Extrahieren von Micro-Frontends aus einer monolithischen Frontend Anwendung

Masterarbeit

Billy J. Lando

Fachbereich 3: Mathematik und Informatik

24. Februar 2022

1. Gutachter: Prof. Dr. Wilhelm Hasselbring
Christian-Albrechts-Universität zu Kiel
2. Gutachter: Prof. Dr. Rainer Koschke
Universität Bremen
Betreut durch: Alexander Krause-Glau, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bremen, 24. Februar 2022

Zusammenfassung

Das Konzept von Microservice-Architekturen in der Softwareentwicklung gewinnt zunehmend an Aufmerksamkeit. Diese Architekturen entstehen oft aus dem Refactoring monolithischer Anwendungen. Viele Microservice-Architekturen kümmern sich wenig um das Frontend beziehungsweise um die Benutzeroberfläche. Oft führt die Zerschlagung zu vielen kleinen, lose gekoppelten Diensten, die in einem weiterhin monolithischen Frontend zusammengefasst sind. Mit einer Micro-Frontend-Architektur können diese lose gekoppelten Systeme systematisch/dynamisch in einer Benutzeroberfläche zusammengeführt werden. Die vorliegende Arbeit gibt einen Überblick über verschiedene Migrationsansätze zu Micro-Frontend-Architekturen und erprobt einen Migrationsansatz am Beispiel des Software-Visualisierungsprogramms ExplorViz.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1. Motivation

Seit dem Aufkommen der Idee der Micro-Frontends [Thoughtworks 2016] wurden unterschiedliche Aspekte und Techniken entwickelt und in vielen Beiträgen diskutiert. Micro-Frontends inspirieren sich von der Microservices-Architektur. Die Hauptidee dahinter ist, eine monolithische Codebasis in kleinere Teile zu zerlegen, ohne dass der Bereitstellungsdurchsatz der Gesamtsoftware verlangsamt werden muss.

Eine allgemeine heuristische Anleitung zur Transformation einer existierenden monolithischen Frontend zu einer Micro-Frontend-Architektur findet sich in den gefundenen Artikeln nicht. Sie lässt sich auch nicht daraus nativ ableiten, weil die Autoren andere Fragen verfolgten oder weil diese sich stark mit Lösungen in speziellen technischen Umfelder beschäftigten.

Aus den Quellen lassen sich aber unterschiedliche Konzepte ableiten, um eine monolithische Anwendung zu zerlegen und in Micro-Frontends umzubauen. Diese unterscheiden sich zum Teil erheblich sowohl in Bezug auf angewandte Techniken, Werkzeugunterstützung und Architektur sowie auch in Bezug auf die verfolgten Ziele. Da angesichts gewachsener, komplexer IT-Umgebungen statt eines Software-Neubaus heutzutage oft eher der Umbau und die Optimierung bestehender Softwarelösungen angestrebt wird, beschäftigt sich die vorliegende Arbeit mit der Frage, was generell wichtige Aspekte bei der Transformation von einer Softwarearchitektur in einer Micro-Frontend-Architektur sein könnten.

1.2. Zielsetzung

Der Umbau einer bestehenden monolithischen Webanwendung in einem Micro-Frontend ist ein größeres Unterfangen, weil viele Abhängigkeiten und Abläufe mit dem Umbau neu gestaltet werden müssen. Die Abgrenzung zwischen monolithische Webanwendung und Micro-Frontend lässt sich gut mit einer Analogie aus dem Wirtschaftsbereich beschreiben. Eine monolithische Anwendung ist wie ein gewachsenes Familienunternehmen, in welchem an verschiedenen Orten einzelne Abteilungen mit unterschiedlichen Aufgaben betraut sind. Notwendige Abhängigkeiten zwischen Mitarbeitergruppen werden meist über Dienstwege gelöst. Analog nehmen in einer monolithischen Softwarearchitektur einzelne Module aus

dem Gesamtcode unterschiedliche Aufgaben wahr, die dabei über interne Schnittstellen miteinander kommunizieren. Analog abgrenzend dazu steht eine Unternehmensholding für die Micro-Frontend-Architektur, in welcher an verschiedenen Orten in unabhängigen Tochterunternehmen Mitarbeitergruppen mit unterschiedlichen Aufgaben betraut sind. In der Analogie entsprechen die Tochterunternehmen als unabhängige Unternehmen den einzelnen Micro-Frontends in der Softwarearchitektur. Die Dienstwege, die in der Holding zu Geschäftsbeziehungen wurden, repräsentieren die externen Schnittstellen, die im Monolithen noch intern waren. Die Holding-Töchter sind nicht nur Teil des Gesamteindrucks der Holding, sondern sie können auch leichter mit externen Geschäftspartnern zusätzlichen Umsatz und Gewinne für die Unternehmensholding generieren. Dieser letzte Aspekt der Analogie deutet auch das Erweiterungspotential von Micro-Frontend an.

Beim Softwareumbau bestehen die Herausforderungen in der Entwicklung einer neuen Architektur und in der Konzeption des Weges, der eine effiziente Zerlegung des Monolithen in Microservices ermöglicht und der gleichzeitig die Kommunikation zwischen den Microservices einfach und überschaubar hält. Angesichts steigender Sicherheits- und Datenschutzanforderungen ist eine weitere Anforderung, das Gesamtkonstrukt robust zu halten und es sicher gegen Angriffe zu machen.

Softwareentwickler entscheiden sich oft dafür, eine Architektur einer anderen vorzuziehen, basierend auf ihrer Erfahrung in früheren Projekten oder basierend auf den wahrgenommenen Vorteilen der neuen Architektur. Daher ist es wichtig zu untersuchen, warum Micro-Frontends eingeführt wurden und die aktuellen Beweggründe hinter ihrer Einführung zu verstehen. Weiter ist zu untersuchen, ob bestimmte Probleme als verbesserungsbedürftig angesehen werden als andere. Um die genannten Aspekte herauszuarbeiten, wird in dieser Arbeit ein Literaturreview zum Stand der Forschung im Bereich Micro-Frontends durchgeführt.

Neben einem Literaturreview sollen mit der expliziten Transformation einer kleineren monolithischen Software in eine Micro-Frontend-Architektur eigene Erfahrungen gesammelt werden. Diese Erfahrungen und die dazugehörige Reflektion sollten hilfreich sein, um Ideen, Arbeitsabläufe, Checklisten, Heuristiken und Strategien als Methodensammlung abzuleiten, die für eine erfolgreiche Transformation hilfreich sein können. In die Reflektion der eigenen Erfahrungen zur Transformation von 'ExplorViz' fließen auch Rechercheergebnisse zum aktuellen Stand der Forschung/Technik mit ein. In der Reflektion sollen die Vorteile und Nachteile der Techniken herausgestellt, die Strategie-Ideen beschrieben und gegebenenfalls vergleichend alternative Ansätze aus der Literatur diskutiert werden.

1.3. Aufbau

Die Arbeit ist in sechs weitere thematische Bereiche unterteilt. Das Kapitel 2 folgt mit den Grundlagen dieser Arbeit. Das Kapitel 3 erläutert die gewählte Methodik zur Recherche und zur Auswahl bestehender Literaturen zum Thema Micro-Frontend. In den gewählten Literaturen sind Einsatzgründen, Vor- und Nachteile und Micro-Frontend-Architektur

Ansätze zu finden. Das Kapitel 4 verschafft einen Überblick zu diesen Punkten aus den gefundenen Literaturen. Die Ergebnisse der Literaturrecherche werden genutzt, um eine Migrationsstrategie für eine exemplarische Software ExplorViz zu entwickeln und ist im Kapitel 5 zu finden. Im Kapitel 6 werden die Implementierungsdetails zu dieser Transformation erläutert und die Arbeiten werden reflektiert. Das Kapitel 7 fasst die wichtigsten Ergebnisse und Erkenntnisse dieser Arbeit nochmal zusammen.

Kapitel 2

Grundlagen

Eine Architektur mit Micro-Frontends verknüpft asynchron auf einer dynamisch generierten Oberfläche alle notwendigen Prozesse und stellt die Ressourcen zusammen, die für die Bewältigung der jeweils aktuellen Aufgabe gebraucht werden. Ein Micro-Frontend nutzt clientseitig den Browser als Betriebssystem. Als eine frühe Form von Micro-Frontends können statische Webseiten gelten, die das HTML-Tag `<frame>` verwendet haben, um unterschiedliche Webseiten in einem Frontend darzustellen [Raggett u. a. 1998]. Der Begriff 'Micro-Frontend' ist relativ neu. Eine frühe Erwähnung fand es bei Thoughtworks [Thoughtworks 2016]. In den nachfolgenden Unterkapiteln wird zur Verdeutlichung der Micro-Frontends auch das klassische Webkonzept beschrieben.

2.1. Typische Webseiten

2.1.1. Strukturmerkmale

Mit Beginn der kommerziellen Phase des Internets um 1990 verstand man unter einer Website eine Kombination aus statischen Websites. Im Gegensatz zu einzelnen Websites, die per URL aufgerufen werden können, meint die Webseite (mit 'e') in der vorliegenden Arbeit immer nur allgemein zu einer 'Domainadresse' das Frontend, welches dem Nutzer im Browser angezeigt wird [Ryan 2010]. Von einer Webanwendung wird in dieser Arbeit gesprochen, wenn dem Nutzer verschiedene dynamische Website im Frontend in einer Gesamtansicht dargestellt werden.

Eine Webanwendung ist eine verteilte Anwendung mit einer Clientseite und mit einem beziehungsweise mehreren Servern. Die Clientseite der Anwendung läuft auf einem Webbrowser, während der Server beziehungsweise die Server als Datenlieferanten im jeweiligen Webanwendungsserver fungieren.

Ein Computer, ein Smartphone oder jedes andere Gerät, mit dem ein Nutzer mittels eines Browsers Webseiten und Daten abruft, wird als Client bezeichnet. Eine Datenquelle für eine Webseite wird als Server bezeichnet, weil sie dem Nutzer die angeforderten Daten liefert. Eine typische Webseite bezieht ihre Informationen fast immer nur von einem Server. Die Kommunikation zwischen den beiden Teilen kann mit dem Client-Server-Modell bezeichnet werden, dessen Hauptaufgabe darin besteht, als Server Anfragen der Nutzer entgegenzunehmen und die Antwort an den Client zu übermitteln. Die Art und Weise, wie

Client und Server miteinander kommunizieren, wird durch die Webanwendungsarchitektur festgelegt. Bei der Entwicklung der Webanwendungen werden meist die Hauptfunktionen in Schichten oder Ebenen unterteilt. Auf diese Weise kann jede Schicht leicht ersetzt und unabhängig aktualisiert werden. Dieses Architekturmuster wird als Multi- oder Three-Tier-Architektur bezeichnet [Ingeno 2018].

Aus der Sicht der funktionalen Beschreibung einer Webseite ist die Präsentationsschicht für die Benutzer über einen Browser zugänglich und besteht aus den Komponenten der Benutzeroberfläche, die die Interaktion des Nutzers mit dem System unterstützen. Sie wird hauptsächlich mit drei Kerntechnologien entwickelt: HTML, CSS und JavaScript. Während HTML der Code ist, der den Inhalt der Website strukturiert, steuert CSS das optische Aussehen der Webseite. JavaScript ist die gängige Programmiersprache der Browser und macht mithilfe von Skripten die Webseite interaktiver, indem die Skripte auf Aktionen des Benutzers reagieren [Ingeno 2018]. Die Benutzeroberfläche beziehungsweise Präsentationsschicht wird auch Frontend genannt. Die Geschäftsschicht nimmt im Gegensatz dazu die Benutzeranfragen von der Präsentationsschicht (Browser) entgegen, verarbeitet sie und bestimmt, ob und wie auf die angeforderten Daten zugegriffen wird. Sie reagiert auf die Nutzeranfragen und merkt sich diese gegebenenfalls. Auch die Arbeitsabläufe, mit denen die Daten und Anfragen das Backend durchlaufen, sind in dieser Schicht kodiert [Ingeno 2018]. Die Erinnerungsschicht beziehungsweise Persistenzschicht ist ein zentraler Ort, der den Zugriff auf den persistenten Speicher einer Anwendung ermöglicht. Die Persistenzschicht ist eng mit der Geschäftsschicht so verbunden, dass die Logik weiß, mit welcher Datenbank sie sprechen muss, und der Datenabrufprozess optimiert wird. Die Geschäftsschicht ist zustandslos. Sie kenne keine Erinnerung nur Regeln. Erst die Persistenzschicht macht die Anwendung individuell, indem es das Erinnern ermöglicht.

2.1.2. Herausforderungen beim Updaten

Gemäß dem Client-Server-Modell können über die Skripte und Programme beim Server verschiedene weitere Dienste oder Module so angesprochen werden, wobei diese informatorischen Abhängigkeiten dem Client verborgen bleiben, weshalb man eine solche Architektur auch als Monolithischen-Architektur bezeichnen kann. Eine monolithische Architektur macht es sehr oft erforderlich, das selbst eine geringfügige Änderung in der Präsentationsschicht, zum Beispiel in CSS-Klassen, eine neue vollständige Bereitstellung des Projekts erfordern. Mit zunehmender Projektgröße sind oft verschiedene Projektteams beteiligt, deren Arbeiten koordiniert werden müssen. Der gesamte Update-Prozess einer solchen Monolithischen-Architektur dauert umso länger, je mehr Module und Funktionalitäten hinzukommen und je mehr Teams an einem solchen Monolithen mitarbeiten.

Die Nutzung von automatisierten Tests ist ein Versuch, den Koordinierungsaufwand für die anwachsende Komplexität des Monolithen bei gleichbleibender Qualität weiterhin in akzeptabler Zeit zu gewährleisten. Ein weiteres Konzept, mit der wachsenden Komplexität umzugehen, liegt in der Standardisierung und Automatisierung von Abläufen. Die meisten Webseiten werden deshalb heutzutage mit Frameworks oder auch Content-

Management-Systeme(CMS) erzeugt, die deren standardisierten Funktionen für bestimmte Anwendungsfelder optimiert sind. Ein Content-Management-System (CMS) ist eine Softwareanwendung, mit der Benutzer digitale Inhalte erstellen, bearbeiten, gemeinsam bearbeiten, veröffentlichen und speichern können. Ein Software-Framework, auch Framework genannt, bezeichnet gemäß D. Riehle und T. Gross [Riehle und Gross 1998] in der Softwaretechnik eine wiederverwendbare Abstraktion, die für bestimmte Anwendungsfälle erstellt werden kann und spezifische Funktionen bereitstellt. Ein Framework ist hilfreich bei der Erstellung individueller Programmanwendungen.

Frameworks wie Shopware[1] und Pimcore[2] werden oft für die Entwicklung individueller Internetshops verwendet. Content-Management-Systeme wie Wordpress[3], Dupral[4] oder TYPO3[5] finden oft im Bereich redaktioneller Informationsseiten ihr Einsatzgebiet. Frameworks oder Content-Management-Systeme gibt es auch für Foren, für Chatsysteme, für Ticketsysteme und auch für viele andere Anwendungsfälle.

2.2. Evolutionäre Entwicklung von Webseiten zu Single-Page-Anwendungen

2.2.1. Evolutionäre Entwicklung

Mit der technischen Weiterentwicklung des JavaScripts, insbesondere für das dynamische Laden von Daten zur Laufzeit, auch als asynchrones JavaScript und XML beziehungsweise als Ajax bekannt, eröffneten sich für die Entwickler innovative Möglichkeiten der Webseitengestaltung. Eine Webseite konnte/sollte/musste, unterstützt durch die weiterentwickelte Technik bei den Browsern, Daten dynamisch intern auf dem Client speichern und auf äußere Impulse(Maus-Klicks, Tastendruck oder auch externe Geräte) unter Berücksichtigung der gespeicherten Daten reagieren können. Parallel mit diesen Wünschen wurden von Entwicklern JavaScript Frameworks für das Clientseitige Rendering entwickelt wie zum Beispiel Jquery(früher, heute veraltet) [6], Angular[7], ReactJS [8], Ember.js[9] oder auch VueJS [10].

Die verbesserten Rendering-Fähigkeiten im Frontend, die einerseits der technischen Weiterentwicklung der (Browser-)Software aber auch merklich den Innovationen im Hardwarebereich geschuldet sind, beschleunigten den Entwicklungstrend, bisherige Aufgaben

[1] https://www.shopware.com/
[2] https://pimcore.com/
[3] https://wordpress.com/
[4] https://www.drupal.org/
[5] https://typo3.org/
[6] https://jquery.com/
[7] https://angular.io/
[8] https://reactjs.org/
[9] https://emberjs.com/
[10] https://vuejs.org/

des Servers in das Frontend zu verlagern. Beispielhaft sei die Sortierung von größeren Datenmengen mit JavaScript genannt.

Im Laufe der Zeit gab es weniger Webseiten, die Benutzeraktivitäten und Datenbankabfragen nur in der Geschäftsschicht des Backends abwickelten und die Präsentationsschicht nur statische Webseiten zur Verfügung stellten. In der modernen Zeit erfolgt das Datenhandling vermehrt über Programmierschnittstellen, die für die partielle Seitengenerierung im Server (Schlagwort: Ajax) oder direkt für die Ausführung im Browser des Benutzers (Schlagwort: Headless) verwendet werden konnte. Heutzutage werden Webseiten, die das gesamte Rendering im Client verarbeiten, oft auch als 'Single Page Application' (SPA) bezeichnet.

2.2.2. Herausforderungen

Dieser moderne Trend brachte nicht nur Vorteile, sondern sie brachte auch neue Herausforderungen. Mit Zunahme der Funktionen im Frontend stiegen auch die Anforderungen an das Design, weil damit auch geeignete Sicherheitseinstellungen bereitzustellen waren und eine gute Performanz auf unterschiedlichen Ausgabegeräten sichergestellt werden musste [Rappl 2021, S. 7]. Mit Blick auch den Wunsch, eine verbesserte Anwendung kontinuierlich bereitstellen zu können, stellen die beiden genannten Aspekte für Single-Page-Anwendungen eine Herausforderung dar. Die in dieser Arbeit bearbeitete Micro-Frontends-Architektur ist ein Konzept, um den neuen Anforderungen und Wünschen gerecht zu werden.

2.3. Monolithische Software-Architektur

2.3.1. Monolith als Metapher

Der Begriff Monolith bezeichnet meist ein geologisches Merkmal und meint einen Festkörper, der aus einem einzigen Stein, Fels oder einen von Menschen geschaffenen Materialblock besteht. In dieser Arbeit wird das Wort breiter und allgemeiner verwendet, wobei die Idee dieselbe bleibt: Ein Monolith ist ein monolithisches Softwareprodukt und meint damit eine einzelne, unteilbare Einheit, die im Allgemeinen zu einer großen Größe anwächst. Eine Webseite mit monolithischer Architektur wird klassischerweise so erstellt, dass ihre Bereitstellung in einem einzigen Block (einzigen Stein) erfolgt.

2.3.2. Strukturmerkmale

Die heute übliche Art, nach welcher die meisten Webseiten konstruiert sind, kann als monolithischen Architektur bezeichnet werden. Die Präsentationsschicht der Webseite entsteht als ein fertiges Paket abrufbarer Dateien(HTML, CSS, JavaScript, Bilder, ...) auf dem

Server, auf welchem die HTTP-Anfragen verarbeitet, auf dem die Geschäftslogik ausgeführt wird und auf dem die Interaktionen in der Persistenzschicht gespeichert werden.

Diese monolithische Architektur hat Vor- und Nachteile. Ein Vorteil ist: Sie ist einfach zu entwickeln, da der initiale Aufwand zum Aufsetzen des Projektes gering ist und der Entwickler alle Systemzustände kontrollieren und bestimmen kann. Außerdem bietet sich innerhalb von Monolithen die Wiederverwendung von Code-Abschnitten zwischen einzelner Software-Komponenten beziehungsweise Module an, was die Existenz von Software Klonen vermindern hilft. Wenn die Software als monolithischer Block besteht, ist es einfacher Funktionstests durchzuführen. Neben diesen Vorteilen gibt es auch Nachteile. Ein wichtiger Nachteil ist die oft per Code explizit definierten, enge Kopplungen zwischen den Teilen und auch innerhalb der einzelnen Blöcke, die mit dem Größer-Werden beziehungsweise mit dem Komplexitätswachstum des Monolithen zunehmend zu potenziellen, schwer zu findenden Fehlerquellen und damit zu wachsenden Herausforderungen werden.

2.4. Microservices

2.4.1. Konzept als Gegenentwurf zu monolithischen Software-Architekturen

Im Vergleich zu klassischen Webseiten mit einer monolithischen Geschäftsschicht schlägt die Idee der Microservice-Architektur eine neue Richtung der Entwicklung ein. Mit Microservices versucht man, die komplexen Verpflichtungen in der Geschäftslogik auf der Serverseite in kleine autonome (unabhängige) Service-Strukturen zu zergliedern. Der Trend zu Microservice-Architekturen kann als Reaktion auf die gewachsene Komplexität der monolithischen Software-Architekturen mit ihren wachsenden Herausforderungen verstanden werden.

2.4.2. Vor- und Nachteile

In größeren, komplexeren Projekte wurde oft die Erfahrung gemacht [Newman 2019] [Hunter II 2017], dass Microservices einige Vorteile mit sich bringen.

- Microservice-Architekturen sind robuster gegenüber Totalausfällen. Ausfälle betreffen meist temporäre nur einzelne Dienste.
- Mehrere Teams können unabhängig voneinander arbeiten.
- Die Reaktion auf Fehler ist schneller und häufiger möglich. Eine agile evolutionäre Entwicklung wird unterstützt.
- Frameworks und Programmiersprachen sind freier wählbar. Spezialisierungen der Mitarbeiter können besser berücksichtigt werden.

- Dienste haben spezifische Ziele, wie zum Beispiel die Erfassung von Stunden in einer Software, was zur Wiederverwendung von Diensten und zur Zusammenfassung von Kenntnissen über bestimmte Aufgaben in Diensten führt.

Neben diese Vorteile sind auch einige Kritikpunkte in der Literatur zu finden.

- Die Entwicklung von Architekturen mit Microservice-basierten Anwendungen wird dadurch erschwert, dass die Geschäftslogik in solchen Anwendungen stark auf viele unabhängige und sich asynchron entwickelnde Microservices verteilt ist [Soldani u. a. 2018].
- Die Auswertung der Software Performanz wird dadurch erschwert, dass die Dienste verteilt und unabhängig entwickelt werden. Dies wird insbesondere dann zum Problem, wenn zur Evaluation der Performanz die Gesamt-Software berücksichtigt werden soll/muss [Soldani u. a. 2018].

2.5. Micro-Frontends

2.5.1. Strukturmerkmale

Die Architektur der Micro-Frontends kann als evolutionäre Weiterentwicklung der Idee der Microservice-Architektur verstanden werden. Während für die Microservices ein gemeinsames Frontend konstruiert wird, geht das Konzept der Micro-Frontends noch ein Schritt weiter und erwartet von jedem Microservice, dass es sich selbst um Darstellung in der Präsentationsschicht beim Client kümmert. Die Micro-Frontends werden damit autonomer als die Microservices. Mit der zusätzlichen Autonomie entstehen auch neue Herausforderungen. Eine gute Webseite sollte zum Beispiel einen Wiedererkennungswert haben, der sich oft an bestimmten Farben, Schrifttypen und anderen Stilelementen festmachen lässt. Die Stilelemente müssen sich im Layouts von jedem Frontend eines Microservices wiederfinden, um für den Nutzer einen stimmigen Gesamteindruck vermitteln zu können. Hier sind gegebenenfalls verschiedene Entwicklergruppen zu koordinieren. Analoges gilt für jede Änderung bei der Layoutvorgabe, denn jede Änderung von Layout-Elementen macht gegebenenfalls auch Änderungen in jedem Micro-Frontend erforderlich. Dieser Nachteil verliert aber an Brisanz, wenn man eine evolutionäre Entwicklung des Frontend-Designs in Kauf nimmt. Dies bedeutet, dass Änderungen der Layoutvorgaben nicht sofort in jedem Micro-Frontend erfolgen müssen, sondern dass die Änderungen auch sukzessive im Laufe der Weiterentwicklungen der Micro-Frontends erfolgen dürfen. Für eine Übergangszeit nimmt man den Alt-Neu-Mischmasch im Gesamtlayout in Kauf.

2.5.2. Vor- und Nachteile

Unter dem Begriff Micro-Frontend versammeln sich verschiedene organisatorische und/oder architektonische Ansätze, um beginnend beim Nutzer und dessen Erwartungen eine

auf dem Browser basierende, dynamisch-adaptive Software zu entwickeln. Dynamisch-adaptiv meint dabei, dass der Nutzer die Webseite schnell an seine Bedürfnisse anpassen kann.

Die sich aus der Untergliederung ergebenden Vor- und Nachteile für die oben erläuterten Microservices gelten auch für die Micro-Frontends. Ein Vorteil der Micro-Frontends im Vergleich zu den Microservices ist aber, dass das Frontend schneller und flexibler an die Erfordernisse angepasst werden kann, weil jedes Micro-Frontend für sein 'Frontend' selbst verantwortlich ist. Diese Autonomie ist gleichzeitig aber auch eine Herausforderung, weil die Webseite mit den Micro-Frontends meist einen Wiedererkennungswert haben soll und weil sich jedes Micro-Frontend-Entwicklerteam an Regeln für das Corporate-Design (CD) halten muss. In dem CD können sich Vorgaben für Farben, Schriftarten, Abstände und andere Stilelemente wiederfinden. Die Stilelemente müssen die Layouts der einen Microservices berücksichtigen und dürfen auf der Webseite nicht mit anderen Micro-Frontends in Konflikt kommen.

Bei größeren Projekten könnte im Idealfall arbeitsorganisatorisch die Entwicklung und Wartung der Micro-Frontend-Architektur auf mehrere Teams so verteilt werden, dass jedes Team sich um ein Micro-Frontend kümmert, ohne die anderen Teams zu stören oder auf diese warten zu müssen.

2.5.3. Herausforderungen bei der Architektur-Transformation

Die Micro-Frontend-Architektur erweitert die Konzepte von Microservices insofern, als jeder Microservice auch für seine Präsentationsschicht, also für seine Frontend-Ansicht, zuständig ist. Es verwandelt monolithische Webanwendungen von einer einzigen Code-basierten Anwendungsarchitektur in eine Anwendung, die mehrere kleine Frontend-Anwendungen zu einem - passende konfigurierten Ganzen kombiniert. Die Idee dahinter ist, eine Webanwendung als Kombination von Funktionen oder Business-Subdomains zu behandeln, die dem Nutzer hilft, schneller, effektiver und ohne Ballast-Informationen in der Frontend-Ansicht seine Informationsbedürfnisse zu befriedigen.

Kapitel 3

Methodik

Wissen über Micro-Frontends zu sammeln sowie Herausforderungen beziehungsweise Heuristiken im Zusammenhang mit Migrationen zu Micro-Frontends-Architekturen zu identifizieren, liefert gute Hinweise für die Migration einer monolithischen Softwarearchitektur in einer Micro-Frontend-Architektur. Die Herausforderungen, die während der Literaturrecherche häufig genannt wurden, sollen in dieser Arbeit zusammen mit Strategien zu deren Bewältigung zusammengestellt werden. Diese Herausforderungen und Strategien werden anschließend als helfende Grundlagen für eine Migration einer bestehenden Software verwendet. Die Literaturerkenntnisse werden auch für eine reflektierende Diskussion der eigenen Erfahrungen zur eigenen Migration genutzt.

Persönliche Anmerkung des Autors dieser Arbeit: *'Ein primäres Interesse für mich als Forscher besteht darin, die verschiedene Themen zukünftiger Forschungsaktivitäten zum Aufbau von Micro-Frontends zu identifizieren'*.

3.1. Forschungsmethodik zum Prozess des Literaturreview

Die systematische Untersuchung der Literatur zum Stand der Praxis beim Thema Micro-Frontends soll die benötigte Wissensbasis schaffen, die die Voraussetzung ist, um daraus neue Ideen, Theorien und Lösungen abzuleiten und zu entwickeln, um Forschungsimplikationen zu analysieren beziehungsweise zu experimentieren und um Ideen für zukünftige Forschungsdimensionen zu entwickeln. Dafür wurden Artikel von verschiedenen (wichtigen) Unternehmen, Blog-Posts, wissenschaftliche Publikationen und Bücher gelesen. Diese Quellen wurden insbesondere auf die Fragen hin untersucht, warum Unternehmen, Softwareentwickler sich für die Arbeit mit Micro-Frontends entscheiden und welche Taktiken und Strategien bei der Transformation zu Micro-Frontend-Architekturen zur Anwendung kamen. Weiter sollte mit dem Literaturreview herausgearbeitet werden, welche Beschränkungen, Nachteile und/oder Probleme von Micro-Frontends aktuell diskutiert und welche Vorteile in der Nutzung von Micro-Frontend-Architekturen gesehen werden.

Der erste Schritt des systematischen Reviews ist die Festlegung des Rechercheumfangs. Angesichts der oben genannten allgemeinen Zielfragen und angesichts der Neuheit der Idee der Micro-Frontends war zu erwarten, dass mit dem Review nicht ausreichend Beiträge zu dem Thema Micro-Frontends zu finden sind, wenn man sich nur auf die klassischen wissenschaftlichen Publikationsorgane beschränkt hätte.

Die Informatik ist eine junge Wissenschaft. Viele neue Ideen werden von den Forschenden nicht in konservativen Wissenschaftspublikation veröffentlicht, sondern in Mitteilungsorganen der manchmal sogenannten graue Literatur veröffentlicht. Im Gegensatz zum Beispiel zu den Naturwissenschaften, in welchen man oft klar zwischen Grundlagenforschung und Anwendungsforschung institutionell und personell unterscheiden kann, findet in der Informatik viel Grundlagenforschung im Bereich von Unternehmen und parallel zur praktischen Arbeit statt. Aus diesem Grunde sind für den Review zum Stand der Diskussion auch problematisch zu zitierende Quellen wie zum Beispiel Blogs zu berücksichtigen. Schwierig zitierbar sind die Quellen deshalb, weil man nicht sicher sein kann, dass die Informationen noch Jahre später verfügbar und nachlesbar sind und/oder dass die Quellen nicht nachträglich ohne Kennzeichnung verändert werden. Die graue Literatur ist auch deshalb schwer zitierfähig, weil wegen des wissenschaftlichen Vergessens bei ihr nicht sicher unterstellt werden kann, dass die Originalquellen in 30 oder 50 Jahren in Bibliotheken oder anderen ähnlichen Wissensspeichern noch verfügbar sind. Für den Themenbereich Micro-Frontends erhält man derzeit leider meist nur über die graue Literatur Einblicke in die aktuell vorherrschenden Trends und Ideen.

Wie oben erläutert, ist aus verschiedenen Gründen (Neuheit, junge Wissenschaft) in der klassischen Literatur eine Erweiterung des zu durchsuchenden Literaturpool um die graue Literatur nötig. Systematisch wird zusätzlich zum grauen Literaturreview-Prozess [Garousi u. a. 2016] der sogenannte 'multivocale Literaturreview' [Garousi u. a. 2017] genutzt, wobei die als 'automatisch übersetzt' gekennzeichneten Texte unberücksichtigt blieben. Der multivocaler Literaturreview-Prozess ist eine Variante des sogenannten systematischen Literaturreview Prozesses, bei dem die graue Literatur wie Blogbeiträge zusätzlich zur veröffentlichten Wissenschaftsliteratur wie zum Beispiel Zeitschriften- und Konferenzbeiträge berücksichtigt wird [Garousi u. a. 2017]. Dieser Prozess erweitert methodisch auch den grauen Literaturreview [Garousi u. a. 2016].

Für den Review in dieser Arbeit wird einem systematischen Ansatz gefolgt, der auf der von Petersen et al. [Petersen u. a. 2008] entwickelten Methodik zur Durchführung systematischer Literaturrecherche im Bereich 'Software-Engineering' basiert. Dessen Methodik erfordert es mehr Aufwand als eine einfache Literaturrecherche, weil unterschiedliche Quellen nach bestimmten Checklisten einzeln bewertet werden sollen, um möglichst qualitative Quellen zu erhalten.

Die aktuell vorliegende Arbeit baut auf einer früheren Bachelorarbeit [Lando 2019] zu dem Thema auf. Die Liste der Artikel, Blog-Beiträge, Paper, Bücher, Konferenz-Vorträge, Kurse, Proof-of-Concepts und weitere Quellen dort ist um neuere Publikationen erweitert worden. Die aktualisierte, nicht systematisch sortierte Literaturliste ist auf Github [11] zu finden.

[11] https://github.com/billyjov/microfrontend-resources

3.2. Aspekte der Methodik zur systematischen Literaturrecherche

Die zur Durchführung der Recherche erforderlichen Daten wurden mithilfe elektronischer Bibliotheken unter anderem bei der Bremer Universitätsbibliothek und mithilfe der Google Suche gesammelt. Neben der allgemeinen Google-Suche zählt dazu auch Googles erweiterte Büchersuche [12] und Google Scholar. Diese wurden ausgewählt, um das Ergebnis aus ihren vordefinierten Quellen von den beliebtesten akademischen Verlagen (zum Beispiel Science Direct[13], ACM[14], Springer[15] oder auch IEEE[16]) zu erhalten. Die anderen Datensätze stammen aus grauer Literatur, die in anderen Datenbanken, Websites, Katalogen, Videos, Vorträgen, Blogs, Büchern gefunden wurden. Folgende Suchbegriffe wurden zur Durchführung der Suche ausgewählt: 'Micro-Frontends', 'Micro frontends', 'micro frontend architecture', 'Migration micro frontends'. Um die Suche intuitiver zu gestalten wurden die Begriffe mit dem folgendem booleschen [17] Ausdruck ausgedrückt:

```
1  ("micro frontends" OR "micro-frontends" OR "microfrontends") AND
2   ("architecture" OR "best practices" OR "challenges" OR "migration" OR "
        benefits" OR "techniques")
```

Die Suchzeichenfolge wurde gemäß den von Brereton et al. [Brereton u. a. 2007] vorgeschlagenen Schritten erstellt.

- Wichtige Begriffe aus den Forschungsfragen und Zielen ableiten, indem man die Hauptkonzepte identifiziert
- Alternative Schreibweisen und Synonyme für wichtige Begriffe identifizieren
- Schlüsselwörter in allen relevanten Quellen überprüfen
- Synonyme und alternativen Schreibweisen mit dem booleschen Ausdruck `OR` definieren
- Wichtige Begriffe mit dem Ausdruck `AND` verknüpfen

3.3. Ein- und Ausschlusskriterien der Literatur

Quellen wurden basierend auf Titeln, Zusammenfassungen und Schlüsselwörtern sowie Volltextlesung ein- oder ausgeschlossen, wobei folgende Einschlusskriterien zur Ergebniseingrenzung angewendet wurden:

[12] https://books.google.com/advanced_book_search
[13] https://www.sciencedirect.com/
[14] https://dl.acm.org/
[15] https://www.springer.com/
[16] https://ieeexplore.ieee.org/
[17] https://southern.libguides.com/google/boolean

- ▻ Quellen wurden im Zeitraum von 2019 bis 2021 veröffentlicht.
- ▻ Studien basieren auf der Diskussion über Micro-Frontends.
- ▻ Quellen befassen sich mit Strategien und/oder Ansätzen für den Aufbau von Micro-Frontends oder Micro-Frontend-Architekturen.
- ▻ Quellen heben Vorteile und Herausforderungen von Migration zu Micro-Frontends beziehungsweise Micro-Frontends-Architekturen hervor.
- ▻ Art der relevanten Literatur wurde eingegrenzt auf
  - wissenschaftlichen Arbeiten,
  - Blog-Posts,
  - Videos,
  - verwandten Problemdiskussionen zu den Open-Communitys, wie zum Beispiel Stack Overflow oder
  - GitHub-Quellen mit Hinweise zum Quellcode, zur Dokumentationen, zu Problemhinweisen oder zu Diskussionen rund um Micro-Frontends Projekte.

Unberücksichtigt blieben insbesondere Quellen wegen folgender Ausschluss- und Abgrenzungskriterien:

- ▻ Aufgrund meiner sprachlichen Kompetenzen wurden nur Quellen auf Englisch und Deutsch verwendet.
- ▻ Automatisiert übersetzte Texte und als solche kenntlich markierte Artikel bleiben unberücksichtigt.
- ▻ Offensichtliche digitale Kopien von bestehenden Artikeln.
- ▻ Kurznachrichten mit wenig neuen Inhalt relativ zu bestehenden Artikeln.

3.4. Qualitätsprüfung der Quellen zum Auswahl

In der grauen Literatur finden sich Beiträge, die aus verschiedenen Gründen veröffentlicht wurden. Gründe waren zumeist persönliches Engagement, Idealismus, Reflektionen, Werbung für eigene Person & Meinung oder Unternehmensmarketing. Die Motivation, eine wissenschaftliche Publikation zu verfassen, konnte bei keiner Quelle aus der grauen Literatur als Hauptgrund für das Veröffentlichen unterstellt werden.

Die Materialien und Ressourcen, die einer der genannten Arten von grauer Literatur zuzuordnen sind, sind vielfältig. Große Konzerne wie Zalando[18], Otto[19] oder auch Google[20]

[18] https://engineering.zalando.com/

[19] https://www.otto.de/jobs/technology/techblog/

[20] https://developers.googleblog.com/

und Microsoft [21] führen Blogs, in welchen Personen zu finden sind, die ihre Meinungen und oft auch ihre detaillierten Erfahrungen teilen. Im Bereich der grauen Literatur gibt es keine Qualitätskontrolle des hochgeladenen Materials, und es gibt keine Institutionen oder Verfahren, die im Rahmen eines Reviews überprüfen, ob die Inhalte valide sind. Es ist auch nicht sichergestellt, dass die Beiträge unverändert bleiben oder dass zumindest Änderungen nachvollziehbar dokumentiert werden. Daher müssen interessierte Wissenschaftler die Aufgabe der Qualitätsprüfung selbst übernehmen, die Qualität der enthaltenen Daten bewerten und die Angaben zumindest kritisch auf ihre Plausibilität hin prüfen.

Zur Qualitätssicherung und Einschränkung der Quellen für diese Arbeit wurde die von Garousi u.a [Garousi u. a. 2017] vorgeschlagene Checkliste auf jede der Quellen aus der grauen Literatur angewendet. Die Checkliste umfasst 18 Punkte, die nach folgenden Kriterien geordnet sind: Autorität des Herstellers, Methodik, Meinungs-äußernder Text, Objektivität, Datum, Neuheit und Wirkung. Jeder Frage wird abhängig von ihrer Antwort ein Wert zugewiesen (ja = 1, teilweise = 0,5, nein = 0).

Zum Abschluss des Prozesses werden die erhaltenen Werte addiert und der Wert anschließend normalisiert:

$$\frac{\text{Summe der erhaltenen Werte}}{\text{Anzahl der ursprünglichen Fragen (18)}}$$

. Das normalisierte Ergebnis bestimmt die Kontroll- und Glaubwürdigkeitsstufe der Quelle. Nach Garousi u.a [Garousi u. a. 2017] gibt es drei Stufen:

- 1. Stufe (Maß = 1): hohe Ausgangskontrolle/hohe Glaubwürdigkeit.
- 2. Stufe (Maß = 0,5): moderate Ausgangskontrolle/moderate Glaubwürdigkeit.
- 3. Stufe (Maß= 0): geringe Ausgangskontrolle/geringe Glaubwürdigkeit.

[21] https://techcommunity.microsoft.com/

Kapitel 4

Ergebnisse aus der Literatur

Herausforderungen zu identifizieren, denen sich die Industrie und Softwareentwickler bei der Bearbeitung von Migrationen zu Micro-Frontends gegenübersieht, war ein Ziel der Literaturrecherche. Zusätzlich sollten mögliche Einsatzgründe sowie Vorteile von Micro-Frontends identifiziert werden. Zudem sollten Schwierigkeiten identifiziert werden, die während der Migration zu Micro-Frontends auftreten könnten, und es sollte verstanden werden, was Wichtiges bei der Zerlegung monolithischer Software beachtet werden sollte beziehungsweise berücksichtigt werden muss. Die kommenden Unterkapitel fassen die Ergebnisse der Literaturrecherche zusammen, die auf 36 ausgewählten Quellen basiert. Neben 28 Quellen aus der grauen Literatur wurden 4 Fachbücher und 4 wissenschaftliche Publikationen berücksichtigt. Die vollständige Auswertung der ausgewählten Literaturen gemäß Garousi u.a [Garousi u. a. 2017] wurde als Google Dokument[22] bereitgestellt.

4.1. Micro-Frontends Einführung in Organisationen und Unternehmen

Obwohl das Konzept relativ jung ist, werden Micro-Frontends von einer Vielzahl von Organisationen genutzt. Immer mehr Unternehmen setzen Micro-Frontends aktiv ein. In einigen Fällen wird anstelle des allgemeinen wertneutralen Begriffs 'Micro-Frontends' eine abweichende, eigene Spezialbezeichnung verwendet wie zum Beispiel: 'Frontend Microservices' oder 'Microservices UI'. Da diese Begriffe gern im Bereich der grauen Literatur genutzt werden, liegt die Vermutung nahe, dass mit solchen Spezialbenennungen die Besonderheit der eigenen Idee hervorgehoben werden soll und dass damit eine intellektuelle Abgrenzung gegen die Konkurrenz aufgebaut werden soll. Die folgende Liste enthält einige Unternehmen und Organisationen, die sich mit Techniken beschäftigen und aktiv einsetzen, die der Autor aus wissenschaftlicher Sicht unter dem Begriff 'Micro-Frontends' zusammenfassen würde: Dazn [Mezzalira 2019a], Bit [Saring 2020] , Alegro [Krzeszowiak 2021] , Spotify [Biomee 2020] , Entando [Entando 2020] , Fiverr [Friedman 2020] , Microsoft [Microsoft 2021] , Zalando [Brockmeyer 2021] , SAP [SAP 2021] , Zeiss [Maric 2020] , Open MRS [MRS 2021] , REWE [Röhrig 2019].

[22] https://tinyurl.com/literatur-review-anhang

4.2. Micro-Frontends - Einsatzgründe, Vorteile und Herausforderungen

4.2.1. Gründe für den Einsatz

Insbesondere auch bei der Grauen Literatur wurde während der Recherche auch ein Augenmerk darauf gelegt, welche Motive für den Einsatz von Micro-Frontend angeführt wurden. Die war möglich, weil viele Software Entwickler und große Konzerne wie DAZN [Mezzalira 2019a] oder Zalando [Brockmeyer 2021], die Micro-Frontends einsetzen, in ihren Publikationen auch begründen, warum sie sich für den Einsatz von Micro-Frontends entschieden haben. Für die Migration zu Micro-Frontends werden je nach Quelle manchmal organisatorische Gründe und/oder manchmal technische Gründe als Motiv für den Einsatz von Micro-Frontends angeführt.

Begründung mit Blick auf arbeitsorganisatorische-marktwirtschaftliche Vorteile

In vielen Unternehmen konzentrieren sich sogenannte Führungs-, Vertriebs- und Produktteams auf eine schnelle Markteinführung, was insbesondere bei kleinen Services mit großer Sicherheit gut zu gewährleisten ist. Die Operations- und Kundenerfolgsteams legen in der Regel Wert auf stabilen und robusten Code, mit dem nach ihrer Einschätzung auch die Kundenzufriedenheit einhergehen sollte. Die Entwickler in solchen Entwicklungsteams schätzen die Micro-Frontends, weil diese eine agile kleinschrittige Entwicklung beim gleichzeitiger großer Unabhängigkeit vom Gesamtprojekt ermöglichen. Die parallele Entwicklung von verschiedenen Micro-Frontends erlaubt eine kleinschrittige, innovative Evolution aller Einzelteile und damit auch die evolutionäre Weiterentwicklung des Gesamtprodukts.

Da Micro-Frontends mit unterschiedlichen Technologien und mit einer anderen internen Struktur entwickelt werden können, müssen die Richtungsentscheidungen, die die Gesamtapplikation betreffen, zwischen den Teams koordiniert werden, während internale technische Entscheidungen von den Teams selbst getroffen werden können [Saring 2020] [Röhrig 2019] [Maric 2020] [Fleischer und Schröder 2021] [Peltonen u. a. 2021].

Die Entwickler bei Otto [Fleischer und Schröder 2021] sind der Meinung, dass die Beteiligung einer größeren Anzahl von Entwickler-Teams oder die Entwicklung von einer großen Anzahl an Funktionalitäten stark dafür sprechen, die geplante Software in einer Micro-Frontend-Architektur zu realisieren beziehungsweise im Artikel heißt es: «[...] das sind die ersten verräterischen Anzeichen dafür, dass eine Micro-Frontend-Architektur in Betracht gezogen werden sollte [...]». Ein Frontend-Monolith wäre gemäß der Entwickler bei Otto nur dann eine passende Lösung, wenn nur eine kleine Anzahl von Entwickler-Teams beteiligt sind oder wenn wenige Funktionalitäten abzudecken/zu entwickeln sind.

Rewe [Röhrig 2019], Zeiss [Maric 2020] sehen in der Micro-Frontend-Architektur besonders den Vorteil, dass man statt eines interdisziplinären-fachübergreifenden Teams mit mehreren funktionsübergreifendes Teams arbeiten kann. Innerhalb des fachübergreifenden

Teams gibt es so weniger teaminterne Reibungsverluste und das Team selbst kann sich stärker auf die End-to-End-Bereitstellung konzentrieren.

Begründung mit Blick auf Technische-informatische Vorteile

Operations- und Kundenerfolgsteams legen tendenziell Wert auf Produktqualitäten wie Stabilität einer Software oder Ausfallsicherheit. Manchmal im Konflikt dazu legen Entwicklerteams bei ihrer Arbeit eher Wert auf Arbeitsqualitäten wie Agilität, Unabhängigkeit, Codewartbarkeit und Innovationsfähigkeit. Dies führt zu technischen Motivationen. In der Literatur werden die technischen Aspekte der Micro-Frontends oft im Vergleich zu den technischen Aufwänden diskutiert, die man bei klassischer Herangehensweise mit monolithischen Softwarestrukturen erwarten würde. In der Literatur [Jackson 2019] [Geers 2020, S. 118-144] [Mezzalira 2021, Kap. 9] [Rappl 2021, S. 149-192] wird auch darauf hingewiesen, dass zum Zusammenführen mehrerer Micro-Frontends im Browser eine einheitliche Integrationstechnik, die für jedes Micro-Frontend geeignet sein muss, die relativ klein bleiben kann.

- **Motivation 1 - Optimierung für Funktionalitäten Entwicklung**: Wenn es um Innovationen geht, kann manchmal die Geschwindigkeit einer Markteinführung den Unterschied zwischen Erfolg und Misserfolg ausmachen. Die erfolgreichsten Unternehmen sind nicht unbedingt die ersten, die die Idee haben, aber sie gehören oft zu den ersten, die eine Innovation in großem Maßstab umsetzen und implementieren. Ein Grund, warum sich Unternehmen für Micro-Frontends entscheiden, ist die Erhöhung der Entwicklungsgeschwindigkeit [ElHousieny 2021] [Mezzalira 2019b] [Peltonen u. a. 2021] [Geers 2020, S. 241-242].

- **Motivation 2 - Die Möglichkeit schaffen, Teile des Frontends inkrementeller zu aktualisieren, zu modernisieren oder neu zu schreiben**: Die Flexibilität von Micro-Frontends ermöglicht inkrementelle Upgrades und schafft einen Raum für den Einsatz der neuesten Technologien. Wenn ein Upgrade mit neuen Funktionen und mit neuer Technik erstellt werden soll, können Teams diese bei Micro-Frontends inkrementell einführen, anstatt Zeit, Geld und Mühe in eine riskante 'Big-Bang'-Veröffentlichung mit einer kompletten Überarbeitung des gesamten Monolithen zu investieren. Der Wunsch, Software mit den neusten Techniken umzusetzen und erproben zu können, motiviert oft den Einsatz von Micro-Frontends [Geers 2020, S. 33, 241] [Mezzalira 2019b] [Pölöskei und Bub 2021] [Jackson 2019] [Peltonen u. a. 2021].

- **Motivation 3 - Unabhängige Bereitstellung und Tests der Funktionalitäten:** Jede Änderung in einem Monolithen würde eine erneute Bereitstellung der gesamten Anwendung erfordern. Dies ist im Vergleich zu Micro-Frontends zeitaufwändig und der gesamte Update-Zyklus muss bei jeder Änderung wiederholt werden. Das bedeutet auch, dass ein häufiger Update-Zyklus vorgenommen werden muss je mehr Änderungen wegen der neuen Technik zu machen sind. Skalierung könnte in einem solchen Szenario für

Entwickler nur ein ferner Traum sein. Für eine vollständige Produktionsbereitstellung muss der Code vollständig getestet werden. Dies ist eine zeitaufwändige Aufgabe, die zu längeren Ausfallzeiten führt. Motivierend sind deshalb Micro-Frontends, weil sie oft flexibler sind, was die Bereitstellung neuer Techniken und die Tests unterschiedlicher Funktionalitäten angeht [Jackson 2019] [Peltonen u. a. 2021] [Mezzalira 2020] [Gilbert 2021, S. 91].

4.2.2. Zusammenfassung der Vorteile und Herausforderungen

Zusammenfassend wurden in der Literatur folgende Vorteile erwähnt:

Vorteile	**Literatur**
Große Teammotivation dank großer technischer Entscheidungsfreiheiten	[Saring 2020] [Röhrig 2019] [Maric 2020] [Fleischer und Schröder 2021] [Prajwal u. a. 2021] [Ball 2019]
Flexibilität bei der Arbeitsorganisation	[Fleischer und Schröder 2021] [Ball 2019] [Jackson 2019] [Peltonen u. a. 2021]
Weniger teaminterne Reibungsverluste dank Fach- statt Funktionsteams	[Röhrig 2019] [Maric 2020]
Schnellere und flexiblere Bereitstellung neuer Funktionalitäten	[Jackson 2019] [Yang u. a. 2019] [Peltonen u. a. 2021] [Lemon 2020] [Prajwal u. a. 2021] [Mezzalira 2021, Kap. 2]
Einfachere Fehleranalyse dank der Modularisierung	[Prajwal u. a. 2021] [Gilbert 2021, S. 79]
Maximierung des Potenzials für 'Lazy Loading' durch separate Bereitstellung der Anwendungen	[Gilbert 2021, S. 79] [Geers 2020, S. 189]

Mit der Entscheidung für Micro-Frontends sind laut Literatur folgende wichtige Herausforderungen zu erwarten:

Herausforderungen	Literatur
Konsistenz in der Benutzeroberfläche zwischen den Micro-Frontends	[Geers 2020, S. 18, 213–235] [Venkatesan 2021] [Peltonen u. a. 2021]
Gute Benutzererfahrung durch schnelle Ladezeiten sicherstellen, wenn Funktionsbereiche nachzuladen sind	[Prajwal u. a. 2021] [Geers 2020, S. S190-211] [Peltonen u. a. 2021] [Fernando 2020]
Code-Duplizierung und Code-Effizienz	[Prajwal u. a. 2021] [Lemon 2020] [Ball 2019] [Geers 2020, S. 17-18] [Peltonen u. a. 2021] [Biomee 2020]
Planung der Schnittstellen für die (potentielle) Kommunikation zwischen Micro-Frontends	[Lando 2019] [Prajwal u. a. 2021]
Datenabhängigkeiten zwischen den Micro-Frontends und deren Dokumentation	[Geers 2020, S. 236-250] [Lando 2019]
Entwicklung eines möglich allgemeingültigen und gleichzeitig robust-schlanken Gerüsts für das Frontend	[Jackson 2019] [Geers 2020, S. 118-144] [Mezzalira 2021, Kap. 9] [Rappl 2021, S. 149-192]

4.3. Migrationsansatz

Für die Architektur einer Micro-Frontend-Anwendung gibt es verschiedene Varianten. Um die beste Variante für ein Projekt zu wählen, muss der Kontext, in dem man tätig ist, verstanden werden. Eine Variante zu wählen, bedeutet Architekturziele zu kennen, diese zu priorisieren und mit den verfügbaren Optionen zu vergleichen. Für die Planung einer neuen Architektur im Allgemeinen beziehungsweise für die Planung einer Migration in Micro-Frontend-Architektur im Speziellen sollten zuvor unter anderem die folgenden Fragen geklärt werden:

- Was sind die Rahmenbedingungen der Software? (technische Rahmenbedingungen, benötigte Flexibilität, Ausgabegeräte)
- Welche Funktionen oder Dienste sollen bereitgestellt werden?
- Welche Vor- und Nachteile haben alternative Architekturen im Vergleich zu den Micro-Frontends?
- Welche Testszenarien müssen zur Validierung einer erfolgreichen Migration erfüllt werden?

Die Migration eines Projekts von einer Architektur in eine andere ist oft eine schwierige und meist auch kostspielige Aufgabe. Dabei können verschiedene Wege beschritten werden, die alle jeweils Vor- und Nachteile haben.

Geers [Geers 2020, S. 254] nennt in seinem Buch basierend auf seinen Erfahrungen drei Strategien zur Migration von der monolithischen Software (Monolithen) in eine Micro-Frontends-Architektur.

- **'Stück-für-Stück' oder 'Backend-zuerst' Strategie**: Bei dieser Methode wird ein System gemäß der Funktionalität des Backends stückweise in einem Microservice migriert, die jeweils die extrahierte Funktionalität erledigen. Nach der Migration einer Funktionalität ersetzt das Entwickler-Team dann auch die zugehörige Benutzeroberfläche des Monolithen durch die Benutzeroberfläche des Micro-Frontends und vervollständigt damit den Microservice.
- **'Frontend-zuerst' Strategie**: Diese Strategie setzt voraus, dass der Monolith sowohl das Frontend als auch das Backend beinhaltet. In diesem Fall wird das Frontend des Monolithen durch neue Micro-Frontend-Anwendungen ersetzt. Die Frontends kommunizieren in der Übergangsphase weiter über die Schnittstellen mit dem alten Monolithen. In der zweiten Phase wird dann das Backend mit einem 'Stück-für-Stück'-Ansatz migriert.
- **'Greenfield und big bang' Strategie**: In diesem Ansatz werden alle Micro-Frontends separat entwickelt, während der aktuelle Monolith weiterhin im Einsatz bleibt. Sobald die Implementierung aller Funktionen beziehungsweise Micro-Frontends abgeschlossen ist, wird der eingehende Datenverkehr auf das neue System weitergeleitet und das alte System wird abgeschaltet.

Mezzalira [Mezzalira 2021, Kap. 3] hat, mit dem Ziel sich auf das Wesentliche zu konzentrieren, vier Kernthemen herausgearbeitet, die man im Vorfeld der Planung abarbeiten muss, bevor mit dem Bau der Architektur eines Micro-Frontend-Projekts begonnen wird oder bevor eine bestehende Webseitensoftware zu einer Micro-Frontend-Architektur umgebaut wird. Seine Methodik nennt er 'Micro-Frontends descisions framework' und seine vier Säulen lassen sich wie folgt skizzieren:

- **Definition der Micro-Frontends**: In dieser Säule definiert man, welche Aufgaben und Ziele die einzelnen Micro-Frontends im Gesamtkontext erfüllen sollen, wobei eine größtmögliche Zustandsautonomie der Micro-Frontends angestrebt werden sollte. Zustandsautonomie meint, dass die Geschäftslogik eines Micro-Frontends unabhängig vom Zustand/Datenbestand anderer Micro-Frontends funktionieren kann.
- **Zusammenstellung**: In dieser Säule definiert man, wie einzelne Dienste zusammengeführt werden.
- **Routing**: In diesem Kernbereich definiert man, wie man von einem Dienst zu einem anderen navigieren kann.
- **Kommunikation**: In diesem Kernaspekt definiert man, wie einzelne Dienste Daten austauschen können [Mezzalira 2021, Kap. 3].

Die Punkte Zusammenstellung, Routing und Kommunikation werden häufig [Geers 2020, S. 42] [Rappl 2021, Kap. 10] [Gilbert 2021, S. 90] in Quellen erwähnt, da diese Kernpunkte für den Entwurf von Micro-Frontends wichtig sind.

Wie die nachfolgenden Unterkapitel zeigen werden, werden in der Literatur je nach gewähltem Migrationskonzept noch weitere Aspekte und Heuristiken genannt, die bei einer Migration zu Micro-Frontends zu betrachten sind.

4.3.1. Analyse von Domäne und Kontext

Jede Software soll bestimmte Aufgaben lösen oder vereinfachen. Dieser Ansatz, oft mit dem Schlagwort Domain-Driven-Design (DDD) verknüpft, beginnt mit der Frage, in welchem Kontext beziehungsweise in welcher Domäne eine Webanwendung ihren Dienst verrichten soll, und versucht auf inhaltlicher Ebene die verschiedenen, möglichst voneinander unabhängigen Funktionalitäten herauszuarbeiten. Dieses Wissen wird dann als Grundlage genommen, um die verschiedenen Micro-Frontends zu entwerfen.

Ziel

Das Ziel ist eine gemeinsame Sprache bei Programmierer und Kunden so zu entwickeln, dass beide gemeinsam verstehen, was das aktuelle System macht und wie es funktionieren soll. Der Kontext meint das Verständnis des Geschäfts, meint die Umgebung, in der man tätig ist, und meint auch das Ergebnis, das angestrebt wird, welches mit dem Kontext verknüpft ist. Die funktionalen Anforderungen der Anwendung stehen dabei im Zentrum der Analyse. Ein Micro-Frontend sollte Domänenwissen in sich so kapseln, dass es unabhängig von anderen Micro-Frontends seine Geschäftslogik ausgeführt werden kann.

Methodik

Domain-Driven-Design(DDD) ist ein häufig genutzter Ansatz, um mit Kunden/Anwender zusammen die Software und ihrer Funktionen zu konzipieren und zu strukturieren. DDD geht von der Annahme aus, dass Softwarearchitekturen basierend auf Domänen und Unterdomänen entworfen werden sollten [Evans 2014]. Eine Domäne bezieht sich auf den Raum des Problems, in dem gehandelt wird, in dem bestimmte Entitäten vorkommen und in dem definierbare Regeln gelten. Eine Domäne kann in Unterdomäne(Subdomain) aufgeteilt werden. Domain-Driven-Design kann gut beim Entwurf von Microservices verwendet werden. Aber auch für den Entwurf von Micro-Frontends lässt sich die Strategieform verwenden, weil man damit beim Entwurf von Micro-Frontends klaren Grenzen ziehen kann. Begonnen wird mit der Analyse der Geschäftsdomäne, um die funktionalen Anforderungen der Anwendung zu verstehen. Das Ergebnis dieses Schrittes ist eine informelle Beschreibung der Domäne [Rappl 2021, S. 46-55] [Friedman 2020]. DDD verwendet das Wort 'Module' als Alias für Pakete, die quasi als Container einen bestimmten Satz von Funktionalitäten

in der Anwendung definieren. Die Benennung passt auch gut zum Wording in der vorliegenden Arbeit, in welcher mit 'Module' ein Micro-Frontend inklusive seines Microservice bezeichnet wird.

4.3.2. Analyse durch Abhängigkeiten identifizieren

Im Allgemeinen sollte sich die Funktionalität in einem Micro-Frontend auf einen begrenzten Kontext beschränken. Ein begrenzter Kontext markiert die Grenze eines bestimmten Domänenmodells. Wenn festgestellt wird, dass ein Modul (Micro-Frontend) mit anderen Modulen wechselwirkt und die Art der Ergebnisse vom Datenbestand anderer Module abhängig ist, so ist dies ein Zeichen dafür, dass man möglicherweise die Domänen- und Kontextanalyse erneut durchführen muss und die modulare Architektur weiter verfeinern muss.

Ziel

Mit der Analyse will man die Grenzen eines Micro-Frontends identifizieren.

Methodik

Domain-Driven-Design kennt als Methodik das Konzept von 'begrenzter Kontext' [Evans 2014, S. 335]. Ein begrenzter Kontext wird verwendet, um die Grenzen der Funktionalität einer Unterdomäne zu definieren. Es meint also einen Bereich, in dem nur die Funktionalität einer bestimmten Domäne sinnvoll ist. Mit begrenzten Kontexten lassen sich einzelne Micro-Frontends leichter eingrenzen und gegebenenfalls nach Subdomänen gruppieren [Pölöskei und Bub 2021] [Rappl 2021, S. 47]. Geers [Geers 2020, S. 239] empfiehlt zusätzlich zur DDD auch eine Analyse der bestehenden Seitenstruktur. Die Seitenstrukturanalyse soll besonders hilfreich sein, wenn eine Software bereits produktiv eingesetzt wird. Aus unterschiedlichen Seiten lassen sich damit Gruppen oder einzelne Funktionalitäten leicht identifizieren, die später in der neuen Architektur als Micro-Frontends wirken könnten. Mezzalira [Mezzalira 2021, Kap. 9] ist der Meinung, dass die Analyse der Nutzerinteraktionen und Datenflüssen ebenfalls, Informationen über denkbare Grenzen und Abhängigkeiten zwischen Micro-Frontends liefern können.

4.3.3. Entwurf der Micro-Frontends aus dem Monolithen

Auch wenn Zugriff auf den vorhandenen Code im Monolithen möglich ist, ist nicht immer klar, was mit dem vorhandenen Code geschehen soll, wenn mit der Migration von Funktionen zu Ihren neuen Micro-Frontends begonnen wird. Sollte der Code unverändert verschoben oder die Funktionalität neu implementiert werden? Jedes Mal, wenn ein Micro-Frontend extrahiert und in einen Dienst umgewandelt wird, schrumpft die monolithische Anwendung. Sobald genügend Module konvertiert wurden, verschwindet der Monolith

entweder ganz oder wird klein genug, um ein weiterer Dienst zu werden. Bei der Entwurf von Micro-Frontends sollte optimalerweise einige Punkte betrachtet werden. Welche Funktionalität kann (bereits) als Dienst extrahiert werden? Können Abhängigkeiten dieses Dienstes unkompliziert aufgelöst werden?

Ziel

Ziel ist es, herauszufinden, wie ein Micro-Frontend aus technischer Sicht sinnvollerweise betrachtet werden sollte.

Methodik

Mehrere Micro-Frontends können in derselben Ansicht co-existieren oder es gibt nur ein Micro-Frontend pro Ansicht. Mezzalira[Mezzalira 2021, Kap. 3] schlägt im ersten Teil der vorgeschlagenen 'Micro-Frontends descision Frameworks' zwei Hauptstrategien vor: Die Horizontale und die Vertikale Aufteilung. Rappl [Rappl 2021, S. 210-211] und Geers [Geers 2020, S. 9-15] stellen diese beiden Strategien als mögliche Ansätze zur Aufteilung von Micro-Frontends im Hinblick auf die Organisationsstruktur der Entwickler Teams.

Bei der horizontalen Aufteilung befinden sich mehrere Micro-Frontends in derselben Ansicht. Dieser Ansatz bietet eine große Flexibilität, denn es können einige Micro-Frontends in verschiedenen Ansichten wiederverwendet werden.

Die vertikale Aufteilung unterteilt die Anwendung in vertikalen Schichten. Jeder Schicht wird von der Datenbank bis zur Benutzeroberfläche erstellt. Dieser Ansatz bezieht sich auf die Microservices-Architektur. Der Hauptunterschied besteht jedoch darin, dass ein Dienst auch seine Benutzeroberfläche enthält. Diese Erweiterung des Dienstes macht ein zentrales Frontend-Team überflüssig und nähert den 'Self-Contained Systems'[23] Ansatz an. Ein 'Self-Contained System' besteht aus einem Backend Microservice, gegebenenfalls einer Datenbank und einem Frontend Microservice. Ein Vorteil eines vertikalen Ansatzes besteht darin, dass eine Problemdomäne wirklich nach Belieben in kleinere Teile aufgeteilt werden kann, sodass man sich bei der Entwicklung auf eine einzelne Unterdomäne beziehungsweise auf ein einzelnes Micro-Frontend beschränken kann.

4.3.4. Zusammenstellung unterschiedlicher Dienste

Das endgültige Erscheinungsbild der integrierten Micro-Frontends sollte nicht anders aussehen als bei einer Website, die man mit einem Frontend-Monolithen erstellen würde. Mit anderen Worten, wenn auf den Benutzer die Gesamtwebseite einheitlich wirkt und wenn der Benutzer Grenzen zwischen Micro-Frontends nicht erkennt, dann ist dies ein Indiz dafür, dass die Komposition der Micro-Frontends und deren Integration erfolgreich war.

[23] https://scs-architecture.org/

Ziel

Das endgültige Erscheinungsbild der integrierten Micro-Frontend-Webseite sollte nicht anders aussehen als eine Website, die ein Frontend-Monolith erstellen würde.

Methodik

Drei Optionen bieten sich gemäß Rappl [Rappl 2021, Kap. 7-9], Mezzalira [Mezzalira 2021, Kap. 3] und Geers [Geers 2020, Kap. 3-5] im Browser für die Zusammenführung unterschiedlicher Micro-Frontends an.

- **Clientseitig**: Bei dieser Technik werden verschiedene Micro-Frontends vom Browser zu Beginn oder während der Nutzungszeit direkt geladen.
- **Serverseitig**: In diesem Fall erstellt der Ursprungsserver die Ansicht, indem er alle verschiedenen Micro-Frontends abruft und die endgültige Seite zusammenstellt. Einer der größten Vorteile der serverseitigen Integration besteht darin, dass die Seite bereits fertig aufgebaut ist, wenn sie den Browser des Nutzers erreicht. Dadurch wird die initiale Ladezeit der Seite reduziert. Bei dieser Technik ist zu beachten, dass das Debuggen einer zusammengesetzten Website schwierig und unübersichtlich werden kann, weil gespeist aus verschiedenen Microservices die Seite auf dem Server zusammengesetzt wird. Einen geeigneten lokalen Entwicklungsfluss für dieses Muster zu erhalten, kann kompliziert sein [Rappl 2021, S. 107].
- **Edge-Side**: Bei diesem Ansatz wird die endgültige Ansicht bestehend aus verschiedenen Micro-Frontends auf der Content-Delivery-Netzwerk-Ebene zusammengestellt. Ein Content-Delivery-Netzwerk (CDN), auch 'Content Distribution Network' genannt, ist eine Gruppe von geografisch verteilten und miteinander verbundenen Servern. Sie stellen zwischengespeicherte Internetinhalte von einem Netzwerkstandort bereit, der einem Benutzer am nächsten ist, um die Bereitstellung zu beschleunigen. Das Hauptziel eines CDN besteht darin, die Webleistung zu verbessern, indem die Zeit verkürzt wird, die zum Senden von Inhalten an Benutzer erforderlich ist. Die Edge-Side Zusammenstellung nutzt Standard Paradigmen wie 'Server Side Inlcudes (SSI)' und 'Edge Side Includes(SSI)'. 'Edge Side Includes' meint eine Technologie, die es Inhaltsanbietern ermöglicht, ihre Seiten in Fragmente mit individuellen Caching-Eigenschaften zu unterteilen. 'Server Side Includes' sind Direktiven, die in HTML-Seiten platziert und auf dem Server ausgewertet werden, während die Seiten bereitgestellt werden. Durch die Edge-Side Zusammenstellung und seine Caching Strategien kann dies Verfahren schneller sein als die Serverseitige Zusammenstellung [Geers 2020, S. 73].

4.3.5. Routing zwischen einzelner Dienste

Routing meint hier den gesamten Prozess, mit dem der Benutzer zu verschiedenen Seiten einer Website navigiert wird. Wenn die Geschäftsfunktionalität in mehrere kleinere

logische Codeteile unterteilt wird, muss darüber nachgedacht werden, wie die von den einzelnen Diensten zurückgegebenen Daten zusammenwirken und wie mit impliziten Abhängigkeiten umzugehen ist.

Ziel

Der Benutzer möchte keine Probleme bei der Verwendung einer Anwendung haben. Er kümmert sich nicht um die gewählte Architektur und darum, welche technische Lösung darunter verwendet wird. Benutzer sind an Webseiten gewöhnt und kennen das Wechseln zwischen verschiedenen Teilen einer Anwendung. Das Wechseln zwischen unterschiedlichen Micro-Frontends ist so zu gestalten, dass es das Zusammengehörigkeitsgefühl nicht stört.

Methodik

Viele JavaScript Frameworks wie zum Beispiel Angular mit `@angular/router` oder Ember.js mit `@ember/routing` bringen Routing Lösungen mit. Dieses eröffnet die Möglichkeit zwischen den Seiten zu navigieren. Ohne einen Seitenreload werden nach jeden Klick nur die betroffenen Teile ausgetauscht, die angefragt oder geändert wurden.

Beim Routing wird zwischen Soft-Links und Hard-Links unterschieden. Bei Soft-Links wird die Transition zwischen zwei Seiten oder zwischen Fragmenten einer Webseite ohne einen kompletten Seitenreload erfolgen, während bei hart-verlinkten Navigationsvorgängen die Ressourcen neu vom Server geladen und gerendert werden. Für eine monolithische Webseite, bei der die Seiten serverseitig gerendert sind, kommen Hard-Links zum Einsatz. Bei Single-Page-Anwendungen ist Soft-Navigation ein Standard.

Bei Micro-Frontends wird eine weitere Abstraktionsebene gebraucht, wenn Soft-Links zwischen einzelne Micro-Frontends gewünscht werden. Diese Ebene nennt man meist 'Applikation Shell', 'Anwendungsshell' oder einfach 'App Shell'[Geers 2020, S. 120]. Dieses Micro-Frontend fungiert für andere Micro-Frontends als erster Einstiegspunkt in die Plattform. Da es sich bei dieser Shell Anwendung um einen gemeinsam genutzten Code handelt, sollte diese so einfach wie möglich gehalten werden. Es sollte keine Geschäftslogik enthalten. Manchmal sind auch Themen, die alle Micro-Frontends betreffen, wie die Authentifizierung in die 'App Shell' integriert.

Die 'App Shell' ist für die Darstellung der ersten Seite für unseren Benutzer und auch für das Aufrufen der jeweiligen Micro-Frontends verantwortlich, während der Benutzer durch unsere Plattform navigiert. Sie agiert als zentraler Punkt für die Navigation, indem sie Ereignisse von Micro-Frontends bekommt und indem sie basierend auf den jeweiligen Ereignissen die nötigsten Navigationsaufgaben übernimmt. Die Shell hat nur zu entscheiden, ob ein Reload des Browsers nötig ist oder ob Soft-Navigation durchzuführen ist. Diese Strategie setzt voraus, dass die 'App Shell' alle URLs der Anwendung kennen muss. Wenn ein Micro-Frontend eine vorhandene URL ändert oder eine neue URL hinzufügen möchte, muss auch die 'App Shell' angepasst und erneut bereitgestellt werden, was eine zusätzliche

Kopplung darstellt. Im optimalen Fall sollte zwischen Micro-Frontends und 'App Shell' eine, keine oder nur eine sehr geringe Kopplung existieren. Um diese zusätzliche Kopplung auszulösen, könnte man die 'App Shell' so gestalten, dass sie nur Weiterleitungen zwischen Micro-Frontends abarbeitet. Jedes Micro-Frontends muss in diesem Fall seinen eigenen Router haben, der die eingehenden URLs jeweils bestimmten Seiten zuordnet. Diese Konvention bringt den Vorteil, dass die Routing-Regeln innerhalb der 'App Shell' und damit die mögliche Zahl der 'App Shell'-Updates klein gehalten wird.

Mezallira [Mezzalira 2021, Kap. 4] fasst die Zuständigkeit der 'Applikation Shell' in folgenden Punkten zusammen:

- Behandlung von Fehlern, wenn ein Micro-Frontend nicht geladen werden kann
- Abrufen globaler Einstellungen
- Abrufen der verfügbaren Routen und der zugehörigen Micro-Frontends zum Laden
- Umgang mit dem anfänglichen Benutzer und mit initialen Daten
- Festlegen von Monitoring-, Logging- oder Marketingbibliotheken, da diese oft Anwendungsübergreifend verwendet werden.

4.3.6. Kommunikation zwischen der Dienste

In einer monolithischen Anwendung, die in einem einzelnen Prozess ausgeführt wird, rufen Komponenten einander mit Methoden- oder Funktionsaufrufen innerhalb der Code-Basis auf. Im Gegensatz dazu wird für interaktive Anwendungen mit Micro-Frontends ein Modell für die Kommunikation benötigt. Obwohl die Kommunikation zwischen den Anwendungen durch die Aufteilung der Dienste so weit wie möglich reduziert wird, bleibt trotzdem fast immer ein Rest an Applikation-übergreifender Kommunikation erforderlich. Die Kommunikation zwischen Micro-Frontends sollte deshalb (parameter-)minimal und einfach gestaltet werden, und globale Zustands- und Framework-spezifische Lösungen so weit wie möglich zu vermeiden. Weitergehend sollte so oft wie möglich native Browser Funktionalitäten genutzt werden [Geers 2021].

Ziel

Ziel ist es, nur den für die Integration notwendigen Mindestdatensatz auszutauschen. Wenn sich zwei oder mehr Micro-Frontends viele Nachrichten teilen, um ihre minimale Funktionalität bereitzustellen, sind sie möglicherweise zu eng gekoppelt. Der sehr wichtiger Aspekt bei der Implementierung eines Kommunikationsmusters in Micro-Frontends ist die geringe Kopplung.

Methodik

Bei der Kommunikation zwischen Micro-Frontends weist Mezzalira [Mezzalira 2021, Kap. 3]) darauf hin, dass verschiedene (Micro-)Frontends eigenständige Einheiten sind und komplett voneinander entkoppelt werden sollten. Bei mehreren Micro-Frontends auf derselben Seite empfiehlt er die Verwendung benutzerdefinierter Ereignisse(engl.: *Custom Events*) oder einer Ereignissender-Bibliothek (engl.: *Event Emitter*), wenn Benachrichtigungen über Benutzerinteraktionen oder Ereignisse auf der Seite ausgetauscht werden sollen. Die Interaktion von JavaScript mit HTML wird durch Ereignisse behandelt, die auftreten, wenn der Nutzer relevante Eingabeaktionen macht oder wenn der Browser eine Seite manipuliert. Wenn beispielsweise die Seite geladen wird, wird sie als Ereignis bezeichnet. Wenn der Nutzer auf eine Schaltfläche klickt, ist dieser Klick ebenfalls ein Ereignis.

Der Browser bietet eine Schnittstelle für sogenannte *Custom Events*, welche die Möglichkeit bietet, das Standard-Event-Listener-System mit den eigenen Namen und Daten zu erweitern. Ergänzend dazu empfiehlt Geers [Geers 2020, S. 46-47] Namensräume zu verwenden. Namensräume sind aus einzelnen Teil-Namen syntaktisch eindeutig zusammengesetzt, wobei die Teil-Namen oft Informationen über strukturelle Ähnlichkeiten oder ähnliche Ursprünge transportieren. Sie ermöglichen es, Objekte mit ähnlichen Namen, aber unterschiedlichen Ursprüngen zu unterscheiden. Ein Namensraum hilft bei der Vermeidung von Konflikten. Wertvoll für die tägliche Arbeit ist, dass sie auch oft kennzeichnen können, wem sie zuzuordnen sind. Eine benannte Browsernachricht könnte im Code Namensraum-codiert zum Beispiel wie folgt aussehen:

```
1 new CustomEvent("landscapes:token", {
2   detail: {
3     token: "xxxxxxxx",
4   },
5 });
```

Mit *landscapes:token* ist der als Namensraum kodierte Ausdruck für die Nachricht, wobei der Teil-Name *landscapes* die Herkunft und der Teil-Name *token* die Bezeichnung der Nachricht repräsentiert. Der durch einen Namensraum kodierte Ausdruck erlaubt gleichzeitig die strukturierte systematische Weiterverarbeitung der Nachricht.

Für den Datenaustausch zwischen Seiten (oder auch Micro-Frontends) stehen technisch verschiedene Optionen zur Verfügung. Es kann eine Webspeicherlösung, die auf lokalem oder Sitzungsspeicher (engl.: *local- und session storage*) basiert, verwendet werden, wo beim Laden ein Micro-Frontend die benötigten Informationen findet[Mezzalira 2021, Kap. 4]. Eine andere Möglichkeit besteht darin, dass das sich installierende Micro-Frontend eine Anfrage an das Backend stellt, um die benötigten Informationen abzurufen. In einigen Fällen könnte auch die 'Applikation Shell' als Schnittstelle für das Weiterleiten von Ereignissen und Informationen zwischen zwei oder mehrere Micro-Frontends genutzt werden.

4.3.7. Fehlerbehandlung und Fehlerbehebung

Beim Konzipieren einer Webanwendung reicht es nicht aus, sich nur auf einen einzigen goldenen Weg der Nutzerinteraktion zu beschränken und nur für diesen 'Weg' die Webanwendung zu entwickeln. Oft muss man mit Respekt auf die Verschiedenheit der Menschen und ihrer Gedanken Variation von Eingaben und ihre Reihenfolge zulassen. Der Hinweis auf Fehler und ihre Heilung ist ein wichtiger Teil der Benutzererfahrung bei vielen Anwendungen. Bei guter Ausführung führt ein durchdachtes Fehlerhandling dazu, dass sich die Benutzer informieren und richtig zum gewünschten Ziel geführt fühlen.

Micro-Frontends werden als kleinerer Module aus verschiedenen Quellen geladen. Jede Anfrage hat eine gewisse Chance, fehlzuschlagen. Neben den Fehlern der Micro-Frontends ist daher die Wahrscheinlichkeit relativ hoch, dass einige Module nicht geladen werden und deswegen ein Fehler auftritt oder ein Prozess auf halben Wege stecken bleibt.

Ziel

Jedes Micro-Frontend ist für den Umgang mit seinen eigenen Fehlern verantwortlich. Ziel ist es, Fehler zu verhindern, die andere integrierte Micro-Frontends beeinträchtigen. Wann immer möglich, möchte man verhindern, dass Ausfälle in einem Teil der Webseite auf andere Teile übergehen. Wenn Micro-Frontends das Erstellen einer Benutzeroberfläche zur Laufzeit erfordern, dann sollte der Umgang mit möglichen 404 Fehlern[24] für eine oder mehrere Dateien oder Micro-Frontends gut geregelt sein [Geers 2020, S. 180-182]. Zudem ist ein gutes Fehlermanagement anzustreben, um bei Informationsaustausch zwischen Micro-Frontends zu verhindern, dass Fehler in diesen Nachrichten zu einem Ausfall der Anwendung führt.

Methodik

Wenn als denkbarer Fehlerfall ein Micro-Frontend abstürzt, sollte die Hauptanwendung einschließlich aller anderen Module weiterhin funktionieren. Um die Benutzererfahrung nicht zu beeinträchtigen, wird empfohlen, in solchen Fällen auf einem alternativen Inhalt zurückzugreifen oder einen bestimmten Teil der Anwendung einfach auszublenden [Mezzalira 2021, Kap. 2]. Je nach gewählte Kommunikationstechnik sollte deshalb beim Empfangen einer Nachricht beziehungsweise von Daten aus anderen Micro-Frontends auch ein Errorhandling eingebaut werden.

4.3.8. Automatisierte Tests

Automatisiertes Testen ist zum Herzstück moderner Softwareentwicklung geworden. Eine gute Testabdeckung reduziert den Bedarf an manuellen Tests. Tests sind ein systematischer Versuch in einem Softwaresystem Defekte zu finden. Mit Blick auf diese Trends sollte

[24] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

jedes Micro-Frontend über eine eigene umfassende automatisierter Tests verfügen, die die Qualität und Korrektheit des Codes und seiner internen Logik sicherstellen [Geers 2020, S. 262].

Micro-Frontends basieren auf dem Konzept entkoppelter Anwendungen. Diese Forderung kann jedoch nicht immer erfüllt werden. Manchmal ist eine Kommunikation zwischen Anwendungen erforderlich. Beispielsweise müsste ein Micro-Frontend für die Navigation eine Benachrichtigung senden, wenn ein anderes Micro-Frontend vom Benutzer in dem Navigationsmenü ausgewählt wurde. Jede Kopplung zwischen Micro-Frontends ist eine Integration, die vor der Bereitstellung in der Produktion getestet werden muss, um dem Nutzer später ein zusammenhängendes Produkterlebnis bieten zu können.

Basierend auf der von Mike Cohn [Cohn 2009] eingeführten 'Testpyramide' können Softwaresysteme auf unterschiedlichen Ebenen getestet werden. Die 'Testpyramide' ist eine Metapher, die hilft, Softwaretests zu gruppieren und den mittleren Zeitbedarf für Tests abzuschätzen. Zur Abdeckung verschiedener Funktionalitäten basierend auf die vorgeschlagene Pyramide bieten sich automatisierte Unit-Tests, Integrationstests und End-to-End-Tests an, die dann auch leicht zu einem Arbeitsschritt in einem Continous-Delivery-Prozess werden können.

Unit-Tests bilden die Basis der Pyramide und sind nahe dem Quellcode der Anwendung. Sie bestehen darin, einzelne Methoden und Funktionen der von der Software verwendeten Klassen, Komponenten oder Module zu testen. Unit-Tests sind im Allgemeinen recht günstig zu automatisieren und können sehr schnell von einem Continuous Integration Server ausgeführt werden [El-Morabea und El-Garem 2021].

Integrationstests liegen in der Mitte der Pyramide und überprüfen, ob verschiedene Module oder Dienste, die von einer Anwendung verwendet werden, gut zusammenarbeiten. Die Ausführung Integrationstests ist im Vergleich zu Unit-Tests meist Zeit- und Ressourcenaufwendiger , da hierfür oft mehrere Module einer Anwendung geladen und ausgeführt werden müssen [El-Morabea und El-Garem 2021].

End-to-End-Tests repräsentieren die Spitze der Pyramide und bilden nach dem Pars-pro-toto-Prinzip[25] ein ausgewähltes Benutzerverhalten mit der Software in einer vollständigen Anwendungsumgebung nach. Mit ihnen überprüft man, ob verschiedene Benutzerabläufe wie erwartet funktionieren. Die Tests können einfach sein, wie das Laden einer Webseite, oder auch sehr komplexe, wie die Nachstellung eines kompletten Kaufprozesses mit Produktauswahl und Bezahlprozess [El-Morabea und El-Garem 2021].

Ziel

Ziel der automatisierten Tests ist es, die Systemrobustheit zu prüfen und sicherzustellen, dass die Datenintegrität zwischen verschiedenen Systemkomponenten und Systemen immer gewahrt bleibt sowie dass mögliche Fehler korrekt erkannt und abgefangen werden.

[25] https://www.lexico.com/definition/pars_pro_toto

Methodik

Jedes Micro-Frontends wird für sich auf verschiedenen Ebenen (Unit-Tests, Integrationstests, Micro-Frontend-internen End-to-End-Tests) getestet. Zusätzlich müssen aber auch die Nutzerinteraktionen über die Micro-Frontends hinweg mit einigen echten End-to-End-Tests abgedeckt werden. Wenn man im Bild der Test-Pyramide bleibt, so ist bei Micro-Frontend-Architekturen oberhalb der End-To-End-Tests noch eine Ebene für die Tests der Gesamtanwendung zu definieren, in welchen die Interaktion zwischen den Micro-Frontends automatisiert getestet wird.

4.3.9. Iterative Implementierung der Micro-Frontends

Migrationen von Software erfolgen häufig in mehreren Schritten. Der agile Ansatz bietet sich deshalb als passendes Vorgehensmodell für die Migration eines Monolithen in eine Micro-Frontends-Architektur an. Der agile Ansatz beinhaltet die Philosophie der iterativen und inkrementellen Softwareentwicklung, die auf einer Zunahme von Funktionserweiterungen und einem zyklischen Release- und Upgrade-Muster basiert. Das Ergebnis der nachfolgenden Iteration ist ein verbessertes Arbeitsinkrement des Produkts. Dies wird wiederholt, bis das Produkt die erforderlichen Funktionen erfüllt.

Das iterative Modell ist eine Methode der agilen Softwareentwicklung und eine bestimmte Implementierung eines Softwareentwicklungslebenszyklus, der sich auf eine anfängliche, vereinfachte Implementierung konzentriert, die dann schrittweise an Komplexität und einem breiteren Funktionsumfang gewinnt, bis das endgültige System vollständig ist. Geers [Geers 2020, S. 242-244] ist der Meinung, dass agile Werte und Praktiken für Teams, die Micro-Frontends-Architekturen entwickeln, hilfreich sein können, denn sie fördern Wissensvermittlung zwischen Teams, weil diese sich bei den regelmäßigen Treffen die Gelegenheit haben, sich auch über Erfahrungen und über Erkenntnisse aus dem Entwicklungsprozess auszutauschen.

Ziel

Das agile Modell bietet die verschiedene Anlässe, wesentliche Konstruktions- oder Planungsfehler im Prozessmodell zu identifizieren und zu beheben. Wichtige wiederkehrende Punkte sind während der Entwicklung bei dem Entwurf und bei der Implementierung zukünftiger Micro-Frontends zu identifizieren und zu dokumentieren.

Methodik

Checklisten sind sehr nützlich in dem täglichen Leben, und sie werden auch gern in der Softwareentwicklung als Kontrollhilfe genutzt. Eine Checkliste könnte beispielsweise unterschiedliche Punkte zu Testfällen und/oder Konfigurationseinheiten beinhalten. Sie ist besonders leistungsstark bei sich wiederholenden Aufgaben, die erledigt und geprüft werden müssen.

4.4. Ausgewählte Ansätze und Methoden für Micro-Frontends-Architekturen

In folgenden werden ausgewählte Techniken, Strukturen und Strategien vorgestellt, die den Aufbau von Micro-Frontends oder die Transformation zu einer Micro-Frontend-Architektur ermöglichen. Bei manchen Ansätzen können unterschiedliche Teile manchmal sogar aus mehreren Quellen kommend zur Build- oder Laufzeit, auf der Client- oder Serverseite [Jackson 2019] [Rappl 2021, Kap. 7-9] [Mezzalira 2021, Kap. 3] [Geers 2020, Kap. 3-5] zusammengebaut werden, wobei diese in einigen Projekten auch programmgesteuert miteinander verknüpft oder verbunden wurden.

Das Themengebiet Suchmaschinenoptimierung (SEO) wurde angesichts seiner Komplexität ausgeblendet. Bei Micro-Frontends können sich hinter ein und demselben Link viele verschiedene Seitenvariationen verbergen, was es insbesondere für Suchmaschinen schwer macht, diese verschiedenen Variationen vernünftig und einfach zu verlinken. Die Hinweise in der Literatur [Fink und Flatow 2014] lassen vermuten, dass eine Single-Page-Anwendung mit Micro-Frontends für auf eine gute Suchmaschinen-Indexierung angewiesene Präsentationswebseite eine eher schlechte Lösung darstellt, weil sich meist die vielen Varianten der Single-Page nur schwer indexieren und verlinken lassen.

4.4.1. 'Versionierte-Pakete' durch Integration während des Builds

Bei diesem Ansatz werden zur Build-Zeit die separaten Micro-Frontends mit einer Datei namens `package.json` gebündelt und als Information an die Container-App gesendet. Die Datei `package.json` ist das Herzstück jedes Frontend-Projekts, die für die Abhängigkeit die Syntax wie NodeJS[26] und für den Build-Prozess den Node Package Manager NPM[27] verwendet. Die Datei enthält wichtige Metadaten zu einem Projekt, die für die Veröffentlichung erforderlich sind. Diese definiert auch die funktionalen Attribute eines Projekts, welches NPM verwendet, um Abhängigkeiten zu installieren, um Skripte auszuführen und um den Einstiegspunkt zum jeweiligen Paket zu identifizieren.

Dank der Datei `package.json` können doppelte gemeinsame Abhängigkeiten zwischen den Micro-Frontends leicht erkannt und entfernt werden. Nachteilig ist jedoch, dass die Container-Anwendung immer neu erstellt werden muss, wenn ein neues Update für eines der Micro-Frontends veröffentlicht wird.

Geers spricht in seinem Buch [Geers 2020, S. 224-226] von 'Versionierten-Pakete', weil es unterschiedliche Versionen von solchen Paketen geben könnte. Die Möglichkeit, verschiedene Versionen nebeneinander zu verwenden, ist wesentlich für die Anforderung der unabhängigen Bereitstellung. Dank der 'Versionierten-Pakete' behalten die Entwickler der verschiedenen Micro-Frontends die Kontrolle über ihre Upgrades und auch die Kontrolle über die Tests für ihre weiterentwickelten Versionen, bevor sie diese produktiv bereitstellen

[26] `https://nodejs.org/`
[27] `https://www.npmjs.com/`

[Geers 2020, S. 224-226]. Dennoch bleibt bei diesem Ansatz die Tatsache nachteilig, dass die gesamte Hauptanwendung bei jedem Update von einem der verwendeten Micro-Frontends neu gebaut, bereitgestellt und heruntergeladen werden muss [Jackson 2019] [Venkatesan 2021] [Fleischer und Schröder 2021].

4.4.2. Laufzeitintegration

Für Micro-Frontend-Architekturen ist die Laufzeitintegration eine wichtige Technik, und man meint damit das Abrufen und Zusammenführen von Ressourcen von unabhängig bereitgestellten Ressourcen URL im Browser. Jedes Micro-Frontend muss dafür idealerweise die Basisanforderung erfüllen, dass es eigenständig - also frei von Projektabhängigkeiten (engl.: *Dependencies*) - kodiert ist. Drei wichtige Konzepte, wie diese Eigenständigkeit erreicht werden kann, werden in den nachfolgenden Kapiteln ausführlicher beschrieben.

Webkomponenten und 'Custom Elements'

Diese Technik der Webkomponenten (engl.: *web components*) nutzt eine Standard-Schnittstelle im Browser zum Definieren neuer Komponenten, welche sich dann als Tags ins HTML integrieren lassen. Jede Webseite kann dabei individuell selbst definierte Webkomponenten bereitstellen und nutzen. Die meisten Browser (Google Chrome, Microsoft Edge, Mozilla Firefox, Opera Mini) haben die Standard-Schnittstelle implementiert. Lediglich der Safari und andere Webkit-basierte Browser unterstützen zurzeit wegen eines Fehler (Bugs) die Technik nur eingeschränkt als 'autonomous custom elements' [28]. Die neuen Komponenten können die gesamte HTML-Struktur und das 'Document Object Model' (DOM) nutzen. Die Standardmethode zum Erstellen dieser Komponenten erlaubt den Webkomponenten auch eine native Kapselung und Isolierung eine eigenen DOM, was hilfreich ist, um Konflikte zwischen verschiedenen Frameworks, Bibliotheken oder Versionen zu vermeiden. Das DOM kann visualisiert werden als eine Top-Down-Darstellung aller HTML Elemente, aus denen eine Webseite besteht. Diese Darstellung folgt einer baumförmigen Struktur. Es übersetzt den Inhalt eines HTML-Dokuments in ein standardisiertes Objekt mit Eigenschaften, Attributen und Methoden, auf das funktionale Programmiersprachen wie JavaScript leicht zugreifen können.

Im Kontext von Webkomponenten werden Begriffe genutzt, die charakteristische Eigenschaften derselben treffend beschreiben, die hier kurz skizziert seien:

- **Custom Elements**: Dies ist ein benutzerdefiniertes Element, welches ein nativen `HTMLElement` erweitert. Ein 'Custom Element' kann als semantisches Element definiert werden und bei gegebener Architektur auch über Frameworks hinweg in jedem Browser gemeinsam genutzt werden[Farrell 2019, S. 86].
- **Shadow-DOM**: Dieser bezeichnet den isolierten und gekapselten Bereich eines 'Custom Elements', in welchem Stile, Javascript und HTML-Strukturen für die Komponente

[28] `https://tinyurl.com/caniuse-webcomponents`

festgelegt werden. Dies ermöglicht auch die Abstraktion der Komponente intern von der Nutzungsumgebung. Die Nutzungsumgebung der Komponente muss sich also nicht darum kümmern, wie die Komponente intern implementiert ist, und sie kann sich auf die Bereitstellung der Schnittstelle beschränken. Das Shadow-DOM folgt bis auf die Aspekte der Kapselung den gleichen Regeln wie der gesamte 'Document Object Model'. Das Shadow-DOM kann man sich als separiertes DOM für das 'Custom Element' vorstellen. Beispielsweise könnte ein 'Custom Element' 'book', welches einen Seitentitel und eine Menüschaltfläche enthielte, die folgende Struktur für das Shadow-DOM aufweisen:

```
1 <my-book>
2   <header>
3     <div>
4       <button>
```

Mit Shadow-DOM können die benötigten Elemente `<header>` , `<div>` und `<button>` in einer Bereichs-bezogenen Unterstruktur so platziert werden, so dass die Stylesheets der Dokumentebene die Schaltfläche im Shadow-DOM nicht 'versehentlich' neu formatieren. Dieser abgekapselte Teilbaum wird 'Shadow tree' genannt. Die 'Shadow Root' ist die Wurzel des 'Shadow tree' Teilbaum. Das Element, an das der Baum angehängt ist (`<my-book>`), wird als 'Shadow host' bezeichnet.

```
1 <my-book>
2   #shadow-root
3     <header>
4       <div>
5         <button>
```

Die Abbildung 4.1 veranschaulicht die Repräsentation der HTML-Elemente in einer baumförmige Struktur mit (siehe Abbildung 4.1 ❷) und ohne (siehe Abbildung 4.1 ❶) Shadow-DOM.

Das Shadow-DOM löst das Kapselungsproblem, indem es ein separates DOM an die Webkomponente anhängt [Farrell 2019, S. 354]. Bei Bedarf kann diese Kapselung auch aufgelöst werden. Webkomponenten kennen zwei Modi, die einen direkter Zugriff auf den Shadow-DOM ermöglichen. Der 'open'-Modus erlaubt, dass der Baum unterhalb von 'Shadow Root' zum Ändern geöffnet ist und auf die 'Shadow Root'-Referenz des Elements zugegriffen werden kann. Der 'closed'-Modus bewirkt, dass der Baum unterhalb von 'Shadow Root' zum Ändern geschlossen ist und nicht darauf zugegriffen werden kann, es sei denn, die Referenz wurde explizit von seinem Ersteller gespeichert [Farrell 2019, S. 364].

- **HTML Templates**: Für Webkomponenten mit komplexerem Output-Design kann ein Entwickler einen HTML-Block auf der Webseite definieren, den die Webkomponente ver-

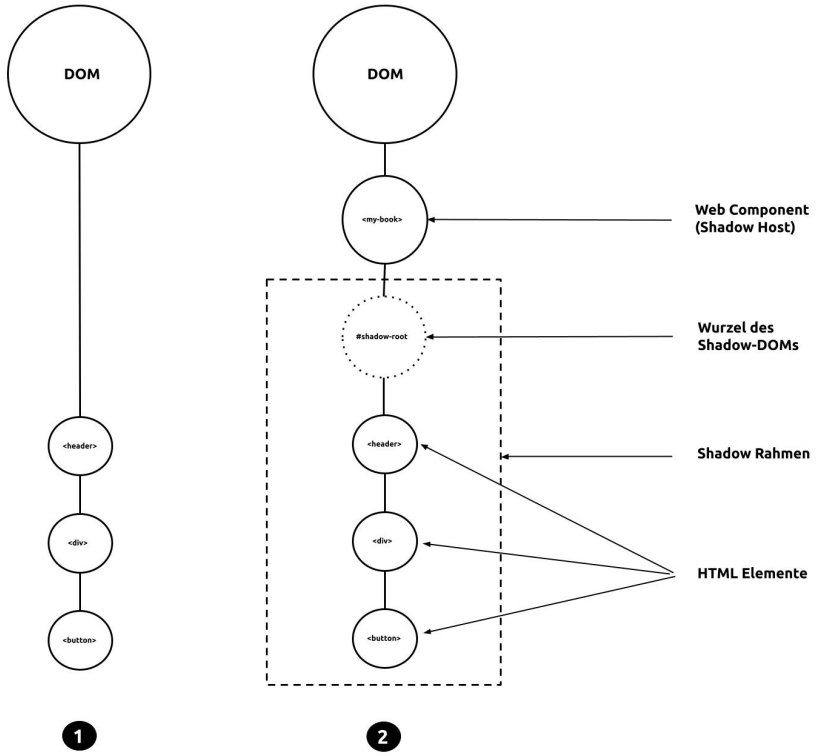


Abbildung 4.1. Baumförmige Struktur eines DOMs mit ❷ und ohne ❶ Shadow-DOM.

wenden kann. Dadurch kann die Ausgabe einer Webkomponente leicht geändert und an verschiedenen Layout-Anforderungen angepasst werden. Auch wird das Programmieren einfacher, weil man HTML nicht als Datentyp String im JavaScript-Code definieren muss. 'Templates' werden in einem `<template>`-Tag definiert. Es ist empfehlenswert, dem Template eine ID zuzuweisen, damit die Webkomponenten darauf verweisen oder zugreifen können [Farrell 2019, S. 321].

Die JavaScript-Klasse der Webkomponente kann auf diese Vorlage zugreifen und diese auch klonen, um sicherzustellen, dass überall dort, wo das Template verwendet wird, ein eindeutiges DOM-Fragment erstellt wird. Das DOM-Fragment kann auch modifiziert und direkt zum Shadow-DOM hinzugefügt werden.

Zusätzlich zu den Templates gibt es noch die sogenannten 'Slots'. Mit 'Slots' können Standard-Texte/Standard-Elemente in Vorlagen übersteuert werden. Beispielsweise könnte der Absatz 'Hello World' im Template der zweiten `<muster-element>`-Webkomponente an der entsprechenden Stelle im HTML per Slot mit einer `<h1>`-Überschrift übersteuert werden, wenn man im HTML folgende Syntax verwendet:

```
<template id="muster-element">
  <p><slot name="message">Hello World</slot></p>
</template>
...
<muster-element>
  <!-- zeigt 'Hello World' an -->
</muster-element>
...
<muster-element>
  <h1 slot="message">Hi my Message</h1>
</muster-element>
```

'Slots' sind hilfreiche Elemente zum Ersetzen von Inhalten [Farrell 2019, S. 346].

Der Einsatz von 'Web Components' gibt Softwareentwicklern bei wenigen Einschränkungen viel Flexibilität und Freiheit, um oft und nahezu überall spezifische Micro-Frontends einzuführen [Fleischer und Schröder 2021] [Geers 2020, S. 86] [Rappl 2021, S. 134].

Laut HTML-Spezifikation[29] kann nur ein benutzerdefiniertes Element definiert und abgerufen werden. Dieses kann nicht zur Laufzeit aktualisiert werden und kann somit nicht aus dem Browser unregistriert werden. Damit ist gemeint, dass nach dem Laden und Registrieren eines Custom Element dasselbe nicht mehr entfernt werden kann, ohne die gesamte Seite zu aktualisieren [Rappl 2021, S. 146]. Eine vordefinierte Fehlermeldung wird an dieser Stelle in der Browser Konsole angezeigt wie diese Diskussion[30] zeigt.

Die Verwendung von Webkomponenten als Integrationstechnik für Micro-Frontends bietet den Vorteil, dass es auf Standard Schnittstellen beim Browser basiert. Dadurch ist die Wahrscheinlichkeit geringer, dass eine Änderung an dem Standard die bestehende Kompatibilität für Browsers abbricht [Rappl 2021, S. 141] [Geers 2020, S. 97]. Shadow-DOM und 'Custom Elements' können bei Bedarf in Kombination verwendet werden und bieten für Micro-Frontends die Möglichkeit, isolierte Komponente zu entwickeln. Webkomponenten schaffen den Raum, für den Einsatz bei Micro-Frontends von verschiedenen Technologien oder von verschiedenen Versionen gleicher Technologie [Mezzalira 2021, Kap. 4].

Die Unterstützung für 'Custom Elements' bei älteren Browsers ist einfach dank der sogenannten Polyfills[31], die fehlende Funktionen bei bestimmten Browsers ergänzen helfen. Geers [Geers 2020, S. 97] weist aber darauf hin, dass der Einsatz von Polyfills bei Shadow-DOM einen nicht unerheblichen Aufwand erfordert. Aus diesem Grund schlägt er vor, dass bei Anwendungen, die auf älteren Browsern ausgeführt werden müssen, die das 'Shadow-DOM' nicht unterstützen, Alternativen - wie beispielsweise das manuelle Namespacing bei CSS-Klassen - in Betracht zu ziehen seien [Geers 2020, S. 97]. Einige Autoren weisen auf den Nachteil hin, dass es bei Webkomponenten keine Standards zum serverseitigen

[29] https://html.spec.whatwg.org/multipage/custom-elements.html#dom-window-customelements

[30] https://github.com/angular-extensions/elements/issues/91

[31] https://developer.mozilla.org/en-US/docs/Glossary/Polyfill

Rendern gibt [Geers 2020, S. 97] [Rappl 2021, S. 141].

Die mit Webkomponenten erstellten Frontendansichten können zusammen mit dem Backend der Microservices serverseitig zu einem Micro-Frontend verpackt werden. Wenn dieser Ansatz gewählt wird, können Frontend-Anwendungen auf das Routing reduziert werden. Dann trifft das Routing die Entscheidungen darüber, welcher Satz von Komponenten angezeigt werden soll, und wie die Orchestrierung von Ereignissen zwischen verschiedenen Webkomponenten organisiert wird.

Bei der Weiterentwicklung von Github[32] – einem web-basierten Service für Entwicklerteams mit der Möglichkeit zum Code-Austausch auf Grundlage der Open-Source-Versionskontrollsoftware git – kamen Webkomponenten zum Einsatz, um die schrittweise Migration seinen Frontend- Monolithen in eine Micro-Frontends-Architektur umzubauen [Oddsson 2021]. Die existierende modulare Architektur wurde schrittweise zu Webkomponenten extrahiert. Ein Modul nach dem Anderem wurde zu Webkomponenten umgebaut, in dem bestehenden Projekt integriert und anschließend werden mit optionalen Einstellungsmöglichkeiten die Daten an dem Komponenten weitergereicht, wenn dies erforderlich war.

Generell weist Mezzalira in seinem Buch [Mezzalira 2021, Kap. 4] darauf hin, dass es besonders wichtig ist die Kommunikation zwischen einem von einer Webkomponente umschlossenen Micro-Frontend und dem Rest der Ansicht entkoppelt beizubehalten. Zu beachten ist dass, Micro-Frontend geschlossen für Erweiterungen aber offen für die Kommunikation mit anderen Komponenten während andere nicht Micro-Frontends Komponenten auch offen für Erweiterung sein sollten [Mezzalira 2021, Kap. 4]. Im Hinblick auf Search-Engine-Optimierung (SEO), ist zu erwähnen, dass Inhalte, die innerhalb der Shadow-DOM gerendert werden, schwer von Webcrawlern zu indexieren ist, da diese Inhalte versteckt sind [Mezzalira 2021, Kap. 4].

Ein kürzlich von Liebel [Liebel 2022] publizierter Nachteil von Custom-Elements sei an dieser Stelle auch erwähnt. Man kann in einer Single-Page-Anwendung ein Custom-Element nur genau einmal definieren. Dies kann insbesondere bei Update zu Problemen führen, weil das Update das alte Custom-Element nicht ohne einen Reload der Seite überschreiben kann. Das Konzept 'Scoped Custom Element Registry'[33], das bei der Web Platform Incubator Community Group (WICG) als Vorschlag eingereicht wurde, soll hier Abhilfe schaffen. Aktuell soll für eine solche Registry schon ein Polyfill[34] vorhanden sein. Die Funktion wurde bisher nicht geprüft.

JavaScript Bibliothek 'Single-SPA'

'Single-SPA'[35] ist ein beliebtes Framework für Micro-Frontends. Es stellt als spezialisiertes Framework viele Funktionalitäten bereit, welche eine einfache Integration anderen

[32] https://github.com/

[33] https://github.com/WICG/webcomponents/blob/gh-pages/proposals/Scoped-Custom-Element-Registries.md

[34] https://github.com/webcomponents/polyfills/tree/master/packages/scoped-custom-element-registry

[35] https://single-spa.js.org/

Funktions-spezifischer Frameworks ohne größere Interdependenzen und ohne gegenseitige Störungen erlauben.

'Single-SPA' zielt darauf ab, die Micro-Frontends zur Laufzeit auf der Client-Seite programmatisch zusammenzufügen [Gilbert 2021, S. 80-89] [Rappl 2021, S. 159] [Geers 2020, S. 134-144]. Laut der offizieller Dokumentation[36] von 'Single-SPA', bietet das JavaScript-Framework eine Umgebung an, mit der man mehrere JavaScript Micro-Frontends auf einer Webseite in einer Frontend-Anwendung zusammenführen kann.

In einer 'Single-SPA'-basierten Architektur hat man eine einzige HTML-Datei, die als Ausgangspunkt für die Micro-Frontends dient. Sie enthält den JavaScript-Code als Verwaltungscode und ordnet dem Code von jedem Micro-Frontend eindeutige URL-Präfixe zu. Die verschiedenen Teams stellen ihre Micro-Frontends dann jeweils als JavaScript-Objekte zur Verfügung, die sich mit Blick auf die Verwaltung der Single-Page-Application und mit Blick auf die Kommunikation zu anderen Micro-Frontends an die definierten Schnittstellen und Konventionen halten müssen.

Die Organisation der Micro-Frontends in einer 'Single-SPA' Webseite wird in folgenden zwei Teilen organisiert:

- **Root Config**: Sie repräsentiert die zentrale Einstellung der gesamten Applikation. Dieser ist für das Rendern der HTML-Seite und das Laden der unterschiedlichen nötigen JavaScript Dateien aller Micro-Frontends verantwortlich. Die Root Config ist der Basiscontainer, der seinen HTML-Code mit den untergeordneten Anwendungen teilt, die bei 'single-SPA' registriert sind. Jedes Micro-Frontend wird registriert mit einem Namen, mit einer Funktion zum Laden des Codes des Micro-Frontends und mit einer Funktion, die bestimmt, wann das Micro-Frontend aktiv sein soll. Die Micro-Frontends müssen nur wissen, wie sie sich selbst vom DOM ein- und aushängen, haben aber keine eigene HTML-Seite. Die Root Config verwendet zur Spezifikation eine Import-Maps[37] in einem Skript-Tag oder als eigenständige Datei, um die aktuell bereitgestellten Versionen der Browser-internen Module der Anwendungen in den Basiscontainer zu laden. Die Import-Maps enthält eine Browserspezifikation für das Aliasing von 'Importbezeichnern' an eine URL. Der Importbezeichner dient als Index, dem das Skript von 'Single-SPA' folgen kann, wenn es nach Applikationen sucht, die ausgeführt werden sollen. Nachteilig an den Import-Maps ist die schlechte Browserunterstützung[38] bei der nativen Verwendung. Die fehlende Browserunterstützung kann zum Beispiel mit Open-Source-Tool SystemJS[39] überwunden werden. Bei 'Single-SPA' selbst sind diese Funktionalitäten schon integriert, sodass man die Import-Maps direkt in Projekten einsetzen kann.

  Das Layout der Frontend-Ansichten der untergeordneten Services kann in der Index-Datei der Root-Config-Anwendung angegeben werden. 'Single-SPA' bietet zusätzlich

[36] https://single-spa.js.org/docs/getting-started-overview
[37] https://tinyurl.com/import-maps-the-basic-idea
[38] https://caniuse.com/?search=importmap
[39] https://github.com/systemjs/systemjs

auch eine optionale Layout-Engine[40], die beispielsweise beim Routing hilft.

- **One-Pager-Services**: In der offiziellen Dokumentation von 'Single-SPA' werden die Anwendungen 'Applications' genannt. Eine Application ist eine auf einem Rechner laufende Anwendung, während Code zu Micro-Frontends mindestens auf zwei (Client und Server) läuft. Weil der Begriff 'Application' leicht zu Verwirrungen und Denkfehlern führen kann, wird in der vorliegenden Arbeit besser von Services statt 'Applications' gesprochen.

  Die One-Pager-Services werden in JavaScript Module mit den von 'Single-SPA' zur Verfügung gestellten Hilfsmitteln gepackt und mit der Root Config kombiniert. Die einzelnen Services können clientseitig - oft mit der Hilfe von kleinen Erweiterungen - auf unterschiedlichen Frameworks beziehungsweise mit unterschiedlichen Bibliotheken wie React, Angular oder auch VueJS und viele weitere realisiert werden. Jeder Micro-Frontend-Service sollte Methoden kennen, um sich selbst in ein DOM einzuhängen oder sich selbst in ein DOM auszuhängen.

Diverse Organisationen[41] [Argarwal 2021] setzen 'Single-SPA' aktiv ein.
Mit der empfohlenen 'Single-SPA'-Architektur kann eine Webseite einige Vorteile bieten, unter anderem:

- Verbesserte anfängliche Ladezeit dank Lazy-Load-Code.
- Konfliktfreie Verwendung von verschiedenen JavaScript-Frameworks wie ReactJS, VueJS, AngularJS, Angular, Ember.js auf derselben Seite ohne Seitenaktualisierung
- Unabhängige Auslieferung der einzelnen Services.

Geers in seinem Buch [Geers 2020, S. 140] weist explizit darauf hin, dass die Micro-Frontends oder andere One-Pager-Services keine Kontrolle über das umgebende HTML-Dokument haben dürfen. Ein Micro-Frontend darf also nur den Inhalt seines Root-DOM-Knotens ändern. Eine Folge davon ist, dass generierte Metadaten bei Crawling/Indexierung der Webseite gleich bleiben. Metadaten sind zusätzliche wichtige Informationen zu einem HTML-Dokument. Diese werden Mithilfe von META-Elemente angegeben und enthalten Informationen wie Eigenschaften des HTML-Dokuments, eine Liste von Schlüsselwörtern, Dokumentautor und vieles mehr. Gleichzeitig ist es aber denkbar, dass bestimmte Metadaten sehr spezifisch und wichtig für einen Seitentyp sind. Während diese Informationen nicht auf der Seite selbst angezeigt werden, können sie von Suchmaschinen und Webcrawlern gelesen werden und spielen eine zentrale Rolle für die Suchmaschinenoptimierung (SEO). Die Entwicklung eines Mechanismus zur effektiven Verwaltung von META-Elementen über alle Micro-Frontends hinweg erfordert in solchen Fällen einige zusätzliche Arbeit und Komplexität [Geers 2020, S. 140].

[40] https://single-spa.js.org/docs/layout-overview/
[41] https://single-spa.js.org/users/

Module-Federation

Module-Federation wurde auch schon als 'Game-Changer' [Jackson 2020] in der JavaScript-Architektur bezeichnet. Dieses ist in JavaScript Bibliothek Webpack ab Version 5 als Plugin [42] integriert und ermöglicht Anwendungen im Wesentlichen, Code von auf verschiedenen Servern gespeicherten Modulen auszuführen, während sie Teil einer gebündelten Anwendung sind.

Webpack ist ein Modul-Bundler für JavaScript-Anwendungen. Sie verwendet eine Reihe von JavaScript-Dateien zusammen mit Informationen zu Abhängigkeiten wie zu Bilddateien, die eine Anwendung benötigt, und erstellt daraus einen sogenannten Abhängigkeitsgraphen. Der Ausdruck Abhängigkeitsgraph meint hier die Darstellung der Anordnung und Verknüpfung dieser Dateien und Abhängigkeiten innerhalb einer Anwendung. Der Abhängigkeitsgraph zeigt, wie die Dateien miteinander interagieren [Owens 2020, S. 17]. In Verbindung mit Webpack Version 5 veröffentlicht, bietet das Plugin 'Module-Federation' die Möglichkeit der Laufzeitintegration für Micro-Frontend-Anwendungen. Der Abhängigkeitsgraph bildet in diesem Kontext dann nicht nur die Abhängigkeiten von lokalen Quelldateien und die Abhängigkeiten aus externen Bibliotheken ab, die sich in dem generierten Ordner `node_modules` befinden, sondern er zeigt auch Abhängigkeiten in der Cloud sowie in den JavaScript-Bundles auf, die in ein Micro-Frontend während der Laufzeit integriert werden können, beziehungsweise mit denen ein Micro-Frontend während der Laufzeit interagieren kann.

Mit dem Plugin 'Module-Federation' können JavaScript-Anwendungen Code dynamisch nachladen und Abhängigkeiten teilen. Dieser Laufzeitprozess bietet eine große Flexibilität bei JavaScript-basierten Web Applikationen, das auf anderen Wege nur schwer zu erreichen wäre. Beispielsweise erlaubt es diese Eigenschaft, ein Micro-Frontend zu ersetzen, ohne dafür das gesamte Projekt neu zu erstellen und ohne dafür die gesamte Applikation neu zu laden.

Wenn eine Anwendung das Plugin 'Modul-Federation' in ihrem Builds verwendet, und zusätzlich noch eine Abhängigkeit benötigt, um den verbundenen Code bereitzustellen, dann kann man mit Webpack auch diese zusätzliche Abhängigkeit vom Ursprung des verbundenes Builds herunterladen. Beim eigentlichen Prozess der Codezusammenführung muss man also nur eine Datei mit der Information bereitstellen, wo Webpack 5 den erforderlichen abhängigen Code finden kann.

Da das Plugin 'Modul-Federation' einerseits Code von Externen Quellen nachlädt und andererseits auch Code auf externen Quellen ausführen lässt, ist eine begriffliche Unterscheidung dieser beiden Zustände nötig. Als Begriffe werden **Remote** und **Host** verwendet. Der Begriff Remote bezieht sich auf von extern nachzuladenden Anwendung oder Module, die in die Anwendung des Benutzers geladen werden, während sich der Host auf die Anwendung bezieht, die der Benutzer zur Laufzeit über seinen Browser besucht.

Jeder Webpack-Build kann ein Host sein, der ein Container zum Laden anderer Builds ist. Es kann auch als Remote dienen, also ein Micro-Frontend, das vom Server erst noch

[42] `https://webpack.js.org/concepts/module-federation/`

geladen werden soll. Jede Anwendung kann je nach Kontext eine Remote-Anwendung oder auch eine Host-Anwendung sein; jede Anwendung kann je nach Kontext 'Verbrauchsmaterial' oder auch 'Verbraucher' anderer Module im System sein. Bidirektionale Hosts oder auch unidirektionale Hosts können leicht mit Webpack-Konfigurationen eingerichtet werden.

Die Anforderungen an 'Module-Federation' sind im aktuellen Migrationsprojekt nicht zu groß, da es nicht den Haupteinstiegspunkt und auch nicht eine andere vollständige Anwendung laden muss. Es muss nur der benötigte Code nachladen werden, der in vielen Fällen nur einige Kilobyte groß ist.

Eine JavaScript Datei, häufig `remoteEntry.js` genannt, enthält eine Liste der von der Remote-Anwendung bereitgestellten Dateien sowie die Anweisungen zum Laden dieser Dateien. Dies wird während der Integration der Remote-Anwendung in die Host-Anwendung verwendet und dient quasi als Vertrag zwischen Host und Remote.

Eine wichtige Funktion von Webpack ist das Entfernen von dupliziertem Code. Mit Modul-Federation bedient eine Host-Anwendung gegebenenfalls auch andere Remote-Anwendungen mit Abhängigkeiten. Wenn keine gemeinsam nutzbare Abhängigkeit vorhanden ist, kann die Remote-Anwendung automatisch ihre eigene herunterladen [Owens 2020, S. 133].

Grundsätzlich registriert die Host-Anwendung beim Start alle verfügbare Versionen, die für den jeweiligen Benutzer freigegeben wurden und genutzt werden. Diese werden in der Datei `webpack.config.js` festgelegt. Jedes Mal, wenn eine Datei `remoteEntry.js` von einer Remote-Anwendung geladen wird, fügt die Remote-Anwendung auch demselben Bereich ihre Versionen hinzu; dies passiert jedoch nur dann, wenn genau diese Versionen noch nicht in der Host-Anwendung vorhanden sind.

Zur Verwaltung dieser Versionen verwendet 'Modul-Federation' eine semantische Versionierung, also ein standardisiertes Benennungssystem für Versionsnummern von Software-Releases. Laut semver.org[43], die offizielle Webseite zu diesem Standard, sollte eine Versionsnummer der Form von *MAJOR.MINOR.PATCH* (beispielsweise, `1.2.5`) gehorchen. Die MAJOR-Version wird hochgezählt, wenn inkompatible Schnittstellen-Änderungen vorgenommen wurden; die MINOR-Version wird hochgezählt, wenn neue Funktionen oder Funktionalitäten hinzugefügt wurden, ohne die Kompatibilität zwischen den verschiedenen Versionen der Major-Version zu stören; und die PATCH-Version wird hochgezählt, wenn Fehlerkorrekturen oder andere Änderungen wie Benennungen vorgenommen wurden, die keinen Einfluss auf die Kompatibilität zwischen den Versionen der aktuellen Minor-Version haben.

Wenn eine Host-Anwendung beispielsweise eine Abhängigkeit A in der Version `15.0.0` besitzt, wird diese beim initialen Laden der Anwendung den freigegebenen Kontext eingefügt. Wenn ein Remote-Micro-Frontend dieselbe Abhängigkeit in der gleichen Version teilt, wird es dem freigegebenen Kontext **nicht** eingefügt, da diese bereits vorhanden ist. Sollte das Remote-Micro-Frontend Abhängigkeit A in der Version 15.0.1 besitzen, würde der

[43] `https://semver.org`

freigegebene Kontext zwei Versionen haben: 15.0.0 vom Host-Anwendung und `15.0.1` vom Remote. In einem solchen Fall wird von Registrierung gesprochen, denn alle einzigartigen Versionen, die von allen Remotes und Hosts geteilt wurden, werden registriert. Entsprechend der semantischen Versionierung wird die kompatible Version mit der höchsten Versionsnummer gewählt - also `15.0.1`. Sind zwei Versionen, wie beispielsweise im Falle von `15.1.1` und `16.1.0`, nicht zueinander kompatibel, werden zwei unterschiedliche Versionen konfliktfrei isoliert bereitgestellt. Alternativ können vom Entwickler eigene Regeln definiert werden, wie sich 'Modul-Federation' in diesem Fall verhalten soll [44] [Steyer 2020].

M. Steyer [Steyer 2021] hebt einige Herausforderungen beim Umgang mit 'Modul-Federation' hervor. Wenn für ein Micro-Frontend beispielsweise Angular geladen wird, das zum Beispiel von der Container Anwendung 'Shell' in einer abweichenden Framework-Version verwendet wird, dann beginnt Module-Federation, Angular zweimal zu laden, einmal als Version für die Shell und einmal als Version für die Remote Anwendung (Micro-Frontend).

Micro-Frontends, die auf 'Module-Federation' basieren, können das Problem mit den Versionskonflikten also auflösen. Für Micro-Frontends eröffnet 'Modul-Federation' auch Vorteile in Zusammenarbeit mit weiteren Webpack-Plugins, weil es Werkzeuge zum Reduzieren der generierten Datei-Größen und somit zur Reduzierung der Ladezeiten der Micro-Frontends anbietet [Rappl 2021, S. 216]. Mit 'Module-Federation' können Micro-Frontends sowohl auf Single-Page-Anwendungen als auch in unterschiedlichen Seiten zusammengestellt werden. Die Technik lässt breiten Raum für den Einsatz von nativen Browser Techniken wie zum Beispiel der Nutzung von DOM-Ereignissen zum Nachrichtenaustausch.

[44] `https://webpack.js.org/concepts/module-federation/`

Kapitel 5

Erfahrungen mit der Migration der Software: ExplorViz

5.1. ExplorViz: Struktur und Migrationsentscheidungen

Die Software 'ExplorViz'[45] visualisiert die Daten und Beziehungen von Softwarelandschaften in Form einer interaktiven 3D-Stadmetapher inklusive der jeweiligen relevanten Beziehungsinformationen [Fittkau u. a. 2017]. Die Stadt-Landschaften, die jeweils verschiedene Softwaresysteme repräsentieren, werden zur Laufzeit mithilfe von Laufzeitüberwachungsdaten generiert. Es verwendet dynamische Analysetechniken, um in Echtzeit die Kommunikation innerhalb der Architektur darzustellen. Diese Darstellung soll beim Nutzer ein Verständnis der Kommunikation zwischen Anwendungen fördern, indem es das Einfügen zusätzlicher Abstraktionsebenen zur Darstellung der Abhängigkeiten zwischen den Anwendungen erlaubt. Mithilfe der generierten Landschaften soll die Architektur von Programmen und Systemen und einige Details zur Kommunikation innerhalb der Anwendungsarchitektur übersichtlicher und mental zugänglicher gestaltbar sein. 'ExplorViz' basiert auf modernen Web Technologien wie JavaScript, WebGL [46] oder auch HTML5 Canvas[47].

Die Software erlaubt verschiedene Ansichten. In der abstrakten Ansicht visualisiert ExplorViz die gesamte Softwarelandschaft, bestehend aus den überwachten Anwendungen und deren Verflechtungen. In der Detailansicht visualisiert ExplorViz nur eine einzelne Anwendung, wobei die vorhandenen Instanzen und die Spuren von Methodenaufrufen zwischen diese als 3D-Modell visualisiert werden können.

5.1.1. Strukturen

Die Software ExplorViz ist ein 'Open-Source-Forschungssoftware' und wird ständig weiterentwickelt. Sie ist bereits in verschiedenen Bereichen wie zum Beispiel bei der Performanz Analyse einer Software, bei der Echtzeit Visualisierung oder auch in der Lehre zur Förderung des Softwareverständnisses zum Einsatz gekommen [Fittkau u. a. 2017]. Als

[45] https://www.explorviz.net/

[46] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[47] https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

'Open-Source-Forschungssoftware' dient sie gleichzeitig als Erfahrungs- und Experimentierplattform. Es wurde bisher vieles im Bereich Modularisierung [Zirkelbach u. a. 2020] [Zirkelbach u. a. 2019b] [Zirkelbach u. a. 2019a] und Microservices [Zirkelbach u. a. 2018] für ExplorViz untersucht. ExplorViz umfasst im Backend folgende sieben Microservices: User-Service, Landscape-Service, Adapter-Service, Trace-Service, Collab-Service, XRCollab-Service und Frontend-Service. Nachfolgende werden die Aufgaben der Services kurz umrissen und die Abbildung 5.1 veranschaulicht die Softwarearchitektur von ExplorViz mit den genannten Microservices.

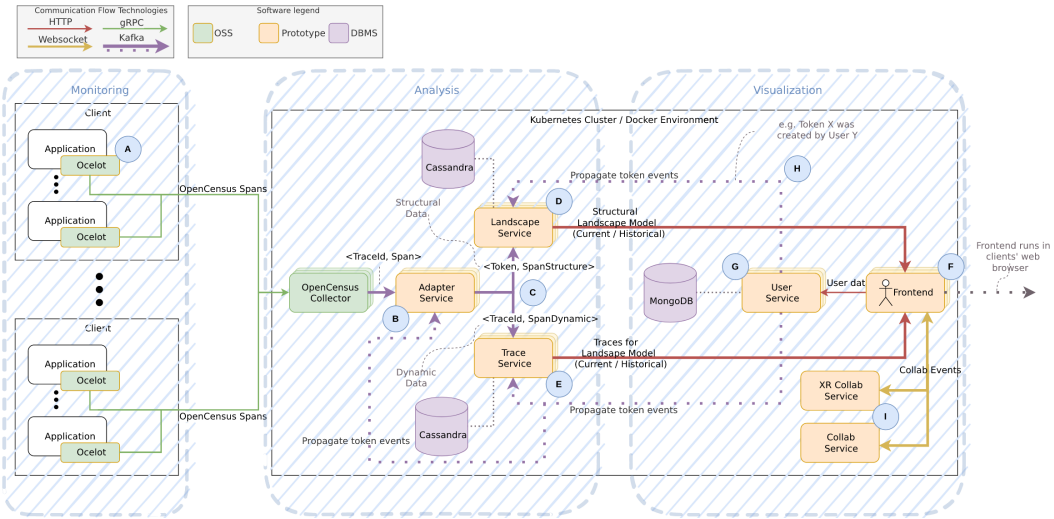


Abbildung 5.1. ExplorViz Microservices Architektur (Quelle: ExplorViz Entwickler-Team.)

Der *User-Service* (Abbildung 5.1-G) verwaltet Nutzerdaten und kümmert sich unter anderem um die Authentifizierung und Verwaltung der Nutzerrechte. Der Zugriff auf Backend Schnittstellen wird mit JSON Web Tokens (JWT) gesichert. Dieser Service muss jeden Zugriff von einem Nutzer authentifizieren, bevor auf diese Daten zugegriffen und ausgeliefert werden können.

Der *Landscape-Service* (Abbildung 5.1-D) verarbeitet und stellt Strukturdaten überwachter Anwendungen bereit. Anwendungen werden entsprechend der Zusammensetzung der Softwarehierarchie in einer umfassenden Landschaft gruppiert. Die strukturellen Daten werden persistiert.

Der *Adapter-Service* (Abbildung 5.1-B) ist ein Dienst, der alle von überwachten Anwendungen ausgegebenen Daten aufbereitet. Unter anderem führt der Adapter-Service auf jedem Datensatz eine Validierung durch. Für alle zu analysierenden Anwendungen sollte deshalb vom Nutzer ein sogenannter 'Landscape Token' angelegt und verwendet werden. Zudem kümmert sich dieser Dienst um die Strukturierung/Gruppierung der zu überwachenden Daten aus den verschiedenen Anwendungen und repräsentiert deren

aktuellen Stand.

Der *Trace-Service* (Abbildung 5.1-E) aggregiert dynamisch die Informationen aus überwachter Anwendungen, beispielsweise Methodenaufrufe mit Zeitliche-Informationen, und konstruiert basierend auf die Metadaten daraus Spuren. Die so entstandenen Spuren werden in einer Datenbank gespeichert und werden, sofern ein 'Landscape Token' gegeben ist, diesem zurückgeliefert.

Der *Collab-Service* und der *XRCollab-Service* (Abbildung 5.1-I) kommen zum Einsatz, wenn mehrere Nutzer bei der überwachten Software zusammenarbeiten und wenn deren Einsatz visualisiert werden soll.

Der *Frontend-Service* (Abbildung 5.1-F) dient dabei zur Anzeige der Daten der verschiedenen Backend-Microservices.

Eingesetzte Technologien

Da im Rahmen dieser Arbeit der Fokus auf das Frontend gelegt wird, ist die nachfolgende Beschreibung auf die dafür relevanten Technologien fokussiert. Die aktuelle Version von ExplorViz [48] Frontend ist eine Web-Applikation, die basierend auf dem JavaScript-Framework Ember.js [49] als Open-Source-Projekt entwickelt wurde. Ember.js erleichtert mit dem Konzept Model-View-Controller (MVC) die Entwicklung clientseitiger Single-Page-Webanwendungen. Das Framework unterstützt die Programmiersprache TypeScript, was auch im 'ExplorViz'-Kontext eingesetzt wird. TypeScript erweitert die Typ-konvertierten Programmiersprache um einer typgebundenen Programmiersprache mit einer Transpiler für entsprechenden Typ und Syntaxprüfungen. TypeScript wird stark von Microsoft als Open-Source-Projekt weiterentwickelt und kommt zum Beispiel in wichtigen JavaScript-Framework wie Angular zum Einsatz. Die Open-Source-Bibliothek Twitter Bootstrap [50] wird als Designsystem für die Umsetzung der UI Elemente verwendet. Ein Designsystem besteht aus UI-Komponenten und einem klar definierten visuellen Stil, der sowohl als Codeimplementierungen als auch als Designartefakte veröffentlicht wird. Wenn es von allen Produktteams übernommen wird, entsteht eine einheitlichere Benutzeroberfläche. Twitter Bootstrap ist eine Sammlung praktischer, wiederverwendbarer Codes, die in HTML, CSS und JavaScript geschrieben wurden. Es ermöglicht Entwicklern und Designern schnell vollständig mobile und Desktop freundliche Websites zu erstellen. Der Mechanismus zur Nutzer-Authentifizierung übernimmt ein externer Dienst Namens Auth0[51]. Auth0 ist ein sicherer und universeller Dienst, der die Authentifizierungs- und Authorisierungsfunktionalität gewährleistet. Es funktioniert auf der Basis von JSON Web Tokens[52], was ein Sicherheitsstandard zur Austausch von Sicherheitsinformationen zwischen Client und Server ist. Auth0 verwendet verschiedene Identitätsanbieter (zum Beispiel Google und

[48] https://www.explorviz.net/
[49] https://emberjs.com/
[50] https://getbootstrap.com/
[51] https://auth0.com/
[52] https://jwt.io/

Github) und passt zu einer Reihe von Plattformen, einschließlich sozialer Netzwerke wie Twitter und Facebook.

Die Backend Microservices werden in Java implementiert und verwenden diverse Technologien wie WebSockets[53], Apache Kafka[54] und Datenbanksysteme wie MongoDB[55] zur Verarbeitung und Persistierung von Datensätze aus Anwendungen.

Die Kommunikation zwischen Backend Microservices und das Frontend bei ExplorViz wird mit REST API auch RESTful API genannt gewährleistet. Eine API (Application Program Interface) ist ein Regelwerk, das es verschiedenen Programmen ermöglicht, miteinander zu kommunizieren. REST steht für den Ausdruck ***RE****presentational* ***S****tate* ***T****ransfer* und ist ein Architekturstil, der eine Reihe von Regeln definiert, um Schnittstellen zu erstellen. Bei einer Client-Server-Kommunikation schlägt REST vor, ein Objekt der vom Client (zum Beispiel ein Micro-Frontend) angeforderten Daten zu erstellen und die Werte des Objekts als Antwort an den Benutzer zu senden [Masse 2011]. Jedes Micro-Frontends spricht eine REST API URLs (falls eine solche vorhanden ist) an, um Domänen spezifische Informationen zu holen.

Technische Ziele

Nach der Entscheidung, die ExplorViz Frontend Anwendung auf Micro-Frontends umzustellen, wurden zusammen mit dem aktuellen Entwickler-Team einige Ziele identifiziert, die bei der Weiterentwicklung angestrebt werden sollten.

Eine Rahmenbedingung war, dass nach der Migration dasselbe JavaScript-Framework verwendet werden soll, was derzeit verwendet wird. So soll den anderen Mitentwicklern ermöglicht werden, ihre bisherigen Erfahrungen auch zukünftig in die migrierte Umgebung einbringen zu können. Entsprechend soll der vorhandene Quellcode mit möglichst wenig Änderungen übernommen werden, um die Weiterentwicklung bestehende Funktionalitäten in der Zukunft ohne große Einarbeitungszeiten ermöglichen zu können. Hinter der Rahmenbedingung steht der Wunsch, das Rad nicht jedesmal neu erfinden zu wollen.

Als Ziel wurde mit Blick auf die Vorteile der Micro-Frontends die Steigerung des evolutionären Entwicklungspotentials bei der Frontend-Darstellung angestrebt. Dies meint zum Beispiel, dass die Micro-Frontends möglichst mit nativen Techniken des Browsers so zu gestalten sind, dass eine unabhängige Evolution und eine konzeptionelle Erweiterung von 'ExplorViz' möglich bleibt und möglichst sogar einfacher wird.

Eine der größeren Herausforderungen bei der Implementierung vieler Micro-Frontend-Architekturen ist der Umgang mit der Authentifizierung. Für das Teilen eines gemeinsamen Zustandes zwischen mehrere Micro-Frontends wird entsprechend eine sichere und robuste Lösung gefordert.

Weiter sollten die Micro-Frontends auch so gestaltet werden, dass sie eine unabhängige Entwicklung und Bereitstellung ermöglichen. Jedes Micro-Frontend sollte ein unabhängiges

[53] https://tinyurl.com/websocket-api

[54] https://kafka.apache.org/

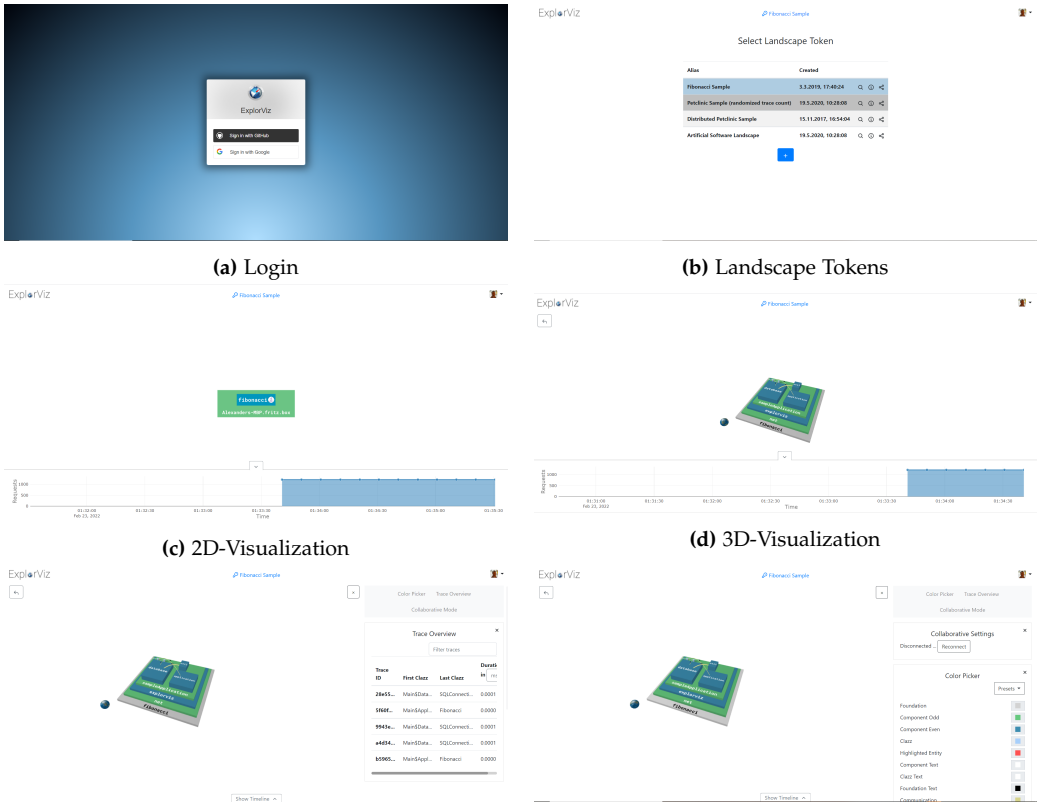
[55] https://www.mongodb.com/

Artefakt sein, das in jeder Umgebung bereitgestellt werden kann.

Um den Aufwand bei der Bereitstellung und Zusammenführung der einzelnen Micro-Frontends gering zu halten, war eine Hauptforderung eine adäquate Projekt-Struktur zu finden, die mit einer kleinen Anzahl an Entwicklern bewältigbar ist. Angesichts der zeitlichen Beschränkungen und angesichts des Prototypen-Charakters des Projektes kam man überein, das automatische Testen auf einige exemplarische Bereiche zu beschränken.

5.1.2. Strukturen der Ansichten im bisherigen System

Das aktuelle ExplorViz Frontend besteht aus den folgenden Ansichten:



(a) Login

(b) Landscape Tokens

(c) 2D-Visualization

(d) 3D-Visualization

(e) Trace Overview

(f) Color Picker und Collaborative Mode

Abbildung 5.2. Ansichten in ExplorViz

▻ **Login**: Es bietet dem Nutzer die Möglichkeit sich, in das System entweder über ein Google- oder ein Github-Konto anzumelden (Abbildung 5.2a).

- **Landscape**: Diese Ansicht fungiert als interaktive Menü und zeigt alle verfügbaren Token zur gegebenen Landschaftsvisualisierung an (Abbildung 5.2b).

- **Visualization**: In dieser Ansicht werden die aktuellen, zur Visualisierung freigegebenen Daten angezeigt (Abbildung 5.2c und Abbildung 5.2d). Interaktiv kann man mit einem Maus-Rechtsklick einen Kontext-Menü öffnen, über den man die folgende drei Ansichten auf der sogenannten Visualisierungseite aktivieren kann:

  - **Trace Overview**: Hier werden die beobachtete Spuren von Methodenausführungen in überwachten Softwareanwendungen tabellarisch angezeigt. Die angezeigte Informationen sind Ergebnis der Aggregation dynamischer Informationen überwachter Anwendungen, d.h Methodenaufrufe mit zeitlichen Informationen (Abbildung 5.2e).

  - **Collaborative Mode**: Diese Anzeige gibt die Möglichkeit für die bestehende Visualisierung weitere Nutzer hinzuzufügen (Abbildung 5.2f).

  - **Color Picker**: Hier wird dem Nutzer die Möglichkeit gegeben verschiedene Farbschema für die angezeigte von Landschaften auszuwählen (Abbildung 5.2f).

Diese Ansichten lassen sich auf die im Abschnitt 4.3.1 vorgestellte Domäne abbilden. So können die Seiten *Login*, *Landscape*, *Visualization* als Domäne betrachtet werden. *Trace Overview*, *Color Picker* und *Collaborative Mode* bilden hingegen die Unterdomäne der von *Visualization*.

Neben diese Ansichten besitzen einige Seiten auch eine eigene weiterführende Navigation. Bis auf die Login-Maske nutzen alle weiteren Ansichten die Navigation zum Nutzerprofil, das Informationen wie Nutzername, Nutzer ID, Profilbild bereitstellt und (wenn vorhanden) einen Abmelde-Button verfügbar macht.

Um uns genügend Informationen zu liefern, und um zu verstehen, wie die Migration zu Micro-Frontends funktionieren wird, wurde der Authentifizierungsfluss für bestehende Nutzer und die Erfahrung innerhalb der Anwendung für authentifizierte Nutzer analysiert. Unified Modeling Language (UML)[56] bot sich als Werkzeug zur Visualisierung der Prozessabläufe innerhalb der Anwendung an. Sie ist eine Modellierungssprache im Bereich der Softwareentwicklung, die darauf abzielt, Standardmethoden zur Visualisierung des Entwurfs eines Systems festzulegen. Die Interaktion des Systems mit dem Nutzer ist nachfolgend in der Abbildung 5.3 als UML-Sequenzdiagramm dargestellt.

5.2. Abhängigkeiten identifizieren

Aus verschiedenen Domäne lassen sich einzelne Kontexte und Grenzen für zukünftige Micro-Frontends identifizieren. Der *Login*-Kontext lässt sich aus einzelne Domäne extrahieren. Zusätzlich ist dieser Kontext die Voraussetzung, damit der Nutzer weitere Funktionalitäten verwenden kann. Es folgt daraus, dass einen potenziellen Informationsaustausch wie

[56] https://www.omg.org/spec/UML

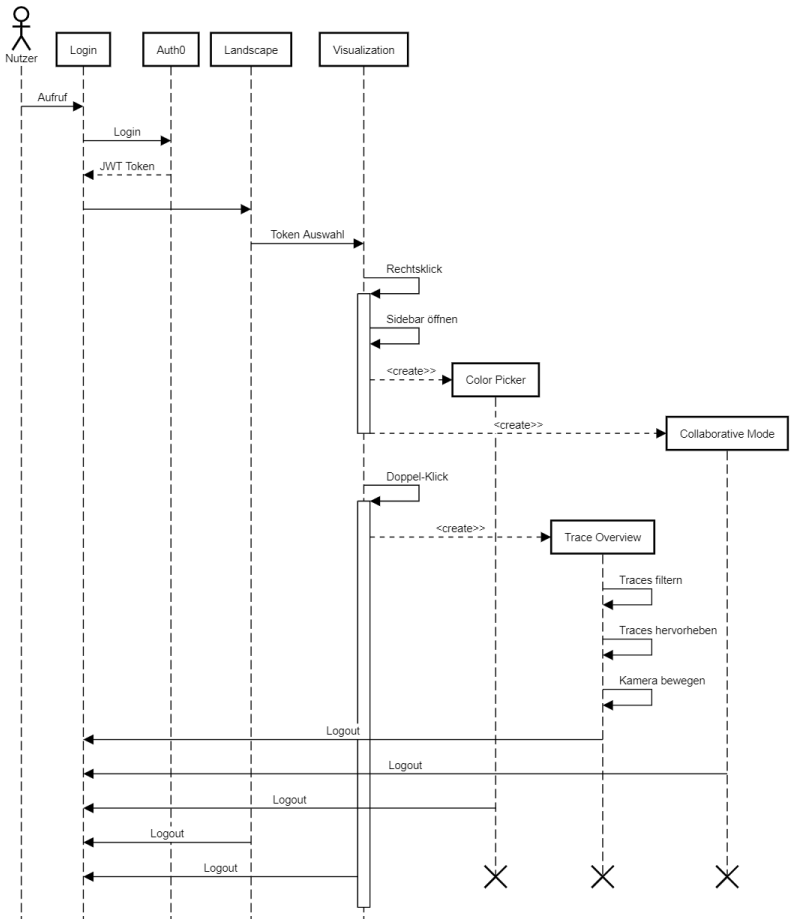


Abbildung 5.3. UML-Sequenzdiagramm vom Nutzer Workflow in ExplorViz

der Status der angemeldete Nutzer zwischen *Login* und weitere Funktionalitäten geben wird.

Aus den Nutzerinteraktionen (Abbildung 5.3) lässt sich auch ableiten, dass die Tokens, die auf der *Landscape* Seite aufgelistet werden, als Schlüssel agieren, um bestimmte Landschaften laden zu können. Mit der Auswahl eines solchen Schlüssels wird die passende Landschaft gerendert und es wird die Möglichkeit gegeben, Auskunft über die Landschaft zu bekommen. Als Idee folgt daraus, dass die Schlüssel (Token) weitergereicht werden könnten, wenn die Micro-Frontends miteinander kommunizieren.

Über den *Visualization*-Domain können gegebenenfalls Unterdomänen aktiviert werden. Eine Unterdomäne fungiert dabei einerseits als Konfigurationsoption (*Color Picker*), andererseits als Überblick (*Trace Overview*) beziehungsweise als Option zur Zusammenarbeit

(*Collaborative Mode*). Es ist darauf zu achten, dass die Daten in beide Richtungen ausgetauscht werden können, wobei auf der einen Seite die *Visualization* als Unterdomäne zu finden ist und auf der Gegenseite die Unterdomäne *Color Picker*, *Trace Overview* und/oder *Collaborative Mode* als Teil der Verbindung zu finden ist.

5.3. Entwurf der Micro-Frontends aus dem Monolithen

ExplorViz wird derzeit als Single-Page-Anwendung mit unterschiedlichen Seiten entwickelt. Die im Abschnitt 4.3.3 vorgestellte Technik bietet sich zur vertikalen Aufteilung der bestehenden Anwendung in verschiedene Micro-Frontends an. Bei einer vertikalen Aufteilung wird aufgrund der Beschaffenheit immer ein Micro-Frontend nach dem anderen geladen. In dieser Architektur-Konstellation müssen sich die verschiedenen Micro-Frontends nicht mit möglichen Abhängigkeitskonflikten zwischen Versionen derselben Bibliothek beschäftigen. Dies verringert die Wahrscheinlichkeit von Laufzeitfehlern. Außerdem wird nur eine CSS-Datei zur gleichen Zeit für das aktive Micro-Frontend geladen, was Versions-bedingte Stil-Konflikte vermeiden hilft.

Bei der Identifizierung der Abhängigkeiten zeigte sich auch, dass sämtliche Komponenten Domänen-übergreifend verwendet werden. Dies beeinträchtigt die Vertikale Aufteilung nicht und ermöglicht wegen der Unabhängigkeit der Module eine robuste Micro-Frontend-Architektur.

Die Micro-Frontends *Landscape* und *Visualization* lassen sich so migrieren, dass sie eine vertikaler Aufteilung unterstützen. Sie werden dabei dieselben Inhalte darstellen wie die ursprüngliche Applikation. Die Unterdomänen *Color Picker*, *Trace Overview*, *Collaborative Mode* sind Teil der *Visualization*-Domäne und lassen sich deshalb nicht so migrieren, dass sie eine vertikale Aufteilung unterstützen. Für diese müsste man auf die Horizontale Aufteilung zurückgreifen, weil die drei Micro-Frontends sich die Ansicht mit *Visualization* teilen. Sie behalten die Rolle, die sie auch schon in der monolithischen Anwendungen einnahmen.

Bevor die Micro-Frontends zur Darstellung genutzt werden können, muss der Nutzer authentifiziert werden. Da die Authentifizierung von jedem Nutzer aufgerufen wird, wird der Domäne *Login* als ein Micro-Frontend betrachtet. Da ExplorViz auf den externen Anbieter 'Auth0' zugreift, reduziert sich der notwendige Code zur Authentifizierung stark. Dies führte zu der Idee, das Micro-Frontend *Login* mit einem weiteren Dienst zu kombinieren, um dieses Frontend global auf der Seite agieren zu lassen.

Die folgende Tabelle listet die geplanten Micro-Frontends auf:

Micro-Frontend	Verantwortung	Authentifizierter Bereich?
Login	Authentifizierung der Nutzer	Nein
Landscape	Laden und Bereitstellung der Token zur Landschaften Generierung	Ja
Visualization	Generierung und Anzeige der Code Struktur als Landschaft	Ja
Color Picker	Einstellung verschiedener Farbschema zur Rendering der Landschaften	Ja
Trace Overview	Anzeige der beobachteten Spuren in der Code Struktur und Dynamische Navigation und Bewegung der Landschaft	Ja
Collaborative Mode	Möglichkeit geben zur Zusammenarbeit mit anderen Nutzer	Ja

5.4. Technische Aspekte zu einigen Diensten

Weil das Frontend als Micro-Frontend umzusetzen werden solle, würde mit Blick auf die bisherigen Erfahrungen Ember.js als Client-basiertes Framework ausgewählt. Weiterhin sollte die Authentifizierung durch der externer Dienst Auth0 beibehalten werden. Im Entwurf wurde deshalb das Micro-Frontend *Login* weggelassen.

Das JSON Web Token Konzept wird verwendet, um die Authentizität der Daten in den Micro-Frontends sicherzustellen und gegen Angriffe abzusichern.

Es wurde geplant, dass die im Abschnitt 4.3.5 vorgestellte Anwendungsshell als Container für einzelne Micro-Frontends fungiert. Diese Shell ist für das Ein- und Aushängen von Micro-Frontends zuständig. Weiter stellt die Shell die nötigen Schnittstellen für die Kommunikation der Micro-Frontends untereinander bereit. Weiterhin soll beziehungsweise muss sie während der Benutzersitzung sicherstellen, dass immer alle Authentifizierungsinformationen verfügbar sind.

5.5. Routing und Authentifizierung in der Anwendungsshell

Die Entscheidung zur Micro-Frontend-Architektur sowie die angestrebte vertikale Aufteilung der Hauptdomänen macht ein clientseitige Routing erforderlich. Über das Routing kann in der Anwendungsshell immer eindeutig definiert werden, welches Micro-Frontend gegebenenfalls nachzuladen ist.

Nach den Vorüberlegungen, die im vorherigen Kapitel beschrieben wurden, soll die Anwendungsshell auch für die Authentifizierung sicherstellen.

Sie prüft deshalb immer der Authentifizierungsstatus, sobald eine URL angefordert wird. Das meint, wenn ein nicht autorisierter Nutzer versucht, über direkten Link (auch *deep Link* genannt) auf einen authentifizierten Teil des Systems zuzugreifen, darf die Anwendungsshell dies nur zulassen, wenn der Nutzer über ein gültiges JSON Web Token verfügt. Wenn kein gültiges JSON Web Token vorhanden ist, dann sollte der Nutzer aus Sicherheitsgründen sofort zur aktuellen Login-Seite weitergeleitet werden. Über diesen Weg wird auch eine Sicherheitsstufe gegen Cross-Site-Scripting-Angriffe eingebaut, weil die bestehende Seite verlassen wird und geladene Scripte ihre Wirkung verlieren.

5.6. Kommunikation zwischen Micro-Frontends

Eine vorteilhafte Eigenschaft von Micro-Frontends ist, dass auf der Oberfläche asynchron nebeneinander verschiedene Prozesse ablaufen und gestartet werden können. Für die Umsetzung bietet sich also an, die DOM-Ereignisse zu verwenden, wenn die einen Micro-Frontend Daten und Nachrichten untereinander austauschen sollen. Über diesen Architekturansatz können entsprechend der weiter oben erwähnten Absprachen große Teile von bestehendem Code übernommen und die Einführung von neuer Technologie minimiert werden. Diese Herangehensweise erlaubt auch, dass zukünftig evolutionär Änderungen in das System einfließen zu lassen.

Da ein gültiges JSON Web Token für die Authentifizierung Micro-Frontends übergreifend erforderlich ist, wird ein gemeinsamer Zustand gebraucht. Der Browser bietet in diesem Sinne native Werkzeuge wie zum Beispiel den der Session Speicher (engl.: *session Storage*) an, um Daten zwischenzuspeichern.

Die Anwendungsshell kann den internen Speicher zum Beispiel als Schnittstelle zum Speichern der Daten verwenden. Auf diese Weise kann die Anwendungsshell den verfügbaren Speicherplatz überprüfen und sicherstellen, dass Daten nicht von anderen Micro-Frontends überschrieben werden.

Der Session Speicher ermöglicht auch das Speichern von Schlüssel-Wert-Paare, die mit dem lokalen Browser-Fenster verknüpft und auf dieses beschränkt, was die Sicherheit gegen Cross-Site-Scripting erhöht. Der Schutz gilt insbesondere auch (muss gelten), wenn (weil) die im Browser gespeicherten Daten nach dem Schließen des Browserfensters automatisch

gelöscht werden. Damit werden auch die von Auth0 generierten Token gelöscht, die während der Session von der Anwendungsshell dort gespeichert und von anderen Micro-Frontends verwendet wurden.

Mit Blick auf das Ziel der Robustheit, werden Default-Daten immer dann verwendet, wenn ein Micro-Frontend Daten aus externen, für den User nicht verfügbaren Micro-Frontends braucht, um bestimmte Daten zu verarbeiten. Die Default-Daten ersetzen echte Daten durch eine Art Simulation der Abhängigkeiten und Daten, um einem kontrollierten und planbaren Ablauf des Micro-Frontend-Prozesses zu ermöglichen. Die Default-Daten in Micro-Frontends können je nach Anwendungsfall statische Platzhalter oder einfache Callback-Funktionen oder aber auch Datensätze zur Ausgabe von Default-Ereignissen sein. Das Vorgehen erinnert ein wenig an dem Mock-Konzept im Bereich des automatisierten Software-Testens, aber hier das Ziel, die Software-Robustheit zu steigern und die Autonomie der Micro-Frontends zu erhöhen.

Während des laufenden Betriebs ist zu erwarten, dass manchmal größere Webkomponenten, externen Dateien oder auch Remote Anwendungen via Module-Federation geladen werden müssen. Solche Ladevorgänge können aus den verschiedensten Gründen (Lastspitzen, Pflegeroutinen, ...) manchmal länger als erwartet andauern. Bei solchen Zeitverzögerungen wird die Anwendungsshell Ladespinner und Platzhalter bereitstellen, um den User darüber zu informieren, dass hier noch mindestens ein Prozess auf eine Antwort wartet.

5.7. Automatisierte Tests

Neben exemplarische Unit-Tests in ausgewählten Micro-Frontends ist die Autonomie der Services eine Kerneigenschaft der Micro-Frontend-Architektur. Um die Funktionen der Kommunikation abzusichern, sind neben den API-Tests, also Tests gegen Schnittstellen, auch End-to-End-Tests wichtig. Die Anwendungsshell übernimmt Kernaufgaben über Micro-Frontends hinweg, deshalb soll insbesondere für dieses Micro-Frontend eine Anzahl automatisierter Tests entwickelt werden, welche die Korrektheit der Navigation zwischen den Micro-Frontends unter ausgewählten Rahmenbedingungen überprüfen und welche die Mechanismen des Nachrichten-Austauschs mithilfe von Simulationen testen.

5.8. Iterative Implementierung der Micro-Frontends

5.8.1. Technische Entscheidungen

Da die ursprünglichen Entwickler von ExplorViz nicht an der Migration beteiligt sind, musste man sich zusätzlich einen Eindruck verschaffen, wie der Code vor der Migration intern strukturiert ist. In diesem Rahmen musste man sich in das verwendete JavaScript Framework Ember.js einarbeiten und die Grundzüge der Architektur von ExplorViz

verstehen lernen. Die Vorbereitungen waren Voraussetzung dafür, um belastbare Entscheidungen treffen zu können, welche Technologien und Techniken zur Zusammenstellung der Micro-Frontends aussichtsreich verwendbar wären. Darauf aufbauend ließen sich Ideen entwickeln, welche Lösungen sich für die Migration anbieten und als geeignet erscheinen könnten. Eine systematische Form der Einarbeitung erlaubt das sogenannte Software-Prototyping. Software-Prototyping wird verwendet, um Entwicklungsvorschläge zu bewerten und Kernfunktionalitäten vor der umfassenden Implementierung auszuprobieren. Es erleichtert das Verstehen von Anforderungen, die benutzer- beziehungsweise problemspezifisch sein können und/oder die sich während der Konzeption des ersten Produktdesigns nicht sofort als relevant erkannt werden können. Das Software-Prototyping wurde in der vorliegenden Arbeit deshalb auf ExplorViz angewendet. Dazu wurden in Form von Evolutionären Prototypen erstellt, mit denen dann die in der Literatur gefundenen und im Abschnitt 4.4 ausgewählten Ansätze und Methoden für Micro-Frontends-Architekturen analysiert wurden.

Prototyping mit 'Single-SPA'

Gemäß der Vorvereinbarung, die Strukturänderungen möglich gering zu halten, entstand die Zielvorstellung, im Frontend einen ähnlichen Workflow wie im Backend Microservices zu aufzubauen, so dass wie bisher mehrere Dienste gleichzeitig als unabhängige Prozesse gestartet werden können. Bei der Suche nach unterstützenden Frameworks stieß man auf die Routing-Engine 'Single-SPA'. Damit erschien ein Prototyping effizient machbar zu sein, weil es einen Framework-unabhängigen Router bereitstellt, der mit einem SPA-Aktivator gekoppelt ist. Dieses Konzept erlaubt die gewünschte Orchestrierung mehrerer Single-Page-Anwendung-Instanzen in einer einzigen Anwendung. In der offiziellen Dokumentation von 'Single-SPA' wird auf die Verknüpfbarkeit mit Ember.js `single-spa-ember`[57] hingewiesen. Diese Möglichkeit war ein weiteres Argument dafür die aktuelle ExplorViz Implementierung in Ember.js in mehreren 'Single-SPA' Micro-Frontends aufzuteilen und zusammenzustellen. Weiter bietet 'Single-SPA' die Möglichkeit dynamisch eine HTML Seite so zu laden, dass immer auch das bisherige Verhalten von Ember.js, nämlich die gesamte HTML Seite inklusive externe JavaScript und CSS Dateien initial auszuwerten, beibehalten werden konnte.

Die vermutlich nötigen Einstellungen waren ausreichend auf der Webseite von 'Single-SPA' dokumentiert. Zusätzlich zu den im Abschnitt 4.4.2 vorgestellte Eigenschaften von 'Single-SPA', wird mit *single-spa-inspector* [58] auch ein Werkzeug zum Debuggen mehrere aktive Micro-Frontends angeboten. Außerdem wird das Framework ständig von der Entwickler-Community weiterentwickelt.

Mit Blick auf die genannten Eigenschaften war die Hoffnung groß, dass 'Single-SPA' sich optimal für die Migration von ExplorViz eignen könnte. Wegen dieser guten Aussichten wurde ein Software-Prototyping begonnen, um zu prüfen, wie gut sich 'Single-SPA' für

[57] `https://single-spa.js.org/docs/ecosystem-ember`

[58] `https://single-spa.js.org/docs/devtools`

die Migration von ExplorViz in eine Micro-Frontends-Architektur eignet. Es entstand der folgende Prototyp [59]. Beim Software-Prototyping wurde insbesondere versucht, möglichst viel vom originalen Code in Micro-Frontends aufzuteilen und die bisherigen, einzelnen Domänen wie zum Beispiel 'Landscape' möglichst strukturell zu erhalten.

Eine erste Erfahrung war, dass sich die CSS Dateien einzelner Applikationen nicht einfach voneinander isolieren lassen, ohne dass die zu Konflikten eventuell mit Klassen in anderen Micro-Frontend führen könnte. Für das eigenen Prototyping war eine Diskussion [60] mit einem 'Single-SPA' Entwickler hilfreich. In einem Kommentar [61] wurde angeregt, dass die gewünschte Isolierung der Stile nur mit Präfixe in CSS-Klassen erreichbar wäre.

Bei den Ideen und Umsetzungsvarianten mit Ember.js und 'Single-SPA' konnten leider noch einige andere Herausforderungen gefunden werden, die im Folgenden kurz erläutert werden.

Es kann vorkommen, dass mehrere Instanzen von Ember.js auf dieselbe Seite geladen werden müssen. Dies führt in der Regel zum Rendern einer weißen leeren Seite. Bei manchen Abhängigkeiten der Anwendung wie `@glimmer` wird in der Dokumentation darauf hingewiesen, dass eine Nutzung nicht möglich wäre, weil diese unter Umständen mit anderen bereits geladenen Version in Konflikt kommen könne[62] [63].

Als Versuch eines Workarounds wurde in der `Ember.js`-Anwendung die standardmäßige Root-URL geändert. Standardmäßig ist diese URL ein einfacher `'/'` (Slash). Mit einer Änderung zum Beispiel auf `'/meine-url'` kann ein Micro-Frontend mit der URL `http://mydomain.com/meine-url` aufgerufen werden. Leider unterscheidet die Ember.js Router mit Blick auf das 'Slash' am Ende zwischen `http://mydomain.com/meine-url` und `http://mydomain.com/meine-url/`, wobei die zweite, Problem-verursachende Variante auftreten kann, wenn man ein Micro-Frontend mit einem anderen zusammenführt. Solche Eigenschaften sind anti-robust und sind als potenzielle, schwer debugbare Fehlerquellen prädestiniert.

Zudem ist zu beachten, dass, sobald die nötigsten Einstellungen von Micro-Frontends vorgenommen werden sollen, der Nutzer im Entwickler Modus während des Startvorgangs der Anwendung manuell die Einstellungen vornehmen müsste, die unter folgender Link `https://tinyurl.com/ember-mfe-issue-1` beschrieben sind. Die Texte auf den nachfolgenden Github Referenzen beschreiben die verschiedenen gefundenen Probleme:

- `https://github.com/single-spa/single-spa-ember/issues/21`
- `https://github.com/single-spa/single-spa/issues/362`
- `https://github.com/single-spa/single-spa-ember/issues/17`
- `https://github.com/ember-micro-frontends/root-config/issues/3`

[59] `https://github.com/billyjov/poc-explorviz`
[60] `https://github.com/single-spa/single-spa/issues/362`
[61] `https://tinyurl.com/single-spa-issue-364-comment`
[62] `https://github.com/glimmerjs/glimmer-vm/issues/1252`
[63] `https://github.com/single-spa/single-spa-ember/issues/21`

▻ https://github.com/ember-micro-frontends/root-config/issues/2

Prototyping mit 'Webpack Module-Federation'

Ein alternativer Lösungsansatz, der den Start mehrerer Micro-Frontends als unabhängige Prozesse ermöglicht und der gleichzeitig eine optimale Lösung für den Umgang mit unterschiedlichen Versionen der Abhängigkeiten bieten könnte, wäre die im Abschnitt 4.4.2 vorgestellte 'Module-Federation'. Sie bietet für die clientseitigen Micro-Frontend-Komposition eine einfache Möglichkeit zum asynchronen Laden von JavaScript-Bundles. Aktuell ist eine Einbindung noch komplex, aber es ist zu erwarten, dass in Zukunft die Umsetzung einfacher werden könnten.

Eine Voraussetzung, um das Plugin 'Module-Federation' zu verwenden, ist die Bereitstellung von *Webpack* in der Version 5. Standardmäßig bietet Ember CLI keine Unterstützung dafür an. Aber Ember CLI, die Befehlszeilenschnittstelle von Ember.js, bietet eine Standardprojektstruktur, eine Reihe von Entwicklungstools und ein Add-on-System, die es Ember.js-Entwicklern ermöglichen, sich auf die Entwicklung von Ember.js-Anwendungen zu konzentrieren. Das Projekt ExplorViz setzt bereit die `ember-auto-import` Abhängigkeit ein, die in Ember CLI die Integration von Webpack-Einstellungen ermöglicht.

Um eine Remote Anwendung in der Ember.js-Anwendungsshell zu integrieren, könnten `ember-auto-import` und *Webpack 5 Module-Federation* genutzt werden. Für Micro-Frontend-Architektur ist aber auch zu fordern, dass der Code als Remote-Anwendung nachgeladen werden kann. Wenn versucht wird, Code aus der Ember.js-App als Remote-Anwendung zu veröffentlichen, wird dies nativ nicht funktionieren, weil `ember-auto-import` den eigenen Code der App überhaupt nicht verarbeitet. `ember-auto-import` verwendet einen Webpack-Build nur für die Abhängigkeiten von Drittanbietern (nicht die von Ember Core) und verwaltet die Verknüpfung mit der klassischen Nicht-Webpack-Build-Pipeline, die die App erstellt. Für das Nachladen würde sich eher ein anderes Modul namens *Embroider*[64] eignen, weil *Embroider* die gesamte App und Abhängigkeiten mit Webpack erstellt.

Für das Erstellen einer Micro-Frontend-Architektur ist mehr erforderlich, als nur das Build-System dazu zu bringen, einige der Komponenten in der Ember.js-App in Remote-Anwendungen umzuwandeln und diese zu einem Gesamtpaket zusammenzubauen. Typische Komponenten in einer Ember.js-App hängen implizit von vielen Abhängigkeiten ab, die als Modul-Importe nicht sichtbar sind. Sie verlassen sich auf die Laufzeitauflösung der anderen Komponenten, Helfer, Modifikatoren und Dienste, die sie verwenden. Und sie hängen implizit davon ab, dass die Rendering-Engine von Glimmer, eine von Ember.js verwendete Templating-Engine vorhanden ist.

Da Embroider sich noch in der frühe Entwicklungsphase befindet, sind derzeit alle Ember.js-Erweiterungen leider damit nicht kompatibel wie in der Dokumentation [65] darauf hingewiesen wird. Aus diesem Grund lässt sich derzeit ExplorViz nicht mit Ember.js und 'Module-Federation' umsetzen.

[64] `https://github.com/embroider-build/embroider`
[65] `https://github.com/embroider-build/embroider`

Die vielen impliziten Abhängigkeiten in Ember.js haben das Potential für viele Fehlerquellen und damit für wenig robusten Code. Man darf auf die Weiterentwicklung von Embroider gespannt sein und eventuell hoffen, dass sich in Zukunft damit in einfacher Weise gute Architekturen für robuste, gut gekapselte Micro-Frontend-Anwendungen schaffen lassen.

Prototyping mit 'Webkomponenten'

Bei Webkomponenten wird zwischen nativen und Framework-spezifische Webkomponenten unterschieden. Zusätzlich zu den im Abschnitt 4.4.2 vorgestellten Eigenschaften, lassen sich Webkomponenten mit unterschiedlichem JavaScript Framework kombinieren. Das 'Custom elements everywhere'-Projekt [66], gibt einen Übersicht basierend auf 30 Testfällen, wie gut verschiedene JavaScript Frameworks Webkomponente, speziell 'Custom Elements' unterstützt werden. Ember.js wird dort nicht aufgelistet, weil das Framework aktuell keine Standardlösung anbietet. Von Entwickler aus der Community wurden unterschiedliche Erweiterungen entwickelt, die die Möglichkeit geben Teile oder ganze Ember.js-Single-Page-Anwendungen als 'Custom Elements' zu definieren. Im Rahmen des Software-Prototypings wurden verschiedene Erweiterungen unterschiedlich tiefgehend ausprobiert. Von allen Erweiterungen bot nur `ember-custom-elements` [67] in der Version `2.1.0-1` eine einzige Lösungsmöglichkeit an, eine vollständige Ember.js-Single-Page-Anwendung in Webkomponente 'Custom Elements' zu kapseln.

Beispielsaft wurde ein Formular zur Eingabe von 'Todo-Listen' als Ember.js-Single-Page-Anwendung und eine Integration in einer in React entwickelte Anwendung programmiert, die die Verträglichkeit jeweils zwischen Ember.js und weitere JavaScript-Frameworks prüfte. Der Quellcode ist auf Github[68] zu finden. Nach jedem Schritt prüft die React-Anwendung die Formularkorrektheit.

In einem zweiten Software-Prototyp wurde eine Anwendungsshell in Ember.js erstellt, die zwei exemplarisch Micro-Frontends integriert. Das eine Micro-Frontend repräsentierte das oben genannte ToDo-Formular, während das andere Micro-Frontend einfach nur Texte und Platzhalterdaten anzeigte. Diese Micro-Frontends wurden als 'Custom Elements' mithilfe von `ember-custom-elements` gebaut, damit sie als JavaScript-Datei (und CSS-Dateien) von der Anwendungsshell verarbeitet werden konnten. Dafür wurde ein Skript entwickelt, dass zur Reduzierung der initialen Ladezeit die nötigten Dateien nur On-Demand (mit Lazy-Loading) lädt.

Ember.js stellt ein Routing-System `@ember/routing` zur verfügung, das eng mit dem Browser-URL-System verzahnt ist. Dieses Routing-System wird zur Navigation zwischen den beispielhaften Micro-Frontends verwendet und erlaubt, dass eine Seite durch eine eindeutige URL so dargestellt wird, dass der Nutzer, wenn aus irgendeinem Grund die

[66] `https://custom-elements-everywhere.com/`

[67] `https://github.com/Ravenstine/ember-custom-elements`

[68] `https://github.com/billyjov/ember-react-microfrontend`

Seite refresht beziehungsweise neu geladen werden muss, direkt zur gewünschten Ansicht weitergeleitet wird.

Während des Software-Prototypings wurden verschiedene Probleme erkannt. Zum Beispiel wird immer nur die als letztes geladene Datei (und das entsprechende Micro-Frontend) angezeigt, wenn von Ember.js auf der Seite mehrere Instanzen mit den jeweiligen, die Micro-Frontends repräsentierenden JavaScript Dateien geladen werden. Beim Debugging werden dabei in der Entwickler-Konsole des Browsers Warnungen angezeigt, die meist nicht mit dem Problem zusammenhängen. Wie bereits festgestellt scheinen mehrere Instanzen von Ember.js auf der selbe Seite sich gegenseitig zu stören. Auch war das Routing-System von Ember.js während des Prototypings nicht robust genug, um Konflikte in Routen Deklarationen zwischen Micro-Frontends und Anwendungsshell immer sicher aufzulösen.

Die gefundenen Problemen in Zusammenhang mit Ember.js und `ember-custom-elements` wurden auf Github unter folgenden Referenzen aufgeführt und dokumentiert:

▻ `https://github.com/Ravenstine/ember-custom-elements/issues/18`

▻ `https://github.com/Ravenstine/ember-custom-elements/issues/19`

Die Anwendungsshell wird häufig klein genug gehalten und übernimmt die Rolle des Dirigenten im Orchester der Micro-Frontends, weshalb das Routing-Problem problematisch ist. Eine Idee zur Lösung dazu war, die in Ember.js programmierte Anwendungsshell zu entlasten und ein eigenes Skript für das Lazy-Loading zu implementieren. Mit Blick auf das Ziel, in ExplorViz schon angewandte Technologien zu verwenden, wurde in einem zweiten Prototyping-Versuch auf das Framework Angular zurückgegriffen, um die Anwendungsshell zu implementieren. Angular basiert auf der JavaScript erweiternden Programmiersprache TypeScript und bietet Lösungen zur Anbindung und Erstellung von Webkomponenten. Es unterstützt auf verschiedene Wege zur Umsetzung des Lazy-Loading Konzept. Angular stellte sich als guter Kandidat für die weitere Umsetzung der Anwendungsshell heraus. Vorteilhaft war auch die vollständige Unterstützung von Webpack Version 5, so dass sich zukünftig weitere Micro-Frontends mit 'Module-Federation' ohne externe Abhängigkeiten unkompliziert in die Anwendungsshell integrieren lassen sollten.

Um im Weiteren auch das genannte Konfliktproblem beim Routing zu lösen, wurde im Abstimmung mit der ExplorViz-Entwicklerteam beim weiteren Prototyping auf Soft-Links in der Anwendung verzichtet. Das bedeutet, dass während der Navigation zwischen den Micro-Frontends immer ein Browserreload erfolgen wird, damit garantiert immer höchstens eine Ember.js-Micro-Frontend Javascript-Datei zur Laufzeit auf der Seite präsent ist.

Mit Blick auf die obigen Erfahrungen und die angesammelten Erfahrungen der anderen Entwickler im ExplorViz-Team werden die Micro-Frontends für ExplorViz zukünftig mit dem JavaScript Framework Ember.js entwickelt und als 'Custom Elements' mit der Bibliothek `ember-custom-element` gebaut, um anschließend mit einer minimale Angular

Anwendungsshell auf der Webseite zusammengestellt zu werden. Die Kommunikation mit den Backend Microservices wie zum Beispiel der *User-Service*, bleibt weiterhin mit dem REST API sowie mit den Live-Ereignissen aus den Services gewährleistet.

5.8.2. Migrationstechnik

Von den im Abschnitt 4.3 vorgeschlagenen Techniken wurde unter anderem wegen des aktuellen kleinen Entwicklungsteams die 'Greenfield'-Strategie zur Umsetzung der Transformation/Migration ausgewählt. Das meint, dass die Micro-Frontend-Architektur separat neu (auf der grünen Wiese) entwickelt wird, während das aktuelle produktiv System parallel im Einsatz ist. Auf diese Weise kann ExplorViz ihren Benutzern einen inkrementellen Mehrwert bieten, indem mit realen Daten vergleichend getestet wird, ob alles wie bisher funktioniert und ob den Datenverkehr ähnlich wie in der bestehenden Single-Page-Anwendung dargestellt wird. In dieser 'Greenfield'-Strategie wird also die bestehende lauffähige Anwendung als Vergleichsbasis für Fehlersuche und für Suche nach unerwarteten Systemverhalten verwendet.

Um der Überblick über Dienste zu behalten und vor allem der Aufwand bei der Aufsetzung der Infrastruktur zur Versionierung der Quellcode und Zusammenstellung zu einer Anwendung so gering wie möglich zu behalten, werden alle Micro-Frontends in einem Mono-Repository entwickelt. Mono-Repository (auch Monorepo genannt) [Brito u. a. 2018] beschreibt ein Konzept, bei dem Änderungen am Code unabhängiger (!) Anwendungen oder Bibliotheken über ein einziges gemeinsames Versionskontrollprogramm verwaltet werden, statt für jede Einzelanwendung eine eigenständige Versionskontrolle zu definieren. Die Versionsarchivierung kann beispielsweise mit 'git'[69] in einem Git Repository erfolgen.

Ein Monorepo macht es einfach, mehrere Projekte gleichzeitig herunterzuladen und zu aktualisieren und gemeinsame Abhängigkeiten zu verwalten. Diese Entscheidung erlaubte es, wie bei den Zielvereinbarungen im Abschnitt 5.1 gefordert, den Aufwand bei der Bereitstellung und Zusammenführung von Codes gering zu halten.

Basierend auf den oben genannten Entscheidungen und Erfahrungen wurde folgende Micro-Frontend-Architektur in der Abbildung 5.4 entworfen:

[69] https://git-scm.com/

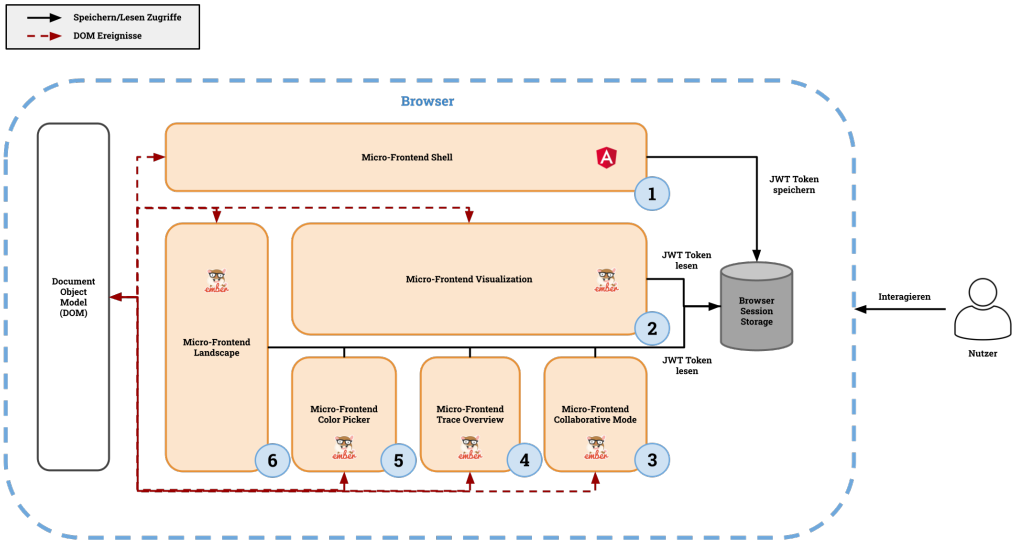


Abbildung 5.4. ExplorViz Micro-Frontend Architektur

5.8.3. Verantwortlichkeiten der Anwendungsshell

Aus den implementierten Prototypen, lassen sich folgende Verantwortlichkeiten der Anwendungsshell zusammenfassen:

- **Kommunikationsschnittstelle**: Wenn nötig, stellt die Shell (Abbildung 5.4-1) verschiedene kleine Schnittstellen zum Speichern, Löschen, Anfordern oder auch Bearbeiten von im Browser gespeicherte Daten. Sie prüft ebenfalls auf die korrekte Anwendung der geteilten Daten, beispielsweise prüft es den JSON Web Token zur Aktivierung eines geschützten Bereichs der Webseite. Die Shell leitet gegebenenfalls Daten von einem Micro-Frontend zum anderen Micro-Frontend weiter, wenn Daten von Ereignissen geholt werden.

- **Routing-Verwaltung**: Die Anwendungsshell lädt basierend auf der geladenen oder eingestellten Konfiguration das entsprechende Micro-Frontend. Die verschiedenen URLs, die zum Laden der/des Micro-Frontends benötigt werden, werden zur Laufzeit bestimmt und abgefragt. Die Anwendungsshell hat nur eine Entscheidungslogik und weiß zu keinem Zeitpunkt, wann sie wie viele URLs verarbeitet hat oder verarbeiten muss.

- **Anwendungsinitialisierer**: Jede Webseite braucht eine initiale Index HTML-Datei. Eine Startseite mit Defaulteinstellungen wird von der Anwendungsshell zur Verfügung ge-

stellt. Dabei werden nur die Bereiche von der Anwendungsshell angezeigt, die basierend auf der Authentifierungsstatus angezeigt werden dürfen.

- **Anwendungen Orchester**: Jedes Micro-Frontend wird der bestehende HTML DOM-Struktur hinzugefügt. Die Anwendungsshell entscheidet und bestimmt, an welcher Position in der Webseite das Micro-Frontend gerendert werden muss. Alle nötigste JavaScript- und gegebenenfalls CSS-Dateien werden von der Anwendungsshell nachgeladen.
- **Platzhalter**: Die Anwendungsshell sollte in der Lage sein, Daten oder auch Platzhalter anzuzeigen. Letzteres erfolgt zum Beispiel dann wenn Micro-Frontends durch Netzwerk Verzögerung oder Ausfall nicht sofort angezeigt werden kann.

5.8.4. Programmier-Paradigmen für die Implementierung eines Micro-Frontends

- **Erstellung des Projekts**: Solange Command-Line Interfaces zur Anlage von Strukturen für die gewählte Framework verfügbar sind, sind diese auch zu nutzen. Dies gilt beispielsweise bei Angular für den Angular CLI Befehl `ng new my-micro-frontend` oder bei Ember.js für den Ember CLI Befehl `ember new my-micro-frontend`.
- **Transformation zu Micro-Frontend**: Wenn als Technik beispielsweise Webkomponenten verwendet wird, sind in der Anwendung die benötigten Pakete zu installieren und die nötigen Einstellungen vorzunehmen.
- **Anwendungslogik und Unit-Tests**: Für die Anwendungslogik sind die dazu gehörigen Unit-Tests zu entwickeln. Im Vorfeld der Umsetzung eines Micro-Frontends sollte für das Projekt oder das Micro-Frontend eine passende Entwicklungsstrategie gewählt werden (beispielsweise Test-Driven-Development oder auch Behavior Driven Development).
- **Funktionstests**: Um das Verhalten eines Benutzers automatisiert zu simulieren und um das korrekte Funktionieren der Micro-Frontends in der Micro-Frontend-Architektur zu prüfen, sind spezifische End-to-End Tests zu entwickeln.
- **Dokumentationspflicht der Datenformate**: Die Daten, die von einem Micro-Frontend empfangen oder gesendet werden können, sind in einer Dokumentation zu erfassen.
- **Akzeptanzprüfung in der Anwendungsshell**: Nach der Implementierung des Micro-Frontend ist die fehlerfreie Integration in die Anwendungsshell zu prüfen. Insbesondere ist auch darauf zu achten, dass das Micro-Frontend nicht die Funktion der anderen Micro-Frontend stört.

Kapitel 6

Technische Umsetzung der Migration

Das ExplorViz Micro-Frontend-Repository ist auf Github unter dem Link[70] verfügbar. Entsprechend der im Abschnitt 5.1.1 genannten Vereinbarungen enthält das Repository alle Micro-Frontends und auch weitere Informationen in der Datei README.md des Hauptverzeichnisses. Die Datei beinhaltet Anweisungen zum Bauen und Starten der Anwendung sowie Informationen zum Starten der Backend Microservices, die für die Funktionsfähigkeit der Anwendung erforderlich sind. In seiner grundsätzlichen Struktur besteht die Anwendung aus einer Anwendungsshell, die während der gesamten Sitzung des Benutzers verfügbar ist und verschiedene Micro-Fontends wie 'Landscape', 'Visualization' und 'Trace Overview' (nach-)lädt. Cypress[71] wird als End-to-End-Test Framework verwendet, um ausgewählte Anwendungsfälle automatisiert zu testen. Entsprechen der Vereinbarungen mit dem Entwicklerteam wurde aus dem Monolithen ein Großteil des Codes migrierend kopiert. 'Migrierend kopiert' meint, dass die grundsätzliche Logik der Submethoden erhalten blieb und dass die meisten Funktionen ohne Änderungen in die neu erstellten Micro-Frontends übernommen wurden.

6.1. Projektstruktur

Die Abbildung 6.1 zeigt die Projektstruktur des Repositorys. Jeder Ordner in dieser Abbildung repräsentiert ein eigenständiges Micro-Frontend, was mithilfe externer Programme wie NPM gebaut und dann gestartet werden kann. Die bei Micro-Frontends oft gefundene und oft erwünschte Poly-Repository-Struktur wird auf diese Weise in dieser Arbeit im Gesamt-Repository simuliert. Die Aufsplittung in einzelne Micro-Frontend-Repositorys wäre leicht möglich. Jeder Ordner eines Micro-Frontends enthält neben seinen Programmcode immer folgende Dateien und Ordner:

[70] https://github.com/billyjov/poc-explorviz-shell
[71] https://www.cypress.io/

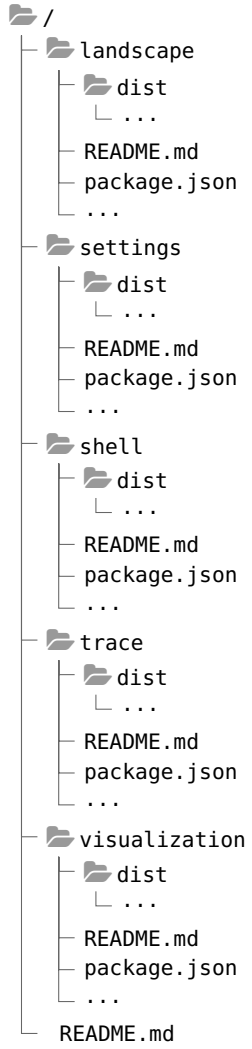


Abbildung 6.1 Projektstruktur des Repositorys

- **`README.md`**: Diese Datei enthält Micro-Frontend spezifische Information zum Starten, Bauen oder für das Ausführen der automatisierten Tests. Zudem findet man in der Datei Informationen, in welcher Form Daten an andere Micro-Frontends gesendet und in welcher Form von anderen Micro-Frontends empfangen werden. Auch findet man hier Informationen zu den jeweiligen technischen Systemvoraussetzungen.
- **`package.json`**: Diese Datei wird von dem Paketmanager NPM verwendet. Darin findet NPM die Metadaten zum Projekt wie zum Beispiel Name und Version eines Moduls. Wei-

ter sind darin funktionale Attribute des Projekts, Abhängigkeiten zu weiteren Modulen und/oder auch benutzerdefinierte Skripte definiert.

- **`dist`**: Dieser Ordner wird generiert, wenn der lauffähige Programmcode gebaut wird. Er enthält Dateien mit schwer lesbaren optimiertem Code, welche für die direkte Verwendung auf einem Produktivserver vorbereitet sind.
- **`<andere Dateien>`**: Alle anderen Dateien und Ordner sind Sprach- und Framework spezifisch angeordnet und repräsentieren den generativen Code des Micro-Frontends. Diese Dateien sind quasi die DNA des Micro-Frontends.

6.2. Anwendungsshell und Integration der Authentifizierung

6.2.1. Micro-Frontend 'Shell'

Die Anwendungsshell, wie oben schon erwähnt (und in der Abbildung 5.4-1 dargestellt) als Angular Anwendung programmiert, ist für die gesamte Dauer der Benutzersitzung präsent. Ihre interne Struktur nutzt als Standard das Angular Command-Line Interface (CLI), dem offiziellen Tool zum Initialisieren und Arbeiten mit Angular-Projekten. Damit die Anwendungsshell auch als Module-Federation-Host agieren und gegebenenfalls Remote Anwendungen laden kann, mussten zusätzlich auch die Einstellungen für Module-Federation vorgenommen werden. Zur Vereinfachung der Arbeit wurde dafür die von Manfred Steyer[72] entwickelte Angular Bibliothek `@angular-architects/module-federation` verwendet, die die Möglichkeit bietet, mit einem einzelnen Kommando die Grundstruktur von Webpack Version 5 Module-Federation in einer Angular CLI Umgebung zu erstellen. Der entsprechende Befehl lautete:

```
1 ng add @angular-architects/module-federation --project host --port 4200
```

Der genannte `ng add-Befehl` ist ein Befehl der Angular CLI und triggert neben dem Herunterladen der benötigen JavaScript Pakete einige weitere Aktionen. In vorliegenden Fall werden dies Webpack sowie `@angular-architects/module-federation` selbst heruntergeladen. Im Zuge des Befehls werden auch die benötigten Webpack Dateien `webpack.config.js` und `webpack.config.prod.js` angelegt, in welchen die Einstellungen für Module-Federation jeweils für den Entwicklungs- und Produktive-Modus zu finden sind. Weiter wird auch eine TypeScript-Datei `bootstrap.ts` generiert, in welcher die Angular spezifische Einstellungen zum Starten der Anwendung geschrieben werden. Zu guter letzte werden im Zuge des Befehls die Einstellungen in den existierenden `angular.json` und `main.ts` überschrieben. Der im obigen Befehl angegebene Option `--project host` führt dazu, dass die aktuelle Anwendung in der Module-Federation Architektur als Host Anwendung agiert

[72] `https://github.com/angular-architects/module-federation-plugin`

und die zweite Option `--port 4200` definiert den Port, unter welchem die Anwendung gestartet und aufgerufen wird. Diese Optionen finden sich dann auch in den Dateien `webpack.config.js` beziehungsweise `webpack.config.prod.js` wieder. Module-Federation wird in den JavaScript-Dateien als Plugin über folgende Einstellungen importiert.

```
const ModuleFederationPlugin = require("webpack/lib/container/
    ModuleFederationPlugin");
```

Die echt eindeutige Name `shell` der Anwendung ist unter den Optionen `module.exports` wie folgt angegeben:

```
output: {
    uniqueName: "shell",
    ...
},
```

Die Option `plugins` bietet die Möglichkeit, das Plugin 'Module-Federation' aufzurufen. Man kann in die Option gegebenenfalls auch Remote-Anwendungen oder geteilte Bibliotheken eintragen, wie der folgende Code exemplarisch zeigt:

```
module.exports = {
   ...
  plugins: [
      new ModuleFederationPlugin({
          remotes: {
              // Eintragen von remote Anwendungen
          },
          shared: share({
            "@angular/core": { singleton: true, strictVersion: true,
                requiredVersion: 'auto' },
            "@angular/common": { singleton: true, strictVersion: true,
                requiredVersion: 'auto' },
            "@angular/common/http": { singleton: true, strictVersion: true,
                requiredVersion: 'auto' },
            "@angular/router": { singleton: true, strictVersion: true,
                requiredVersion: 'auto' },
          })
        ...
      }),
      ....
  ],
  ...
}
```

Weil die Anwendungsshell eine immer präsente Anwendung ist, wird in der Einstellungen keine Einstellung zur Veröffentlichung von 'shell' Code eingetragen. Wenn per Remote Anwendungen nachgeladen werden müssen, sind sie dementsprechend in der `remotes` Option einzutragen.

Anlog wie beim `plugin` Objekt werden im `shared` Objekt werden alle Bibliotheken aufgelistet, die geteilt werden sollen. Die erweiterte Schnittstelle von 'Module-Federation' nutzt dafür folgenden benannte Eigenschaften:

- Die **singleton**-Eigenschaft gibt mit einem booleschen Wert an, ob die Bibliothek nur einmal geladen werden sollte.
- Die **requiredVersion**-Eigenschaft beschreibt, welche Version der Bibliothek geladen werden sollte. Das Schlüsselwort *auto* delegiert bei mehrere vorhandenen Versionen die Entscheidung der Auswahl an Webpack. Für die Entscheidung wird das Konzept der Semantische Versionierung genutzt.
- Die **strictVersion**-Eigenschaft dient dem Fehler- und Exception-Management. Mit der gesetzten Eigenschaft wird eine Exception geworfen, wenn versucht wird, mehrere, nicht kompatible Versionen Anwendungs-übergreifend zu teilen.

Beispielsweise Micro-Frontends mit Webkomponenten konnten nicht mit Module-Federation als Anwendungsshell geladen werden. Dafür wurde alternativ eine Angular Bibliothek (`@angular-extensions/elements` [73]) eingesetzt, die externe JavaScript Dateien per 'Lazy Loading' in die Anwendung geladen hat. Diese Open-Source-Bibliothek bietet dafür eine einfache Syntax an. Wie das nachfolgende Code-Beispiel zeigt, fungiert als Schnittstelle ein Attribut eines als Webkomponente definierten HTML-Element, dem direkt der Pfad (URL) zur gewünschten gegebene JavaScript-Datei *micro-frontend.js* zugewiesen wird.

```
1 <my-webcomponent-micro-frontend
2   *axLazyElement="my-awesome/path/micro-frontend.js"
3 ></my-webcomponent-micro-frontend>
```

Das Attribut `*axLazyElement` im HTML-Webkomponente `my-webcomponent-micro-frontend` der die `@angular-extensions/elements`-Bibliothek definiert dabei implizit, dass die per Pfad/URL definierte Datei mit dem 'Lazy loading' Mechanismus der Bibliothek geladen werden soll. Der Ladevorgang wird also erst gestartet, wenn das Micro-Frontend echt besucht und aktiviert wird.

6.2.2. Integration der Authentifizierung

Die Authentifizierung von ExplorViz basiert auf einen externen Dienst `Auth0`. Die Anwendungsshell fordert von `Auth0` eine JSON Web Token an und schreibt die Nutzerinformationen inklusive des Tokens nach dem erfolgreichen Einloggen in das 'Session Storage'.

[73] `https://github.com/angular-extensions/elements`

Alle Micro-Frontends, die aktiviert werden, rufen das Token und gegebenenfalls Nutzerinformationen aus der 'Session Storage' ab und schicken diese Daten die Schnittstelle für ihre Domäne mit. Im diesen serverseitigen Backend- beziehungsweise Domänenbereich muss jedes Micro-Frontend eine Logik zur Validierung der JSON Web Token implementiert haben und nur wenn sichergestellt ist, dass der Benutzer berechtigt ist, auf der Inhalt zuzugreifen, dürfen diese auch als Antwort an das Micro-Frontend zurückgesandt werden.

Zum Beispiel, um die verschiedene URLs der Micro-Frontends zu schützen, ist ein sogenannter 'Guard' mit Angular implementiert worden. Dieser überprüft, ob ein gültiges JSON Web Token vorhanden ist, bevor ein authentifizierter Bereich geladen werden kann.

Technisch wurden die JavaScript Bibliotheken Auth0-js [74] und Auth0-lock[75] genutzt, die sich in jedes JavaScript-Projekt und in jedes separate Micro-Frontend integrieren ließen. Da der bestehende ExplorViz-Monolith im Frontend die Programmiersprache TypeScript nutzt, konnte der Code und die Logik leicht übertragen werden.

6.3. Konsistenz der Benutzeroberfläche

Eine wichtige Anforderung bei der Migration ist es, das der optische Gesamteindruck der verschiedenen Micro-Frontends zu einem einheitlichen Gesamteindruck führt. Dieser kann durch Vorgabe eines 'Stil-Frameworks' beziehungsweise ein Designsystem-Framework erreicht werden. In allen Micro-Frontends wurde deshalb Bootstrap verwendet, weil es Anbindungen zu weiteren JavaScript-Frameworks erlaubt. Bootstrap verwendet beispielsweise in seiner Version 4 intern die JavaScript Bibliothek JQuery[76] um dynamische Reaktionen auf einer Webseite zu ermöglichen. Um den Einsatz von Bootstrap mit anderen Frameworks zu vereinfachen, sind im Laufe der Zeit viele Framework-spezifische Bootstrap Bibliotheken entstanden. Diese übernehmen oft das komplette CSS von Bootstrap und nutzen die jeweiligen Framework-spezifischen Techniken, um bestimmtes dynamisches Verhalten erzeugen zu können. Eine Implementierung für Ember.js ist `ember-bootstrap`[77], für Angular gibt es `@ng-bootstrap/ng-bootstrap` [78]. Während dieser Migration wurden alle oben Genannten bei Bedarf genutzt. In der Anwendungsshell wurde zur Darstellung der Elemente auf der Seite `@ng-bootstrap/ng-bootstrap` genutzt. In jedem Ember.js basierten Micro-Frontend kam `ember-bootstrap` zum Einsatz.

6.4. Micro-Frontend 'Landscape'

Nach der Extraktion aus dem Monolithen soll das Micro-Frontend 'Landscape' (Abbildung 5.4-6) eine tabellarische Liste von Tokens generieren, die zur Rendering von Land-

[74] `https://github.com/auth0/auth0.js/`
[75] `https://github.com/auth0/lock`
[76] `https://jquery.com/`
[77] `https://www.ember-bootstrap.com/`
[78] `https://ng-bootstrap.github.io/`

schaften erforderlich sind. Dieses mit Ember.js entwickelte Micro-Frontend erzeugt nach dem Kommando `ember new landscape` in der Root-Ordner der Mono-Repositorys ein neuer Ordner und legt nach der Installation aller Bibliotheken und Abhängigkeiten das gewünschte `landscape`-Projekt an. Um die Anwendung erweiterbar zu halten, sollte die generierte Single-Page-Anwendung auch die Programmiersprache TypeScript unterstützen. Da Ember CLI dies standardmäßig nicht unterstützt, wird auch automatisch die Bibliothek `ember-cli-typescript`[79] mit dem Befehl `ember install ember-cli-typescript` installiert und zusätzlich in den Root-Ordner der generierten Anwendung eingefügt. Dieser Prozess installiert alle notwendige Abhängigkeiten, passt gegebenenfalls Dateien an und generiert TypeScript-spezifische Einstellungsdateien.

6.4.1. Implementierung im Standalone-Modus

Um die Anwendung zu befähigen, Micro-Frontend beziehungsweise Webkomponente zu nutzen, sollte in der Datei `app/app.js` der mit Ember.js entwickelten Anwendung die nötigen Einstellungen vorgenommen werden. Die Funktion `customElement` aus der `ember-custom-elements` Bibliothek wird importiert und als Dekorator für die Klasse Definition verwendet. Mit Dekorator ist hier ein Entwurfsmuster (Musterlösung für ein wiederkehrendes Entwurfsproblem), mit dem eine Instanz einer Klasse um ein 'Verhalten' entweder statisch oder auch dynamisch erweitert werden kann, ohne das Verhalten anderer Instanzen derselben Klasse zu verändern [Mu und Jiang 2011]. In JavaScript und auch in TypeScript wird ein Dekorator mit dem Präfix '@' gekennzeichnet, wie der nachfolgende Code-Auszug zeigt.

```
1 import { customElement } from "ember-custom-elements";
2
3 @customElement("ember-landscapes-app", { useShadowRoot: true })
4 export default class App extends Application {
5
6 }
7 ...
```

Im obigen Beispiel wird das Attribut nach dem Instanziierung der Webkomponente auf das HTML-Element `<ember-landscapes-app></ember-landscapes-app>` angewendet. Das Attribut `useShadowRoot` aktiviert den Shadow-DOM als Eigenschaft der Webkomponente. Ein Dekorator kann also als statische Konfigurationskonstante betrachtet werden.

Das Ember-CLI-Buildsystem generiert standardmäßig zwei JavaScript Dateien: eine Datei für den Anwendungscode und eine Datei für die Abhängigkeiten. Analoges gilt für die Definitionen der Stylesheets (CSS). Um eine spätere Integration in der Anwendungsshell zu vereinfachen, wurde eine Ember.js Bibliothek, `ember-cli-concat` eingesetzt, um das Resultat des Buildsystems auf jeweils eine Datei für JavaScript und eine Datei für CSS zu

[79] `https://github.com/typed-ember/ember-cli-typescript`

reduzieren. Die notwendige Einstellungen für die Bibliothek sind in der Dokumentation[80] zu finden.

Zur Automatisierung des Build-Prozesses wurden einige Skripte entwickelt. Diese befinden sich in der Datei `package.json` und können mit dem Node-Package-Manager (NPM) aufgerufen werden. Beispielsweise lässt sich mit dem Befehl `npm run build:all` das Micro-Frontend 'Landscape' als 'Custom Element' bauen und die nötigsten Dateien generieren und an einem Zielort zur Konsumierung kopieren. Alle verfügbaren Skripte sind in der Datei README.md[81] dokumentiert.

Zur Implementierung der Ablauflogik sowie für die dazugehörigen Unit-Tests wurde die Logik inklusive bestehende Tests aus dem Monolithen übernommen. Die Abhängigkeiten zu den anderen Bereichen des Monolithen wurden entfernt beziehungsweise durch passende (Daten-)Strukturen ersetzt.

Eine große Herausforderung für den Standalone Modus, war der Umgang mit der Authentifizierung, denn jede REST API Abfrage an das zugehörige Microservice erfordert ein JSON Web Token. Mit der Abwesenheit der Anwendungsshell in diesem Modus muss eine Alternative zur Verfügung gestellt werden. Die Nutzung des externen Dienstes Auth0 Authentifizierung bietet der Vorteil, dass der Code zur Integration der Authentifizierung minimal bleibt. So konnte dieser Code auch in der Micro-Frontend 'Landscape' übertragen werden. Dort kommt er nur zum Einsatz wenn, kein JSON Web Token (beispielsweise in Standalone Modus) gefunden wurde. Mit der folgenden If-else-Anweisung lässt sich in der Datei `app/routes/index.ts` überprüfen ob dieser Token vorhanden ist.

```
1 if (!sessionStorage.getItem("accessToken")) {
2   this.replaceWith("login");
3 } else {
4   this.transitionTo("landscapes");
5 }
```

Sollte der JSON Web Token nicht vorhanden sein, wird der Nutzer zu der Login Maske weitergeleitet; ansonsten landet er auf der 'Landscape'-Seite. So kann ein ähnliches Verhalten wie in der Anwendungsshell erreicht werden.

Eine weitere Anforderung dieses Micro-Frontends ist die Möglichkeit zu geben, ausgewählte Landschaft-Token an das Micro-Frontend 'Visualization' weiterleiten zu können. Dafür werden im Abschnitt 4.3.6 vorgestellten DOM-Ereignisse genutzt. Sobald ein Nutzer auf ein Landschaftstoken in der Tabelle klickt, wird ein Ereignis angetriggert. Das Ereignis soll zum einen das Navigieren zu dem Micro-Frontend 'Visualization' in der Anwendungsshell anstoßen und zum anderen den Wert des Tokens mitsenden. Der folgende Code-Auszug veranschaulicht dieses Ereignis.

[80] https://github.com/sir-dunxalot/ember-cli-concat#documentation

[81] https://github.com/billyjov/poc-explorviz-shell/blob/master/landscape/README.md

```
window.dispatchEvent(
  new CustomEvent("landscapes:navigate-to-visualization", {
    detail: { token },
  })
);
```

Der Präfix `landscapes:` wird als Bezeichner verwendet, um in der Anwendungsshell mögliche Konflikte während der Integration mit anderen Ereignisse zu vermeiden.

Für dieses Micro-Frontend wurden bislang keine End-to-End-Tests entwickelt. Die weiteren Details zum Micro-Frontend und zu der Kommunikation sind in der Datei README.md[82] im Root-Ordner dokumentiert worden.

6.4.2. Integration in der Anwendungsshell

Ein wichtiger Punkt bei der Transformation des Monolithen Zum Micro-Frontend-Architektur ist die Integration der implementierten Micro-Frontends in die Anwendungsshell. Dafür wurde in der Anwendungsshell ein 'Web Component' beziehungsweise Webkomponente mit dem Namen `ember-landscapes-app` geschaffen, die einfach folgendermaßen in die HTML-Datei eingebunden wird.

```
<ember-landscapes-app *axLazyElement="elementUrl"></ember-landscapes-app>
```

Der Schlüsselwert `'elementUrl'` im obigen Beispiel bezeichnet die Variable mit dem Pfad zur JavaScript-Datei dieses Micro-Frontends. Der eigentliche Pfad ist in der Typescript-Datei der Komponente festgelegt.

```
 public elementUrl = 'assets/landscapes/landscapes.js';
```

Wie oben schon erwähnt, wird darauf verzichtet, die Quellenvariabilität der Micro-Frontend-Architektur durch verschiedene Server explizit nachzuweisen. Zur Nachweis der möglichen Quellenvielfalt wurde stattdessen `assets`-Ordner verwendet. Jedesmal, wenn nach Änderungen das Gesamtobjekt mit seinen verschiedenen Microservices neu gebaut wurde, würden ansonsten die erstellen Skripte in den `assets`-Ordner des jeweiligen Micro-Frontends hineingeschrieben werden. Dies sollte vermieden werden. Wenn in Zukunft einzelnen Micro-Frontend auf Server ausgelagert werden, so ist nur in der Variable `'elementUrl'` der Pfad durch die neue URL zu ersetzen.

Mithilfe von der JavaScript Bibliothek RxJS [83] und mit für den Anwendungsfall angepassten Funktionen konnten in der Anwendungsshell die Ereignisse für Landschaft-Token unkompliziert verwaltet werden. Technisch gesehen implementiert RxJS das Entwurfsmuster Observer. Ein Abonnement muss deshalb mit `subscribe()` aufgerufen werden, um die

[82] https://github.com/billyjov/poc-explorviz-shell/blob/master/landscape/README.md
[83] https://rxjs.dev/

neuste Daten aus dem Ereignis zu bekommen und gegebenenfalls verarbeiten. Folgender Quellcode-Auszug illustriert die grundsätzliche Struktur des asynchrones Nachrichten-Management.

```
...
fromEvent(window, "landscapes:navigate-to-visualization").subscribe(
  (event: Partial<CustomEvent>) => {
    // 1. navigieren
    // 2. Mit event.detail auf dem Landschaft-Token zugreifen.
  }
);
...
```

6.5. Micro-Frontend 'Visualization'

Ziel des 'Visualization'-Micro-Frontend (Abbildung 5.4-2) ist es, mit dem vorhandenen Landschaft-Token, Daten abzufragen und gewählte Anwendungen in einer zweidimensionalen Darstellung oder aber in einer drei-dimensionale Darstellung zu rendern.

Die 'Visualization'-Domäne ist von allen Micro-Frontends wahrscheinlich diejenige mit den komplexesten Anforderungen, da diese mehrere Bausteine zum Rendern und zum Navigationsmanagement innerhalb einer Landschaft braucht.

Zur Anlage des neuen Micro-Frontends wird in diesem Kontext auf die Ember-CLI zurückgegriffen, wobei auch das Plugin `ember-cli-typescript` zum Einsatz kam, um die zunehmende Bedeutung von TypeScript in der Frontend-Entwicklung gerecht zu werden und um mit Blick auf die Wiederverwendung von vorhandenem Code zukünftig auch TypeScript zu unterstützen.

6.5.1. Implementierung im Standalone-Modus

Die iterativen Transformationsschritte im Micro-Frontend 'Visualization' waren ähnlich wie beim Micro-Frontend 'Landscape'. Zuerst wurden der vorhandene Code nach Abhängigkeiten für das Landschaft-Rendering durchgesucht und anschließend wurde dann der entsprechende Code sukzessive für die Wiederverwendung beziehungsweise zur Kapselung angepasst. Zu den Anpassungen gehörten beispielsweise die Pfadangaben für die Imports-Stellen im Code, die auf bestimmte Funktionen und Methoden verweisen, deren Ort sich im Rahmen der Transformation verschoben hatte.

Bei der Transformation der Anwendung in 'Custom Elements' entstand eine Dauerschleife im Quellcode der `ember-custom-elements`-Bibliothek. Die Rekursion hat dazu geführt, dass die Anwendung nicht korrekt gebaut wurde. Als temporäre Lösung wurde im Rahmen der Arbeit eine angepasste Version `ember-custom-elements-patch` [84] der Bibliothek

[84] `https://www.npmjs.com/package/ember-custom-elements-patch`

unter Betrachtung der originale Lizenz-Bedingungen der Bibliothek veröffentlicht.

Eine mögliche Ursache könnte die große Anzahl der Abhängigkeiten zur Rendering der Landschaft sein, weil diese Rekursion sämtliche Abhängigkeiten aus der Anwendung zu bearbeiten versucht. Der Abbruch der Rekursion wird über die Exception `Maximum call stack size`-Exception [85] erzwungen. Diese JavaScript Exception 'Maximum call stack size', manchmal auch 'zu viele Rekursion'-Fehler genannt tritt auf, wenn zu viele Funktionsaufrufe vorhanden sind. Zum Zeitpunkt der Entwicklung dieser Arbeit war das Problem von den Entwicklern der `ember-custom-elements`-Bibliothek noch nicht abschließend gelöst. Eine Diskussion zu dem Thema ist auf Github[86] zu finden.

Die in der vorliegenden Arbeit genutzte, temporäre Lösung ändert nicht die grundsätzliche Verwendung der Bibliothek. Für das Micro-Frontend wurden die Einstellungen wie im Micro-Frontend 'Landscape' vorgenommen. Die weiteren Details zum Micro-Frontend und zum Build-Prozess sind in der Datei README.md [87] dokumentiert worden.

Dieses Micro-Frontend sollte das interaktive Verhalten anbieten, per Rechtsklick ein Kontextmenü aufzumachen. Ein entsprechendes DOM-Ereignis wurde jeden wählbaren Punkt definiert, wobei die Daten zentraler Bestandteil innerhalb des Micro-Frontends sind, wie der folgende Code-Ausschnitt veranschaulicht.

```
1  ...
2  window.dispatchEvent(
3    new CustomEvent("visualization:open-sidebar", {
4      detail: {
5        landscapeData: this.landscapeData,
6        applicationTraces: this.applicationTraces,
7      },
8    })
9  );
10 ...
```

Der Name des Ereignisses wird ähnlich wie bei anderen Micro-Frontends auch zusätzlich zu dem Funktions-beschreibenden Namen mit dem Micro-Frontend-spezifischen Präfix `'visualization'` ausgezeichnet. In dem Micro-Frontend zur Darstellung der Nachrichtendetails können sich immer auch Datensätze befinden, die auf andere Micro-Frontends verweisen und/oder gegebenenfalls in anderem Micro-Frontends angezeigt werden. Alle solche Ereignisse sind in der Datei README.md[88] dokumentiert.

Unit-Tests wurden aus dem vorhandenen Code genauso wie die Logik übernommen. Für dieses Micro-Frontend wurden bislang keine End-to-End-Tests entwickelt. Wie bei dem im vorherigen Kapitel beschriebenen Micro-Frontend 'Landscape' sind im Micro-Frontend 'Visualization' für die Darstellung der Elemente Authentifizierungen erforderlich um Daten

[85] https://stackoverflow.com/q/6095530/

[86] https://github.com/Ravenstine/ember-custom-elements/issues/19

[87] https://github.com/billyjov/poc-explorviz-shell/blob/master/visualization/README.md

[88] https://github.com/billyjov/poc-explorviz-shell/blob/master/visualization/README.md

aus dem Microservice aufzurufen. Es wurde in analoger Weise wie beim Micro-Frontend 'Landscape' gelöst.

6.5.2. Integration in der Anwendungsshell

Ähnlich wie bei dem Micro-Frontend 'Landscape' sollte das Micro-Frontend 'Visualization' als eine Komponente in der Anwendungsshell integriert werden. In der HTML-Datei der Komponente findet sich deshalb folgender Code-Ausschnitt:

```
1 <ember-visualizations-app *axLazyElement="elementUrl"></ember-visualizations-app>
```

Ein Migrationsproblem ergab sich bei dem Micro-Frontend beziehungsweise der Webkomponente 'Visualization', weil sich das Kontextmenü oft nicht über den rechten Mausklick öffnen ließ. Nach Untersuchung des Verhaltens konnte herausgefunden werden, dass das Rechtsklick-Event nur dann ausgelöst wurde, wenn es in der Anwendungsshell aktiv genutzt wurde. Grund für die beschränkte Aktivität war,, dass sich DOM-Ereignisse beziehungsweise DOM-Events innerhalb des Shadow-DOMs anders verhalten als bei klassischer DOM. Im Gegensatz zum klassischen DOM werden beim Shadow-DOM Ereignisse an Elementen innerhalb des Shadow-DOMs automatisch an die übergeordnete Komponente im klassischen DOM weitergeleitet. Um dieses Verhalten zu unterbinden, bietet sich ein Wechsel zu dem 'open'-Modus des Shadow-DOMs an. Die im Abschnitt 4.4.2 beschriebene Modus. Mit diesem lässt sich Klick-Event außerhalb des Shadow-DOMs auslösen. Diese Technik ist für den hier beschriebenen Anwendungsfall unproblematisch, da Bootstrap als Design-Bibliothek eingesetzt wird und da die CSS Klassen dank genutzter Präfixe so ausgezeichnet sind, dass die Stile anderer Micro-Frontends auf der Seite selbst dann nicht überschrieben werden, wenn der 'open'-Modus genutzt wird.

6.6. Micro-Frontends 'Color Picker' und 'Collaborative Mode'

Das Micro-Frontend 'Color Picker' (Abbildung 5.4-5) soll die Möglichkeit bieten verschiedene Farben für die Anzeige der Landschaften einzustellen. Ähnlich wie bei den vorherigen Micro-Frontends führte der erste Versuch, aus der Ember.js Anwendung Webkomponenten zu bauen und anschließend in der Anwendungsshell zu konsumieren, zu grundsätzlichen Problemen mit der Architektur. Die nachfolgenden Kapitel beschrieben die Machbarkeit-Implementierung des Micro-Frontends 'Trace Overview' (Abbildung 5.4-4) und zeigt, wie angesichts ähnlicher Probleme eine analoge Lösung für 'Color Picker' aussehen könnte.

Wie bereits oben bei der Beschreibung der Prototypen Phase erwähnt, können auf derselben Seite mehrere Instanzen von Ember.js nicht kollisionsfrei verwendet werden. Da das Micro-Frontend 'Color Picker' eine Unterdomäne des Micro-Frontends 'Visualization' ist, wird eine solche Kollision auftreten.. Um dieses Problem zu lösen, wurde versucht den

Build-Prozess so zu ändern, daß während des Builds die Ember.js Kern-Instanzen von dem Anwendungsspezifische Logik getrennt waren. Dadurch musste man nur der letzte Teil in der Anwendungsshell importieren. Der Versuch glückte und ist lauffähig. Als nachteilig stellte sich jedoch heraus, dass dadurch der Grad der Kopplung zwischen Micro-Frontends höher wird. Damit gehen wichtige Vorteile von Micro-Frontend Architektur verloren.

Das bei den Micro-Frontends 'Collaborative mode' und 'Trace Overview' auf Grund struktureller Ähnlichkeiten ähnliche Probleme wie beim Micro-Frontend 'Color Picker' zu erwarten waren, wurde in Absprache mit dem Entwicklerteam an dieser Stelle entschieden, für eines der drei genannten Micro-Frontends mit Module-Federation plus einem JavaScript Framework einen Micro-Frontend-Prototypen zu entwickeln.

Für die im nachfolgenden beschriebene Machbarkeitsimplementierung wurde das Micro-Frontend 'Trace Overview' ausgewählt, weil es unter anderem auch die Funktionalität der bidirektionalen Kommunikation mit dem Micro-Frontend 'Visualization' erfordert. Als JavaScript Framework für die Realisierung dieses Micro-Frontend wurde sich für Angular entschieden, weil Angular von dem Framework gut unterstützt wird, weil das Framework bereits das moderne TypeScript als JavaScript-Spracherweiterung unterstützt und auch weil Teile des vorhandenen Codes wiederverwendet werden konnten.

6.7. Micro-Frontend 'Trace Overview' mit Module-Federation

Das Ziel von 'Trace Overview' (Abbildung 5.4-4) als Prototyp ist es, Daten mit Zeitspannen und Spuren der Methoden-Aufrufe aus den überwachten Anwendungen tabellarisch anzuzeigen. Zudem sollte es aus diesen Daten die Navigation und Bewegung in der Landschaftsansicht erlauben können. Bewegung meint, dass der Nutzer durch Klicks in dieser Tabelle verschiedene Stellen in der Landschaft bewegen kann.

Zur Transformation zu Micro-Frontend, dessen Grundgerüst mit dem Angular CLI generiert wurde, wird der `@angular-architects/module-federation` Plugin verwendet, um nötigste Dateien zu generieren. Das Kommando `ng add @angular-architects/module-federation` wird ähnlich wie bei der Anwendungsshell die Webpack Einstellungsdateien anlegen. Da 'Trace Overview' in der Darstellung der Anwendung genutzt wird, hat das Angular in dem Module-Federation Kontext die Rolle der Remote Anwendung. In der `webpack.config.js` wurde entsprechend der Rolle des Micro-Frontends wie in folgendem Code Ausschnitt eingestellt.

```
...
 plugins: [
    new ModuleFederationPlugin({
        name: "trace",
        filename: "remoteEntry.js",
        exposes: {
            './Component': './src/app/app.component.ts',
        },
        shared: share({
          "@angular/core": { singleton: true, strictVersion: true, requiredVersion
              : 'auto' },
          "@angular/common": { singleton: true, strictVersion: true,
              requiredVersion: 'auto' },
          "@angular/common/http": { singleton: true, strictVersion: true,
              requiredVersion: 'auto' },
          "@angular/router": { singleton: true, strictVersion: true,
              requiredVersion: 'auto' },
          ...sharedMappings.getDescriptors()
        })
    }),
    sharedMappings.getPlugin()
  ],
```

Wegen der Rolle als Remote Anwendung sind in dem Module-Federation Plugin Objekt zwei weitere Eigenschaften hinzugekommen. `filename` definiert dabei den Namen der generierten Datei, die als 'Vertrag' zwischen Anwendungsshell und Micro-Frontend dient. Das Objekt `exposes` listet auf, welche Komponenten für die Welt außerhalb der Kapselung beziehungsweise für die Anwendungsshell zur Nutzung eingetragen werden.

6.7.1. Implementierung im Standalone-Modus

Anders als bei den Micro-Frontends 'Landscape' und 'Visualization' sind in 'Trace Overview' JSON Web Token und Daten aus der Authentifizierung hier nicht erforderlich, um die Funktionalität zur Verfügung stellen zu können. Die gewünschte Daten kommen aus dem Micro-Frontend 'Visualization' und werden tabellarisch dargestellt.

Damit die Tabelle im Standalone-Modus bedienbar bleibt und die verschiedenen Funktionalitäten unabhängig von der Anwendungsshell entwickelt werden können, werden Platzhalterdaten (Default-Daten) gebraucht. Die Daten dafür sind in dem Projekt statisch als JSON Dateien zur Verfügung zu stellen. Zum Start der Anwendung überprüft das Micro-Frontend auf Basis der boolesche Variable `isShellPresent`, ob die Anwendungsshell verfügbar ist. Je nach Zustand setzt sie entweder die Daten aus dem Anwendungsshell

oder aus den Platzhalterdaten.

Wegen des Charakters als Prototyp wurden in der vorliegenden Arbeit für dieses Micro-Frontend bisher keine End-to-End Tests entwickelt.

6.7.2. Integration in der Anwendungsshell

Für die Umsetzung von 'Trace Overview' ist das Micro-Frontend als unabhängiges Prozess in die Anwendungsshell zu integrieren. Der folgende Code Ausschnitt zeigt, wie unkompliziert das Micro-Frontend in der Datei `webpack.config.js` der Anwendungsshell registriert wird.

Der Folgender Code Ausschnitt zeigt wie eine solche Einstellung vorzunehmen ist:

```
1 new ModuleFederationPlugin({
2   remotes: {
3     mfe1: "mfe1@http://localhost:4500/remoteEntry.js",
4   },
5 });
6 ...
```

Unter `remotes` wird Mithilfe der Name (`mfe1`) und URL (`http://localhost:4500/remoteEntry.js`) des Codes des Micro-Frontends auf dieses verwiesen.

Zur Anzeige dieses Micro-Frontend wurde eine Host Komponente in der Anwendungsshell erstellt. Zusätzlich wurde auf dem Entwurfsmuster Observer mit RxJS zurückgegriffen, um die Daten aus dem Micro-Frontend 'Visualization' zu empfangen und verarbeiten. Die eine Richtung der Kommunikation ist das Empfangen der Daten zum Anzeigen und der Nachrichtentransfer vom Micro-Frontend 'Visualization' zum Micro-Frontend 'Trace Overview'. Dieses Ereignis wird asynchron ausgeführt und befüllt zur Laufzeit zwei Variablen `landscapeData` und `isShellPresent` in 'Trace Overview', wie der folgende Code zeigt:

```
1 ...
2 this.subscriptions.add(
3   this.evtBridgeService.landscapeDataSubject.subscribe((val) => {
4     (componentRef.instance as any).landscapeData = val.landscapeData;
5     (componentRef.instance as any).isShellPresent = true;
6   })
7 );
8 ...
```

Die zweite Richtung der Kommunikation von dem Micro-Frontend 'Trace Overview' zum Micro-Frontend 'Visualization' wird beispielsweise bei der Bewegung der Spuren in der Landschaft realisiert. Das folgende Code-Beispiel aus dem Micro-Frontend 'Trace

Overview' zeigt, wie das DOM-Ergeinis namens `trace:move-camera-to` zur Bewegung der Ansicht in dem Micro-Frontend 'Visualization' implementiert wurde.

```
1 public moveCameraTo(step: Class | undefined): void {
2     window.dispatchEvent(
3       new CustomEvent('trace:move-camera-to', {
4         detail: {
5           step,
6         },
7       })
8     );
9   }
```

In dieser Methode `moveCameraTo(...)` werden die Bewegungsschritte aus dem Micro-Frontend 'Trace Overview' als DOM-Ereignisse an 'Visualization' gesendet. Diese können dort empfangen und verarbeitet werden, was über folgenden Code erreicht wird:

```
1 fromEvent(window, "trace:move-camera-to").subscribe((event: CustomEvent) => {
2   if (event.detail) {
3     this.moveCameraTo(event.detail.step);
4   }
5 });
```

Die `fromEvent`-Methode aus der RxJS Bibliothek implementiert intern das Observer Muster und ermöglicht jedes Subscriber auf die Benachrichtigung zu hören und Daten auszulesen, wodurch eine interne Ausführung mit den empfangenen Daten innerhalb von Micro-Frontend 'Visualization' gewährleistet wird.

Auf eine vollständige Auflistung aller Nachrichten wird an dieser Stelle verzichtet. Sie sind in README.md[89] Datei im Hauptordner des Micro-Frontends 'Trace Overview' dokumentiert.

6.8. Automatisierte Tests

Für das Micro-Frontend 'Trace Overview' wurden End-to-End-Tests umgesetzt, weil dieses verschiedene diverse Anwendungsfälle bietet, um alle wesentlichen Funktionalitäten mit Testfällen in automatisierten Tests angesichts des zeitlich begrenzten Rahmen der Arbeit abdecken zu können. Die Tests mit dem browser-basierten Testsystem Cypress wurden in JavaScript und in Teilen in TypeScript entwickelt. Cypress bietet dabei die Möglichkeit, die Tests der Anwendung in verschiedenen modernen Browsern wie Google Chrome oder Mozilla Firefox ausführen zu lassen. Auch unterstützt Cypress das Framework Angular, das bereits in dem Micro-Frontend 'Trace Overview' zum Einsatz kommt.

[89] https://github.com/billyjov/poc-explorviz-shell/blob/master/trace/README.md

Das Micro-Frontend 'Trace Overview' stellt die beobachtete Spuren von Methodenausführungen in überwachten Softwareanwendungen tabellarisch dar. Mit dem Befehl `ng add @cypress/schematic` ließen sich End-to-End-Tests für die Cypress in dem Micro-Frontend 'Trace Overview hinzufügen. Mit dem Befehl installierte man alle notwendigen Abhängigkeiten für Cypress und legte zusätzlich alle benötigten Ordner und Dateien an, um danach die Entwicklung der Tests ohne zusätzliche Aufwand bei den Einstellungen beginnen zu können.

Nach dem Programmieren startet der Befehl `npm run e2e` die jeweils fertigen End-to-End-Tests für das Micro-Frontend 'Trace Overview'. Cypress öffnet nach der Ausführung der Befehl automatisch ein Browser Fenster für die Visualisierung der Tests. Zusätzlich wird auch eine Bedienungsoberfläche von Cypress geöffnet, die die Möglichkeit bietet, existierende Test-Dateien auszuführen. So konnten die Testszenarien 'Seite öffnen', 'Tabelle Anzeigen', 'Tabelle Bedienen' mit verschiedenen Testfällen abgedeckt werden.

6.9. Technische Reflektion zur Transformation

Die extrahierten Micro-Frontends basierten auf den im Monolithen 'ExplorViz' vorgefundenen internen Domänen. Entsprechend sind die extrahierten Micro-Frontends um Domänen wie 'Landscape' und 'Visualization' oder Unterdomänen wie 'Color Picker' und 'Trace Overview' herum modelliert worden. Für die Migrationsarbeiten war es hilfreich, dass der bestehende Monolith 'ExplorViz' während der Entwicklung als Vergleichsanwendung herangezogen werden konnte, um die korrekte Migration sicherzustellen. Mit der Migration wurden Abhängigkeiten so reduziert, was zu mehr Autonomie innerhalb des Projekts führte. Die in dieser Arbeit gemachten Erfahrungen bestätigten die Erwartungen aus der Literatur [Mezzalira 2019b] [Geers 2020, S. 33, 241] [Pölöskei und Bub 2021] [Jackson 2019] [Saring 2020].

Die Migration zu den Micro-Frontends erzwang das Konzept mit den Platzhalterdaten. Das Erstellen einfacher Platzhalterdaten wiederum half dabei, einen realistischen Eindruck des Layouts zu bekommen und gleichzeitig systematisch die Abhängigkeiten zum Fremdcode während der lokalen Entwicklung zu eliminieren. Die Transformation des Quellcodes in verschiedene Micro-Frontends mit bestimmten Zuständigkeiten half während des Programmierens, die Funktionalitäten innerhalb des Monolithen besser zu überblicken, weil die Zahl der zu beachtenden Parameter in den Micro-Frontends reduziert wurde beziehungsweise weil die Komplexität innerhalb der Micro-Frontend abnahm. Ein leichteres Verstehen lässt die in der Literatur erhofften schnelleren Entwicklungszyklen plausibel erscheinen, weil die Auswirkungen von Änderungen von den Entwicklern leichter überblickt und weil damit Änderungen schneller umgesetzt werden können [Prajwal u. a. 2021] [Jackson 2019] [Lemon 2020] [Yang u. a. 2019].

Das Prototyping mit verschiedenen Ansätzen, um eine passende Micro-Frontend-Architektur für ExplorViz war zeitaufwendig, weil verschiedene Technologien ausprobiert werden mussten. Besonders hilfreich war mit einer theoretischen Vorauswahl die Zahl der

Ansätze einzuschränken, für die Prototypen erstellt werden sollten. Eine Vorauswahl der Ansätze wird oft durch problemspezifische Eigenheiten festgelegt. Insgesamt stellte sich das Prototyping als effizient heraus und hat geholfen, um wichtige Erkenntnisse in einer frühen Phase der Migration zu erkennen und um verschiedene technische und architektonische Entscheidungen frühzeitig zu treffen (siehe Abschnitt 5.8.1). Es bestätigte somit die Erfahrung aus der Literatur, dass man Eigenschaften aus verschiedenen Micro-Frontends Ansätzen vergleichen sollte, um den am besten passenden Ansatz zu finden [Geers 2020, S. 156-169] [Mezzalira 2021, Kap. 4] [Venkatesan 2021].

Die Erfahrung, dass mit der Komplexität von Projekten die Einarbeitungszeit überproportional ansteigt [Geers 2020, S. 252] [Mezzalira 2021, Kap. 5] [Pölöskei und Bub 2021], konnte mit den eigenen Erfahrungen bestätigt werden. Die Migration von ExplorViz in eine Micro-Frontends-Architektur war zeitaufwendig, besonders weil sich der Autor mit dem bestehenden System nicht auskannte. Zu Beginn musste trotz der guten Unterstützung durch die bisherigen Entwickler viel ausprobiert und experimentiert werden, um möglichst passend zum bisherigen System eine angemessene Lösung für jeweilige Einzelaspekte zu finden (siehe Abschnitt 5.8.1).

Die Migration zu den Micro-Frontends erzwang die Definition von Platzhalterdaten. Das Erstellen einfacher Platzhalterdaten wiederum half dabei, einen realistischen Eindruck des Layouts zu bekommen und gleichzeitig systematisch die Abhängigkeiten zum Fremdcode während der lokalen Entwicklung zu eliminieren. Die Transformation des Quellcodes in verschiedene Micro-Frontends mit bestimmten Zuständigkeiten half während des Programmierens, die Funktionalitäten innerhalb des Monolithen besser zu überblicken, weil die Zahl der zu beachtenden Parameter in den Micro-Frontends reduziert wurde und weil dadurch die Code-Komplexität innerhalb der einzelnen Micro-Frontends abnahm. Ein leichteres Verstehen lässt die in der Literatur erhofften schnelleren Entwicklungszyklen plausibel erscheinen, weil die Auswirkungen von Änderungen von den Entwicklern leichter überblickt und weil damit Änderungen schneller umgesetzt werden können [Jackson 2019] [Yang u. a. 2019] [Lemon 2020] [Prajwal u. a. 2021].

Während der Implementierung stellte sich außerdem heraus, dass auf bestehende Tools und Command Line Interface (CLI) wie Angular CLI und Ember CLI stets zugegriffen werden musste, um den initialen Aufwand beim Aufsetzen und Konfiguration von Micro-Frontends zu reduzieren. Diese immer benötigten Tools sind wie ein Singleton in der Micro-Frontends-Architektur. Dieser Aspekt der Micro-Frontends, dass bei der Transformation zu Micro-Frontends die übergreifenden Strukturen zu analysieren sind, wurde in der Literatur [Fleischer und Schröder 2021] nur teilweise behandelt. Bei zukünftigen Umbauten von Softwarearchitekturen sollte man auf solche Punkte achten, um frühzeitig zu entscheiden, ob man sich frühzeitig für ein bestimmtes Framework entscheidet oder ob man eine Vielfalt von Frameworks und gegebenenfalls einen Wildwuchs an Frameworkversionen zulassen will.

Während der Migration stellte der zweistufige Routing-Ansatz, bei dem die Anwendungsshell eine einfache Navigation zwischen den Seiten durchführt und bei dem das

jeweilige Micro-Frontend die eigentliche Seite bestimmt, als sehr nützlich heraus. So konnte die Anwendungsshell in Bezug auf Code und in Bezug auf das Anwendungsprofil schlank gehalten werden (siehe Abschnitt 6.2.1). Wenn man unterstellt, dass weniger Code oft robusteren Code impliziert, dann bestätigt die Erfahrung die in der Literatur gefundene Erwartung, dass Micro-Frontend zu robusteren Code führen sollen [Geers 2020, S. 32] [Rappl 2021, S. 185] [Mezzalira 2021, Kap. 5].

Während der Migration musste zudem ein gemeinsames Vokabular beim Erstellen von Ereignissen zwischen Micro-Frontends definiert werden, um diverse mögliche Konflikte zwischen Nachrichten zu vermeiden. Diese gemeinsame Sprache war hilfreich, um die Module leichter gegeneinander in ihren Aufgaben und Funktionen abzugrenzen. Dieser Aspekt, dass über alle Micro-Frontends hinweg eine gemeinsame Sprache beziehungsweise bestimmte Konventionen benötigt werden, wird in der Literatur oft zusammenfassend als 'Verstehen der Domäne' bezeichnet [Geers 2020, S. 236-250] [Mezzalira 2021, Kap. 4].

Das automatisierte Testen auf verschiedenen Ebenen waren grundsätzlich leichter möglich, weil weniger Code und weniger Abhängigkeiten zu berücksichtigen waren (siehe Abschnitt 6.8). Aufgrund der zeitlichen Beschränkung dieser Arbeit konnte der Autor nur wenige Erfahrungen zum Entwerfen automatisierter Tests sammeln. Es bestätigte sich aber die Erwartung aus der Literatur, dass sich leicht verschiedene Teststrategien einführen ließen und dass die Tests unabhängig voneinander entwickelt werden konnten [Mezzalira 2021, Kap. 6] [Jackson 2019]. Dabei ist aber zu beachten, dass die Tests zwischen Micro-Frontends eine Herausforderung bleibt.

Mit Blick auf die Analogie zum monolithischen Familienunternehmen wird die gefundenen immer notwendige Codebasis durch die abteilungsübergreifenden Stabsstellen repräsentiert, wobei in Holdings abgesehen von Service-Unternehmen keinen expliziten Gegenpart zu Stabsstellen gibt, weil Töchter in einer Holding eher unabhängig voneinander agieren. Da sich dieser Aspekt auch in der Analogie der wirtschaftlichen Organisationsstruktur wiederfinden lässt, kann man vermuten, dass wirtschaftliche Organisationsstrukturen und Softwarearchitekturen viele weitere Ähnlichkeiten haben könnten [Rappl 2021, S. 201-212] [Mezzalira 2021, Kap. 4] [Argarwal 2021].

Kapitel 7

Fazit und Ausblick

7.1. Fazit

Für eine Migration einer monolithischen Frontend-Anwendung in eine Micro-Frontends-Architektur entschieden sich viele Entwickler gemäß der Recherche im 'multivocalen Literaturreview'-Prozess, weil sich so Teile der Software kleinteiliger und damit schneller entwickeln lassen und weil die Weiterentwicklung der Gesamtsoftware sich natürlicher auf mehrere unabhängige Entwicklerteams aufteilen lässt. Auch das Verstehen der Domäne wird damit erleichtert, wie unter anderem die Reflektionen zur Migration der Visualisierungssoftware ExplorViz in eine Micro-Frontend-Architektur bestätigten. Die Abgrenzung der Aufgaben in Micro-Frontends zwang zu einer vereinheitlichten strukturierten Benennung des Routings und damit zu Vereinfachung des Verstehens der Struktur. Die weiteren in der Literatur genannten Vorteile wie unabhängige Weiterentwicklung, einfacheres Testen und leichte Erweiterbarkeit bestätigten sich bei den Migrationsarbeiten in dieser Arbeit.

So wie in monolithischen Unternehmen oft Stabsstellen benötigt werden, so konnte im Rahmen der Migration für ExplorViz mit der Anwendungsshell eine schlanke und robuste Gerüststruktur für die Micro-Frontends gefunden werden. Die monolithische Software 'ExplorViz' wurde in sechs Micro-Frontends zerlegt. Für die Migration wurde als Strategie ein evolutionär-iterativer Zerlegungsprozess mit parallel aktiven monolithischen Altsystem für Tests gewählt. Die Zerlegung selbst orientierte sich an der bestehenden Seitenstruktur von 'ExplorViz'. Weil während der Migrationsentwicklung der Output vom bisherigen System als Testreferenz verwendet wurde, wurden automatisierte Tests zur Software-Absicherung erst im Nachgang geschrieben.

Der in der Literatur erwartete Vorteil, dass die Micro-Frontends unabhängig voneinander definierbar sind, konnte nur bedingt bestätigt werden, sobald man sich die Details der bewusst experimentell ausgerichteten Migration von ExplorViz anschaut. Zwei Micro-Frontends sind als Webkomponenten umgesetzt worden. Weiter wurde ein Micro-Frontend mit der Technik der Module-Federation umgesetzt, während ein anderes Micro-Frontend als übergeordnete Anwendung agiert. Der Versuch, das Micro-Frontend 'Color Picker' mit dem Framework Ember.js umzusetzen, führte zu Problemen, die in ähnlicher Weise auch bei den Micro-Frontends 'Collaborative mode' und 'Trace Overview' zu erwarten gewesen waren. Für 'Trace Overview' konnte über effizienten Prototyp-Implementation mit Angular im Kontext von Module-Federation ein grundsätzlicher Lösungsweg gefunden werden.

7.2. Ausblick

Viele Aspekte rund um Micro-Frontends mussten offen bleiben. Für die Nutzung von einer Visualiserungssoftware wie 'ExplorViz' ist eine Micro-Frontends-Architektur sicher sinnvoll, weil hier viele unterschiedliche Microservices zusammengeführt werden müssen. Zu großen Schwierigkeiten führt bei Micro-Frontends, die als Single-Page-Anwendung arbeiten, die Suchmaschinenoptimierung (SEO), wenn für die verschiedenen Varianten der Seite nur immer der gleiche Link zur Verfügung steht. Hier ist in Zukunft zu überlegen, wie man für Micro-Frontends leicht aussagekräftige Sitemap-Dateien erstellen kann.

Schon mit Blick auf Suchmaschinenoptimierung sind Micro-Frontends für klassische Webseiten wenig geeignet, aber zum Beispiel für die Cockpit-Bildschirme in Autos könnten die organisatorischen und technischen Vorteile wirklich hilfreich sein, weil hier viele unterschiedliche Entwicklerteams aus unterschiedlichen Firmen mit vielen unterschiedlichen Techniken zusammenarbeiten müssen/sollen.

In dieser Arbeit wurde eine Migration von einer Single-Page-Anwendung zu Micro-Frontends durchgeführt. Ausgehend von den hier vorgestellten Ergebnissen, wäre eine systematische Untersuchung von Performanz Unterscheidung beider Architekturen in die Zukunft durchzuführen.

Glossar

Ajax Der Begriff '**A**synchronous **J**avaScript **a**nd **X**ml' steht für ein browser-seitiges Kommunikationsprotokoll und dem dazugehörigen Softwarekonzept. Es erlaubt das asynchronen Nachladen von Daten ohne die Seite als Ganzes neu laden zu müssen . 7, 8

CSS Die formale Steuersprache '**C**ascading **S**tyle **S**heets' dient zur optisch ansprechenden Gestaltung und Strukturierung von Textdokumenten im HTML-Format und wird den Browsern unterstützt . 6, 8, 49, 54, 58, 59, 61, 65, 72, 73, 78

DOM Das '**D**ocument **O**bject **M**odel' ist eine Schnittstelle zwischen HTML und JavaScript . 36–38, 41, 42, 78

HTML Das '**H**yper**T**ext **M**arkup **L**anguage' ist eine Variante vom XML und ist die gängige Markup-Sprache für Text-Dokumente im Internet, die das Gruppieren und Strukturieren von Textinformationen nach inhaltlichen und formalen Kriterien für Suchmaschinen, für die Anzeige und für andere Ausgabegeräte (Drucker, Screenreader, Hololens, ...) erlaubt . 5, 6, 8, 28, 31, 36–39, 41, 42, 49, 58, 64, 65, 71, 73, 75, 78

JSON Das '**J**ava**S**cript **O**bject **N**otation' repräsentiert ein textbasiertes Datenaustauschformat und für den Austausch von Datenstrukturen, Objekten und strukturierten Daten in der Programmierung verwendet . 48, 49, 55, 56, 64, 74

NPM Das Programm '**N**ode **P**ackage **M**anager' bezeichnet ein Tool zum Veröffentlichen und Verwalten von Open-Source NodeJS-Projekten in der Cloud. NPM kann aber auch den NPM-Client als Befehlszeilen-Dienstprogramm meinen, dass für die Interaktion mit dem Repository zuständig ist und bei der Paketinstallation, Versionsverwaltung und Abhängigkeitsverwaltung hilft . 35, 67, 68, 74

URL Der '**U**niform **R**esource **L**ocator' verweist auf den Ort einer Webressource im Computernetzwerk oder im Internet und enthält Informationen zum Mechanismus des Abrufens der Webressource. Weil man die URL auch als eindeutigen Namen der Ressource betrachten kann, wird oft URL und URI ('**U**niform **R**esource **I**dentifier') synonym verwendet . 5, 29, 30, 36, 41, 56, 59, 61, 64, 71, 72, 75, 81

XML Die 'E**x**tensible **M**arkup **L**anguage' repräsentiert ein textbasiertes Datenformat mit einer standardisierten Auszeichnungssyntax, welche zum Austausch und Speichern von strukturierten Daten im Textformat dient . 7

Appendix: Ausgewählte Literatur für Review

[Argarwal 2021] A. Argarwal. How Micro-frontend frameworks are replacing legacy monoliths. Servian, 2021. URL: https://servian.dev/how-micro-frontend-frameworks-are-replacing-legacy-monoliths-f66f34d06a2 (besucht am 21. 10. 2021). (Siehe Seiten 42, 85)

[Ball 2019] K. Ball. Microfrontends: the good, the bad, and the ugly. Zendev, 2019. URL: https://zendev.com/2019/06/17/microfrontends-good-bad-ugly.html (besucht am 27. 10. 2021). (Siehe Seiten 22, 23)

[Biomee 2020] M. Biomee. Micro Fronends — Spotify Approach Iframes. 2020. URL: https://medium.com/@m.biomee/micro-fronends-spotify-approach-iframes-part-2-bb15c14449bf (besucht am 31. 08. 2021). (Siehe Seiten 19, 23)

[Brockmeyer 2021] J. Brockmeyer. Micro Frontends: from Fragments to Renderers (Part 1). Zalando, 2021. URL: https://engineering.zalando.com/posts/2021/03/micro-frontends-part1.html (besucht am 20. 08. 2021). (Siehe Seiten 19, 20)

[ElHousieny 2021] R. ElHousieny. Micro-Frontends: What, why, and how. März 2021. URL: https://levelup.gitconnected.com/micro-frontends-what-why-and-how-bf61f1f0a729 (besucht am 07. 08. 2021). (Siehe Seite 21)

[Entando 2020] Entando. Microfrontends. Entando, 2020. URL: https://dev.entando.org/next/tutorials/micro-frontends/ (besucht am 31. 08. 2021). (Siehe Seite 19)

[Fernando 2020] A. Fernando. How We Achieved Smooth Navigation Across Microfrontends. Bits und Pieces, 2020. URL: https://blog.bitsrc.io/how-we-achieved-smooth-navigation-across-micro-frontends-42130577924d (besucht am 27. 10. 2021). (Siehe Seite 23)

[Fleischer und Schröder 2021] A. Fleischer und H. Schröder. Frontends with Microservices – Why the choice of frontend architecture also impacts team. Otto, 2021. URL: https://www.otto.de/jobs/technology/techblog/artikel/frontends-with-microservices.php (besucht am 08. 08. 2021). (Siehe Seiten 20, 22, 36, 39, 84)

[Friedman 2020] E. Friedman. Domain-Driven Design Case study: Introducing Fiverr Logo Maker. Fiverr, 2020. URL: https://medium.com/fiverr-engineering/domain-driven-design-case-study-introducing-fiverr-logo-maker-94f0339e41aa (besucht am 22. 08. 2021). (Siehe Seiten 19, 25)

[Geers 2020] M. Geers. Micro Frontends in Action. Manning Publications, Aug. 2020. URL: https://www.manning.com/books/micro-frontends-in-action. (Siehe Seiten 21–29, 31–36, 39–42, 83–85)

[Geers 2021] M. Geers. Micro Frontends: extending the microservice idea to frontend development. Neuland, 2021. URL: https://micro-frontends.org/ (besucht am 01. 09. 2021). (Siehe Seite 30)

[Gilbert 2021] J. Gilbert. Software Architecture Patterns for Serverless Systems: Architecting for innovation with events, autonomous services, and micro frontends. Packt Publishing, 2021. URL: https://www.packtpub.com/product/software-architecture-patterns-for-serverless-systems/9781800207035. (Siehe Seiten 22, 25, 41)

[Jackson 2019] C. Jackson. Micro Frontends. Thoughtworks, 2019. URL: https://martinfowler.com/articles/micro-frontends.html (besucht am 01. 10. 2021). (Siehe Seiten 21–23, 35, 36, 83–85)

[Jackson 2020] Z. Jackson. Webpack 5 Module Federation: A game-changer in JavaScript architecture. 2020. URL: https://indepth.dev/posts/1173/webpack-5-module-federation-a-game-changer-in-javascript-architecture (besucht am 10. 08. 2021). (Siehe Seite 43)

[Krzeszowiak 2021] M. Krzeszowiak. CSS Architecture and Performance in Micro Frontends. Allegro, 2021. URL: https://blog.allegro.tech/2021/07/css-architecture-and-performance-of-micro-frontends.html (besucht am 31. 08. 2021). (Siehe Seite 19)

[Lemon 2020] S. Lemon. Problems with Micro-frontends. 2020. URL: https://medium.com/swlh/problems-with-micro-frontends-8a8fc32a7d58 (besucht am 27. 10. 2021). (Siehe Seiten 22, 23, 83, 84)

[Liebel 2022] C. Liebel. Micro-Frontends mit Web Components. 2022. URL: https://www.heise.de/developer/artikel/Micro-Frontends-mit-Web-Components-6328145.html (besucht am 25. 01. 2022). (Siehe Seite 40)

[Maric 2020] B. Maric. Our journey to microfrontends. Zeiss, 2020. URL: https://blogs.zeiss.com/tech/our-journey-to-microfrontends/ (besucht am 11. 08. 2021). (Siehe Seiten 19, 20, 22)

[Mezzalira 2019a] L. Mezzalira. Adopting a Micro-frontends architecture. DAZN, 2019. URL: https://medium.com/dazn-tech/adopting-a-micro-frontends-architecture-e283e6a3c4f3 (besucht am 31. 08. 2021). (Siehe Seiten 19, 20)

[Mezzalira 2019b] L. Mezzalira. Micro-frontends, the future of Frontend architectures. DAZN, 2019. URL: https://medium.com/dazn-tech/micro-frontends-the-future-of-frontend-architectures-5867ceded39a (besucht am 20. 08. 2021). (Siehe Seiten 21, 83)

[Mezzalira 2020] L. Mezzalira. Lessons from DAZN: Scaling Your Project with Micro-Frontends. DAZN, 2020. URL: https://www.infoq.com/presentations/dazn-microfrontend/ (besucht am 10. 10. 2021). (Siehe Seite 22)

[Mezzalira 2021] L. Mezzalira. Building Micro-Frontends: Scaling Teams and Projects Empowering Developers. O'Reilly Media, 2021. URL: https://www.oreilly.com/library/view/building-micro-frontends/9781492082989/. (Siehe Seiten 21–24, 26–28, 30–32, 35, 39, 40, 84, 85)

[Microsoft 2021] Microsoft. Redux Micro-Frontend. Microsoft, 2021. URL: https://github.com/microsoft/redux-micro-frontend (besucht am 20. 08. 2021). (Siehe Seite 19)

[MRS 2021] O. MRS. Open MRS. Open MRS, 2021. URL: https://github.com/openmrs/openmrs-esm-home (besucht am 20. 08. 2021). (Siehe Seite 19)

[Oddsson 2021] K. Oddsson. How we use Web Components at GitHub. GitHub, 2021. URL: https://github.blog/2021-05-04-how-we-use-web-components-at-github/ (besucht am 10. 10. 2021). (Siehe Seite 40)

[Peltonen u. a. 2021] S. Peltonen, L. Mezzalira und D. Taibi. Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review. *Information and Software Technology* 136 (Aug. 2021), Seite 106571. URL: https://doi.org/10.1016/j.infsof.2021.106571. (Siehe Seiten 20–23)

[Pölöskei und Bub 2021] I. Pölöskei und U. Bub. Enterprise-Level Migration to Micro Frontends in a Multi-Vendor Environment. *Acta Polytechnica Hungarica* 18.8 (2021), Seiten 7–25. URL: https://doi.org/10.12700/aph.18.8.2021.8.1. (Siehe Seiten 21, 26, 83, 84)

[Prajwal u. a. 2021] Prajwal, J. V. Parekh und R. Shettar. A Brief Review of Micro-frontends. *United International Journal for Research and Technology* 01 (2021), Seiten 123–126. URL: https://uijrt.com/articles/v2/i8/UIJRTV2I80017.pdf. (Siehe Seiten 22, 23, 83, 84)

[Rappl 2021] F. Rappl. The Art of Micro Frontends: Build websites using compositional UIs that grow naturally as your application scales. Packt Publishing, 2021. URL: https://www.packtpub.com/product/the-art-of-micro-frontends/9781800563568. (Siehe Seiten 8, 21, 23, 25–28, 35, 39–41, 45, 85)

[Röhrig 2019] N. Röhrig. Micro-Frontends im REWE Shop - Evolution eines Headers. Rewe Digital, 2019. URL: https://speakerdeck.com/drunknzombiecow/micro-frontends-im-rewe-shop-evolution-eines-headers-1 (besucht am 10. 08. 2021). (Siehe Seiten 19, 20, 22)

[SAP 2021] SAP. The Enterprise-Ready Micro Frontend Framework. SAP, 2021. URL: https://luigi-project.io/ (besucht am 20. 08. 2021). (Siehe Seite 19)

[Saring 2020] J. Saring. How We Build Micro Frontends: Building micro-frontends to speed up and scale our web development process. Bits und Pieces, Aug. 2020. URL: https://blog.bitsrc.io/how-we-build-micro-front-ends-d3eeeac0acfc (besucht am 01. 09. 2021). (Siehe Seiten 19, 20, 22, 83)

[Steyer 2020] M. Steyer. Getting Out of Version-Mismatch-Hell with Module Federation. Angular Architects, 2020. URL: https://www.angulararchitects.io/aktuelles/getting-out-of-version-mismatch-hell-with-module-federation/ (besucht am 10. 10. 2021). (Siehe Seite 45)

[Steyer 2021] M. Steyer. Pitfalls with Module Federation and Angular. Angular Architects, 2021. URL: https://www.angulararchitects.io/aktuelles/pitfalls-with-module-federation-and-angular/ (besucht am 10. 10. 2021). (Siehe Seite 45)

[Venkatesan 2021] P. Venkatesan. Scaling Applications Using Micro-Frontends: A checklist of problems that require solving while starting a microfrontend application. ThoughtWorks, 2021. URL: https://prasans.info/scaling-applications-using-microfrontends/ (besucht am 01. 09. 2021). (Siehe Seiten 23, 36, 84)

[Yang u. a. 2019] C. Yang, C. Liu und Z. Su. Research and Application of Micro Frontends. *IOP Conference Series: Materials Science and Engineering* 490 (Apr. 2019), Seite 062082. URL: https://doi.org/10.1088/1757-899x/490/6/062082. (Siehe Seiten 22, 83, 84)

Literaturverzeichnis

[Brereton u. a. 2007] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner und M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software* 80.4 (Apr. 2007), Seiten 571–583. URL: https://doi.org/10.1016/j.jss.2006.07.009. (Siehe Seite 15)

[Brito u. a. 2018] G. Brito, R. Terra und M. T. Valente. Monorepos: A Multivocal Literature Review. *ArXiv* abs/1810.09477 (2018). (Siehe Seite 63)

[Cohn 2009] M. Cohn. Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional, 2009, Seite 312. (Siehe Seite 33)

[Evans 2014] E. Evans. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley, 2014. (Siehe Seiten 25, 26)

[Farrell 2019] B. Farrell. Web Components in Action. Manning Publications, 2019. (Siehe Seiten 36–39)

[Fink und Flatow 2014] G. Fink und I. Flatow. Pro Single Page Application Development: Using Backbone.js and ASP.NET. Apress, 2014, Seite 269. (Siehe Seite 35)

[Fittkau u. a. 2017] F. Fittkau, A. Krause und W. Hasselbring. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology* 87 (Juli 2017), Seiten 259–277. URL: https://doi.org/10.1016/j.infsof.2016.07.004. (Siehe Seite 47)

[Garousi u. a. 2017] V. Garousi, M. Felderer und M. Mäntylä. Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering (Juli 2017). (Siehe Seiten 14, 17, 19)

[Garousi u. a. 2016] V. Garousi, M. Felderer und M. V. Mäntylä. The need for multivocal literature reviews in software engineering. In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, Juni 2016. URL: https://doi.org/10.1145/2915970.2916008. (Siehe Seite 14)

[Hunter II 2017] T. Hunter II. Advanced Microservices : A Hands-on Approach to Microservice Infrastructure and Tooling. Apress, 2017, Seiten 1–2. (Siehe Seite 9)

[Ingeno 2018] J. Ingeno. Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts. Packt Publishing, 2018, Seiten 213–223. (Siehe Seite 6)

[Lando 2019] B. Lando. Microservices Architekturen für Webanwendungen: Micro Frontends (Apr. 2019). (Siehe Seiten 14, 23)

[Masse 2011] M. Masse. REST API Design Rulebook. O'Reilly Media, 2011, Seiten 5–6. (Siehe Seite 50)

[El-Morabea und El-Garem 2021] K. El-Morabea und H. El-Garem. Testing Pyramid. In: *Modularizing Legacy Projects Using TDD: Test-Driven Development with XCTest for iOS*. Berkeley, CA: Apress, 2021, Seiten 65–83. URL: https://doi.org/10.1007/978-1-4842-7428-6_4. (Siehe Seite 33)

[Mu und Jiang 2011] H. Mu und S. Jiang. Design patterns in software development. In: *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. Beijing, China: IEEE, Juli 2011. (Siehe Seite 73)

[Newman 2019] S. Newman. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, 2019, Seite 6. (Siehe Seite 9)

[Owens 2020] T. Owens. Webpack 5 Up and Running: A quick and practical introduction to the JavaScript application bundler. Packt Publishing, 2020. (Siehe Seiten 43, 44)

[Petersen u. a. 2008] K. Petersen, R. Feldt, S. Mujtaba und M. Mattsson. Systematic Mapping Studies in Software Engineering. *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering* 17 (Juni 2008). (Siehe Seite 14)

[Raggett u. a. 1998] D. Raggett, A. Le Hors und I. Jacobs. HTML 4.0 Specification | W3C Recommendation. Eclipse Foundation, 1998. URL: https://www.immagic.com/eLibrary/ARCHIVES/SUPRSDED/W3C/W980424S.pdf%20https://www.w3.org/TR/1998/REC-html40-19980424/ (besucht am 15. 10. 2021). (Siehe Seite 5)

[Riehle und Gross 1998] D. Riehle und T. Gross. Role Model Based Framework Design and Integration. In: Band 33. Okt. 1998, Seiten 117–133. (Siehe Seite 7)

[Ryan 2010] J. Ryan. A History of the Internet and the Digital Future. Reaktion Books, 2010, Seiten 99–104. (Siehe Seite 5)

[Soldani u. a. 2018] J. Soldani, D. A. Tamburri und W.-J. V. D. Heuvel. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software* 146 (Dez. 2018), Seiten 215–232. URL: https://doi.org/10.1016/j.jss.2018.09.082. (Siehe Seite 10)

[Thoughtworks 2016] Thoughtworks. Thoughtworks - TECHNOLOGY RADAR. Thoughtworks. 2016. URL: https://www.thoughtworks.com/radar/techniques/micro-frontends (besucht am 01. 09. 2021). (Siehe Seiten 1, 5)

[Zirkelbach u. a. 2018] C. Zirkelbach, A. Krause und W. Hasselbring. On the Modernization of ExplorViz towards a Microservice Architecture. In: *Software Engineering*. 2018. (Siehe Seite 48)

[Zirkelbach u. a. 2019a] C. Zirkelbach, A. Krause und W. Hasselbring. On the Modularization of ExplorViz towards Collaborative Open Source Development. en. Kiel, 2019. URL: https://macau.uni-kiel.de/receive/macau_mods_00002031. (Siehe Seite 48)

[Zirkelbach u. a. 2020] C. Zirkelbach, A. Krause und W. Hasselbring. The Collaborative Modularization and Reengineering Approach CORAL for Open Source Research Software. In: 2020. (Siehe Seite 48)

[Zirkelbach u. a. 2019b] C. Zirkelbach, A. Krause-Glau und W. Hasselbring. Modularization of Research Software for Collaborative Open Source Development. In: Juli 2019. (Siehe Seite 48)