Freie Universität Berlin
Institut für Informatik
Takustr. 9
14195 Berlin

adesso SE
Prinzenstr. 34
10969 Berlin


**Bachelorarbeit**

# Evaluierung von Microfrontends: Technische Ansätze und Umsetzung in einer Beispielanwendung

Viktor Korneev
B.Sc. Informatik
Matrikel-Nr. 5092713

Berlin, 24.11.2023

Betreut von Prof. Dr. Lutz Prechelt und Prof. Dr. Claudia Müller-Birn,
sowie Daniel Dienhardt (adesso SE)

**Abstract**

Die Entwicklung von Web-Anwendungen erfolgt äußerst dynamisch. Fortlaufend entstehen neue Technologien und Ansätze, die ihre Vorgänger ablösen oder unterstützen. Diese rasante Entwicklung erfordert ständig neue Herangehensweisen, um Effizienz, Flexibilität und Benutzererfahrung zu verbessern. In der Entwicklung von Backend-Anwendungen hat sich die Microservice-Architektur als Standard etabliert, wobei immer mehr bedeutende Akteure auf diese Architektur setzen. Im Gegensatz dazu ähnelt die Frontend-Architektur oft noch einem Monolithen.

Trotz dieser Unterschiede haben sich Microfrontends in den letzten Jahren als vielversprechende Architektur erwiesen. Sie ermöglichen die Aufteilung einer Frontend-Anwendung in mehrere unabhängige Module.

Im Rahmen dieser Bachelorarbeit werden potenzielle technische Ansätze erforscht, wobei Vor- und Nachteile der einzelnen Ansätze beleuchtet werden, um den optimalen Ansatz zu ermitteln. Dieser wird später in der Arbeit an einer Beispielanwendung umgesetzt und der Umsetzungsprozess dokumentiert. Zudem werden weitere hilfreiche Tools vorgestellt.

Nach der Umsetzung erfolgt eine Evaluierung, in der festgestellt wird, ob sich die Umsetzung lohnt. Es werden Erkenntnisse gewonnen und Schlussfolgerungen gezogen.

Freie Universität Berlin

## Fachbereich Mathematik, Informatik und Physik

## SELBSTSTÄNDIGKEITSERKLÄRUNG

| | |
|---|---|
| Name: Korneev | (BITTE nur Block- oder Maschinenschrift verwenden.) |
| Vorname(n): Viktor | |
| Studiengang: B.Sc. Informatik | |
| Matr. Nr.: 5092713 | |

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Datum: 24.11.23 Unterschrift: Korneev

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

Das Web - oder World Wide Web - wurde am Anfang der 1990er Jahre erfunden und hat sich seit dem hochgradig geändert. Die Entwicklung davon kann in drei Iterationen aufgeteilt werden, wobei eine Iteration nicht vollständig die vorherige ersetzt, sondern mit ihr co-existiert.
In der ersten Iteration hatte man nur begrenzte Funktionsmöglichkeiten. Die Features waren damals hauptsächlich "read-only", d.h. es waren keine Interaktionen des Users mit der User Interface möglich [1, S. 4]. Dynamische Webseiten konnten damals nur von erfahrenen Entwicklern mithilfe von **Common Gateway Interface (CGI)** ausgeliefert werden, die meisten aber hatten entweder kein Know-How oder wollten kein Geld ausgeben, daher nutzten sie **Server-side Rendering (SSR)**. Um Inkonsistenz zu vermeiden und Wiederverwendung von Code zu gewährleisten wurde in HTML ein Tag *<frameset>* zur Verfügung gestellt, welcher es erlaubte die wiedervorkommende Komponenten wie z.b. Header, Footer etc. an mehreren Stellen einzusetzen. Er löste einige Probleme, erschuff aber dafür einige neue, wie z.B. schlechte Perfomanz und Schwierigkeiten im Umgang mit **Uniform Resource Locators (URL)** [2, S. 5].
Auf der Suche nach alternativen Möglichkeiten zur Vereinfachung der Webarchitektur vor allem mit der stets wachsenden Komplexität wurde **Server Side Includes (SSI)** eingeführt. Es handelte sich hierbei um eine Methode, nach der einige Stellen im HTML Code mit Platzhaltern besetzt waren. In diesen Platzhaltern befanden sich Server-Befehle, die zur gegebenen Zeit ausgeführt und dynamisch angepasst wurden.
Die zweite Iteration der Entwicklung des Webs war bereits nicht weit entfernt. Mit ihr wurde das Gebiet der Userinteraktionen noch einmal erweitert [1, S. 5]. Die Vorreitertechnologien waren damals PHP und JavaScript. Vorallem durch das kostengünstige PHP wurde Server-side Rendering de facto Standard und vorher erwähnte Server Side Include geriet in Vergessenheit [2, S. 5].
In der anschließenden dritten Iteration konnte man programmierbare Webanwendungen ausliefern, wodurch die Möglichkeiten an Funktionalitäten stark zunahmen [1, S. 5]. **SSR** hatte keine Priorität mehr. Um Daten dynamisch zur Laufzeit zu laden wurde **Asynchronous JavaScript**

**mit XML (AJAX)** vorgestellt, somit landete ein Teil des zu ausführenden Code beim Client. Somit begann die Aufteilung der Anwendungen in zwei Hauptbestandteile - Client-side und Server-side bzw. Frontend und Backend [2, S. 7].
Das heißt schon damals begann die Modularisierung der Anwendungen von einer vollständigen monolitischen Architektur her zu (zwei) Modulen. Diese Entwicklung, wie auch alle zuvor, brachte sowohl Vorteile als auch neue Herausforderungen mit sich. Die Komplexität der Anwendungen stieg an, die Sicherheitsvorkehrung wurden ebenso immer komplexer, da man gewährleisten musste, dass die zur Verfügung gestellten Programmierschnittstellen (**API**s) nur nach bestimmten Bedingungen Daten lieferten.

Wie man sehen kann passiert die Weiterentwicklung von Webanwendungen sehr zügig. Immer mehr große Spieler wie Netflix [3], LinkedIn [4] und Zalando [5] setzen auf die Microservice-Architektur. Obwohl die meisten Frontend-Anwendung weiterhin monolitisch sind, haben sich Microfrontends in den letzten Jahren als eine vielversprechende Architektur erwiesen, die die Aufteilung einer Frontend-Anwendung in mehrere unabhängige Module ermöglicht.

## 1.2 Zielsetzung

Diese Bachelorarbeit zielt darauf ab, die Auswirkungen von Microfrontends auf verschiedene Aspekte der Softwareentwicklung, Wartung und Benutzererfahrung zu evaluieren. Dabei sollen verschiedene Ansätze untersucht, in einem Projekt umgesetzt und das Ergebnis bewertet werden.

Zuerst sollen folgende grundlegende Fragen zu Micro-Frontend beantwortet werden:

- Was ein Microfrontend überhaupt ist und ob es unterschiedliche Variationen davon gibt?
- Welche Vor- und Nachteile haben Microfrontends im Vergleich zu einer monolithischen Architektur?
- Wann sind Microfrontends sinnvoll bzw. wovon ist die Sinnhaftigkeit abhängig - Anzahl der Entwickler im Team, Größe der Anwendung, Anzahl verschiedener Domänen in der Anwendung?

Danach geht es zur Umsetzung und mit konkreten Fragen dazu weiter:

- Welche technische Ansätze gibt es?
- Was ist beim Entwurf der Architektur zu beachten und wie könnte eine sinnvolle Architektur aussehen?

- Wie hoch ist der Aufwand der Aufteilung einer bestehenden Anwendung in Micro-Frontends? Kann hier ein geschätztes Return on Investment im Bezug auf den Aufwand der Entwicklung angegeben werden?
- Welche Herausforderungen ergeben sich bei der Migration oder aber auch bei initialen Entwicklung mit Microfrontend Architektur?
- Welchen Einfluss auf Deployment (CI/CD, d. Veröffentlichung) gibt es?
- Wie werden Daten weitergereicht? Von Microfrontend zu Microfrontend oder ausschließlich über Server-Aufrufe?
- Nach welchen Kriterien und Prinzipien soll der Schnitt erfolgen? Große/kleine Microfrontends?

## 1.3 Aufbau

Im zweiten Kapitel werden die Grundlagen von Microservice- und Microfrontend-Architektur erläutert. Das dritte Kapitel legt die Anforderungen an die Literatursuche dar, präsentiert die Ergebnisse der Recherche und stellt Bedingungen vor. Im vierten Kapitel erfolgt die Vorstellung und Analyse einer Beispielanwendung, gefolgt von der Ausarbeitung eines Ablaufplans für die Umsetzung der Migration. Das abschließende Kapitel widmet sich der tatsächlichen Migration, dokumentiert sämtliche wesentlichen Schritte sowie auftretende Probleme. Eine Auswertung verschiedener Anwendungsversionen sowie eine Zusammenfassung schließen die Arbeit ab.

In dieser Arbeit werden durchgehend englische Begriffe verwendet, da sich viele davon nur schwer ins Deutsche übersetzen lassen und im deutschsprachigen Raum in der Praxis gebräuchlich sind. Begriffe, die über den angenommenen Kenntnisstand der Leser hinausgehen könnten, werden kurz erläutert.

Abkürzungen werden beim erstmaligen Vorkommen ausgeschrieben und im weiteren Verlauf der Arbeit in Kurzform verwendet.

Der wichtigste Begriff dieser Arbeit, *Microfrontend(s)*, kann und wird auf verschiedene Weisen geschrieben. Zur Erleichterung der Lesung wird die genannte Form verwendet.

Des Weiteren wird häufig der Begriff 'Modul' (englisch 'module') verwendet. Um Konsistenz zu gewährleisten, wird die englische Variante benutzt, da diese auch im Code verwendet wird. Dabei wird für die Leserlichkeit auch die deutsche Schreibweise beibehalten: das Module (die Modules).

# 2 Grundlegende Aspekte von Micro-Frontends und Microservice-Architektur

## 2.1 Entstehung

In der Einleitung wurde der Tag *<frameset>* präsentiert, der es ermöglichte bestimmte Komponenten wieder zu verwenden. In der Tat kann dieser Tag als eine frühe Version von Microfrontends betrachtet werden [6].
Um die Entstehung von Microfrontends besser verstehen zu können, schauen wir auf den Ursprung der Microservice-Architektur.

## 2.2 Was ist eine Microservice-Architektur?

Microservices sind unabhängige Services, die separat veröffentlicht werden können und die nach Business-Domäne deutlich strukturiert sind [7, Kapitel 1] [8]. Eine Microservice-Architektur ist dementsprechend ein architektur-technischer Ansatz, bei dem die ganze Backend-Anwendung in solche Services aufgeteilt wird.

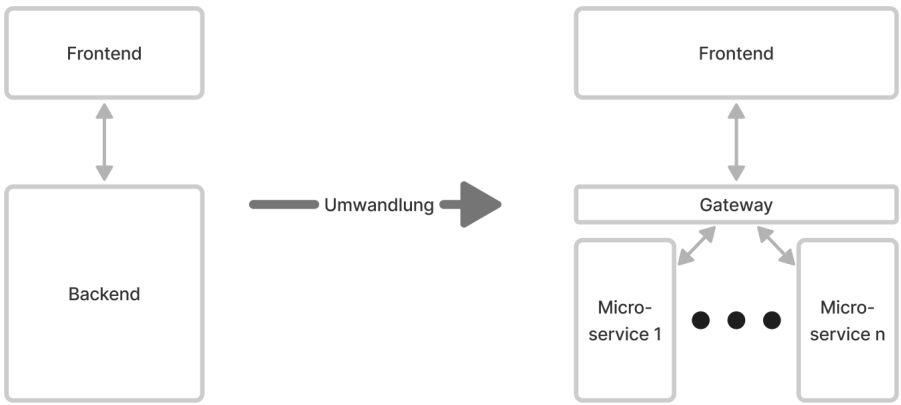


**Abb. 2.1:** Umwandlung in Microservices.

Ein Service umfasst die Funktionalität einer bestimmten Domäne (bspw. Warenkorb, Produktsuche, Bezahlung) oder auch einer bestimmten Aufgabe (bspw. Authentifizierung, Registrieren anderer Services) und kommuniziert mit anderen Services mit Hilfe von Netzwerk. Alle Services zusammen stellen somit eine komplette Anwendung dar [7, Kapitel 1]. In einer sogenannten Systematic Mapping Study [9] haben D.Taibi et al. 42 Studien zu den Architekturprinzipien analysiert, um daraus Vorteile, Nachteile und Herausforderungen zu entnehmen und zusammenzufassen. Daraus ergaben sich unter anderem folgende Vorteile:

- *Bessere Wartung*. In allen analysierten Papers wurden Vorteile beim Warten bestehender Anwendung hervorgehoben.
- *Unterschiedliche Sprachen*. Durch die Aufteilung des Backends in verschiedene Services, lassen sich diese mit unterschiedlichen Sprachen implementieren.
- *Wiederverwendung*. Ein Service kann an unendlich vielen anderen Stellen eingesetzt und eine Änderung muss dementsprechend auch nur einmalig vorgenommen werden.
- *Einfacheres Deployment*. Die Services können auch einzeln deployed werden, d.h. wenn die Code-Basis in einem Service geändert wurde, können alle anderen Service weiterhin am Laufen bleiben und es muss nur ein Service neu veröffentlicht werden. Das vereinfacht den Continious Integration Prozess und reduziert in der Regel eine mögliche Downtime.

Allerdings wurde in einigen Studien auch auf folgende Nachteile einer Microservice-Architektur hingewiesen:

- *Komplexität des Testens*. Viele Komponenten und Muster deren Zusammenarbeit erschweren oft das Testen.
- *Komplexität des Implementierens*. Das Implementieren von neuen Microservices ist in der Regel schwerer als die Erweiterung eines Monolithen um gewünschte Funktionalität.
- *Netzwerk Probleme*. Da die einzelnen Microservices unter einander über das Netzwerk kommunizieren, können es auch hier typische Probleme für die Netzkommunikation ergeben:
  - Durch *Latenz* kann die Kommunikationszeit zwischen den Services zunehmen.
  - *Bandbreite* hat ebenfalls einen Einfluss auf die Kommunikation zwischen den Services und kann diese einschränken, demnach muss die Bandbreite bei der Implementierung beachtet werden sowie für normale als auch für hohe Datenübertragungsrate.

- *Größerer Speicherverbrauch.* Falls jeder Microservice in einer separaten Virtual Machine läuft, ergibt sich größerer Speicherverbrauch.
- *Authorisierung.* Jeder Microservice stellt Schnittstellen bereit, über die mit anderen Services und/oder mit Frontend kommuniziert wird. Dadurch entsteht höhere Komplexität bei der Autorisierung, da die Services in der Regel untereinander verflochten sind und es an jedem Eingangsknoten sichergestellt werden muss, dass der Aufruf legitim ist.
- *Anforderung an Automatisierung.* Im Entwicklerteam sollen auch DevOps Kenntnisse vorhanden sein, um Microservicearchitektur sinnvoll nutzen zu können. Ansonsten bringt es noch mehr Probleme mit sich.

Seitdem die Aufteilung der Anwendungen in Frontend und Backend bzw. Client- und Serverside sich etabliert hat, eröffneten sich immer mehr Möglichkeiten im Bezug darauf, was man mit solch einer Anwendung machen kann. Im Laufe der letzten Jahrzehnten ähnelten sich Web-Anwendungen immer mehr den üblichen Desktop-Anwendungen, die sonst auf dem Rechner installiert sein müssten [2, S.7]. Das führte hingegen dazu, dass die Größe und die Komplexität der Web-Anwendungen enorm anstieg. Man konnte nun die Entwicklung von Frontend und Backend von einander entkoppeln, sodass beispielsweise ein Team sich komplett mit dem Frontend beschäftigt und das andere Team mit der Entwicklung des Backends. Vorteilhaft dabei ist vor allem, dass die User Experience viel besser wurde, andererseits wurde die Logik immer komplexer und bei großen, stetig wachsenden Projekten führte dies fast unvermeidbar zu Problemen. Eine solche Frontend-Anwendung war aufwendig zu skalieren, zu warten und weiterzuentwickeln [10].

## 2.3 Was ist eine Microfrontend-Architektur?

Laut [11] ist Microfrontend-Architektur ein Architekturstil, in dem mehrere kleine Frontend-Anwendungen in eine große ganze Anwendung zusammengesetzt werden. Die Idee dabei ist, dass das große Ganze als Kombination von kleinen Teilen angesehen wird, für die jeweils unterschiedliche Teams zuständig sind. Jedes Team hat dabei unabhängige von anderen Aufgaben, auf die sie sich fokussieren [12]. Wenn die Entwickler einen Monolithen haben, wird sehr viel Code geschrieben, der harmonisch funktioniert soll. Das führt dazu, dass die Entwickler die Softwarelogik überentwickeln können, und dies kann auf lange Sicht nur schwer gewartet werden [13, S.9]. Durch die Entstehung von immer mehr Bugs und durch immer komplexeren und unverständlichen Code, verzichten Firmen mit großen Monolithen darauf, viel in die neuen Features zu investieren [13, S.9-10].

Die Entstehung von Microfrontends wird maßgeblich durch die kontinuierlich wachsende Berechnungskapazität und -geschwindigkeit der Hardware sowie die hohen Erwartungen der Nutzer beeinflusst. Dieser Aspekt gewinnt zunehmend an Bedeutung, da die technologischen Fortschritte es ermöglichen, immer komplexere und reaktionsschnellere Benutzeroberflächen zu entwickeln, die den anspruchsvollen Anforderungen der modernen Nutzer gerecht werden.

Luca Mezzalira definiert in [13] folgende vier Prinzipien von Microfrontends:

- Autonomous database
- Business domain representation
- Single-team ownership
- Independent deployment

Eine Ähnlichkeit oder Überschneidung mit Microservice-Architektur ist erkennbar.
Es gibt folgende Arten von Micro Frontends [13, S.3-8]:

- Single Page Application (SPA)
  - SPAs sind Webanwendungen, die sich auf einer einzigen HTML-Seite befinden und AJAX verwenden, um dynamische Inhalte zu laden, anstatt separate Seiten vom Server abzurufen.
  - Bei SPA Micro Frontends wird die Anwendung in verschiedene Modules oder Komponenten aufgeteilt, die unabhängig voneinander entwickelt und ausgeliefert werden

können. Diese Modules können in einer gemeinsamen Container-SPA eingebettet werden.

  - Diese Architektur ist gut geeignet, wenn Sie eine reichhaltige, interaktive Benutzeroberfläche erstellen und die Vorteile von SPAs nutzen möchten.

- Isomorphic
  - Isomorphe Micro Frontends kombinieren serverseitiges Rendering (SSR) mit clientseitigem Rendering. Dies bedeutet, dass sowohl auf dem Server als auch im Browser gerenderte Versionen der Benutzeroberfläche vorhanden sind.
  - Diese Architektur bietet eine bessere Leistung und Suchmaschinenfreundlichkeit als herkömmliche SPAs, da Inhalte beim ersten Laden schneller angezeigt werden.
  - Isomorphe Micro Frontends sind nützlich, wenn Sie die Vorteile von SPAs nutzen möchten, aber auch auf eine bessere SEO und Ladegeschwindigkeit achten müssen.

- Static Pages
  - Bei dieser Art von Micro Frontends werden die Frontend-Komponenten in einzelne statische HTML-Seiten aufgeteilt, die unabhängig voneinander ausgeliefert werden können.
  - Dies ist besonders nützlich für Inhaltsseiten, die selten aktualisiert werden und hauptsächlich aus Text und Multimedia bestehen. Sie können von einem Content-Management-System (CMS) generiert und als eigenständige Mikro-Frontends bereitgestellt werden.
  - Diese Architektur kann die Wartbarkeit und Skalierbarkeit von Inhaltsseiten verbessern.

- Jamstack
  - Jamstack ist eine moderne Webentwicklungsarchitektur, die auf JavaScript, APIs und Markup (daher der Name Jamstack) basiert.
  - Bei Jamstack Micro Frontends werden verschiedene Frontend-Komponenten oder -Services als eigenständige Module entwickelt und über APIs integriert. Die Benutzeroberfläche wird statisch generiert und über Content Delivery Networks (CDNs) ausgeliefert.
  - Dies ermöglicht eine hohe Leistung, Skalierbarkeit und Sicherheit und ist gut geeignet für Anwendungen mit vielen externen Services und Inhalten.

Im nächsten Kapitel wird genauer auf die Microfrontends eingegangen.

# 3 Literaturrecherche

## 3.1 Gestaltung der Suche

### 3.1.1 Wie wird gesucht?

Es werden Sachbücher und wissenschaftliche Publikationen untersucht. Außerdem die sogenannte “graue Literatur”, d.h. Blog-Posts bzw. Netzartikel. Solche Quellen sind daher grau, weil sie bereits in Kürze nicht mehr verfügbar sein könnten und deren Inhalt von Autoren geändert werden kann ohne jegliche Kennzeichnung und die haben öfters keine Quellenangaben. Nicht desto trotz können sie wichtige Hinweise und Erkenntnisse liefern, da sie sehr nah den aktuellen Entwicklungsstand wiederspiegeln.

Da diese Arbeit vor allem das Ziel verfolgt die praktischen Aspekte und Fragestellungen zu untersuchen, werden die Quellen vor allem für die Eigenbereicherung auf dem Gebiet der Web-Entwicklung und Darstellung bzw. Wiedergeben der Feststellungen benutzt, aber nicht für eine systematische wissenschaftliche Analyse, um zu einem wissenschaftlichen Konsens zu kommen. Nach der durchgeführten Umsetzung an einer eigenen Beispielanwendung sollen die Feststellung aus der Literaturrecherche bewertet und ergänzt werden.

Suche nach den Quellen:

Für die Literatursuche werden hauptsächlich Google Suche und Google Scholar verwendet. Es werden Quellen aus bekanntesten akademischen Verlagen ausgewählt wie zum Beispiel Research Gate, Science Direct, ACM, IOP Science und anderen.

Sobald es mehr in Richtung Umsetzung geht, werden immer häufiger einfachere Quellen in Anspruch genommen, wie zum Beispiel Youtube und verschiedene Foren (StackOverflow, GitHub, etc.).

### 3.1.2 Was wird gesucht?

- Quellen aus den letzten fünf Jahren. Hauptsächlich, dennoch für Erläuterung der Geschichte der Entwicklung in Einzelfällen auch ältere.

- Quellen zur Diskussion über Microfrontends.
- Quellen, die dabei helfen sollen, eine Strategie zur Migration zu Microfrontends oder Ansätze zum Aufbau vom Microfrontends zu entwickeln.
- Quellen, die auf Vor- und Nachteile von Microfrontends eingehen.
- Quellen, in denen persönliche Eindrücke und Erfahrung in Arbeit mit Microfrontends erläutert werden.

Art der Literatur:

- Bücher.
- wissenschaftliche Arbeiten ( Masterarbeiten, Papers oder Ähnliches).
- Netzartikel.
- Videos.
- Diskussionen zum Thema in verschiedenen Communities wie z.B. Stack Overflow oder GitHub.
- Sprachen: Deutsch, Englisch.

## 3.2 Ergebnisse der Literaturrecherche

Ich möchte diesen Abschnitt mit einem Zitat beginnen, das meiner Meinung nach das Problem perfekt widerspiegelt, das durch Microarchitektur gelöst wird:

> "...working with a handful of people on a new project feels fantastic. Every developer has an overview of all functionality... This changes when the project's scope and the team size increases. Suddenly one developer can't know every edge of the system anymore...Discussions inside the team are more cumbersome. Before, team members made decisions at the coffee machine. Now you need formal meetings to get everyone on the same page... **At some point, adding new developers to a team does not increase productivity**"[14, S. 3].

In diesem Abschnitt werden die durch die Literaturrecherche gewonnenen Einblicke und Erkenntnisse zusammengeführt und analysiert.

### 3.2.1 Team als Grundbaustein

#### Aufteilung Team

Jedes Team hat eine Mission oder eine Domäne, mit der es sich beschäftigt. Beispielsweise:

- Team Inspire - hilft eine passende Ware zu finden.
- Team Decide - hilft eine Kaufentscheidung zu treffen.
- Team Checkout - leitet durch den Bezahlvorgang.

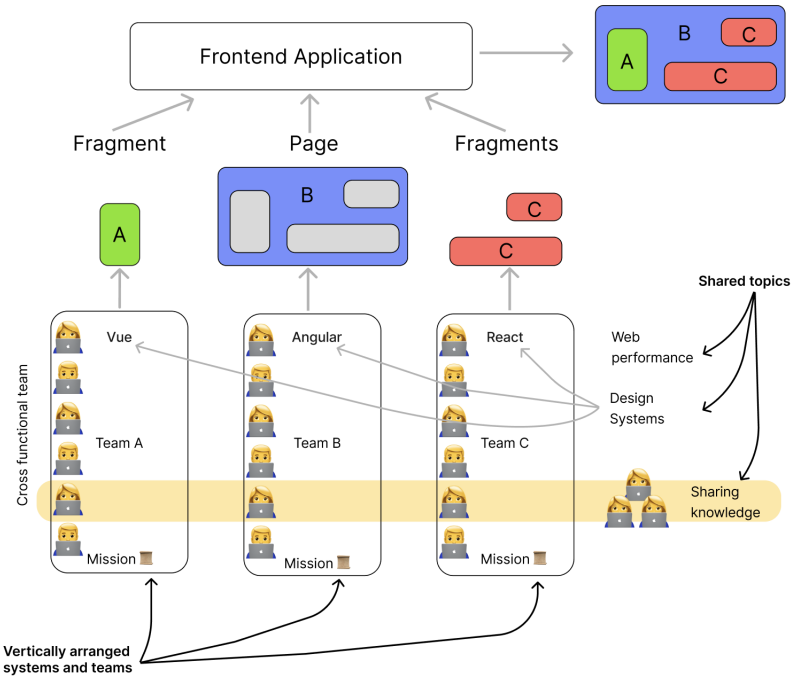


**Abb. 3.1:** Microfrontends und Organisation [14, S. 5].

Ein wesentlicher Unterschied zwischen Microfrontends und anderen Architekturansätzen liegt in der Teamstruktur. Grundsätzlich kann zwischen Spezialistenteams und cross-funktionalen Teams unterschieden werden. Bei der ersten Variante werden alle Mitarbeiter nach ihrer Funktion gruppiert, sodass beispielsweise Frontendentwickler gemeinsam arbeiten, Business Analysten gemeinsam tätig sind usw.. Die Spezialisten auf ihrem Gebiet streben nach Perfektion und der besten Lösung in ihrem Bereich. Daher kann die Zusammenarbeit mit anderen Spezialisten dabei hilfreich sein [14, S. 6-7].

Im Gegensatz dazu stehen die cross-funktionalen Teams, die besonders bei einer Microfrontend-Architektur zum Einsatz kommen. Solche Teams bestehen aus Spezialisten aus verschiedenen Gebieten – von Frontend bis Backend, einschließlich Anforderungsingenieuren. Die Ergebnisse eines solchen Teams mögen nicht unbedingt die höchste Ingenieurskunst aufweisen, da die Teams kleiner sind. Aber, wie in [14, S. 6-7] betont wird, hat dies einen enormen Vorteil: jedes Teammitglied identifiziert sich stärker mit dem Produkt.

Es existiert ebenfalls vordefinierte Kommunikation zwischen den Teams, die sich auf verschiedene gemeinsame Probleme und Themen bezieht. Hierzu können beispielsweise das Design oder auch die Authentifizierung gehören.

#### Fallbeispiel: neues Feature

In einer Anwendung mit einem System aus Spezialistenteams stellt sich die Frage: Wie kann ein neues Feature implementiert werden?

Die Aufgabe muss alle Teams durchlaufen, angefangen bei den Anforderungsingenieuren bis hin zu den DevOps. Dies führt dazu, dass viele Personen an jeder Kleinigkeit mitarbeiten, ohne sich dabei auf ein spezifisches Gebiet zu konzentrieren.

Abgesehen davon, dass jedes solche Feature mehr Aufwand erfordert, leidet auch die menschliche Psyche, da das Multitasking zwischen verschiedenen Aufgaben und Anforderungen zu mehr Stress führt und die Produktivität senkt [15].

#### Ziele des Teams

Während Microfrontend-Architektur ein Architekturstil ist[11], ist ein Microfrontend ein Module, welches eine Seite oder auch ein Fragment einer Seite darstellt. Ein Fragment ist hier praktisch eine eingebettete Mini-Anwendung, die isoliert ist von dem Rest der Seite [14, S. 3-4].

Die Aufgabe eines einzelnen Teams ist entsprechend ihrer Mission ein Module zur Verfügung zu stellen, d.h. HTML-, CSS- und JavaScript-Dateien, die für bestimmte Features notwendig sind. Unterschiedliche Teams sollten dabei in der Theorie keine Libraries und keinen Code teilen, in der Praxis ist schwer umzusetzen.

### 3.2.2 Vorteile von Microfrontends

Viele Vorteile von Microfrontends ähneln sich den von Microservices, weil die grundlegende Idee ähnlich ist. Es ergeben sich folgende Vorteile:

### Deployment

- Unabhängiges Deployment
  Microfrontends ermöglichen es, einzelne Teile einer Anwendung unabhängig voneinander zu aktualisieren und bereitzustellen [2, S. 18]. Dies reduziert das Risiko von Ausfallzeiten und ermöglicht es, neue Funktionen oder Fehlerkorrekturen schneller auf den Markt zu bringen.

- Skalierbarkeit
  Jedes Microfrontend kann individuell skaliert werden, je nach Bedarf. Dies ist besonders nützlich, wenn bestimmte Teile der Anwendung eine höhere Last erzeugen als andere. Es verhindert, dass Ressourcen verschwendet werden [14, S. 17] [16].

### Fehlersuche

- Isolierung von Fehlern
  Bei Microfrontends ist es einfacher, Fehler einzugrenzen und zu isolieren, da sie unabhängig voneinander funktionieren. Wenn ein Fehler in einem Microfrontend auftritt, beeinflusst dies nicht zwangsläufig den gesamten Anwendungsstack [16].

- Rollback-Fähigkeit
  Falls ein Update eines Microfrontends unerwartete Probleme verursacht, ist es oft einfacher, auf die vorherige Version zurückzukehren, da andere Teile der Anwendung davon unberührt bleiben.

### Verständnis & Einarbeitung

- Klare Abgrenzung der Verantwortlichkeiten
  Microfrontends ermöglichen es, die Verantwortlichkeiten für verschiedene Teile der Benutzeroberfläche klar zu definieren. Dies erleichtert die Einarbeitung neuer Entwickler, da sie sich auf spezifische Teile der Anwendung konzentrieren können, ohne den gesamten Anwendungscode verstehen zu müssen.

- Technologieunabhängigkeit
  Jedes Microfrontend kann in unterschiedlichen Technologien entwickelt werden, was Teams die Flexibilität gibt, die für ihre Aufgaben am besten geeigneten Technologien zu wäh-

len. Dies bedeutet auch, dass Entwickler mit unterschiedlichen Fachkenntnissen effektiv zusammenarbeiten können.

#### Überarbeitung & Updates

Herausforderung des 'Feature Drucks' bei Updates effektiv bewältigen. Durch die Möglichkeit, Teile der Benutzeroberfläche unabhängig voneinander zu aktualisieren, können Entwickler Updates und neue Funktionen in einem gut abgestimmten Tempo bereitstellen, ohne dabei die Gesamtfunktionalität der Anwendung zu gefährden. Dies schafft Raum für eine flexiblere und stärker kontrollierte Verwaltung von neuen Features, während gleichzeitig die Stabilität der Anwendung gewährleistet bleibt. Damit können Entwicklerteams den Anforderungen der Benutzer und des Marktes gerecht werden, ohne sich in einem Teufelskreis aus unerwünschtem "Feature Druckßu verfangen.

- Gezielte Aktualisierungen
  Mit Microfrontends können Updates gezielt und unabhängig voneinander und Überarbeitungen in verschiedenen Teilen der Anwendung vorgenommen werden. Dies ermöglicht es, ältere oder veraltete Teile der Benutzeroberfläche zu modernisieren, ohne die gesamte Anwendung zu beeinträchtigen.
  - **GitHub** hat mehrere Jahre gebraucht, um die Abhängigkeit von *jQuery* zu entfernen [17].
  - Im Projekt, welches im nächsten Kapitel präsentiert wird, haben wir mehr als drei Sprints (je drei Wochen) gebraucht, um eine externe UI Bibliothek zu aktualisieren. Bei Microfrontends lässt sich jeder Teil separat aktualisieren und somit iterativ in Schritten bewältigen.
- Kontinuierliche Verbesserung
  Teams können kontinuierlich an der Verbesserung und Optimierung einzelner Microfrontends arbeiten, ohne den gesamten Anwendungscode überarbeiten zu müssen. Dies erleichtert die Anpassung an sich ändernde Anforderungen und die Reaktion auf das Feedback der Benutzer.

#### Neues ausprobieren

- Technologie-Experimente
  Microfrontends erlauben es Teams, neue Technologien und Frameworks auszuprobieren,

ohne die gesamte Anwendung aufs Spiel zu setzen. Dies fördert die Innovationsfreudigkeit und ermöglicht es, auf aktuelle Trends und Best Practices in der Webentwicklung zu reagieren [16].

- A/B-Tests und Prototyping
  Durch das Erstellen von neuen Microfrontends für Experimente und Prototypen können Entwickler verschiedene Ansätze testen, um herauszufinden, welche Benutzererfahrung am besten funktioniert. Dies minimiert das Risiko von Fehlinvestitionen in ungetestete Ideen [2, S. 18-19].

### 3.2.3 Nachteile von Microfrontends

Es ist äußerst selten, dass Vorteile ohne begleitende Nachteile auftreten, und die Verwendung der Microfrontend-Architektur ist tatsächlich mit einigen dieser Nachteile verbunden. Wenn Microfrontends in einem Projekt eingesetzt werden, besteht die Gefahr, dass sie mehr Nachteile als Vorteile mit sich bringen, insbesondere wenn sie ohne genaue Überlegung und nur aufgrund eines Trends anstatt aus funktionalen Gründen verwendet werden. Daher ist es von entscheidender Bedeutung, sorgfältig abzuwägen, ob der Einsatz von Microfrontends in einem bestimmten Kontext sinnvoll ist [13].

#### Komplexität der Koordination

Die Aufteilung der Benutzeroberfläche in unabhängige Microfrontends führt zu erhöhter Komplexität bei der Koordination zwischen den verschiedenen Teilen. Das Management von Abhängigkeiten und Kommunikation zwischen den Microfrontends kann kompliziert werden [16] [18].

#### Zustandsverwaltung (Statemanagement)

Die Zustandsverwaltung kann potenziell die schwierigste Herausforderung werden, da ein nicht zwischen verschiedenen Microfrontends synchronisierter Zustand kann in schlimmsten Fällen zu Sicherheits- und Datenschutzproblemen führen [18].

#### Initialer Implementierungsaufwand

Die Einführung von Microfrontends erfordert in der Regel eine umfangreiche Überarbeitung der vorhandenen monolithischen Anwendung oder eine gezielte Planung von Anfang an. Dies kann zu einem erheblichen anfänglichen Implementierungsaufwand führen.

#### Redundanz

In der Softwareentwicklung ist einer der Grundprinzipien der Arbeit, dass man möglich auf redundanten Code verzichtet, damit beispielsweise wenn ein Bug auftritt, dieser nur an einer Stelle behoben werden muss [16].

Durch die Benutzung von Microfrontends durch mehrere Teams entsteht Code-Redundanz und sie kann zu einem Problem werden [14, S. 17].

#### Konsistenz

Damit die einzelnen Microfrontends komplett unabhängig werden, sollten deren Datenbanken ebenfalls unabhängig sein. Öfters müssen aber die Daten zwischen mehreren Microfrontends geteilt werden. Dies kann dazu führen, dass eigentlich gleiche Daten in verschiedenen Systemen unterschiedlichen Stand aufweisen könnten [14, S. 18]

### 3.2.4 Entscheidungsrahmen

Es gibt unterschiedliche Architekturansätze für die Microfrontends und um den besten für jeweiliges Projekt zu wählen.

### 3.2.5 Technische Ansätze

#### Frontend Integration: Wie kommen alle Microfrontends zusammen, ob ganze Seiten oder auch Fragmenten?

#### Routing

Beim Routing geht es um die Integration auf dem Seitenniveau, d.h. Fragmente werden hier nicht betrachtet. Mittels eines HTML Links kann von Seite A zu Seite B weitergeleitet werden. Das ist an sich nicht spektakulär und benötigt keine Microfrontend-Ansätze. So funktionieren praktisch die meisten Verlinkungen im Internet. Interessant hingegen wird es, wenn man eine gute Benutzererfahrung bieten möchte, in dem eine Client-side Navigation bereit gestellt wird, welche die nächste Seite rendert, ohne diese immer neu laden zu müssen [14]. Das kann durch **Shared Application Shell** oder durch das Benutzen von **Single-SPA** erreicht werden.

### Komposition

Auf den Seiten, die Fragmente enthalten werden in der Regel Platzhalter anstelle von diesen Fragmenten platziert. Bei der Komposition werden die dann die Platzhalter durch tatsächliche Komponenten ersetzt. Dieser Prozess kann normalerweise auf zwei Arten passieren ( einzeln oder beide in Kombination miteinander).

**Server-side Komposition** Server-side Komposition beschreibt den Ansatz, nach dem alle Bestandteile der Anwendung auf dem Server zusammengesetzt und anschließend an den Client geschickt werden.

Es bringt folgende **Vorteile** mit sich:

- *SEO-Freundlichkeit*: Serverseitige Komposition ermöglicht es, vollständig gerenderte HTML-Seiten an Suchmaschinen zu senden, was die Indexierung und das Ranking in Suchergebnissen verbessert. Dies ist aufgrund der bereits gerenderten Seiten und der Verwendung von traditionellem HTML einfacher [14, S. 48, 82–83].
- *Performance*: Durch die serverseitige Komposition können Sie den Großteil der Arbeit auf dem Server erledigen, was zu schnelleren Ladezeiten und besserer Leistung führen kann, da weniger Rechenleistung und Ressourcen auf dem Client erforderlich sind [14, S. 82-83].
- *Sicherheit*: Serverseitige Komposition ermöglicht eine bessere Kontrolle über die Sicherheit, da kritischer Code auf dem Server ausgeführt wird und weniger Angriffspunkte für Clientseitige Angriffe vorhanden sind.
- *Konsistenz der Benutzererfahrung*: Die Benutzer erleben konsistentes Verhalten, da die Seite beim Laden bereits vollständig gerendert ist und es keine sichtbaren Verschiebungen oder Verzögerungen gibt. Allerdings gibt dazu einen Nachteil, der im weiteren Verlauf erläutert wird.

Nicht desto trotz ergeben sich auch einige **Herausforderungen**:

- *Komplexität auf Serverseite*: Die serverseitige Komposition erfordert zusätzliche Serverressourcen und kann die Serverinfrastruktur komplexer machen [19].
- *Eingeschränkte Interaktivität*: Da die meisten Aktionen auf dem Server verarbeitet werden, kann dies die Interaktivität der Anwendung einschränken und dazu führen, dass bestimmte Teile der Benutzeroberfläche nicht so reaktionsfähig sind wie bei clientseitigen Ansätzen [14, S. 83].

In der nächsten Abbildung sind die Vorteile von Server-side Komposition im Verhältnis zu Client-side Komposition. Es ist klar zu erkennen, dass obwohl die technischen Herausforderung steigern, die User Experience deutlich verbessert werden kann.

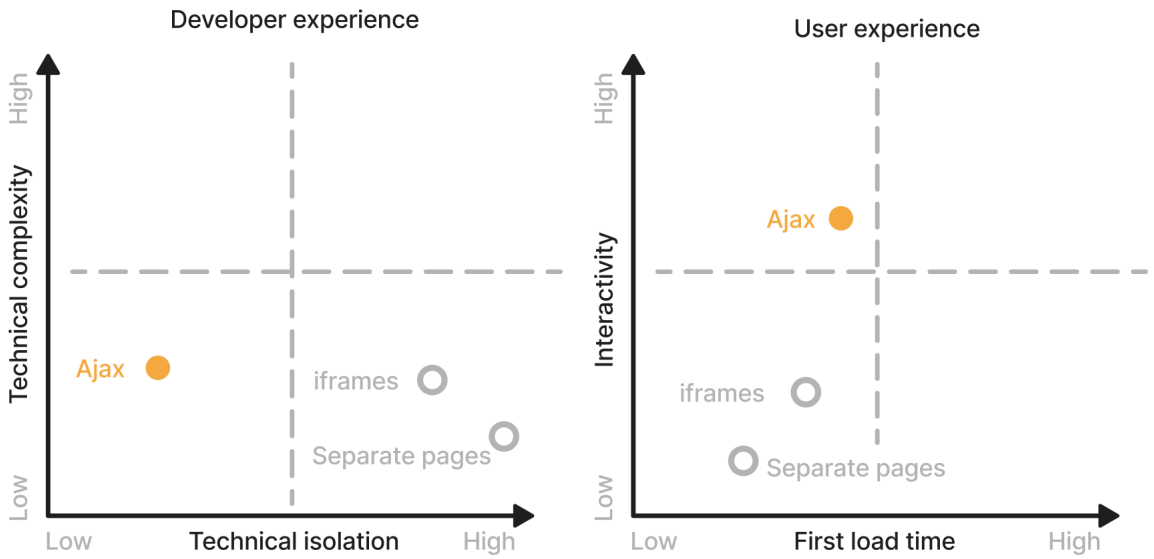


**Abb. 3.2:** Vergleich von Server-side composition zu Methoden von Client-side composition [14, S. 84]

Folgende technische Ansätze sind die gängigsten:

- **Server-Side Includes (SSI)**: SSI ist ein serverseitiger Ansatz, bei dem verschiedene Teile einer Webseite auf dem Server kombiniert werden, bevor sie an den Client gesendet werden. SSI-Tags ermöglichen das Einbetten von Inhalten aus verschiedenen Quellen in eine HTML-Seite. Dies ermöglicht die serverseitige Komposition von Micro Frontends [20].
- **Edge Side Includes (ESI)**: ESI ist eine Erweiterung von SSI, die häufig in Content Delivery Networks (CDNs) verwendet wird. ESI ermöglicht das Caching und die serverseitige Komposition von Inhalten auf Edge-Servern, um die Ladezeiten zu optimieren [21].
- **Podium**: Podium ist ein Framework, das von der Zalando-Gruppe entwickelt wurde und die serverseitige Komposition von Micro Frontends ermöglicht. Es ermöglicht das Rendering von Micro Frontends auf dem Server und das Zusammenführen der Ausgabe, bevor sie an den Client gesendet wird [14, S. 75-77].

- **Tailor**: Tailor ist ein weiteres serverseitiges Rendering-Framework, das von Walmart entwickelt wurde. Es ermöglicht die serverseitige Komposition von Micro Frontends und die Auslieferung optimierter, zusammengesetzter Seiten an den Client [14, S. 73-75].

**Client-side Komposition** Aktuell ist Client-side Komposition de facto Standdard in modernen Web-Anwendungen. Dabei werden alle Bestandteile der Anwendung im Browser des Client zusammengesetzt.

Folge **Vorteile** ergeben:

- *Hohe Flexibilität*: Clientseitige Komposition ermöglicht eine hohe Flexibilität bei der Entwicklung und Integration von Micro Frontends. Verschiedene Teams können unabhängig voneinander an Teilen der Anwendung arbeiten.
- *Interaktivität*: Die clientseitige Komposition ermöglicht eine hohe Interaktivität und dynamische Benutzererlebnisse, da die Verarbeitung auf dem Client stattfindet.

Mehrere **Herausforderung** werden aber erkannt:

- *SEO-Herausforderungen.* Clientseitige Komposition kann SEO-Herausforderungen mit sich bringen, da die Seiten oft dynamisch auf dem Client gerendert werden und Suchmaschinen möglicherweise Schwierigkeiten haben, den Inhalt zu indexieren [14].
- *Initiale Ladezeit.* Clientseitige Komposition kann längere initiale Ladezeiten aufweisen, da der Client möglicherweise mehr Arbeit leisten muss, um die Seite zu rendern, insbesondere wenn viele Komponenten geladen werden müssen.

Aktuel sind folgende Ansätze die gängigsten zur Implementierung von Client-side Komposition:

- **Ajax (Asynchronous JavaScript and XML)**: Ajax ermöglicht das Abrufen von Daten oder Inhalten von einem Server und das dynamische Aktualisieren einer Seite, ohne die gesamte Seite neu zu laden. Dies wird oft in Kombination mit clientseitigen Frameworks und Bibliotheken wie React oder Vue.js verwendet, um Microfrontends clientseitig zu komponieren [14, S. 42-50].
- **iframes**: iframes sind HTML-Elemente, die es ermöglichen, eine separate HTML-Seite in ein anderes Dokument einzubetten. iframes können zur clientseitigen Komposition von Micro Frontends verwendet werden, indem jede Komponente in einem iframe geladen wird. Dies ermöglicht eine gewisse Isolation, birgt jedoch auch Herausforderungen in Bezug auf die Kommunikation zwischen den Komponenten [22].

- **Web Components**: Web Components sind eine Webstandard-Technologie, die es ermöglicht, wiederverwendbare und eigenständige Komponenten zu erstellen. Sie können verwendet werden, um Microfrontends clientseitig zu komponieren, da sie unabhängig entwickelte Komponenten in einer Anwendung integrieren können.Client-Side Routing: In Kombination mit clientseitigem Routing können Micro Frontends in getrennten JavaScript-Dateien entwickelt und bei Bedarf clientseitig geladen und gerendert werden. Dies ermöglicht eine schnelle Navigation zwischen verschiedenen Teilen einer Anwendung [14, S. 92-98].
- **Client-Side Routing**: In Kombination mit clientseitigem Routing können Micro Frontends in getrennten JavaScript-Dateien entwickelt und bei Bedarf clientseitig geladen und gerendert werden. Dies ermöglicht eine schnelle Navigation zwischen verschiedenen Teilen einer Anwendung [14, S. 120-122].

### Kommunikation

In der Theorie sollen Microfrontends nicht untereinander kommunizieren [14, S. 17], allerdings erweist sich das in der Praxis sehr schwer. Für die doch benötigte Kommunikation zwischen einzelnen Microfrontends bietet sich ein *shared* Module, dessen Instanz in die jeweiligen Microfrontends injiziert wird.

### Shared Topics

Der Microfrontend-Ansatz zielt darauf ab, kompakte und effiziente Teams zu formen, die tiefgreifende Kenntnisse ihres Produkts besitzen. Dennoch ist es nicht ausgeschlossen, dass selbst diese Teams miteinander kommunizieren müssen, insbesondere wenn es um Aspekte wie Design oder Performance geht. Die Zusammenarbeit und der Austausch zwischen den Teams sind entscheidend, um das bestmögliche Ergebnis zu erzielen. Besonders relevant wird die Frage, wie man mit Teilen der Anwendung umgeht, die über Team- oder Domänengrenzen hinweggehen. Wie sollte man beispielsweise mit einem Modell (oder einer Klasse) verfahren, das in mehreren Microfrontends verwendet wird? Für die Beantwortung dieser Frage eignet sich gut Domain Driven Design [23]. Dabei werden zwei grundsätzliche Arten von Strategien dargestellt: *Strategic Domain-Driven Design* und *Strategic Domain-Driven Design Strategies*.

Strategic Domain-Driven Design (DDD) betont ein gemeinsames Verständnis der Geschäftsdomäne und fördert die Zusammenarbeit zwischen Teams. Durch die Organisation von Software um begrenzte Kontexte können Abhängigkeiten reduziert werden. Die klare Trennung von Verantwortlichkeiten minimiert das Risiko von 'Spaghetti-Code' und erleichtert Änderungen. Strategic

DDD erleichtert auch die Zusammenarbeit zwischen Teams, indem es eine gemeinsame Sprache schafft. Insgesamt bietet es einen Rahmen, um die Komplexität der Softwareentwicklung zu bewältigen und einen effektiven Prozess zu fördern.

Folgende Strategien werden dabei vorgeschlagen:

- Partnership: Teams aus beiden Kontexten arbeiten zusammen und entwickeln Schnittstellen weiter, um den Entwicklungsbedürfnissen beider Kontexte gerecht zu werden.
- Shared Kernel: Beide Kontexte teilen sich eine gemeinsame Codebasis, und Änderungen am Kernel werden nach Rücksprache mit dem anderen Team vorgenommen.
- Customer-Supplier: Das upstream-Team berücksichtigt die Bedürfnisse des downstream-Teams, und die Beziehung zwischen den Teams ist formalisiert, so dass das upstream-Team der Lieferant und das downstream-Team der Kunde ist.
- Conformist: Das downstream-Team passt sich dem Modell des upstream-Teams an
- Anticorruption Layer: Das downstream-Team erstellt eine Abstraktionsschicht, die den downstream-Kontext vor Änderungen im upstream-Kontext schützt und es dem downstream-Team ermöglicht, mit einem Domänenmodell zu arbeiten, das ihren Bedürfnissen entspricht.
- Open Host Service: Der Zugang zu einem System erfolgt über klar definierte Dienste unter Verwendung eines klar definierten Protokolls. Das Protokoll ist offen, damit jeder, der es benötigt, mit dem System integrieren kann.
- Published Language: Eine dokumentierte Sprache wird für die Eingabe und Ausgabe des Systems verwendet. Es ist nicht notwendig, eine bestimmte Bibliothek oder eine bestimmte Implementierung einer Spezifikation zu verwenden, solange man sich an die veröffentlichte Sprache hält.
- Separate Ways: Die Systeme haben sich zu einem Punkt entwickelt, an dem sie nicht mehr miteinander verbunden sind, und es ist besser, die Kontexte voneinander zu trennen und sie unabhängig voneinander weiterentwickeln zu lassen.

Taktisches Bereichsorientiertes Design (DDD) bietet Prinzipien und Muster, um Abhängigkeiten in Softwareprojekten zu managen. Durch die Aufteilung der Codebasis in kleinere, auf die Geschäftsdomäne basierende Komponenten können Teams Abhängigkeiten zwischen verschiedenen Teilen des Systems reduzieren. Dies erfolgt mithilfe von Konzepten wie Bounded Contexts,

Aggregates und Domain Events, die klare Grenzen setzen und Interaktionen minimieren. Die Anwendung dieser Prinzipien ermöglicht es Teams, ein modulares, wartbares und skalierbares System zu entwickeln, das leichter zu verstehen und zu ändern ist. Eine klare Verantwortlichkeit jeder Komponente minimiert zudem die Notwendigkeit teamübergreifender Kommunikation und Koordination bei Systemänderungen. Taktisches DDD bietet somit einen effektiven Rahmen zur Verwaltung von Abhängigkeiten in Softwareprojekten und verbessert die Gesamtqualität, Flexibilität und Wartbarkeit der Codebasis.

- Bounded Contexts: Durch die Definition und Gruppierung verwandter Konzepte in abgegrenzten Kontexten kann taktisches DDD dazu beitragen, Abhängigkeiten zwischen verschiedenen Teilen des Systems zu identifizieren und zu verwalten.
- Ubiquitous Language: Durch die Schaffung einer gemeinsamen Sprache und eines Verständnisses der Domäne kann taktisches DDD dazu beitragen, Missverständnisse zu reduzieren und Abhängigkeiten zu minimieren.
- Context Mapping: Durch die Erstellung einer Karte der Beziehungen und Interaktionen zwischen verschiedenen abgegrenzten Kontexten kann taktisches DDD potenzielle Abhängigkeiten identifizieren und Bereiche aufzeigen, in denen Kommunikation und Koordination notwendig sind.
- Aggregates: Durch die Gruppierung verwandter Entitäten und die Definition von Transaktionsgrenzen kann taktisches DDD dazu beitragen, Abhängigkeiten zwischen verschiedenen Teilen des Systems zu verwalten und die Auswirkungen von Änderungen an einem Teil des Systems auf andere Teile zu minimieren.
- Domain Events: Durch die Verwendung von Domänenereignissen zur Kommunikation von Änderungen und Updates zwischen verschiedenen Teilen des Systems kann taktisches DDD dazu beitragen, Abhängigkeiten zu reduzieren und eine lose Kopplung zwischen verschiedenen Komponenten zu fördern.

### 3.2.6 Analyse der Migrationsstrategien

Viele Anwendungen sind bereits seit vielen Jahren in Benutzung und in Weiterentwicklung, daher häufen sich Bugs an [8]. Es wäre von Vorteil solche Anwendungen mit einer neuen Architektur neu zu beleben und viele angesammelte Bugs loszuwerden [24]. Nebenbei sollen auch veraltete bzw. nicht mehr benutzte Teile der Anwendung entfernt werden.

Paolo Di Francesco et al haben eine empirische Studie zu Migration zu Microservices durchgeführt. Dabei wurden folgende Aspekte analysiert [25]:

1. Ausgeführte Aufgaben im Rahmen der Migration.
2. Die Herausforderungen, die bei der Migration entstanden sind.

Die Studie bezog sich grundsätzlich auf die Microservice-Architektur mit Bezug zu Backend Anwendungen, dennoch lieferte sie auch eine gute Grundlage für die Autoren von [24], um einen ähnlichen Algorithmus für das Frontend vorzuschlagen:

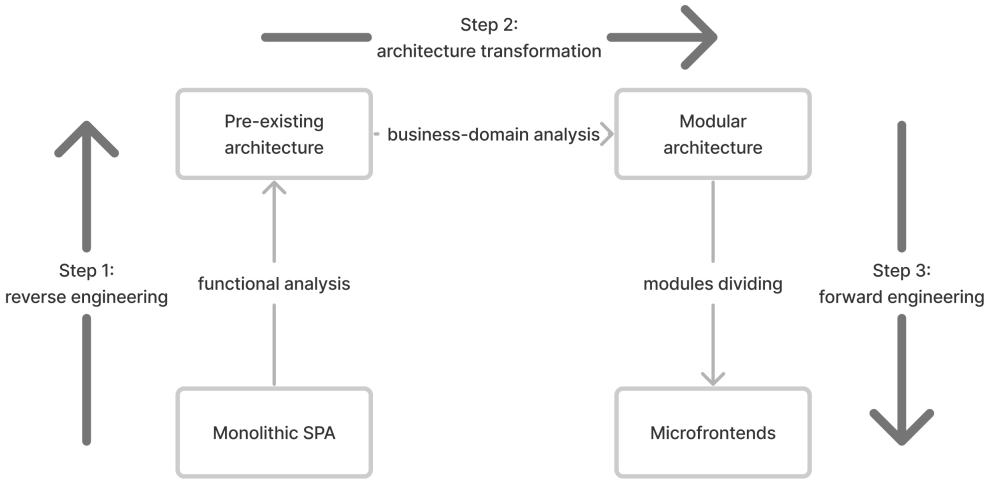


**Abb. 3.3:** Migration zur Microfrontends.

**Schritt 1: Reverse engineering** Im ersten Schritt wird die aktuelle Codebasis untersucht und es wird festgelegt, welche Teile der Anwendung separiert werden sollen [25].

**Schritt 2: Architecture transformation** Hier soll die Anwendung auf dem selben Abstraktionsniveau modularisiert werden [25]. Danach bleibt es weiterhin ein Monolith, aber mit einer micro-basierten Struktur.

**Schritt 3: Forward engineering** Im letzten Schritt folgt die finale Implementierung und Veröffentlichung [25].

In [24, S. 106] wird folgender Vergleich der technischen Lösungen aufgeführt, wobei *Module Federation* als beste Lösung aus dem Vergleich herauskommt.

| Feature | Hyperlinks | Single SPA | i-frames | Module Federation | Tailor.js |
|---|---|---|---|---|---|
| One URL for all application | - | + | + | + | + |
| Lack of framework – specific requirements for microfrontends | + | - | + | + | + |
| Avoiding code duplication | - | + | - | + | + |
| Lack of necessary to add backend server | + | + | + | + | - |
| Applicable to use with SPA | + | + | + | + | - |

**Tabelle 5.1**: Vergleich technischer Lösungen [24, S. 106].

### 3.2.7 Entscheidung für die Migration

Nach einer umfangreichen Recherche wurde entschieden, mit *Module Federation* fortzufahren. Am Anfang dieses Kapitels wurden die Arten von Micro Frontends aufgelistet. Mit Module Federation wird eine Single Page Application implementiert.

Module Federation ist ein Feature von webpack, welches es ermöglicht, verschiedene Versionen der einzelnen Modules aus verschiedenen Build-Systemen dynamisch zu laden. Dadurch wird Entwicklung von Microfrontend-Anwendungen leichter und zugänglicher gemacht. Dabei können die Anwendungen separat von einander entwickelt und veröffentlicht werden. Die Modules, die über mehrere Anwendungen hinweg benutzt werden, können geteilt werden [26]. Webpack (meist geschrieben **webpack**) ist ein sogenannter Opensource-JavaScript-Module-Bundler, deren Aufgabe besteht darin, JavaScript-Dateien für die Nutzung im Browser zusammenzuführen und zu einer Datei zu bündeln. Zusätzlich ist er auch in der Lage, andere Ressourcen zu bündeln [27].

Neben oben genannten Gründen ergeben sich folgende Vorteile für Module Federation:

- Einfache Entwicklung durch viel Dokumentation im Internet.
- Vorteilhaft ist ebenfalls die große Verbreitung von Module Federation, da die Suche nach Hilfe dadurch erleichtert wird.

- Module Federation ermöglicht sowohl Server-side als auch Client-Side Rendering. Standardmäßig wird Client-Side benutzt und es lässt sich zu Server-side umstellen.
- Möglichkeit, verschiedene Technologien für verschiedene Microfrontends zu nutzen, da diese im Endeffekt von webpack zu gebündelten JavaScript-Dateien zusammengeführt werden.

### Zusatztool

Bei der Recherche zu Module Federation, insbesondere beim Anschauen von verschiedenen Anleitungen zur Implementierung wird häufig *Nx* erwähnt. *Nx* ist ein Entwicklungstool, welches durch unter anderem das einfachere Verwalten von Bibliothekversionen, Unterstützung von Module Federation die Produktivität der Entwickler steigern soll. Es kann dabei helfen eine saubere Architektur der Anwendung zu erreichen, wodurch diese besser organisiert und skaliert werden kann [28].

In einem *Nx* Workspace befindet sich der Programmcode in der folgenden Ordnern:

- *apps*: Hier werden die Projekte abgelegt. Es sind beispielsweise Shell-Anwendung oder auch große Teile der Anwendung.
- *libs*: Hier werden Libraries abgelegt. Libraries in diesem Kontext sind wiederverwendbare Bibliotheken oder Module, die innerhalb des Monorepos erstellt werden. Es sollen auch einzelne Features sein.

Von Nx wird es empfohlen nach Regel 80/20 vorzugehen. 20% des Programmcodes werden unter *apps* abgelegt und 80% unter *libs*. Es wird dabei das Ziel verfolgt, den Programmcode so schlank wie möglich zu halten [29].

Ein weiterer wichtiger Vorteil von *Nx*, dass durch Caching die kürzeren Buildzeiten erreicht werden, wodurch die kostbare Zeit der Entwickler gespart wird [28]. Es wichtig zu beachten, dass die einzelnen Libraries und Projekte so wenig wie möglich von einander abhängig sind, insbesondere, dass es keine Kreisabhängigkeiten bestehen, d.h. dass zwei Modules nicht voneinander abhängig sein sollen. Wenn dies zutrifft, bringt das Caching am meisten Vorteile.

Es wurde somit entschieden im weiteren Verlauf der Bachelorarbeit *Module Federation* zusammen mit *Nx* zu benutzen.

### 3.2.8 Dokumentieren von Architektur-Entscheidungen

Der Zeitpunkt, an dem man sich die obere Frage stellt, kann vollkommen unterschiedlich sein. Kann sein, das passiert bei der Planung einer Software ganz am Anfang. Es kann auch sein, das passiert einige Zeit später. Vielleicht betrifft es die ganze Anwendung, vielleicht aber nur ein Fragment, welches an mehreren Stellen eingesetzt werden soll.

Nicht alle Architektur-Entscheidungen können im Voraus getroffen und dokumentiert werden, viele passieren auch schon in der Entwicklung. Das sollte aber die Entwickler nicht davon abschrecken, diese Entscheidungen zu dokumentieren. Die Dokumentation soll aber möglichst kurz und klar gehalten werden.

Für die Entscheidungen eignet sich gut die Vorlage von Michael Nygard [30]:

| Was soll entschieden werden? | |
|---|---|
| **Status** | Was ist der Status? Zum Beispiel vorgeschlagen, akzeptiert, abgelehnt. |
| **Kontext** | Was motiviert uns hier eine Veränderung zu machen? |
| **Entscheidung** | Was ist die Veränderung, über die entschieden wird? |
| **Folgen** | Was wird einfacher und/oder schwieriger durch die Veränderung? |

# 4 Entwicklung der Migrationsstrategie für die gewählte Beispielanwendung

## 4.1 Beschreibung der Anwendung

Die Beispielanwendung, deren Frontend-Monolith in Microfrontends zerlegt werden soll, ist eine interne Anwendung von **adesso SE**.

Die Anwendung kann grundsätzlich in folgende Teile aufgeteilt werden:

- **Datenbank aller Projekte**. Es werden alle Projekte der Haupt- und Töchtergesellschaften erfasst. Dabei werden viele Daten von anderen Diensten der Firma bezogen.
  *Hauptfunktionalitäten*:
  - Suche nach Projekten.
  - Anzeige von einzelnen Projekten.
  - Filtern von gesuchten Projekten.
  - Anschauen von Details zu einzelnen Projekten.
  - Bearbeitung von Projektdaten (für jeweilige Projektleitung).
  - Projektverwaltung - Übersicht eigener Projekte (für Projektleitungen).

  Folgend sind die Hauptseiten dieses Teils der Anwendung schematisch dargestellt. Die Homepage der Anwendung besteht aus einer Tabelle mit Projekt, sowie eine Filter- und Suchfunktion. Initial werden den Nutzern tausende Projekte angezeigt, dabei kann jedes Projekt in der Tabelle aufgeklappt werden, um Details zu sehen.

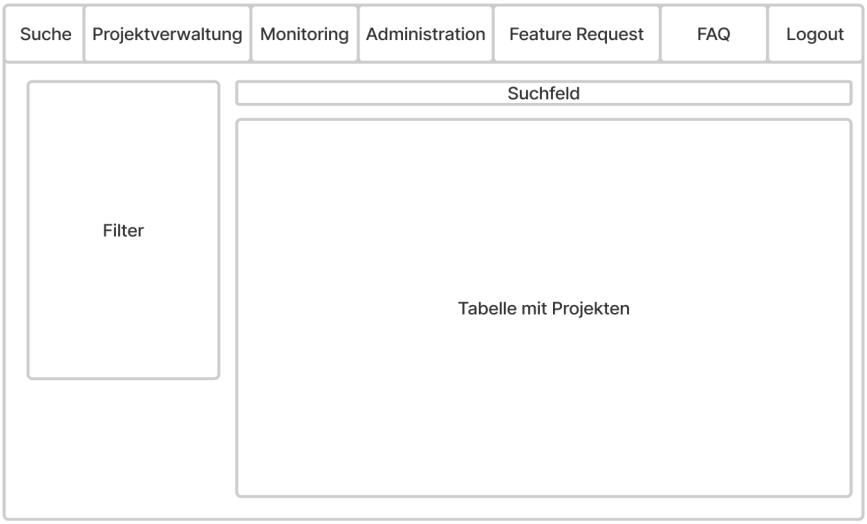


**Abb. 4.1:** Suchseite.


**Abb. 4.2:** Projektdetails.

Die Projektleiter sehen ihre eigenen Projekte im Tab "Projektverwaltung". Auf dieser Seite gibt es eine Übersicht eigener Projekt und deren Status (In Vorbereitung, Laufend, Abgeschlossen). Damit die Daten auch eingepflegt werden, werden keine Projekte abgeschlossen, solange es bestimmte Informationen fehlen. Wenn Projekt lange Zeit nicht gepflegt werden, werden Verantwortliche per E-Mail benachrichtigt.

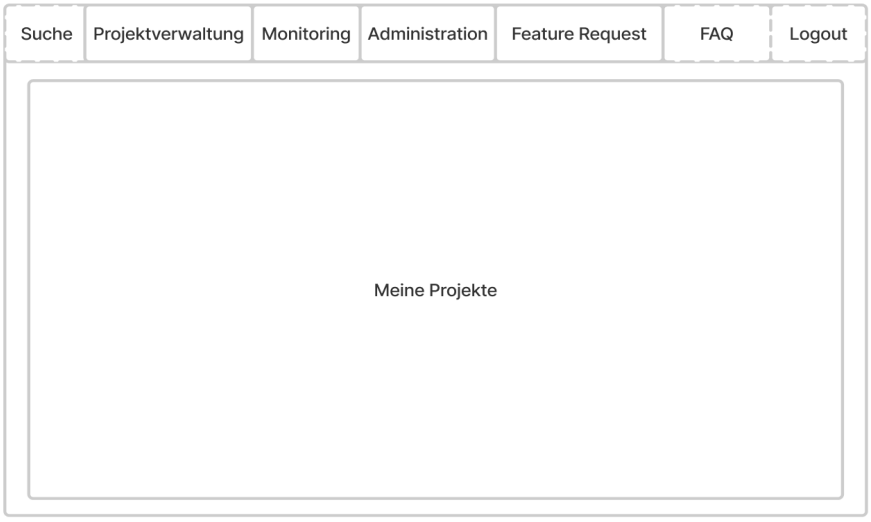


**Abb. 4.3:** Projektverwaltung.

Diese Projektdetails können Projektleiter auf einer separaten Seite auch bearbeiten.

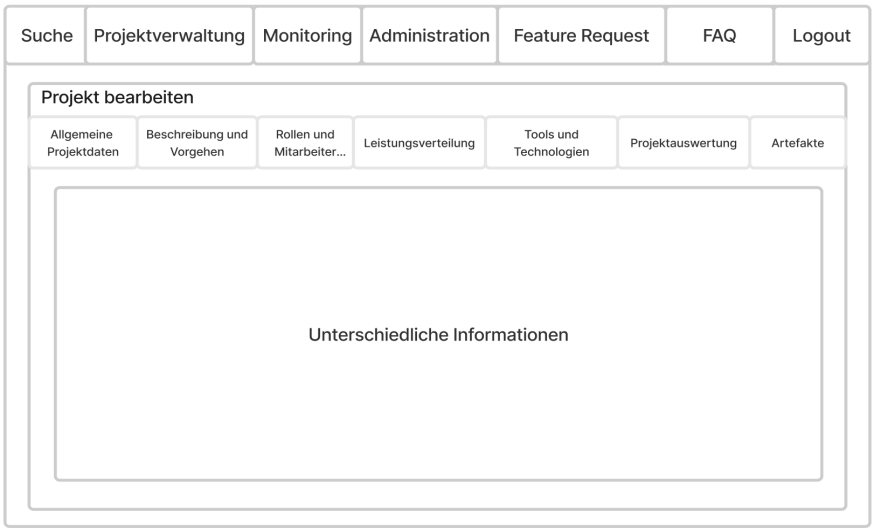


**Abb. 4.4:** Projektbearbeitung.

- **Project Monitoring Tool** wird bei vielen Projekten als internes Reporting Tool genutzt. Nicht alle Projekte gelangen automatisch in dieses Tool.
  *Hauptfunktionalitäten*:
  - Suche nach Projekten.
  - Filtern von gesuchten Projekten.

- Erstellung von Statusberichten für eigene Projekte (für Projektleitungen).
- Eintragen einiger anderen projektrelevanten-organisatorischen Informationen.
- Graphische Darstellung von unterschiedlichen wirtschaftlichen Daten von Projekten.
- Exportieren von Statusberichten, Informationen und Graphiken im PowerPoint und PDF Format.

Die Homeseite des Project Monitoring Tools sieht sehr ähnlich zu der Suche aus. Man sieht ebenfalls eine Tabelle von Projekte, allerdings mit anderen Informationen, die hervorgehoben werden und mit etwas anderen Filtermöglichkeiten. Der Hauptunterschied ist es, dass nur eine Abteilung der Firma alle Monitoring relevanten Projekt auch sieht, alle anderen sehen nur ihre eigenen Projekte, falls diese auch monitoring relevant sind.

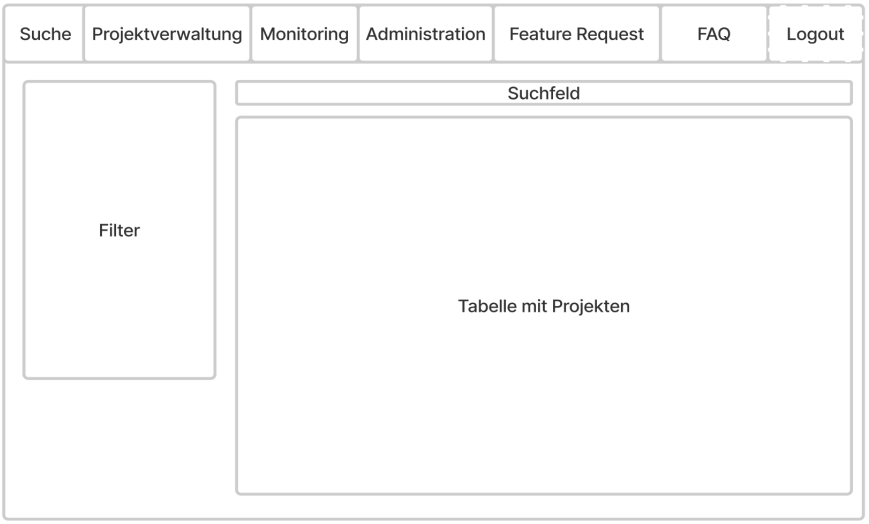


**Abb. 4.5:** Project Monitoring Tool.

Von der Homepage des Project Monitoring Tools kommt man auch zu Detailsseite. Darauf sieht man bestimmte Informationen des jeweiligen Projekt sieht, Statusberichte erstellen kann, sowie Risiken, Termine, Staffing-Angelegenheiten eintragen.

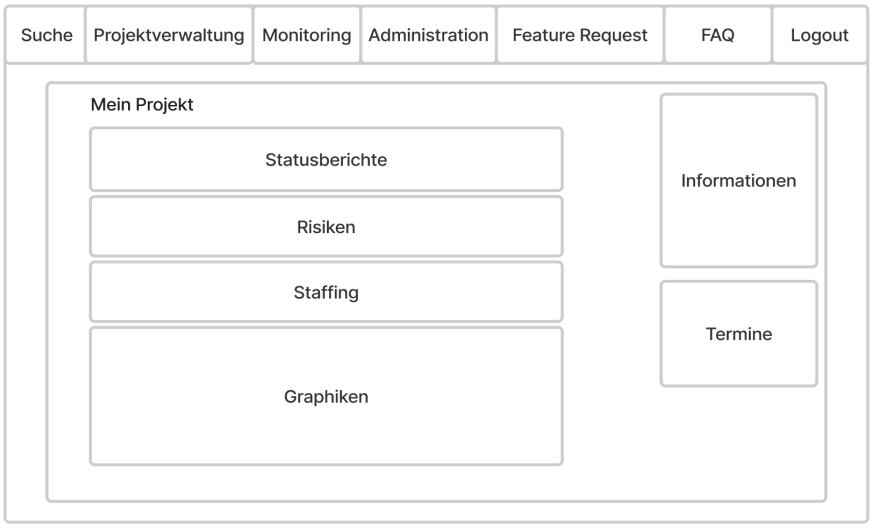


**Abb. 4.6:** Project Monitoring Tool - Details.

- **Karte** Im Rahmen einer Neuentwicklung entsteht gerade ein neues Modul - Karte. Hierbei soll eine Kartenanwendung implementiert werden, in der Projekte geographisch dargestellt werden sollten. Die Anwendung soll in erster Linie der Ausbildung der Entwickler im Umgang mit Geotechnologien dienen. Dennoch lassen sich auch bereits jetzt mehrere Use-Cases ableiten, die auch für die Firma einen Mehrwert bringen.
  Die ersten geplanten *Hauptfunktionalitäten*:
  - Suche nach Projekten auf der Karte.
  - Filtern von gesuchten Projekten auf der Karte.
  - Clustering von Informationen nach bestimmten Vorgaben.
  - Graphische Darstellung von Informationen (zum Beispiel Heatmap).

  Im Moment gibt es nicht viel, was man zur Karte (auch schematisch) zeigen kann. Die Anwendung besteht aktuell aus einer Seite, auf der Karte angezeigt wird mit ein Paar hardgecodeten Projekten.

### 4.1.1 Eingesetzte Technologien

Hier folgt eine kurze Übersicht der verwendeten Technologien, die für die Bachelorarbeit von Bedeutung sind. In ProjektPool/PMT wird im Frontend das Framework **Angular v16.2** verwendet. Für die Karte wurde die Bibliothek **React v18.2.0** gewählt. Für beide Teilprojekte wird **Bitbucket** als Repository verwendet, **Jenkins** als Build-Tool und **npm** als Dependency Management Tool.

## 4.2 Aktueller Stand der Anwendungen

### 4.2.1 Architekturanalyse

In folgender Abbildung ist der grobe Überblick über die Architektur der Anwendung gegeben. Der große Block rechts stellt die gesamte Anwendung da, bestehenden aus zwei Modules - Frontend und Backend (Client und Server).
**Frontend** besteht aus einem einzigen Module. Das heißt es kann nur als ein Ganzes veröffentlicht werden und einzelne Teile des Frontend sind nicht voneinander abgekoppelt.
Im **Backend** kommt Microservice-Ansatz ins Spiel. Es gibt einen Service (Web), der als Gateway für die Anfragen an alle anderen Services dient. Jeder anderer Microservice ist nach dem Single-Responsibility-Prinzip aufgebaut, der besagt, dass jede Klasse für eine bestimmte Aufgabezuständig bzw. jeder Service für bestimmten Umfang an zusammenhängenden Aufgaben zuständig sein dürfte.
Intern werden die Daten in zwei NoSQL Datenbanken gespeichert, für die jeweils einen Adapter Microservice gibt, der für Lesen und Schreiben von Daten zuständig ist.
Im Block Extern sind die externen Datenbanken aufgezeichnet - entweder werden Daten dorthin exportiert oder von dort importiert, sowie ein externer Service von Azure AD - durch ihn werden die Tokens beim Einloggen validiert (oder verworfen).

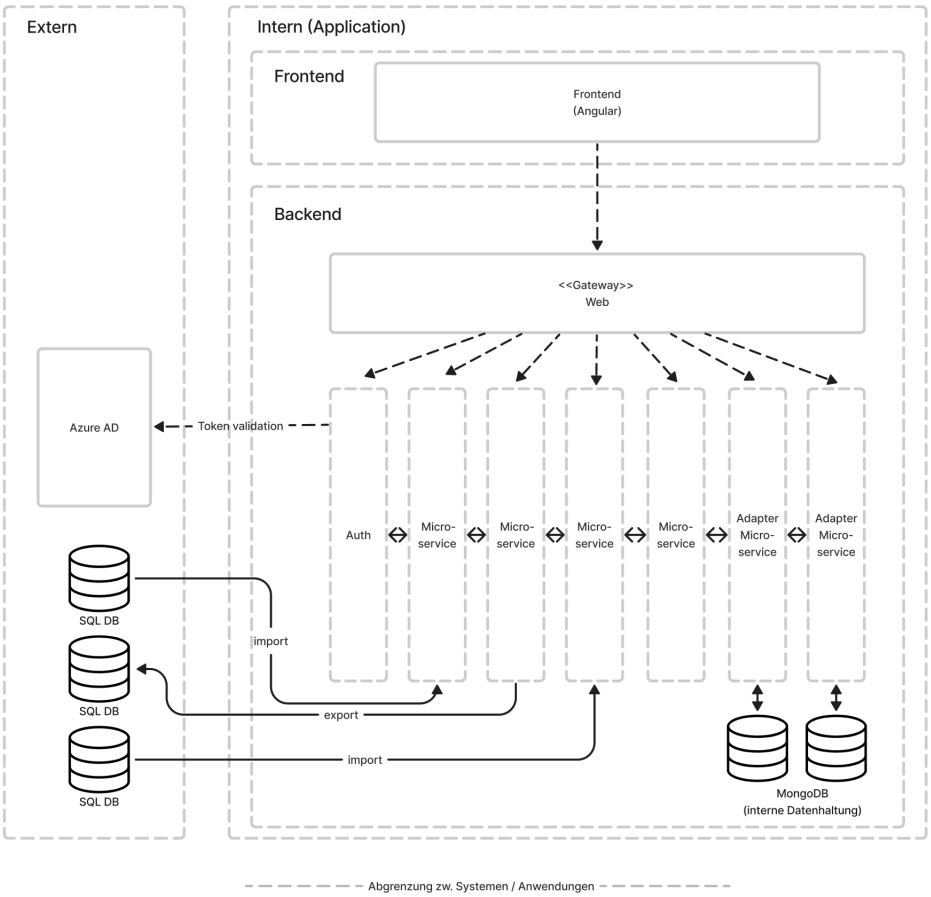


**Abb. 4.7:** Aktuelle Architektur.

Die Ordnerstruktur des Frontends sieht wie folgt aus:

```
Frontend
└─src
  ├─app
  │ ├─auth
  │ ├─components
  │ ├─models
  │ ├─pages
  │ ├─pipes
  │ ├─pmt
  │ │ ├─auth
  │ │ ├─components
  │ │ ├─models
  │ │ ├─pages
  │ │ └─services
  │ ├─services
  │ ├─util
  │ ├─validation
  │ ├─app.components.html
  │ ├─app.components.ts
  │ ├─app.config.ts
  │ ├─app.modules.ts
  │ └─app.routes.ts
  ├─assets
  ├─enviroments
  ├─variables.scss
  ├─styles.scss
  ├─adesso-theme-scss
  └─index.html
```

**Abb. 4.8:** Aktuelle Ordnerstruktur.

Es besteht teilweise eine Struktur, aber sie ist nicht konsistent. Viele Komponenten und Seiten ( an sich auch Komponenten) sind ineinander verschachtelt, schwerer zu finden und das erschwert die Arbeit mit dem Code.

Perspektivisch könnte die Frontendanwendung bestehend aus mehreren Microfrontends wie folgt aussehen:

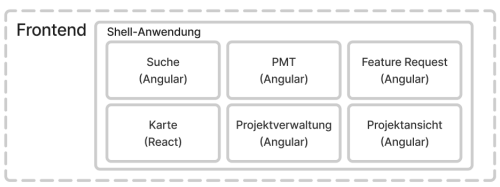


**Abb. 4.9:** Perspektive.

### 4.2.2 Problemanalyse

#### Welche Probleme bestehen aktuell?

- Architekturmodell, nach dem die ganze Angular Frontend-Anwendung aus einem einzigen Module besteht gehört nicht zu den "best practices".
  - Da es sich um ein Ausbildungsprojekt handelt, ist es umso schlimmer, da falsches bzw. nicht aktuelles Wissen vermittelt wird.
- Durch die fehlende Modularisierung sind die Buildzeiten zu lang.
- Durch die fehlende Modularisierung sind auch die Ladezeiten sehr hoch.
  - Kein Lazy Loading ist möglich.
  - Alle Seiten werden für jeden Nutzer geladen. D.h. auch Administrationsbereich wird im Hintergrund auch für Nutzer geladen, die eigentlich dafür so wie so kein Recht haben und nicht nutzen können.
- Kein Einzeldeployment von Features ist möglich.
  - Auch mit einer bereits modularisierten Anwendung wird man die ganze Anwendung auf einmal deployen müssen. Bestimmte Features können also nicht separat deployt werden.
- Struktur der Anwendung ist unübersichtlich.

Die Anwendung wird folgend mit einem Programm zur Visualisierung der Softwarearchitektur Structure101 analysiert. Das Problem wurde erst für die Java-Anwendungen entwickelt, allerdings gibt es auch eine Version, die JavaScript bzw. TypeScript Anwendungen analysieren kann.

In einer Visualisierung nach der Analyse bekommt man einen Score, der die strukturelle Komplexität der Architektur bewertet.

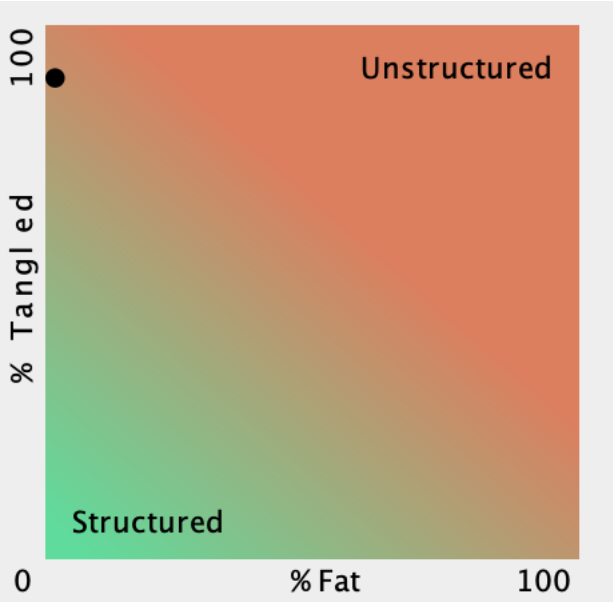


**Abb. 4.10:** Strukturelle Komplexität.

Die wichtigsten Funktionalitäten von Structure101 sind die Analyse der Abhängigkeiten einzelner Softwarekomponenten untereinander und die Visualisierung dessen, was passiert, wenn eine Abhängigkeit an eine andere Stelle verschoben wird. Das bedeutet, dass die gesamte Anwendung in Structure101 refactored werden kann. Anschließend erhält man eine Liste von Aufgaben, die in der tatsächlichen Anwendung durchgeführt werden müssen, um sie zu modularisieren. Die Vorteile liegen darin, dass die Umstrukturierung getestet werden kann, ohne die echte Anwendung zu beeinträchtigen, und dass man später nach einem klaren Plan vorgehen kann.

Leider habe ich von diesem Programm erst zu spät erfahren, sodass es nicht mehr sinnvoll für die Umsetzung der Modularisierung genutzt werden konnte. Es wäre jedoch definitiv lohnenswert, das Programm (oder ein ähnliches) direkt zu Beginn einzusetzen.

Welche Anforderungen gibt es an bzw. von Betrieb?

### 4.2.3 Anforderungserhebung & Qualitätssicherung:

#### Funktionale Anforderungen

- Verbesserung der Build Zeit.
- Deployment Prozess soll leichter sein.
- Einmalige Authentifizierung.

- Developer Experience.
- Entkoppelung externer Modules, damit das Aktualisieren in jedem Module separat durchgeführt werden können.

**Qualitätsanforderungen:**

- Zusammenspiel von verschiedenen Technologien.
- Einarbeitungszeit für neue Teammitglieder soll kürzer und einfacher werden.
- Kommunikation zwischen jeweiligen Microfrontends nur bei besonderem Bedarf (bspw. wenn dadurch bessere Usability erreicht wird).
- Beibehaltung bzw. Verbesserung der SOLID Prinzipien.
- Kompatibilität von verschiedenen Version der Applikationstechnologie.
- Testabhängigkeit. Das Testen soll weiterhin möglich sein.

## 4.3 Migrationsstrategie

### 4.3.1 Vorbereitung

In der Vorbereitung geht es darum, den am besten passenden Ansatz für die Anwendung zu wählen und anhand dessen die Strategie die Umsetzung der Migration zusammen zu stellen. Hierbei soll ein Schema bzw. eine Checkliste erstellt werden, wonach vorgegangen werden soll. Bei der Umsetzung werden diese Schritte kommentiert und ggf. ersetzt. Außerdem wird es auf aufkommende Probleme und deren Lösungsansätze eingegangen.

### 4.3.2 Ablaufplan

Bestehende Angular-Anwendung soll manuell in mehrere Anwendungen aufgeteilt werden - jede Anwendung soll seperat gestartet werden können. In der jeweiligen Microfrontend-Anwendung sind nur dort notwendige Abhängigkeiten vorhanden.

### Iterationen

#### 1: Analyse & Refactoring

Der erste Schritt ist die gesamte Anwendung zu analysieren, um den Code vor der Umsetzung auf möglichst aktuellsten und saubersten Stand zu bringen.

- Alter unbenutzter Code soll gelöscht werden.
- Unbenutzte (externe) Abhängigkeiten sollen entfernt werden.
- Code aktualisieren, falls aktuell die neuere Version der Sprache verwendet wird und deswegen einige Methoden und Ausdrücke einfacher geschrieben werden können.
- Code Stylings nebenbei korrigieren (Abstände etc.).
- Gegebenenfalls Klassen umbenennen und an die relativ richtige Stelle verschieben. Es kann sein, dass viel an der Struktur der Anwendung als falsch bezeichnet werden kann. Hier geht es nicht um eine endgültige Verschiebung und Umstrukturierung von Klassen, sondern darum diese 'falsche' Struktur überall auf dem gleichen Stand ist, sodass man später immer den gleichen Ausgangspunkt hat bei jeder Komponente.

#### 2: Entwurf

Im zweiten Schritt soll festgelegt welche Teile der Anwendung am sinnvollsten separiert werden sollen und können. Diese Teile der Anwendung können später ein separates Microfrontend darstellen. Innerhalb einzelner Teile der Anwendung können ebenfalls Unterteile gefunden werden, die später ein Fragment-Microfrontend darstellen könnten.

#### 3: Modularisierung

Im dritten Schritt erfolgt die Modularisierung der Anwendung. Das Ziel dabei ist es, jeden Teil der Anwendung von einander zu separieren. Dabei wird jedes Module nur dafür benötigte Abhängigkeiten haben. Es sollen zwei weitere Module entstehen - Shared und Core. Das Shared Module beinhaltet dabei alle externe Abhängigkeiten, wie zum Beispiel Angular Material, und interne Komponenten, die in mehreren Modules verwendet werden. Es lohnt sich dabei das Shared Module in zwei weitere Module unterzuteilen - Shared für interne Teile und ThirdParty für externe. Der Inhalt von Code Module werden Services, Models und Guards sein, die ebenfalls in mehrere Modules verwendet werden. Soweit ein einfaches Modules *core* oder *shared* benötigt, wird es dort importiert.

Folgende Ordnerstruktur wird übernommen [31]:

```
app
├── core
│   ├── guards
│   ├── interceptor
│   ├── services
│   ├── constants
│   ├── enums
│   ├── models
│   └── utils
├── features
│   └── feature-a
│       ├── components
│       ├── pages
│       ├── models
│       ├── services
│       ├── feature-a-routing.module.ts
│       ├── feature-a.module.ts
│       └── feature-a.component.html | scss | ts
├── shared
│   ├── components
│   ├── directives
│   └── pipes
├── styles
└── assets
```

**Abb. 4.11:** Empfohlene Ordnerstruktur.

Aufschlüsselung:

- *guards*: In Angular sind Guards Schnittstellen, die verwendet werden, um den Zugriff auf bestimmte Routen zu kontrollieren. Sie ermöglichen es, bestimmte Bedingungen zu überprüfen, bevor eine Route aktiviert wird. Beispiele hierfür sind der 'CanActivate-Guard', der entscheidet, ob eine Route aktiviert werden kann, und der 'CanDeactivate-Guard', der feststellt, ob eine Navigation von einer Route aus erlaubt ist [32].

- *interceptor*: Interceptors in Angular ermöglichen es, HTTP-Anfragen und -Antworten zu abzufangen und zu transformieren. Sie sind nützlich für das Hinzufügen von Authentifizierung, das Bearbeiten von Anforderungen oder das Transformieren von Antwortdaten, bevor sie die Anwendung erreichen [33].

- *services*: Services in Angular sind Singleton-Objekte, die Daten oder Funktionalitäten über Komponenten hinweg teilen. Sie werden verwendet, um Logik zu kapseln, die nicht direkt zu einer bestimmten Komponente gehört, wie z. B. Datenzugriff, Authentifizierungsdienste oder allgemeine Geschäftslogik [34].

- *constants*: In der Regel werden Konstanten in Angular für Werte verwendet, die in der gesamten Anwendung gleich bleiben sollen. Beispielsweise können URLs von APIs, App-Versionen oder andere Werte, die nicht verändert werden sollen, als Konstanten definiert werden.

- *enums*: Enums sind eine Möglichkeit, benannte Konstanten in TypeScript zu erstellen. In Angular werden Enums oft verwendet, um einen Satz von vordefinierten Werten zu definieren, die in der Anwendung verwendet werden können, um Typsicherheit zu gewährleisten.

- *models*: In Angular repräsentieren Models Datenstrukturen oder Entitäten innerhalb der Anwendung. Sie werden verwendet, um die Struktur von Daten zu definieren, die zwischen verschiedenen Teilen der Anwendung ausgetauscht werden, z. B. zwischen Komponenten und Services. Models verbessern die Typsicherheit und erleichtern die Wartung der Anwendung.

- *components*: Komponenten sind die grundlegenden Bausteine einer Angular-Anwendung. Sie sind wiederverwendbare, selbstständige Einheiten, die bestimmte Teile der Benutzeroberfläche repräsentieren und Logik dazu kapseln. Jede Komponente besteht aus einer TypeScript-Klasse, einem HTML-Template und optionalen CSS-Stilen. Komponenten ermöglichen die Strukturierung der Benutzeroberfläche und können hierarchisch angeordnet werden, um komplexe Anwendungen zu erstellen [35].

Ein wichtiger Aspekt bei der Modulasierung ist das Routing - es muss komplett überarbeitet werden, sodass interne Routes einzelner Modules sich in dem Module selbst befinden und der Shell Anwendung nur die Routes vorhanden sind, die auf die Module selbst zeigen. Dadurch wird Lazy-Loading erreicht, was eine bessere Performance und dadurch eine bessere Benutzererfahrung bringen soll. Die Buildzeit wird somit auch verringert, da jedes Module parallel gebaut werden kann. Die Buildzeit wird gemessen und analysiert.

Anschließend soll ein weiteres Mal versucht werden die Anwendung zu bereinigen, in dem die Models und Services aus dem *core* Module angeschaut, ggf. aufgeteilt und anschließend in ein Feature Module verschoben werden. Dadurch soll die Entkoppelung verbessert werden.

**4: Entwurf der Authentifizierung**

Im vierten Schritt soll die Authentifizierung analysiert werden und zwar, wie sie mit verschiedenen Microfrontends funktioniert soll. Es soll eine Vorgehensweise erarbeitet werden, die nach der Aufteilung in Microfrontends umgesetzt werden soll.

**5: Festlegung von Microfrontends**

Im fünften Schritt soll entschieden werden, welche Modules zu Microfrontends werden.

**6: Aufsetzen von *Nx* Workspace & Abkapselung einzelner Libraries und Projekte**

Im sechsten Schritt wird das *Nx* Workspace initialisiert, konfiguriert und mehrere Libraries und Projekte abgekapselt.

**7: Konfiguration von webpack**

Im siebten Schritt wird das webpack für Module Federation konfiguriert.

**8: Evaluierung**

Im letzten Schritt soll die Anwendung evaluiert werden.

# 5 Umsetzung der Migration

## 5.1 Ablauf

### 5.1.1 Analyse & Refactoring

Die Anwendung wurde in diesem Schritt analysiert und refactored. Dabei wurden viele Service- und Model-Klassen entfernt, sowie einige Komponenten, die nicht mehr verwendet werden. Für die Analyse wurden die zahlreichen Funktionen der IDE(Integrated Development Environment) Intellij benutzt.

Die Anwendung wird aktuell als ein großes Ganzes gebaut, d.h. es gibt keine Modules, die als Lazy Chunks gebaut und nur bei Bedarf nachgeladen werden:

```
Initial Chunk Files           | Names         |  Raw Size | Estimated Transfer Size
main.ab157c6c61f9cd96.js      | main          |   3.03 MB |                646.88 kB
styles.3a97fc901e4be47f.css   | styles        | 549.66 kB |                 50.50 kB
scripts.ea1f89fad9bed65e.js   | scripts       | 528.74 kB |                156.25 kB
polyfills.d436a9726019696e.js | polyfills     |  33.02 kB |                 10.64 kB
runtime.8bbad8ba76690b13.js   | runtime       |   1.23 kB |                665 bytes

                              | Initial Total |   4.12 MB |                864.92 kB

Build at: 2023-11-24T16:46:04.102Z - Hash: a943f7cef7ab669e - Time: 49654ms
```

**Abb. 5.1:** Das Build.

### 5.1.2 Entwurf

Zum Entwurf gehört die Festlegung welche Teile der Anwendung zu einzelnen Modulen werden. Es wurde entschieden, dass die Anwendung vor allem nach Domänen aufgeteilt wird. Die Modules

- Core.
- Shared.

- Search.
- Pmt.
- Projektbearbeitung.
- Projektdetails.
- Projektübersicht.
- Admin-Tab.
- Feature-Request.
- FAQ.
- Logout

### 5.1.3 Modularisierung

#### Vorbereitung

Vor Beginn der Modularisierung erstellt man das neue Ordnersystem nach dem Vorschlag aus vorigem Kapitel. Da die Modularisierung von mir komplett gemacht wird, kann man so verfahren, dass es immer weitere Modules entstehen und die ursprünglichen Ordner einfach leer werden und anschließend gelöscht werden.

Falls ein ganzes Team an der Modulasierung arbeitet und dafür mehrere Iterationen benötigt, lässt sich empfehlen das alte Ordnersystem in einen neuen Ordner *legacy* zu verschieben, um eine klare Grenze zwischen dem alten und dem neuen Stand zu entrichten.

Für den Anfang der Modularisierung habe ich mich für Feature-Request Module entschieden, da es bereits gut von anderen Teilen der Anwendung entkoppelt ist.

Die Modularisierung wird in der Intellij ausgeführt, welche zahlreiche Möglichkeiten zur Analyse der Abhängigkeiten sowie eine umfangreiche Suche bietet.

#### Analyse der Struktur

In der folgenden Abbildung wird die Struktur gezeigt. *navigation* ist eine Komponente für den Header der Anwendung und die Komponente *feature-request* befindet sich darin. Im Vergleich dazu werden alle anderen großen Komponenten (Seiten) woanders abgelegt. Eine leichte Inkonsistenz ist zu bemängeln. Um die Ansicht zu vereinfachen, werden nur die relevanten Ordner dargestellt.

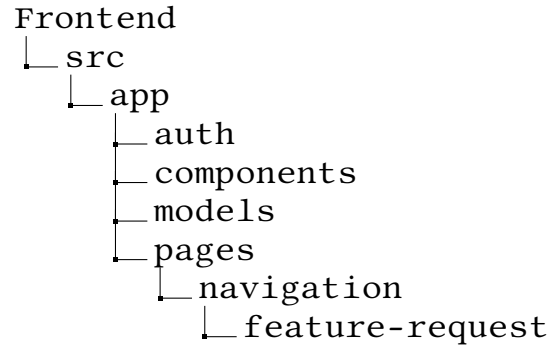


**Abb. 5.2:** Position von feature-request.

In Angular werden die eigenen Komponenten in index.html eingebunden - dabei können grundsätzlich bestimmte Komponente mit einem *selector* ausgewählt werden. Wird eine Anwendung größer, bietet sich die Möglichkeit mithilfe von Routing die Seiten dynamisch zu laden.

Die *index.html* sieht wie folgt aus:

```
<app-header></app-header>
<router-outlet></router-outlet>
```

Die Weiterleitung zur tatsächlichen *feature-request* Seite erreicht man mit der Definition und Registrierung der Routen in *app.module.ts*:

```
export const rootRouterConfig: Routes = [
...
 { path: 'feature-request', component: FeatureRequestComponent, canActivate: [AuthGuard],
    data : {title:titleProjektpool}}
...
];
```

Ein neues Module wird angelegt mit folgendem Befehl:

```
ng g m features/feature-request --project projekt-pool
```

Das neue Module besitzt dann folgende Struktur:

```
import { NgModule } from '@angular/core';

@NgModule({
 declarations: [],
 imports: [],
 providers: []
})
export class FeatureRequestModule {
}
```

Aufschlüsselung [36]:

- Die Annotation *@NgModule* definiert das Module.
- Alle Komponenten, Direktiven und Pipes, die Teil des Modules sind, werden unter *declarations* aufgelistet.
- Andere Modules können unter *imports* importiert werden.
- Alle Services, die Teil des Modules sind, werden unter *providers* aufgelistet.

Als nächstes wird die tatsächliche *feature-request* Komponente in das Modules verschoben, dort registriert und in *app.module.ts* rausgenommen. Als nächster Schritt mussten fast alle Imports von Komponenten und Services in der *feature-request* Komponenten aktualisiert werden. Das Module nutzt mehrere externe Angular Material Module. Es wurde versucht das *AppModule* zu importieren, das führte zu einem *Cyclic dependency between modules* Fehler. Wie der Name sagt, bestand damit eine Kreisabhängigkeit und die Anwendung konnte nicht gebaut werden.

Um das Problem zu lösen wurde nach dem gleichen Schema das neue Module *shared* angelegt und alle in *feature-request* externe Abhängigkeiten dorthin verschoben. Das *shared* Module wurde dann unter *import* eingetragen. Desweiteren nutzt *feature-request* mehrere weitere interne Komponenten, die aber auch in anderen Teilen der Anwendung eingesetzt werden. Die wurden ebenfalls in *shared/components* verschoben. Aufgrund dessen mussten die Imports von internen Komponenten auch überall in der Anwendung aktualisiert werden. Das *shared* Module wird ebenfalls in *app.module.ts* importiert.

Auch das *core* Module wurde erstellt, welches alle domänenübergreifende Services, Models und Guards beinhaltet. Da die Verkopplung von allen diesen drei Arten von Programmdateien in der ganzen Anwendung sehr hoch ist, wurde entschieden alle Dateien auf einmal ins *core* zu verschieben. Wie erwartet, führte dies zu zahlreichen Importfehlern in der ganzen Anwendung. Nachdem alle Imports refactored wurden, konnte das Problem der Kreisabbhängigkeit gelöst werden.

Für die vollständige Modularisierung dieses Teils der Anwendung fehlte noch das allerwichtigste - Routing.

Die Route in *app.module.ts* musste angepasst werden:

```
{
path: 'feature-request',
canActivate: [AuthGuard],
loadChildren: () => import('./features/feature-request/feature-request.module').then(m
    => m.FeatureRequestModule)
```

```
}
```

Eine Route musste ebenfalls in *feature-request.module.ts* definiert und registriert werden:

```
const routes: Routes = [{path: '', component: FeatureRequestComponent}]
```

Somit war das Module nutzbar. Mit *loadChildren* wurde das s.g. Lazy Loading erreicht, welche die initiale Ladezeit sowie die Buildzeiten reduzieren soll.

Ein Schema für die Modularisierung wurde dabei festgestellt:

- Das Module erstellen.
- Die Ordnerstruktur im Module erstellen.
- Alle internen Services, Models, Komponenten und ggf. Styles dorthin verschieben.
- Alle Komponenten und Services im neuen Module deklarieren und aus *app.module.ts* entfernen.
- Alle Import korrigieren.
- Das Routing umsetzen.

Nach diesem Schema wurden alle anderen Teile der Anwendung modularisiert. Dabei wurden weitere Mängel wie unbenutze Imports, unbenutzte Methoden und Variablen in zahlreichen Programmcodedateien entdeckt und entfernt.

Beim Bauen der modularisierten Anwendung sieht man, dass alle Modules als "Lazy Chunk Files"gebaut werden, wodurch die initiale Ladezeit verringert werden sollte, da die Modules nur dann für den User geladen werden, wenn er auch das jeweilige Module aufruft.

| Initial Chunk Files | Names | Raw Size | Estimated Transfer Size |
|---|---|---|---|
| main.bb8c17949be175d2.js | main | 1.45 MB | 301.21 kB |
| styles.5d5617a39ee24c06.css | styles | 549.46 kB | 50.53 kB |
| scripts.ea1f89fad9bed65e.js | scripts | 528.74 kB | 156.25 kB |
| polyfills.f1a81cf077e09999.js | polyfills | 33.02 kB | 10.67 kB |
| runtime.457c36d85bb4fa34.js | runtime | 3.01 kB | 1.45 kB |
| | **Initial Total** | **2.54 MB** | **520.11 kB** |

| Lazy Chunk Files | Names | Raw Size | Estimated Transfer Size |
|---|---|---|---|
| 209.99a289181d58ae46.js | features-pmt-pmt-module | 1.19 MB | 303.42 kB |
| 224.7617de11dc39b152.js | features-projekt-bearbeiten-projekt-bearbeiten-module | 174.40 kB | 30.68 kB |
| 912.b5c96f09a6d4e5b0.js | features-search-search-module | 106.29 kB | 18.90 kB |
| 915.82252dd55ef88ef9.js | features-projekt-details-projekt-details-module | 95.68 kB | 17.80 kB |
| 316.50ce665fbd03a97c.js | features-projekt-uebersicht-projekt-uebersicht-module | 46.64 kB | 9.50 kB |
| 603.13431e9d120a613d.js | features-admin-tab-admin-tab-module | 39.50 kB | 8.76 kB |
| 615.9cc6a9cc64ab337a.js | features-feature-request-feature-request-module | 22.62 kB | 5.09 kB |
| common.7d94ce66600ab4b5.js | common | 18.83 kB | 5.80 kB |
| 42.9dc52240fcc4d3ff.js | features-faq-faq-module | 7.15 kB | 1.51 kB |
| 13.b619e5c80e225fc6.js | features-initial-check-initial-check-module | 3.96 kB | 1.53 kB |
| 908.0d3e5c781a60a81e.js | features-logout-logout-module | 2.32 kB | 1.02 kB |
| 637.4812827009ade4c4.js | features-projekt-details-projekt-details-module | 940 bytes | 485 bytes |

Build at: 2023-11-15T15:59:28.456Z - Hash: 154274778b32b327 - Time: 44206ms

**Abb. 5.3:** Das Build.

### 5.1.4 Entwurf der Authentifizierung

Beim Entwurf des Ablaufplans war mir unklar, wie die Authentifizierung bei Microfrontends funktionieren soll. Mittlerweile hat sich diese Frage geklärt mit dem Ergebnis, dass es keine großen Umstellungen benötigt werden.

Der Authentifizierungsservice sowie alle Guards werden als ein separates Module zu einer *Nx* Library, welche in allen Teilen der Anwendung injiziert wird.

### 5.1.5 Festlegung von Microfrontends

In der Beispielanwendung könnten zahlreiche einzelne Microfrontends entstehen. Zur Veranschaulichung wird aus'PMT' ein Microfrontend entstehen. Zudem wird versucht, die Kartenanwendung (unter Verwendung von React) als eigenständiges Microfrontend zu integrieren.

Die Entscheidung das PMT zu wählen basiert auf vorher diskutierten DDD Prinzipien, insbesondere auf *Bounded Contexts*. Das gewählte Module unterscheidet sich bereits stark in seinen Funktionalitäten und stellt eine separate Domäne dar. Da die Anwendungen auch viel Gemeinsames untereinander haben werden, wird *Shared Kernel* Prinzip durch beispielsweise *core* und *shared* umgesetzt.

### 5.1.6 Aufsetzen von Nx Workspace & Abkapselung einzelner Libraries und Projekte

Zu Beginn muss *Nx* Workspace initialisiert werden:

```
nx init
```

Das automatische Aufsetzen betrifft vor allem die Konfigurationsdateien. Im Anschluss muss die Anwendung manuell nach *apps* verschoben werden. Folgende Ordnerstruktur wird übernommen [37]:

```
Frontend
├─apps
├─libs
├─tools
├─nx.json
├─packagejson
└─tsconfigbasejson
```

**Abb. 5.4:** Struktur von Nx.

Aufschlüsselung:

- *apps* beinhaltet die einzelnen Microfrontends (Projekte). Es wird empfohlen diese so schlank und leichtgewichtig zu gestalten wie möglich. Die schwergewichtige Teile sollen als Libraries von jeder Anwendung importiert werden [37].
- *libs* beinhaltet die Library Projekte verschiedener Arten [37].
- *tools* beinhaltet die Skripten für die Datenbank [37].
- *nx.json* konfiguriert das *Nx* Workspace: was wird gecached usw. [37].
- *tsconfig.base.json* konfiguriert die globalen TypeScript-Einstellungen und weist den Libraries Aliases zu, um die Imports zu erleichtern [37].

Nach Initialisierung von *Nx* werden unter anderem die Skripte in *package.json* aktualisiert und auf *Nx* umgeschrieben (diese können natürlich angepasst bzw. erweitert werden):

```
"scripts": {
"ng": "ng",
"start": "nx serve",
"build": "nx build",
```

```
  "build:prod": "nx build -c container",
  "test": "jest",
  "test:watch": "jest --watch",
  "lint": "nx lint --project projekt-pool",
  "e2e": "nx e2e"
 }
```

Das *Nx* bietet eine Funktionalität ähnlich zu vorhin vorgestelltem Programm *structure101*. Mit folgendem Befehl, lässt sich ein Abhängigkeitsgraph erstellen:

```
npx nx graph
```

In der Abbildung rechts sieht man, dass das *Nx* Workspace aktuell noch nur aus einem einzigen Projekt besteht:

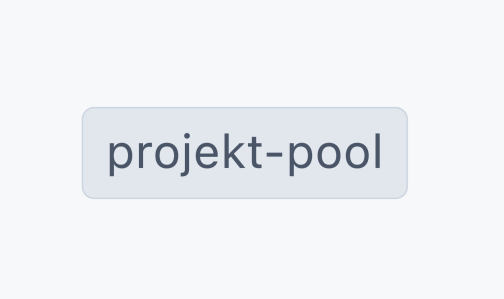


**Abb. 5.5:** Nach Initialisierung.

Weiter sollen Angular *core* und *shared* Module *Nx* Libraries werden. Um den Einstieg zu erleichtern, um große Fehler zu vermeiden, wird erst das *shared* Module in zwei Modules aufgeteilt - *shared* für interne Abhängigkeiten und *third-party* für externe wie z.B. Angular Material.

Das Erstellen der neuen Libraries passiert mit folgendem Befehl:

```
nx generate library
```

Für jede Library wird in *tsconfig.base.json* die Pfaden ergänzt, was das Importieren einfacher macht:

```
"@projektpool/core": ["libs/core/src/index.ts"],
"@projektpool/shared": ["libs/shared/src/index.ts"],
"@projektpool/third-party": ["libs/third-party/src/index.ts"],
```

Das Importieren von *third-party* in *shared*:

```
import { ThirdPartyModule } from '@projektpool/third-party';
```

Dadurch können in Zukunft Probleme vermieden werden, falls die Dateien umstrukturiert bzw. verschoben werden.

Nach Abkapselung von *third-party* *shared* und *core* sieht der Graph wie folgt aus:

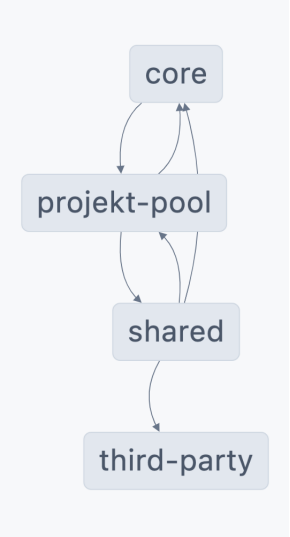


**Abb. 5.6:** Abkapselung von geteilten Libraries.

In der oberen Abbildung sieht man die Pfeile, die von *shared* und *core* zu *projekt-pool* führen. Es liegt daran, dass einige Konfigurations- sowie Authentifizierungsdateien geteilt werden und somit Kreisabhängigkeiten entstehen:

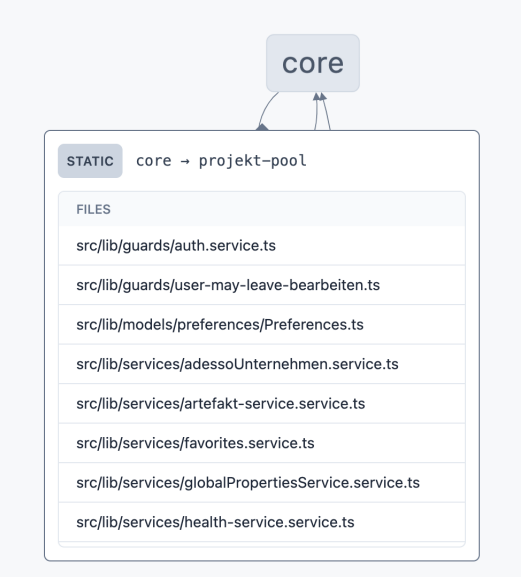


**Abb. 5.7:** Kreisabhängigkeiten.

Löse die Kreisabhängigkeiten durch Abkapselung von Konfiguration und Authentifizierung in eigene Libraries:

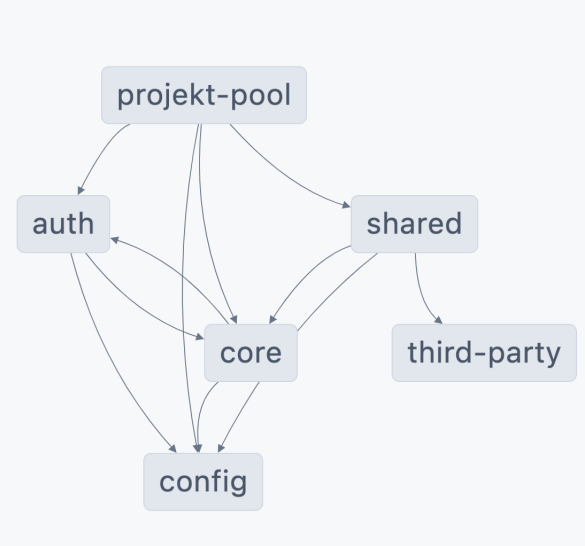


**Abb. 5.8:** Abkapselung von Konfiguration und Authentifizierung.

Im Anschluss werden zwei Projekte erstellt - eines für PMT und ein weiteres für die Karte, und der Programmcode wird dorthin migriert. Da die Karte sich derzeit noch in den Anfängen der Entwicklung befindet, bestehen noch keine Abhängigkeiten zu *auth*.

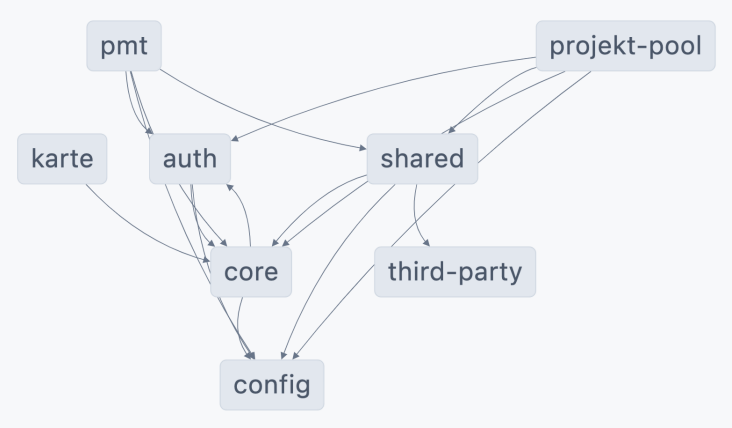


**Abb. 5.9:** Finaler Stand.

### 5.1.7 Konfiguration von webpack

Der webpack wird in *module-federation.config.ts* der jeweiligen Projekte oder Libraries konfiguriert. Beginne mit Microfrontend 'PMT':

```
import { ModuleFederationConfig } from '@nx/webpack';

const config: ModuleFederationConfig = {
  name: 'projekt-pool',
  remotes: ['pmt'],
};
```

```
export default config;
```

Sobald das Microfrontend geteilt wird, muss das Hauptprojekt konfiguriert werden:

```
import { ModuleFederationConfig } from '@nx/webpack';

const config: ModuleFederationConfig = {
 name: 'pmt',
 exposes: {
   './Module': './src/remote-entry.ts',
 },
};
export default config;
```

Es sind die minimalen Konfigurationen, die man beim webpack vornehmen muss. Es gibt einige weitere Konfigurationen, die man für sein jeweiliges Projekt eventuell in Betracht ziehen sollte. Aktuell werden mehrere Modules über das *Nx* verwaltet und auch freigegeben für andere Modules und Projekte. webpack bietet auch die Möglichkeiten Modules zwischen Anwendungen zu teilen.

#### Problem

Die Idee bestand darin, die wenigsten Konfigurationen vorzunehmen, um das PMT auf einfachster Weise in den Projekt-Pool einbinden zu können. Allerdings trafen bereits dabei Bugs auf, die nicht vollständig gelöst werden konnten, was die Weiterentwicklung der Umsetzung unmöglich machte. Leider war keine Lösung der Probleme im zeitlichen Rahmen der Bachelorarbeit möglich.

#### Möglicher Lösungsansatz

Ein möglicher Lösungsansatz bei mehr Zeit wäre es, nicht den webpack in der bereits bestehenen Projekt-Pool Anwendung zu initialisieren, sondern ein neues Projekt anzulegen mit den Standard-konfigurationen und dabei sicherzustellen, dass es funktioniert. Dann wäre der Programmcode inkl. aller Abhängigkeiten zu migrieren. Wobei das bei den Microfrontends keine große Ausgabe gewesen ist, wäre es bei der Hauptanwendung deutlich schwieriger.

### 5.1.8 Evaluierung

Im folgenden wird die Anwendung an mehreren Stellen verglichen:

| Measurable indicator | Pre-existing system (monolithic SPA) | Modular architecture (modularized SPA) | Target system (Micro-frontends) |
|---|---|---|---|
| Production builds building time, ms | 23317.83 | 20162.67 | x |
| Size of the main bundle, mb | 4.12 | 2.54 | x |
| First page average load time, ms | 6960 | 4933 | x |

**Tabelle 5.1**: Vergleich der Auswertungskriterien.

### Buildzeit

Während sich die Größe von Main Bundle einfach ablesen lässt, da diese immer fast genau gleich bleibt, ist die Evaluierung der Buildzeit deutlich schwieriger.

Um die genauesten Daten zu erhalten, wurde ein Bash Skript geschrieben, welches den *ng build*-Befehl mit produktiver Konfiguration 100-mal ausführt. Die gemessenen Zeiten werden summiert und am Ende durch die Anzahl der Läufe geteilt.

Zusätzlich zur Tabelle sind die Ergebnisse in der unteren Abbildung dargestellt. Es ist zu sehen, dass die modularisierte Anwendung um ca. 15% schneller gebaut wird. Über viele Jahre und viele Entwickler kann diese Zahl immer größer werden und die kostbare Entwicklerzeit rauben. Es ist zu beachten, dass bei größeren Anwendungen die Unterschiede auch größer ausfallen könnten und somit eine Modularisierung noch mehr Relevanz hat.

War:

```
Average execution time for 'ng build' over 100 runs: 23317.83 milliseconds
```

Ist:

```
Average execution time for 'ng build' over 100 runs: 20162.67 milliseconds
```

**Abb. 5.10:** Veränderung der Buildzeit.

### Ladezeit

Für die Auswertung der Ladezeit wurde eine Anleitung [38] benutzt, in der die Möglichkeiten aufgelistet werden.

Die zugänglichste Methode ist es, Entwicklertools im Browser zu nutzen, dabei werden folgende drei Faktoren beachtet:

- *DOMContentLoaded*: Dieses Ereignis im Dokument wird ausgelöst, wenn das initiale HTML-Dokument vollständig heruntergeladen und geparst wurde.
- *Load*: Dieses Fensterereignis wird ausgelöst, wenn die Seite vollständig geladen ist, also wenn das HTML (1) und die blockierenden Ressourcen (CSS und JS) heruntergeladen und geparst wurden.
- *Finish*: Das Laden ist vollständig abgeschlossen, wenn (1) und (2) und auch die NICHT-BLOCKIERENDEN JS-Ressourcen heruntergeladen und geparst sind und alle XMLHttpRequests und Promises abgeschlossen sind.

Die Kriterien sind mit Hellblau gekennzeichnet:

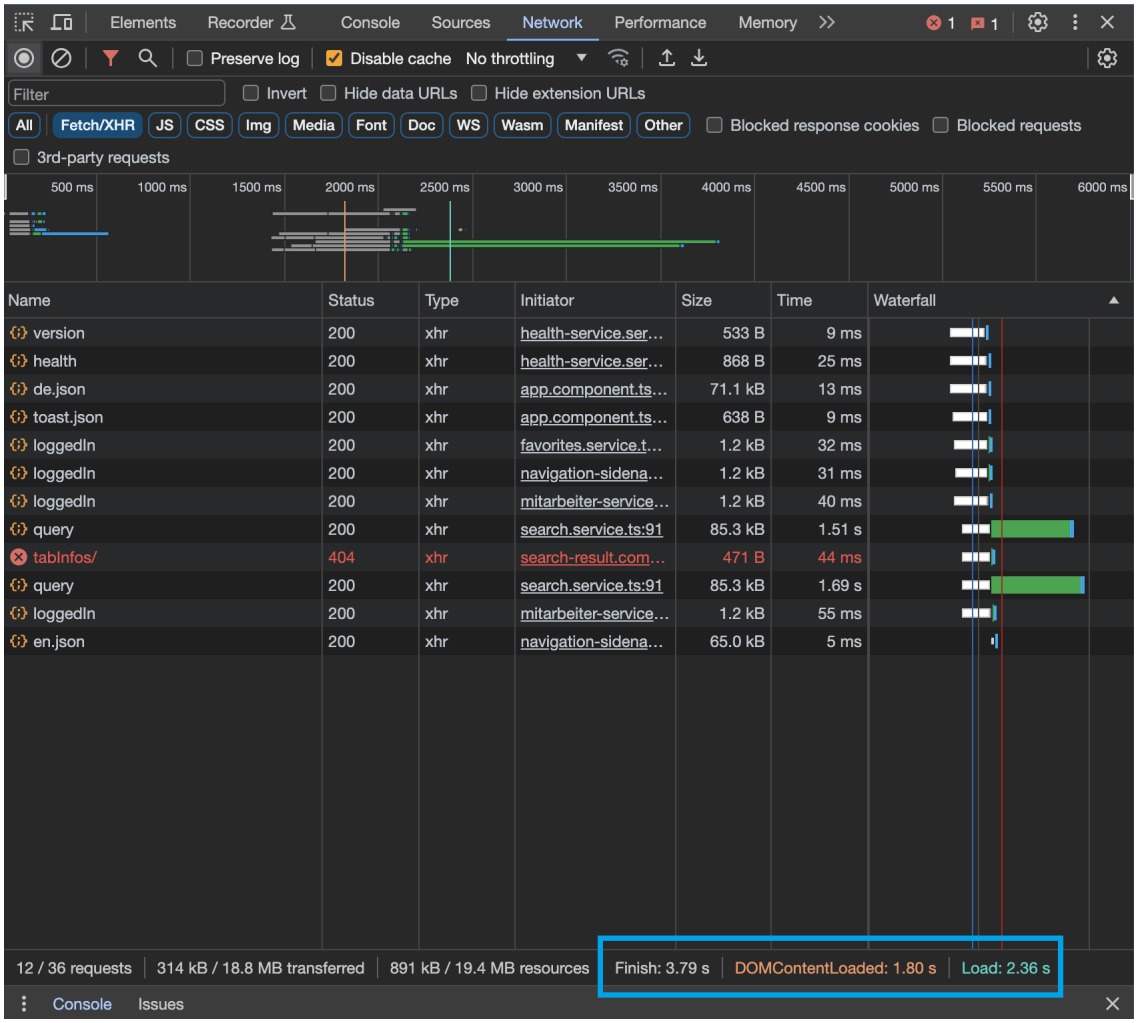


**Abb. 5.11:** Entwicklertool.

Der Finish Wert wird mehrmals ermittelt und der Durchschnitt davon in der oberen Tabelle angegeben.

In der Anleitung wird auch das Open-Source-Tool von Google, *Google Lighthouse*, vorgestellt, mit dem die Anwendung auf verschiedene Weisen untersucht werden kann. Dieses Tool wurde für die Bachelorarbeit zwar erst spät entdeckt, dennoch lohnt es sich grundsätzlich, es bereits bei der Entwicklung einzusetzen.

### Schlussfolgerung

In allen drei Kriterien gibt es eine Verbesserung:

- Die initiale Ladezeit wurde um 30% gekürzt.

- Die Buildgröße des Main Bundles ist ca. 40% kleiner geworden.
- Die Buildzeit hat sich um ca. 15% verbessert.

# 6 Zusammenfassung

Die Entstehung und Bedeutung von Microfrontends wurden untersucht, wobei viele Vor- und Nachteile bei ihrer Nutzung identifiziert wurden. Es existieren zahlreiche Ansätze für die Umsetzung. Vor der Implementierung von Microfrontends ist es entscheidend, die spezifischen Anforderungen und Gegebenheiten der Anwendung zu berücksichtigen. Eine Entscheidungsmatrix kann dabei unterstützen, die verschiedenen Aspekte zu bewerten und Vor- sowie Nachteile abzuwägen. Eine umfassende Dokumentation und Schulung der Entwicklerteams sind ebenfalls von großer Bedeutung, um einen reibungslosen Übergang sicherzustellen.

Insbesondere in großen Anwendungen mit verschiedenen Domänen und zahlreichen Entwicklern könnten Microfrontends sich als vorteilhaft erweisen. Durch die Aufteilung der Entwickler in kleinere Teams, die sich auf spezifische Domänen konzentrieren, wird weniger Multitasking erforderlich, was die Produktivität steigern kann. Alleine durch die Modularisierung einer Anwendung konnten deutliche Verbesserungen in Main Bundle Größe, Lade- und Buildzeit erreicht werden.

Das Deployment kann getrennt werden, wodurch Teile der Anwendung unabhängig voneinander deployed werden können. Dies erleichtert den CI/CD-Prozess und verhindert unnötige Downtime für die gesamte Anwendung. Trotzdem können sich Teile der Anwendung herauskristallisieren, die über mehrere Teams hinweg geteilt werden müssen. Ein effektiver Kommunikationskanal zwischen den Teams ist dabei wichtig, um den Datenaustausch kurz und strukturiert zu halten.

Die Migration einer bestehenden Anwendung ist ein komplexes Unterfangen, das sorgfältig durchdacht werden sollte. Die Nutzung von Domain-Driven Design (DDD) kann eine sinnvolle Aufteilung der Anwendung ermöglichen. Das Zusammenfügen von Microfrontends kann beispielsweise mit Module Federation oder anderen technischen Werkzeugen erfolgen. Dabei ist es ratsam, stets den aktuellen Stand der Softwareentwicklung im Auge zu behalten, da Veränderungen schnell erfolgen können.

Es ist wichtig zu betonen, dass Microfrontends nicht für jede Anwendung geeignet sind, und es gibt auch andere Architekturansätze, die in bestimmten Situationen besser passen könnten. Daher ist eine gründliche Analyse und Abwägung der Vor- und Nachteile unerlässlich, bevor die Einführung von Microfrontends in Erwägung gezogen wird.

Zusammenfassend lässt sich sagen, dass Microfrontends ein vielversprechendes Werkzeug sind, um die Herausforderungen großer und komplexer Frontend-Anwendungen zu bewältigen. Eine fundierte Entscheidung und eine gut durchdachte Implementierung sind entscheidend, um die Vorteile dieses Ansatzes optimal zu nutzen.

# Danksagung

Ich möchte mich zunächst bei Prof. Dr. Lutz Prechelt für die Betreuung dieser Bachelorarbeit herzlich bedanken. Ein besonderer Dank gilt auch adesso SE und meinem Kollegen Daniel Dienhardt, der die Arbeit betreute und mir bei verschiedenen Problemen hilfreich zur Seite stand. Er wies mir in vielen Fällen die Richtung und lieferte neue Ideen. Ebenfalls möchte ich mich herzlich bei der Zweitgutachterin der Arbeit, Prof. Dr. Claudia Müller-Birn, bedanken.

# Literatur


1. VALTER ROESLER Eduardo Barrére, R. W.: *Special Topics in Multimedia, IoT and Web Technologies*. Springer International Publishing, 2020. ISBN 978-3-030-35102-1.
2. RAPPL, F.: *The The Art of Micro Frontends: Build websites using compositional UIs that grow naturally as your application scale*. Packt Publishing, 2021. ISBN 978-1800563568.
3. T.MAURO: *Adopting Microservices at Netflix: Lessons for Architectural Design*. Auch verfügbar unter: https://www.nginx.com/blog/Microservices-at-netflix-architectural-best-practices/.
4. K.PARIKH, S. und: *From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture*. Auch verfügbar unter: https://www.infoq.com/presentations/linkedin-Microservices-urn/.
5. D.KOLESNIKOV: *Using Microservices to Power Fashion Search and Discovery*. Auch verfügbar unter: https://engineering.zalando.com/posts/2017/02/using-microservices-to-power-fashion-search-and-discovery.html.
6. D.RAGGETT A.L.Hors, I.: *HTML 4.0 Specification*. 1998. Auch verfügbar unter: https://www.w3.org/TR/1998/REC-html40-19980424/.
7. NEWMAN, S.: *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2021. ISBN 978-1492034025.
8. F.AUER V.Lenarduzzi, M. u. D.: From monolithic systems to Microservices: An assessment framework. *Information and Software Technology Volume 137, September 2021, 106600*. 2021, ?? Abger. unter DOI: https://doi.org/10.1016/j.infsof.2021.106600.
9. TAIBI D. Lenarduzzi V., P. C.: Microservices: Architecting for Continuous Delivery and DevOps. *Cloud Computing and Services Science. CLOSER 2018 Selected papers. Communications in Computer and Information Science, vol 1073*. 2019, S. 126–151. Abger. unter DOI: 10.1007/978-3-030-29193-8_7.
10. CAIFANG YANG, C. L.; SU, Z.: Research and Application of Micro Frontends. *IOP Conference Series: Materials Science and Engineering*. 2019. Abger. unter DOI: https://iopscience.iop.org/article/10.1088/1757-899X/490/6/062082/meta.

11. FOWLER, M.: *Micro Frontends.* Auch verfügbar unter: https://martinfowler.com/articles/micro-frontends.html.

12. GEERS, M.: *MicroFrontends: extending the microservice idea to frontend development.* Auch verfügbar unter: https://micro-frontends.org/.

13. MEZZALIRA, L.: *Building Micro-Frontends.* O'Reilly Media, Inc., 2021. ISBN 978-1492082996.

14. GEERS, M.: *Micro Frontends in Action.* Manning Publications, 2020. ISBN 978-1617296871.

15. A.TREGUBOV, B. u. J.: Impact of task switching and work interruptions on software development processes. 2017, S. 2. Abger. unter DOI: 10.1145/3084100.3084116.

16. GEER, B. de: *Micro Frontends: A Deep Dive into Pros and Cons?* 2023. Auch verfügbar unter: https://www.linkedin.com/pulse/micro-frontends-deep-dive-pros-cons-boudy-de-geer/.

17. M. MAROHNIĆ K.Oddsson, M.; K.CIRKEL: *Removing jQuery from GitHub.com frontend.* Auch verfügbar unter: https://github.blog/2018-09-06-removing-jquery-from-github-frontend/.

18. N.VENDITTO: *Pros and cons of micro-frontends.* 2023. Auch verfügbar unter: https://microfrontend.dev/architecture/micro-frontends-pros-and-cons/.

19. *Micro Frontends Patterns12: Server Side Composition.* 2021. Auch verfügbar unter: https://dev.to/okmttdhr/micro-frontends-patters-13-server-side-composition-1of5.

20. *Apache httpd Tutorial: Introduction to Server Side Includes.* 2023. Auch verfügbar unter: https://httpd.apache.org/docs/current/howto/ssi.html.

21. *What Is Edge Side Includes (ESI)?* 2018. Auch verfügbar unter: https://www.keycdn.com/support/edge-side-includes.

22. *<iframe>: The Inline Frame element.* Auch verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe?retiredLocale=de.

23. *Domain-Driven Design.* 2014. Auch verfügbar unter: https://www.dependencypoker.com/dependencies-and-strategies/strategies/domain-driven-design.

24. O.VARENYK, N. u. K.: Method of Converting the Monolithic Architecture of a Front-End Application to Microfrontends. 2022, S. 106–116. Abger. unter DOI: https://secs.knute.edu.ua/doc/SECS_2022.pdf.

25. P.D.FRANCESCO, P. u. I.: Migrating Towards Microservice Architectures: An Industrial Survey. 2018, 1?? Abger. unter DOI: http://dx.doi.org/10.1109/ICSA.2018.00012.

26. *Module Federation.* 2022. Auch verfügbar unter: https://github.com/module-federation.

27. *Webpack*. 2021. Auch verfügbar unter: https://de.wikipedia.org/wiki/Webpack.

28. *Why Nx?* Auch verfügbar unter: https://nx.dev/getting-started/why-nx.

29. *Mental model*. Auch verfügbar unter: https://nx.dev/concepts/more-concepts/applications-and-libraries.

30. M.NYGARD: *DOCUMENTING ARCHITECTURE DECISIONS*. Auch verfügbar unter: https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions.

31. S.NATH: *Angular File Structure and Best Practices (that help to scale)*. 2020. Auch verfügbar unter: https://medium.com/@shijin_nath/angular-right-file-structure-and-best-practices-that-help-to-scale-2020-52ce8d967df5.

32. *Common Routing Tasks*. Auch verfügbar unter: https://angular.io/guide/router.

33. *HttpInterceptor*. Auch verfügbar unter: https://angular.io/api/common/http/HttpInterceptor.

34. *Introduction to services and dependency injection*. Auch verfügbar unter: https://angular.io/guide/architecture-services.

35. *Component*. Auch verfügbar unter: https://angular.io/api/core/Component.

36. *Angular Modules and NgModule - Complete Guide*. 2023. Auch verfügbar unter: https://blog.angular-university.io/angular2-ngmodule/.

37. *Integrated Repo Folder Structure*. Auch verfügbar unter: https://nx.dev/concepts/more-concepts/folder-structure.

38. THALHAMMER, A.: *How to measure Initial Load Performance*. 2023. Auch verfügbar unter: https://www.angulararchitects.io/blog/how-to-measure-initial-load-performance/.

39. T.CLEMSON: *Testing Strategies in a Microservice Architecture*. 2014. Auch verfügbar unter: https://martinfowler.com/articles/microservice-testing/#conclusion-test-pyramid.