Kro21]: Es gibt eine Übersicht über viele Frameworks, die die Implementation einer MF-Architektur vereinfachen

Framework	Documentation	Free tooling choice	Technique	Type	Boilerplate generator	Multi-framework support	Base framework
Mosaic9	partly	no	Server-side	service collection	yes	no	
PuzzleJS	no	yes	Server-side	Server-side composition	no	no	
Podium	good	?	Server-side	Server-side composition	no	no	
BigPipe	partly	no	Server-side	Server-side composition	no	?	
Bit	good	no	Server-side	component library	yes	yes	
Luigi	good	no	Server-side	applicationshell	no	no	
Web Components	good	yes	Client-side	HTML-standard	no	yes	Polymer
Single-spa	good	yes	Client-side	applicationshell	yes	yes	
Quiankun	partly	?	Client-side	applicationshell	yes	yes	Single-SPA
Piral	good	yes	Client-side	applicationshell	yes	no	React
FrintJS	partly	no	Client-side	applicationshell	yes	yes	
Mooa	partly	?	Client-side	applicationshell	no	yes	Single-SPA
NgxPlanet	partly	no	Client-side	applicationshell	no	no	
icestark	no	?	Client-side	applicationshell	no	yes	
Isomorphic Layout Composer	partly	?	client and server-side	Server-side composition	no	yes	Single-SPA / TailorX
OpenComponents	partly	no	client and server-side	component library	yes	yes	
ARA	partly	?	client and server-side	applicationshell	yes	yes	Airbnb Hypernova
Webpack5ModuleFederation	good	no	Bundle-loading	?	no	yes	
Systemjs	partly	no	Bundle-loading	?	no	yes	
nuz	partly	?	Bundle-loading	?	yes	?	
One-App	no	no	?	component library	no	no	
Nut	no	?	?	?	?	?	
Cellular JS	no	yes	?	?	?	?	
Misk	partly	?	?	?	yes	no	React
Scalecube-js	no	?	?	?	?	?	
Berial	no	?	?	?	?	?	

A.4.1 *Fehlerbehandlung - Tabelle*

Tabelle A.2: Tabelle nach der Identifikation aller Fehler

Kategorie	**Fehler**
Webpack	Container bereits initialisiert
Webpack	Fehler bei Chunk nachladen
Webpack	Container nicht gefunden
Axios - keine Antwort	ERR_NETWORK
Axios - keine Antwort	ETIMEDOUT
Axios - keine Antwort	ERR_FR_TOO_MANY_REDIRECTS
Axios - keine Antwort	ERR_BAD_OPTION_VALUE
Axios - keine Antwort	ECONNABORTED
Axios - keine Antwort	ERR_DEPRECATED
Axios - keine Antwort	ERR_NOT_SUPPORT
Axios - keine Antwort	ERR_INVALID_URL
Axios - keine Antwort	ERR_CANCELED
Axios - keine Antwort	ETIMEDOUT
Axios - HTTP Code	Unauthorized (401)
Axios - HTTP Code	Conflict (409)
Axios - HTTP Code	TooEarly (425)
Axios - HTTP Code	InternalServerError (500)
Axios - HTTP Code	BadGateway (502)
Axios - HTTP Code	ServiceUnavailable (503)
Axios - HTTP Code	GatewayTimeout (504)
Axios - HTTP Code	VariantAlsoNegotiates (506)
Axios - HTTP Code	InsufficientStorage (507)
Axios - HTTP Code	TooManyRequests (429)
Axios - HTTP Code	Bad Request (400)
Axios - HTTP Code	Forbidden (403)
Axios - HTTP Code	NotFound (404)
Axios - HTTP Code	MethodNotAllowed (405)
Axios - HTTP Code	NotAcceptable (406)
Axios - HTTP Code	ProxyAuthenticationRequired (407)
Axios - HTTP Code	RequestTimeout (408)
Axios - HTTP Code	Gone (410)
Axios - HTTP Code	LengthRequired (411)
Axios - HTTP Code	PreconditionFailed (412)
Axios - HTTP Code	PayloadTooLarge (413)
Axios - HTTP Code	UriTooLong (414)
Axios - HTTP Code	UnsupportedMediaType (415)
Axios - HTTP Code	RangeNotSatisfiable (416)
Axios - HTTP Code	ExpectationFailed (417)
Axios - HTTP Code	ImATeapot (418)
Axios - HTTP Code	MisdirectedRequest (421)
Axios - HTTP Code	UnprocessableEntity (422)
Axios - HTTP Code	Locked (423)
Axios - HTTP Code	FailedDependency (424)
Axios - HTTP Code	PreconditionRequired (428)
Axios - HTTP Code	RequestHeaderFieldsTooLarge (431)
Axios - HTTP Code	UnavailableForLegalReasons (451)
Axios - HTTP Code	NotImplemented (501)
Axios - HTTP Code	HttpVersionNotSupported (505)
Axios - HTTP Code	LoopDetected (508)
Axios - HTTP Code	NotExtended (510)
Axios - HTTP Code	NetworkAuthenticationRequired (511)
Axios - HTTP Code	UpgradeRequired (426)
JavaScript - DOM	HierarchyRequestError

Tabelle A.3: Tabelle nach Identifikation und Analyse der Fehler (gefiltert nach Fehler mit Szenarien). Zur Übersicht gekürzt um die Kategorie. Die vollständige Tabelle ist auf TODO zu finden

Fehler	Szenario	Eintritts-wahrscheinlichkeit	Auswirkung	Lösungsversuch	Lösbarkeit
Container bereits initialisiert	Bug (loaded container map nicht integer)	gering	mittel / hoch	Webseite neu laden	gering
Fehler bei Chunk nachladen	alle Netzwerkfehler	hoch	mittel	erneut versuchen	mittel
ERR_NETWORK	lokaler Neztwerkdienst nicht verfügbar	hoch	hoch	erneut versuchen	mittel
ETIMEDOUT	Server überlastet	mittel	hoch	erneut versuchen	mittel
ERR_FR_TOO_MANY_REDIRECTS	Routingfehler Serverseitig	mittel	mittel	erneut versuchen	mittel
ERR_BAD_OPTION_VALUE	MAnipulierter request header	gering	hoch	ab- und wieder anmelden	hoch
ECONNABORTED	Abbruch-Ereignis geworfen (custom timeout)	mittel	mittel	erneut versuchen	mittel
Unauthorized (401)	JWT token ungültig	gering	hoch	ab- und wieder anmelden	hoch
Conflict (409)	Client nicht bereit für Anfrage	mittel	mittel	erneut versuchen	hoch
TooEarly (425)	In einem TCP 1.3. durchgeführten Handshake	mittel	mittel	erneut versuchen	hoch
InternalServerError (500)	Innerer Server Fehler	mittel	mittel	erneut versuchen	mittel
BadGateway (502)	Ungültige Antwort des Zielservers	mittel	mittel	erneut versuchen	mittel
ServiceUnavailable (503)	Temporärer Ausfall durch bspw. Überlastung	mittel	mittel	erneut veruschen (retry-after)	mittel
GatewayTimeout (504)	Zielserver timeout	mittel	mittel	erneut versuchen	mittel
VariantAlsoNegotiates (506)	allgemeiner Serverfehler	mittel	mittel	erneut versuchen	mittel
InsufficientStorage (507)	Ressource nicht verfügbar	mittel	mittel	erneut versuchen	mittel
Container nicht gefunden	Fehlerhafte remoteEntry.js (Bug)	gering	mittel / hoch	Webseite neu laden	gering

```
/**
* Build routes out of {@link MicroFrontend} objects
*
* @param options Micro frontend routes
* @returns new total routes
*/
private buildRoutes(options: MicroFrontend[]): Routes {
 return options.map((o) => ({
   path: o.routePath,
   loadChildren: () =>
     loadRemoteModule(o)
       .then((m) => m[o.ngModuleName])
       .catch((error: any) => {
         // remap error message for a better generic error implementation
         if (!error.code) {
           error.code = error.message;
         }

         // intercept error flow to inject information
         throw { targetMf: o, reason: error } as MfLoadError;
       }),
 }));
}
```

Listing A.8: (micro-frontend-route.factory.ts) Die Funktion erstellt die Routen anhand einer Liste zur Laufzeit. Somit wird die Fehlerbehandlung auf alle Routen angewandt

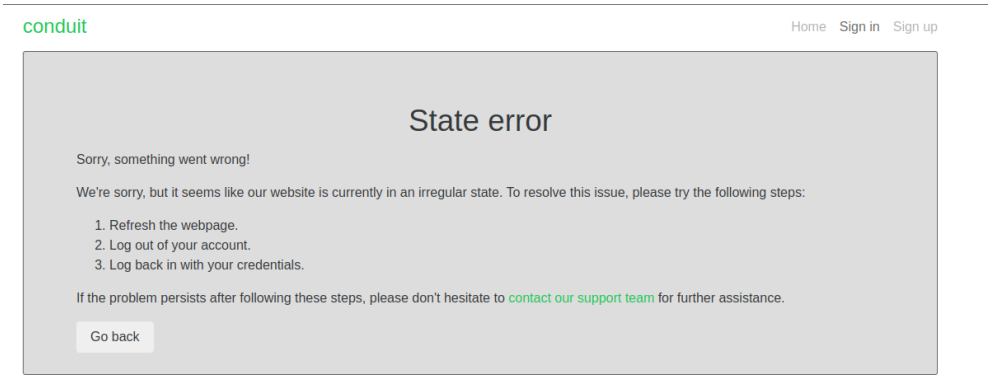


Abbildung A.3: Fehlerbehandlungskomponente Zustandsfehler

```
/**
  * Handles errors by merging different error structure codes to the
  * same format.
  *
  * @param error any error. The structure should look like
  *  {rejection: {reason: any}} | {reason: any}
  */
handleError(errorWrapper: any) {
  let error = errorWrapper.rejection || errorWrapper;
  if (!error?.reason) {
    // unknown error or error that doesn't match the known error structure
    throw errorWrapper;
  }

  this.routeNext(error);
}

/**
  * Determine the error route and navigate to it
  * Saves the given error to the local storage
  *
  * @param error containing additional information about the error cause
  */
private routeNext(error: MfLoadError) {
  const route = this.errorRouter.getRoute(error.reason);
  error.route = route;

  this.storage.set(MF_ERROR_LOAD_TOKEN, error);
  this.router.navigate([route.path]);
}
```

Listing A.9: (error.handler.ts) Filtert Nachladefehler und

A.4.2 *Übersichtsdiagramm zur Fehlerbehandlung*

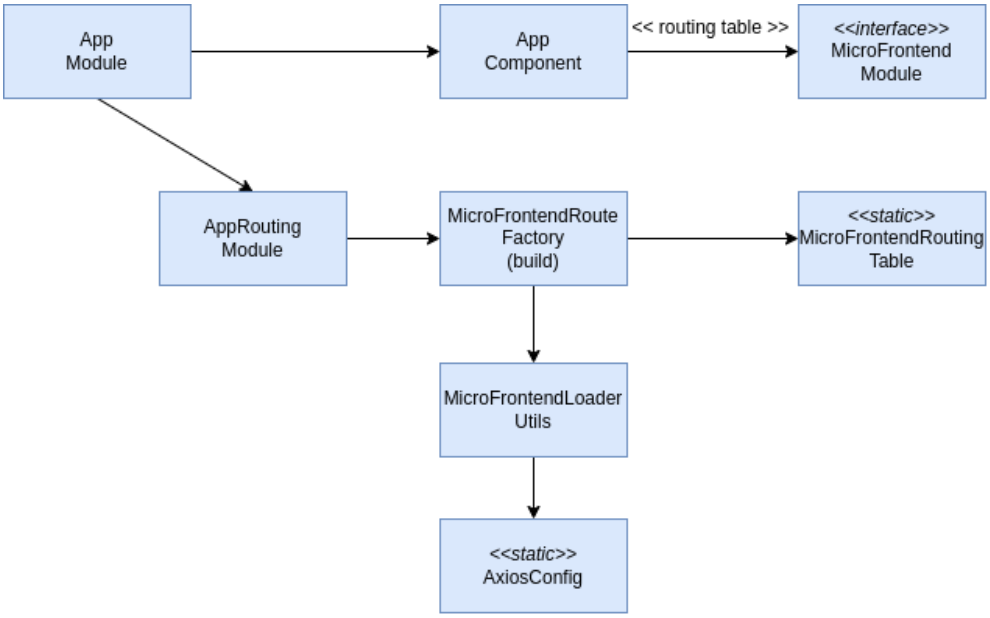


Abbildung A.4: Klassendiagramm ohne Fehlerbehandlung. Es werden alle relevanten Dateien zum Laden und Verwalten der MFs gezeigt

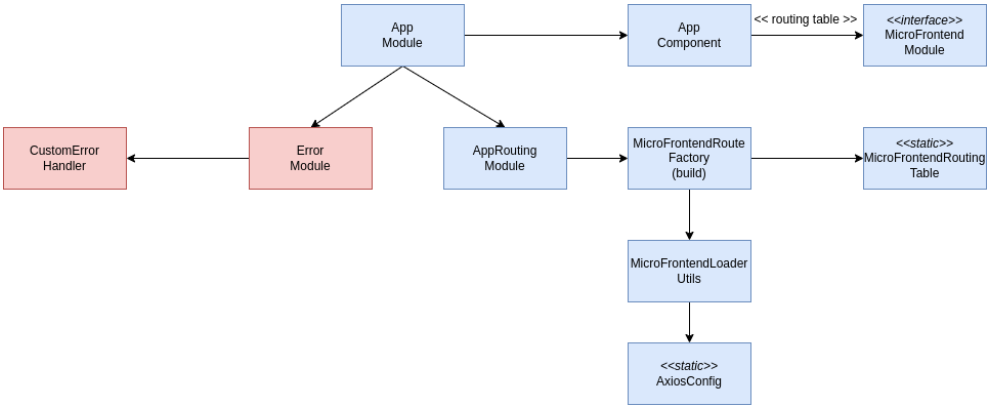


Abbildung A.5: Diagramm nach Durchführung von Schritt 1. Erweiterung durch ein *ErrorModule*, welches den *ErrorHandler* registriert. (rot markiert)

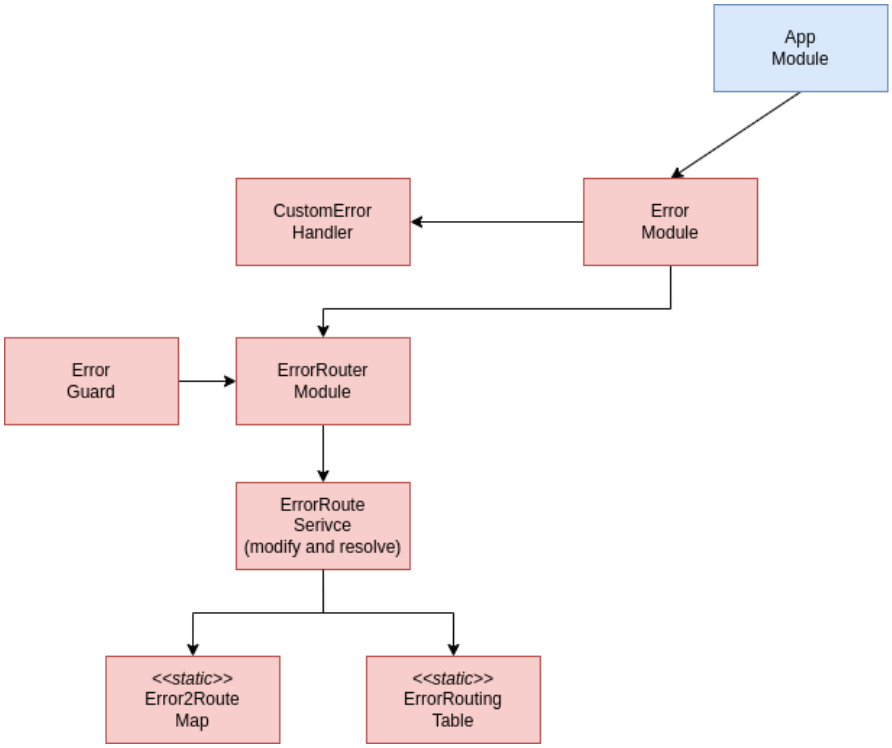


Abbildung A.6: Diagramm nach Schritt 2. Routen und Abbildungen auf Komponenten zur Verwaltung der Fehler sind erstellt. Sie werden durch den *errorRouteSerivce* verwaltet.

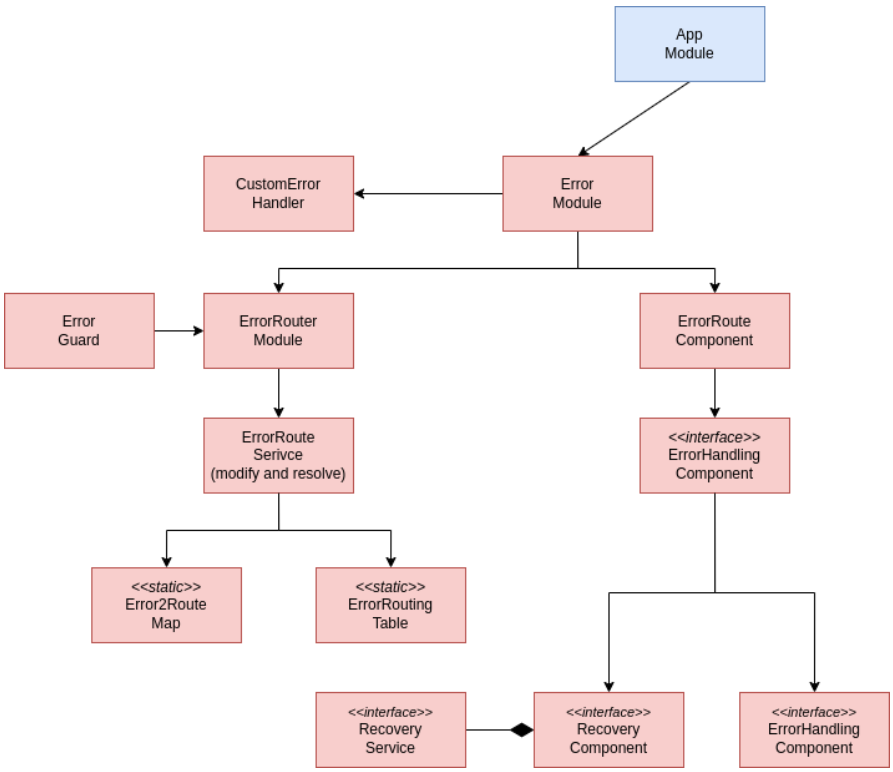


Abbildung A.7: Diagramm nach Schritt 3. Es wurde um die Struktur zur Anzeige der Fehler erweitert

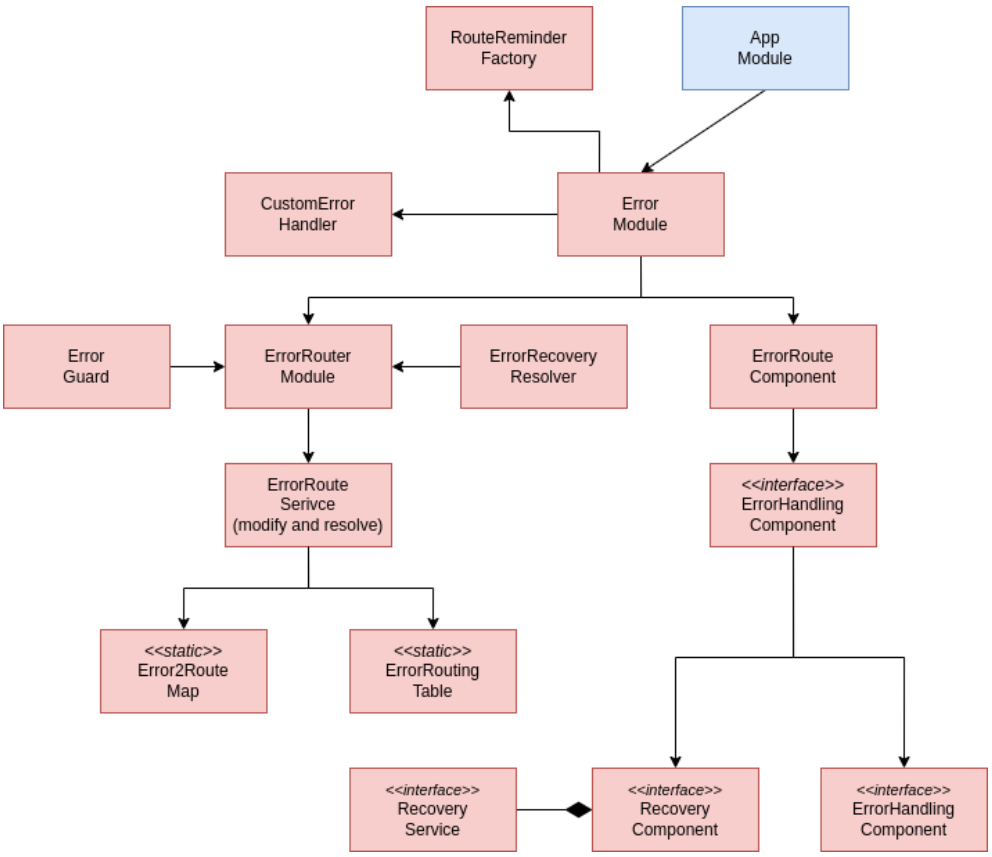


Abbildung A.8: Diagramm nach Schritt 4. Es hat sich um den *ErrorRecoveryResolver* sowie um die *RouteReminderFactory* erweitert

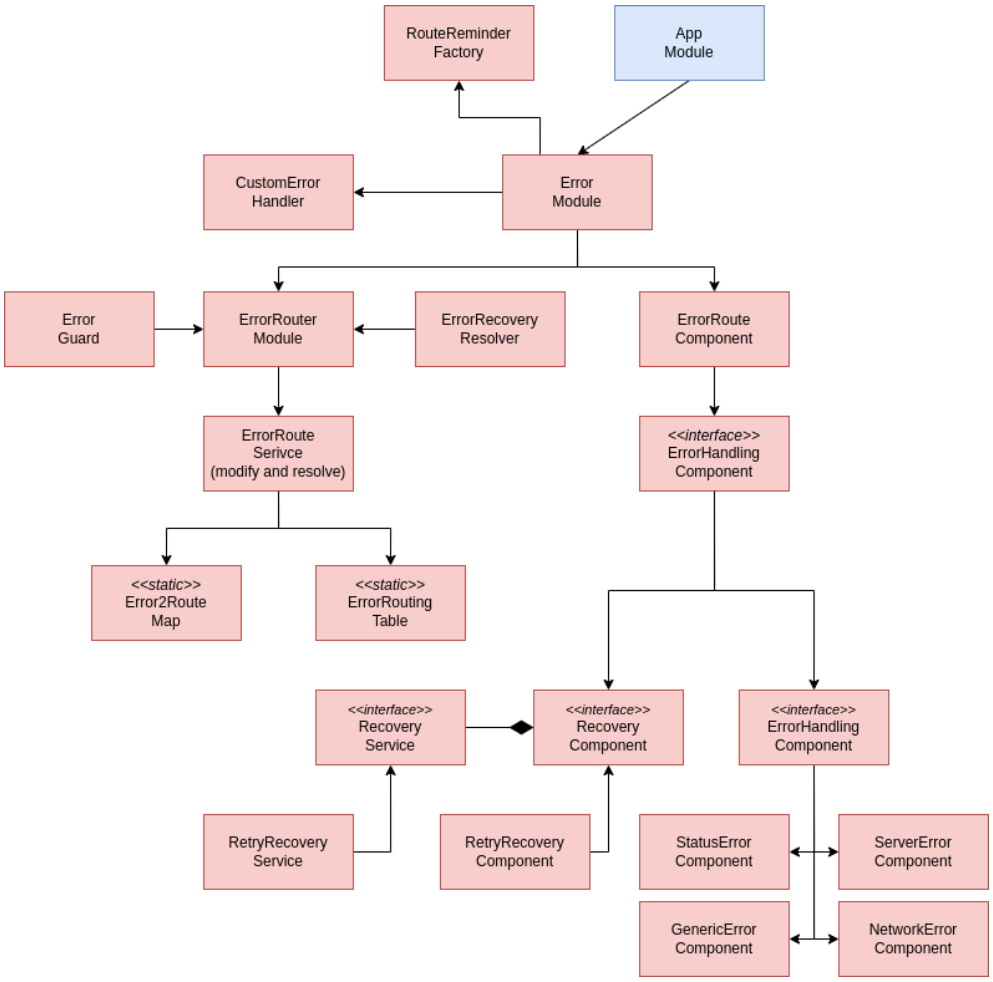


Abbildung A.9: Diagramm nach Schritt 5. Es sind die Komponenten zur Fehlerbehebung- und Behebung dazugekommen.