FERDINAND PORSCHE
FERN FH

Micro Frontends - Module Federation, Monorepo & React Prototype

Bachelorarbeit

eingereicht von: **Philipp Mossier**
Matrikelnummer: 51905481

im Fachhochschul-Bachelorstudiengang Wirtschaftsinformatik (0470)
der Ferdinand Porsche FernFH

zur Erlangung des akademischen Grades

Bachelor of Arts in Business

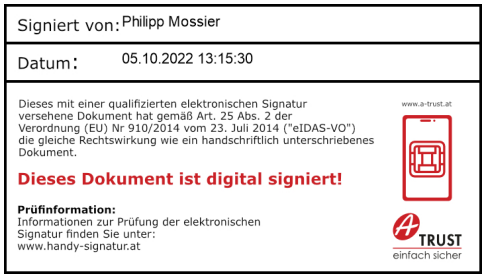
Betreuung und Beurteilung: DI Dr. Werner Toplak

Wiener Neustadt, Oktober 2022

Ehrenwörtliche Erklärung

Ich versichere hiermit,

1. dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Inhalte, die direkt oder indirekt aus fremden Quellen entnommen sind, sind durch entsprechende Quellenangaben gekennzeichnet.
2. dass ich diese Bachelorarbeit bisher weder im Inland noch im Ausland in irgendeiner Form als Prüfungsarbeit zur Beurteilung vorgelegt oder veröffentlicht habe.



Wien Philipp Mossier, 05.10.2022

Unterschrift

Die Kapitel 1 bis 3 der vorliegenden Bachelorarbeit wurden im Rahmen der Lehrveranstaltung „Bachelor Seminar 1“ eingereicht und am 09.09.2022 Datum des Gutachtens als Bachelorarbeit 1 angenommen.

Kurzfassung

Die Anforderungen an das Web steigen von Jahr zu Jahr, was dafür sorgt, dass die Systeme dahinter immer umfangreicher und komplexer werden. Um große und vor allem komplexe Webanwendungen in kleine unabhängige Services aufzuteilen, bedient man sich bei Systemen, die im Hintergrund laufen (Backend), seit Jahren an der äußerst erfolgreichen Microservice-Architektur (MSA).

Im sichtbaren Teil einer Webanwendung (Frontend), wird der darunter liegende Code in der Regel nicht in kleine, unabhängige Teile aufgeteilt, da man lange Zeit keine andere Möglichkeit hatte, außer einen Monolithen zu entwickeln. Dies ändert sich nun mit einem neuen Architekturstil namens Micro Frontends (MF), welcher eine “microservice” ähnliche Alternative zum Monolithen darstellt. MF in die Praxis umzusetzen ist aktuell noch schwierig, da die Erfahrung mit diesem Architekturstil fehlt. Wenn MF falsch eingesetzt werden, bringen diese oft mehr Nachteile als Vorteile.

Diese Arbeit zielt darauf ab, unterschiedliche Technologien miteinander zu vergleichen, um herauszufinden, welche dieser Technologien am besten für MF geeignet sind.
Ein durch die Weighted Scoring Methode aufgestellter Vergleich von 5 verschiedenen MF-Frameworks (Module-Federation, Single-Spa, Piral, Podium oder Mashroom-Server) und ein Vergleich von Monorepo und Multi-Repo, lässt schlussendlich Module-Federation in Kombination mit einem Monorepo als klaren Sieger hervorgehen. Darüber hinaus wurden Lösungswege entdeckt, die den Nachteilen einer Micro Frontend Architektur (MFA) entgegenwirken.

Schlüsselwörter: Micro Frontends, Frontend Architecture, Module Federation, Microservices

Abstract

The demands on the web are increasing from year to year, which means that the systems behind them are becoming more and more complex. In order to separate large and complex software into small manageable parts, the extremely successful microservice architecture (MSA) has been used in the backend for years.

In the visible part of a web application (frontend), the underlying code is usually not divided into small independent parts, because for a long time, there was no other option than developing a monolith. This is now changing with a new architectural style called Micro Frontends (MF), which is a “microservice”-like alternative to the Frontend-Monolith. Putting MF into practice is currently difficult because there is a lack of experience with this architectural style. If MF are used incorrectly, they often bring more disadvantages than advantages.

This work aims to compare different technologies to find out which of these technologies are best suited for MF.
A comparison of 5 different MF frameworks (module federation, single spa, piral, podium or mashroom server) and a comparison of monorepo and multi-repo, based on the weighted scoring method, finally allows module federation in combination with a Monorepo to emerge as the clear winner. In addition, solutions were discovered that counteract the disadvantages of a micro frontend architecture (MFA).

Keywords: Micro Frontends, Frontend Architecture, Module Federation, Microservices

1. Einleitung

Dieses Kapitel beginnt mit der Herausforderung, die sich diese Arbeit stellt, gefolgt von der Motivation hinter dieser Bachelorarbeit. Nach der Nennung des Zielpublikums und des Nutzens für den Leser wird ein Überblick über das allgemeine Ziel dieser Arbeit gegeben. Abschließend erhält der Leser einen Überblick über die im Rahmen dieser Bachelorarbeit verwendeten Methoden.

1.1 Die Herausforderung

Da es im Zusammenhang mit MF noch einige offene Fragen und viel zu erkunden gibt, stellt sich diese Arbeit der Herausforderung, den Umgang mit MF zu erleichtern und möchte dabei vor allem nach Lösungen suchen, welche den Nachteilen dieser Software-Architektur entgegenwirken.

Dabei soll neben der Entwicklung eines Prototyps, die Weighted Scoring Methode helfen, Technologien zu vergleichen, um am Ende den idealen Entwicklungs-Stack für MF zu finden.

Da die MFA ein riesiges Gebiet in der SW-Entwicklung darstellt, ist es wichtig, das Ökosystem dahinter bestmöglich zu analysieren, damit am Ende die richtigen Entscheidungen getroffen werden können.

1.2 Motivation

Das Internet verändert sich drastisch, ursprünglich stellte das Web eine Dokumentenplattform dar, die vollständig aus Hypertext Markup Language (HTML) bestand. Heute leisten Webanwendungen viel mehr Arbeit, als lediglich statische Daten anzuzeigen. Moderne Webseiten müssen heute nicht nur interaktiv sein, um den Ansprüchen des Users gerecht zu werden, sondern müssen auch mit ungeheuren Datenmengen umgehen können. Man könnte argumentieren, dass Websites heutzutage eher Anwendungen sind, die vorgeben, Websites zu sein. Wir können sie verwenden, um Nachrichten zu senden, Online-Informationen zu aktualisieren, einzukaufen und vieles mehr.

Der schnelle Wandel des Internets sorgt dafür, dass auch die Frontend-Entwicklung immer komplexer wird. Webanwendungen von heute müssen schnell geladen werden, auf jedem Gerät reaktionsschnell sein und schnell auf Benutzerinteraktionen reagieren, was eine sorgfältige und durchdachte Entwicklung unerlässlich macht. Wenn die Anwendung klein genug ist und nur eine Handvoll Entwickler/-innen daran arbeiten, ist das Erstellen einer netten Webanwendung eine überschaubare Aufgabe. Bei größeren Projekten stößt ein Monolith jedoch schnell auf Skalierungsprobleme, siehe Kapitel 2.3.1.

Die Einführung moderner Frontend Frameworks ermöglichte es, auch komplexe Web-Anwendungen bei „überschaubarem“ Aufwand zu entwickeln, um schließlich den stetig steigenden Nutzeranforderungen gerecht zu werden. Das sorgte dafür, dass die Qualität und der Umfang einer Frontend Applikation von Jahr zu Jahr stiegen, sodass die dahinter liegenden „Codebases“ immer größer wurden. Nun ist die Branche an einen Punkt gelangt,

an dem man auf Skalierungsprobleme stößt, welche innerhalb einer Monolithen Architektur nicht mehr einfach zu lösen sind. Es wird immer schwieriger, die Frontend-Entwicklung so zu skalieren, dass viele Teams gleichzeitig an einem großen und komplexen Produkt arbeiten können.
An dieser Stelle kommen MF ins Spiel, welche einen neuen Architekturstil in der Frontend-Entwicklung darstellen. Diese Versprechen, Probleme innerhalb großer Softwareprojekte zu lösen, ähnlich wie es „microservices“ bereits seit vielen Jahren erfolgreich im Backend machen, mehr dazu in Kapitel 2.3.2.

Eine Webanwendung besteht im Grunde aus 3 Schichten, einer Datenbank, einem Backend und einem Frontend. Die Datenbank ist für das persistente Speichern der Daten zuständig. Das Backend ist für die Verarbeitung und für die Bereitstellung dieser Daten zuständig, was bedeutet, dass es auf Clientseitige (vom Browser bzw. Frontend kommende) Datenanfragen reagiert.
Das Frontend empfängt diese Daten und bereitet diese strukturiert, sinnvoll und leserlich für den Nutzer auf, damit dieser die gewünschten Inhalte auf einem Gerät seiner Wahl (Smartphone, PC, Notebook, Tablet, TV…) mittels Browser empfangen kann.

Im „Persistence-Layer“ (Datenbank) und API Layer (Backend) werden seit einigen Jahren erfolgreich “microservices” verwendet. Das bedeutet, dass man jedem noch so kleinen Anwendungsfall einen eigenen Service zuordnen kann, der völlig unabhängig und selbstständig agiert. Ein Beispiel für solch ein Service könnte ein Authentifizierungsservice sein, welcher lediglich dafür zuständig ist, einen Nutzer zu authentifizieren (Login, Register). In einer MSA bedeutet das, dass ein Backend Service nur für diesen Teil der Domain definiert wird, welcher komplett getrennt vom Rest der Anwendung betrieben werden kann. Das MSA Prinzip wird auch für Datenbanken angewandt, somit bekäme jede Teil-Domäne (subdomain) eine eigene Datenbank zugeordnet (Nutzer-DB, Produkt-DB, Bilder-DB). Dieser Architekturstil ermöglicht es nicht nur, die Services beliebig miteinander zu kombinieren, sondern können dadurch große Projekte in kleine überschaubare Arbeitspakete unterteilt werden, welche man eigenen Teams zuordnen kann.

Im Frontend stellt die MSA jedoch die Ausnahme dar, was bedeutet, dass der Großteil aller Frontends einem monolithischen Ansatz folgt, bei dem die gesamte Anwendung ein einziges zentrales Element darstellt. Diese Tatsache sorgt dafür, dass die M in der Praxis noch sehr unerprobt sind. Das schmälert jedoch nicht das ungeheure Potenzial, das in diesem Architekturstil steckt. Wenn richtig angewendet, sind MF durchaus in der Lage, Probleme aus der Enterprise Frontend Entwicklung zu lösen.
Wenn wir von MF sprechen, sprechen wir von einem Architekturansatz auf der Frontend-Seite (Präsentations-Schicht) einer Anwendung. Die Bezeichnung MF tauchte erstmals 2016 durch die Firma Thoughtworks im Zuge eines Technologie Radars auf (Thoughtworks, 2016). Dieser Ansatz ermöglicht es, einen großen Frontend-Monolithen in mehrere eigenständige und unabhängige Anwendungen aufzuteilen, ähnlich wie man es innerhalb einer MSA mit Backend Services macht. Durch die starke Inspiration durch die MSA findet man auch in MF viele Ähnlichkeiten.
Dass MF in den letzten Jahren an Popularität gewonnen hat, zeigt die hohe Adoptionsrate großer Technologieunternehmen (Havro IT Solutions, 2022). Nach Analyse einiger MF-Open-Source-Projekte auf GitHub, tauchten bekannte Firmennamen wie Netflix, Microsoft, Amazon, DAZN, Reddit, Epic Games, SAP, Zalando, Sony, Alibaba, Tencent auf, um nur ein

paar davon zu nennen (Mezzalira, 2021). Auch Luca Mezzalira, Autor des Buches „Building micro frontends“ erwähnt einige dieser Unternehmen und deren Bemühungen rund um MF. Die Beliebtheit und kontinuierliche Unterstützung eines technologischen Trends bzw. Open-Source-Projekts spielt dabei eine ganz wesentliche Rolle. Frameworks, Bibliotheken und alle anderen Tools sind heute größtenteils Open Source und nur durch dessen „Contributor“ und dessen Community, die sich rundherum aufbaut, machen solche Projekte langfristig erfolgreich. Die hohe Adoptionsrate von MF durch Technologie Giganten ist zusätzlich ein guter Hinweis auf zukünftige Trends.

Folgende Grafik zeigt, wie zahlreich Open-Source-Beiträge großer Tech-Unternehmen waren (Zandt, 2021).

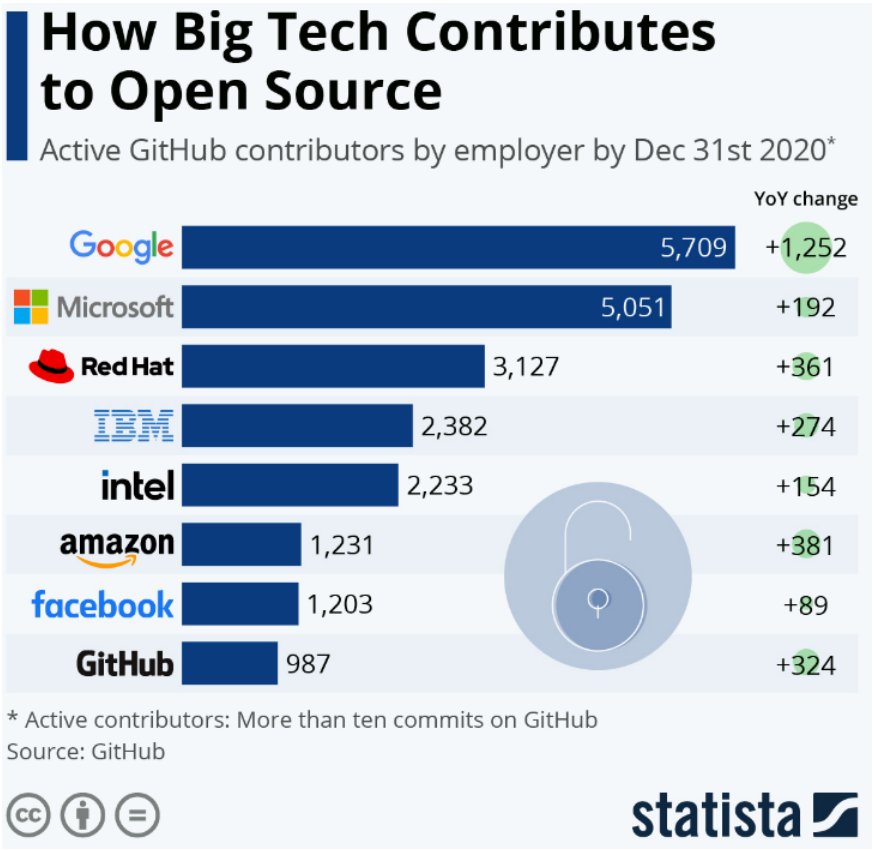


Abbildung 1.1: Open Source Beitrag, großer Tech-Unternehmen (Zandt, 2021).

Obwohl MF immer beliebter werden, findet man auch heute noch bei der Mehrheit der Softwareunternehmen einen monolithischen Ansatz, um ihre Frontends zu erstellen. Da die Komplexität des Clients ständig zunimmt, werden Web-Apps heute nur noch selten in reinem JavaScript erstellt und moderne Frontend-Frameworks zur Hilfe genommen. Langsam gerät das “Frontend Field” an einen Punkt, an dem selbst auf moderne Frameworks basierte Web-Apps immer größer und komplexer werden, welche sich trotz moderner Technologien an die Grenzen eines Monolithen stoßen. MF versuchen diese Skalierungsprobleme im Frontend zu lösen.
Zweifellos liegt viel Potenzial darin, eine große Codebasis in kleinere, überschaubare Teile zu zerlegen, jedoch gibt es keinen typischen Weg zur Implementierung von MF. Die Schier an möglichen Konstellationen einer MFA ist dabei sicherlich ein Grund, warum noch so viel Verwirrung rund um MF herrscht. Ein roter Faden ist nur schwer ausfindig zu machen und

lediglich ein paar sogenannte “Best practices” helfen bei der Reise nach der idealen Micro Frontend Architektur (MFA). Während der Implementation von MF, merkt man schnell, dass man bei vielen Entscheidungen auf sich allein gestellt ist, da es nur sehr wenige Anhaltspunkte für erfolgreiche MFA gibt. Da sich dieser Architekturstil noch so neu ist, kann sich auch in einem Jahr wieder alles ändern und schließlich weiß man nie genau, wohin einen die Reise führt. Diese stetige Ungewissheit macht es schwer langfristig zu planen, jedoch gibt mir das die Motivation einen Beitrag zu der stetig wachsenden MF Community zu leisten.

1.3 Nutzen für Dritte

Diese Arbeit richtet sich an Personen, die sich für MF interessieren, diese bereits einsetzen oder auch an Personen, die unentschlossen sind und nach einer Lösung suchen, um ihre Frontend Anwendung zu skalieren. Da sich MF auf den “Frontend-Teil” von Software-Architekturen bezieht, könnte diese Arbeit besonders für Frontend-Entwickler/-innen oder Software-Architekten und Software-Architektinnen interessant sein. Da MF mit einem großen organisatorischen Mehraufwand einhergehen, ist es auch wichtig, dass Product-Owner, Business Analysts, Scrum-Master und andere Teammitglieder wissen, wer in einer MF-Umgebung arbeiten soll. Die Orchestrierung eines kleinen autonomen Teams, das alles hat, was es braucht, um Werte für den Kunden zu schaffen, führt zu Wissensgebieten, die weit über eine rein technische Angelegenheit hinausgehen und deshalb könnte diese Arbeit auch für Personen ohne einschlägigen technischen Hintergrund interessant sein.

1.4 Ziel

Diese Arbeit zielt darauf ab, unterschiedliche Technologien miteinander zu vergleichen, um herauszufinden, welche dieser Technologien am besten für MF geeignet sind.

Da man für die Entwicklung von MF einen ganzen Stack an Technologien benötigt, wird zusätzlich zu einem Frontend-Framework auch ein MF-Framework benötigt (abgesehen von der Tatsache, dass man gar keine Frameworks verwenden muss).
Auch die Entscheidung, welches Code-Repository man verwenden möchte, muss getroffen werden. Es gibt zwar noch viele weitere Entscheidungen, die bei der Entwicklung einer MFA zu treffen sind, jedoch möchte sich diese Arbeit auf 2 Teile konzentrieren: Code-Repository-Type und MF-Framework. Daher verfolgt diese Arbeit 2 grundlegende Ziele:

Das erste Ziel dieser Arbeit ist es herauszufinden, welcher Repository-Typ (Monorepo oder Multi-Repo) besser für Micro Frontends geeignet ist.

Das zweite Ziel dieser Arbeit ist es herauszufinden, welches der 5 Micro-Frontend-Frameworks (Module-Federation, Single-spa, Piral, Podium oder Mashroom-Server) besser für Micro Frontends geeignet ist.

Um die Ergebnisse der Weighted Scoring Methode zu untermauern, wird anhand des Prototyps geprüft, ob dieser am Ende auch in der Lage ist, häufig auftretende Probleme im Umgang mit MF zu lösen. Aus meiner Sicht bringt es nur wenig die beste Technologie zu

küren, welche am Ende nicht in der Lage ist, häufig auftretende Probleme im Umgang mit MF zu lösen.

Zu Beginn ist es wichtig, den aktuellen Wissensstand der Forschung und Technik rund um MF einzufangen. Dabei gilt es ein theoretisches Fundament aufzubauen, welches möglichst Technologie-unabhängig ist und als Leitfaden für jede Art von MF herangezogen werden kann.

Es ist wichtig, dass es sich bei den Technologien um ein Toolset handelt, welches die Implementierung von MF möglichst einfach hält, hohe Flexibilität bietet und von einer aktiven Community gestärkt wird. Zusätzlich sollte dieses Toolset auch mit den beliebtesten Frontend-Frameworks wie React, Angular und Vue kompatibel sein. Da das Web sehr stark von „Single-Page-Applications“ (SPA, siehe Kapitel 2.2.1) geprägt ist, ist die Unterstützung dieser Frameworks essenziell.

1.5 Forschungsfrage und Hypothese

Diese Arbeit versucht folgende Forschungsfragen zu beantworten:

Welcher Code-Repository-Type (Monorepo oder Multi-Repo) ist für Micro Frontends besser geeignet?

Welches Micro-Frontend-Framework (Module-Federation, Single-Spa, Piral, Podium oder Mashroom-Server) eignet sich besser für Micro Frontends?

Die Arbeit zeigt, dass folgende Hypothesen bestätigt/nicht bestätigt werden können:

Ein Monrepo eignet sich besser für Micro Frontends als ein Multi-Repo.
Module-Federation ist als Micro-Frontend-Framework besser geeignet als Single-Spa, Piral, Podium oder Mashroom-Server.

Was diese Arbeit **nicht** abdeckt:

Diese Arbeit versucht nicht alle Probleme im Umgang mit MF zu analysieren. Dabei werden zum Großteil Probleme aus eigener Erfahrung und jene, die häufig in der Community genannt werden, untersucht. Da der Autor dieser Arbeit auch im beruflichen Umfeld seit 2 Jahren mit MF zu tun hat, werden auch Probleme, die aus eigener Erfahrung im Umgang mit MF entstanden sind, Untersuchungs Bestandteil dieser Arbeit sein.

Da sich diese Arbeit auf den Frontend-Teil von Software-Architekturen fokussiert, werden nur ausgewählte Themen aus dem Backend behandelt wie “Domain Driven Design” und “Microservices”.

1.6 Methoden Überblick

Um das MF-Framework Module Federation innerhalb eines Monorepos zu testen, wird ein Prototyp entwickelt, welche die erste Methode darstellt.

Für den Vergleich der 5 MF-Frameworks (Module-Federation, Single-spa, Piral, Podium oder Mashroom-Server) sowohl als auch für den Vergleich der beiden Code-Repository-Types (Monorepo und Multi-Repo) wird die Weighted Scoring Methode angewendet.

Da auf die technischen Rahmenbedingungen des Prototyps in Kapitel 3 und 4 genauer eingegangen wird, werden diese hier nur kurz angeführt.

Technische Spezifikation des Prototyps:

- Frontend-Framework: React[1],
- Micro-Frontend-Framework: Webpack Module Federation[2],
- Programmiersprache: TypeScript[3],
- Art des Code-Repositories: Monorepo[4],
- Monorepo Anbieter: Nx[5]

Nun folgt ein detaillierter Überblick über den aktuellen Stand des Frontend Ökosystems.

[1] https://reactjs.org/
[2] https://webpack.js.org/concepts/module-federation/
[3] https://www.typescriptlang.org/
[4] https://en.wikipedia.org/wiki/Monorepo
[5] https://nx.dev/

2. Frontend - Derzeitiger Stand von Wissenschaft und Technik

Dieses Kapitel beginnt mit einem Überblick über das aktuelle Frontend Ökosystem welches stark von JavaScript Frameworks geprägt ist. Zu Beginn wird der Einfluss dieser Frameworks auf das Web untersucht, gefolgt von einer detaillierten Beliebtheitsanalyse. Danach wird auf die verfügbaren Typen einer modernen Webapplikation eingegangen (Single Page Applikation oder Universelle Applikation)
Darauf folgt eine kurze Zusammenfassung der Software Architektur Prinzipien Monolith und Microservices, gefolgt von der Microservices Adaptierung im Frontend.
Der größte Teil, Kapitel 2.5, befasst sich mit dem Architekturstil MF und gibt einen detaillierten Überblick über vier fundamentale Entscheidungen, die man in jeder Micro Frontend Architektur treffen muss. Das letzte Kapitel (Kapitel 3) geht grob auf das Konzept des MF Prototyps ein und begründet die Wahl dieser Methode.

2.1 JavaScript Frameworks

2.1.1 JavaScript Frameworks und deren Einfluss auf das Web

Um herauszufinden, wie der MF Architekturstil in das heutige Frontend Ökosystem passt, muss man sich zuerst ansehen, wie und vor allem womit heute moderne Frontends entwickelt werden.
Das Frontend-Ökosystem, auf das wir heute blicken, ist unglaublich umfangreich. Die Zeiten, in denen ein Browser noch ein fix fertiges HTML vom Backend geliefert bekommt, sind längst vorbei. Lange belief sich das Frontend auf HTML, CSS und ein wenig JavaScript. Damals wurden diese Frontends noch als “thin client” bezeichnet, da sich der Großteil der Logik im Backend abspielte (Mezzalira, 2021). Diese Situation endete jedoch spätestens 2014 als das JavaScript Framework AngularJS[6] die “Single-Page-Application” Ära einläutete.
Was eine SPA ist und woran man solche erkennt, wird in Kapitel 2.2 näher behandelt.

Bei sogenannten Frontend Frameworks oder auch JavaScript Frameworks genannt, handelt es sich um ein Rahmenwerk, das Entwickler/-innen unterstützt, um komplexe User Interfaces zu bauen. Theoretisch kann man auch jede noch so moderne Web-Anwendung in “Vanilla” JavaScript ohne jegliches Framework entwickeln, jedoch müsste man wesentlich mehr Zeit dafür investieren, da man auf jegliche Hilfsmittel verzichtet. Auch wenn Frameworks das Entwickeln anspruchsvoller User Interfaces enorm erleichtert haben, sind die Ansprüche der Nutzer und Nutzerinnen in den letzten Jahren stark gestiegen, was die Webentwicklung im Grunde nicht einfacher gemacht hat. Moderne Webseiten müssen heute nicht nur interaktiv sein, um den Ansprüchen des Users gerecht zu werden, sondern müssen auch mit ungeheuren Datenmengen umgehen können.
Durch die immer höher werdenden Anforderungen wurden JavaScript Frameworks im Frontend immer wichtiger, sodass diese seit Jahren als Standard bezeichnet werden können.

[6] https://angular.io/

2.1.2 JavaScript Framework Beliebtheit Analyse

Lange Zeit konnte sich die bereits 2006 erschienene JavaScript Library jQuery[7] im Frontend behaupten, als diese dann 2014 von dem Angular Framework dominiert wurde. Angular setzte damals den Startschuss für moderne SPA welchem weitere JavaScript Frameworks wie React und Vue[8] folgten. Mit der Steigerung der Beliebtheit an SPA begann auch das Web sich stark zu verändern. Daten basierend auf der Anzahl der „Github Stars“ dieser „Repositories“ zeigen, dass sich die Beliebtheit von JavaScript Frameworks zwischen 2014 und 2019 mehr als verzehnfacht hat (Statistics and Data, 2021). Dieser regelrechte Hype um Single Page Applikation setzt sich auch im Jahr 2022 ungehindert fort.
Folgende Abbildung zeigt die Ergebnisse einer Online-Umfrage zu den beliebtesten Web-Frameworks, die zwischen Mai und Juni 2021 durchgeführt wurde.

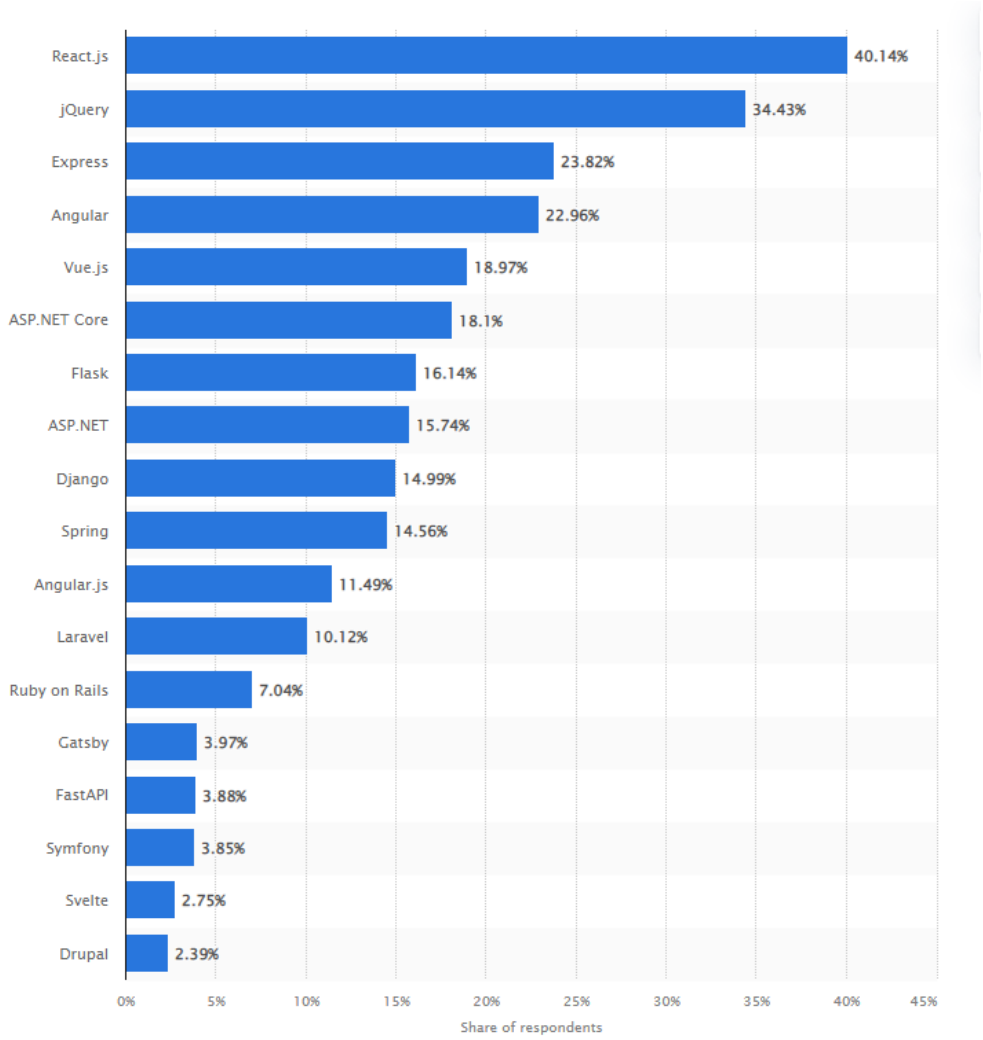


Abbildung 2.1: Online-Umfrage: Beliebteste Web-Frameworks aus 2021 (Statista, 2022)

Wenn man lediglich JavaScript Frameworks berücksichtigt und Backend Frameworks wie Express APS.NET Core weglässt, sieht man, dass man auch im Jahr 2021 von den 3 großen Frontend-Frameworks React, Angular und Vue sprechen kann.

[7] https://jquery.com/
[8] https://vuejs.org/

Wenn man sich die aktuellen Zahlen aus Mai 2022 und die Entwicklungen zum Vorjahr ansehen will, dann ist die Analyse der wöchentlichen Downloads durch den JavaScript Paketmanager npm[9] ein hervorragender Indikator für Beliebtheits Zahlen.

Eine Analyse der wöchentlichen Downloadzahlen zwischen 05.2021 und 05.2022 ergeben bei den JavaScript Frameworks React, Angular, Vue und Svelte folgende Zahlen:

Tabelle 2.1: Wöchentliche NPM Downloadzahlen (eigene Abbildung[10],[11])

Frontend Framework	**Wöchentliche Downloads 05.2022**	**Wöchentliche Downloads 05.2021**	**Zuwachs zum Vorjahr**
React	15,913,392	10,780,547	48%
Vue	3,281,715	2,520,662	30%
Angular	6,408,058	4,832,499	33%
Svelte	308,403	151,375	104%

Auch dieser Vergleich macht deutlich, dass React ganz vorne in der Beliebtheitsskala steht, gefolgt von Angular und Vue. Zum Vergleich wurde auch der Newcomer Svelte miteinbezogen, welcher zwar auf sehr viel Zuspruch stößt, da dieser einiges anders macht, jedoch sind die Downloadzahlen im Vergleich der 3 größten Frontend Frameworks sehr gering.

React ist als Frontend Framework sogar so beliebt, dass ganze Fullstack-Frameworks (Backend & Frontend) auf React aufbauen. Diese Fullstack Frameworks bieten eine Reihe an Zusatzfunktionen an, um auch die serverseitige Entwicklung abzudecken, womit Entwickler/-innen nun in der Lage sind, eine komplette Webanwendung zu entwickeln. Einer der bekanntesten Fullstack Frameworks sind Next.js[12], Remix[13], Blitz.js[14], und Redwood.js[15] Manche dieser Frameworks setzen React voraus, andere Frameworks wie Blitz.js bauen sogar auf React und Next.js auf. Diese Schier an Kombinationen zeigt, wie aktiv das Ökosystem rund um React ist, wobei Gründer von Blitz.js auch kürzlich erwähnten das sie in der Zukunft einen Framework Diagnostischen Ansatz anstreben, um auch Nutzer von Remix, Svelte oder Nuxt bedienen zu können. Bei der riesigen Auswahl an Open Source Frameworks würde vermutlich niemand ahnen, dass aus React Frameworks wie Nextjs mit den Jahren “Multimillion Dollar” schwere Unternehmen entstanden sind.

[9] npmjs.com/
[10] npmjs.com/package/react npmjs.com/package/vue npmjs.com/package/svelte
[11] npmjs.com/package/@angular-devkit/core
[12] https://nextjs.org/
[13] https://remix.run/
[14] https://blitzjs.com/
[15] https://redwoodjs.com/

Um noch einmal auf das Schlusslicht der Tabelle 2.1 zurückzukommen, sollten auch Newcomer wie Svelte, welches beiläufig erwähnt, alles anders als React macht erwähnt werden, da dieses erst kürzlich als das meist beliebteste JavaScript Framework gewählt wurde (Stack Overflow, 2021).
Downloadzahlen geben zwar einen guten Überblick über die Entwicklung in diesem Feld, jedoch sollte man sich nie von Newcomern, bzw. geringen Downloadzahlen abschrecken lassen, da vor allem Neulinge von einer sehr aktiven Open Source Community vorangetrieben werden.

Abbildung 2.2 zeigt eine Umfrage aus 2022 mit dem Titel “State of frontend 2022” an der 3703 Personen aus 125 verschiedenen Ländern teilnahmen. Bei dieser Umfrage wurden die Teilnehmer gefragt, welches Framework sie in der Zukunft lernen möchten. Die verblassten Farben stellen dabei die Verteilungen der Antworten zum Vorjahr 2020/21 dar, woran man gut erkennen kann das immer weniger Personen gewillt sind ältere Frameworks wie Angular und Nuxt.js zu lernen, wohingegen das Interesse zu Newcomern wie Svelte in die entgegengesetzte Richtung entwickelt hat.

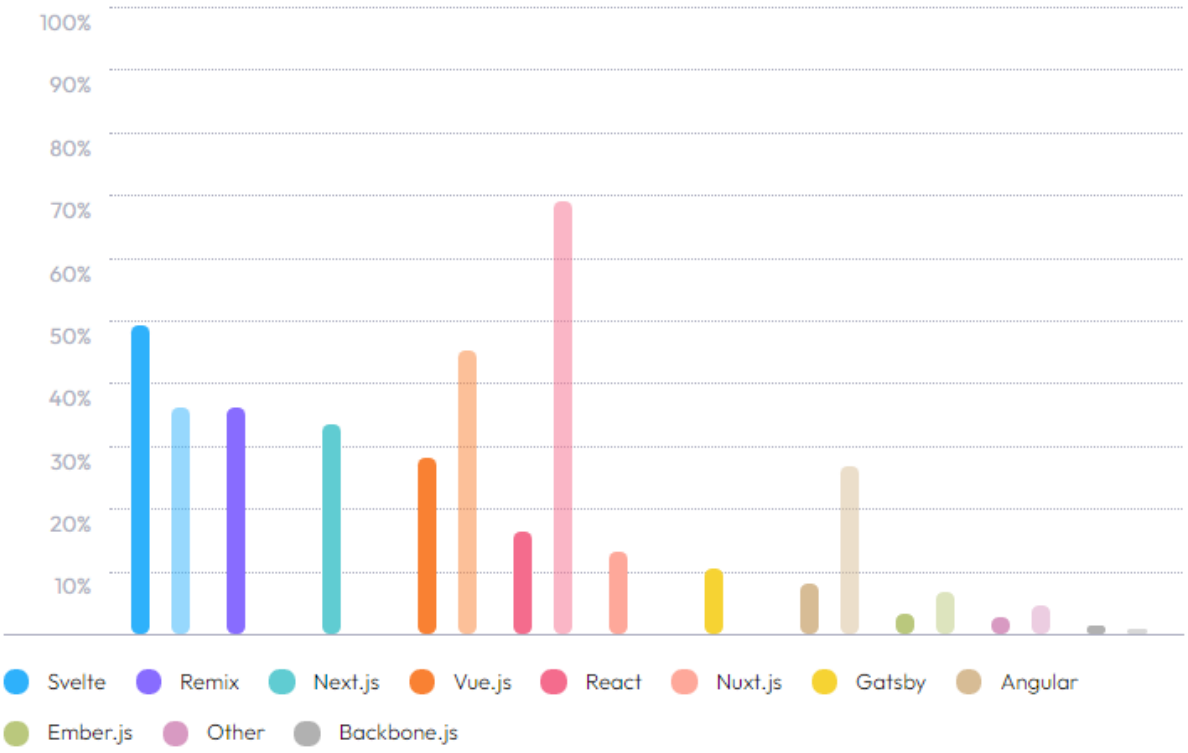


Abbildung 2.2: Umfrage “State of frontend 2022” (The Software House, 2022)

Die schnelle Entwicklung neuer Frameworks ist zudem ein gutes Beispiel für den schnellen Wandel von Frontend-Technologien. Die Technologie die Frontend Entwickler/-innen in einem Jahr gelernt haben, kann im darauffolgendem Jahr schon wieder veraltet sein, womit einige gar nicht umgehen können, andere wiederum schätzen diese stetige Weiterentwicklung und Abwechslung, die einem das Frontend bietet.

In Hinblick auf React kann man jedoch vor einer gewissen Stabilität sprechen, da dieses Framework bereits seit 2016 unangefochten auf Platz 1 der beliebtesten Frontend-Frameworks ist und ihren Vorsprung über die Jahre immer mehr ausbauen konnte. Selbst wenn in einem Jahr keiner mehr neue Apps in React entwickeln würde, dann würde es Jahre dauern, bis der Bedarf für React-Entwickler/-innen zurückginge, da bereits unzählige React Anwendungen live sind, welche regelmäßig durch neue Features erweitert werden, oder gewartet werden müssen.

Ähnlich verhält es sich bei Angular, welches zwar in der steigenden Beliebtheitsskala hinterherhinkt, jedoch in Enterprise Anwendungen und kritischen Infrastruktursystemen wie Banken und Versicherungen nach wie vor sehr beliebt ist. Nach dem Durchforsten etlicher Stellenangebote auf Plattformen wie devjobs, karriere.at und Co., würde ich sogar behaupten, dass Angular in den Banken und Versicherungssektor beliebter als React ist.

2.1.3 JavaScript Alternativen

Obwohl das Verwenden von JavaScript Frameworks heute als Standard gilt, gibt es auch Wege, einem Browser andere Programmiersprachen verständlich zu machen. Web Assembly[16] ermöglicht es mit seinem Compiler auch viele andere Programmiersprachen in JavaScript zu übersetzen und ganz neue Wege zu gehen. Obwohl Web Assembly bereits 2017 erschien, scheint es sich bis heute noch nicht richtig durchgesetzt zu haben, obwohl die Idee, andere Programmiersprachen im Browser zu verwenden, enormes Potential hat. Besonders für Backend-Entwickler/-innen ist dies ein Dorn im Auge, da JavaScript sich doch sehr von statischen Programmiersprachen unterscheidet.
TypeScript ist hier die einzige Ausnahme, welche die Vorzüge einer statischen und typsicheren Sprache ins Frontend bringt. Da TypeScript jedoch am Ende auch zu JavaScript kompiliert und im Grunde ein „Superset“ von JavaScript darstellt, bringt diese neben der dazugewonnenen Typsicherheit keine nennenswerten Vorteile. Trotzdem muss erwähnt werden, dass TypeScript bei Frontend Entwickler/-innen über dem Junior Level durchaus die bevorzugteste Programmiersprache im Frontend ist, was daran liegt, dass die Mehrheit der neuen Open Source Projekte in TypeScript geschrieben werden.

Web-Frameworks fernab von JavaScript wie Django[17] oder Flask[18], welche unter Python laufen, sind zwar weitere Alternativen, gehören jedoch zu serverseitigen Web-Frameworks, weil diese schlussendlich dem Client ein fertiges HTML übergeben und somit mit nur sehr wenig JavaScript auskommen. Ruby on Rails,[19] eine weitere JavaScript Alternative, welche vor einigen Jahren noch sehr beliebt war, ist jedoch ebenfalls im Bereich der serverseitigen Web-Applikationen angesiedelt. Mit ihrer Methode, die Ruby Programmiersprache mit HTML, CSS und JavaScript zu kombinieren, welche schlussendlich auf einem Webserver läuft, stellt diese ebenfalls nur eine serverseitige Alternative zu JavaScript dar.

Das Backend ist durch Cloud und Streaming basierter Dienste ebenfalls stark verändert worden, jedoch sind die technologischen und methodologischen Aspekte viel langlebiger als im Frontend.

[16] https://webassembly.org/
[17] https://www.djangoproject.com/
[18] https://flask.palletsprojects.com/en/2.1.x/
[19] https://rubyonrails.org/

Backend-Frameworks sind im Vergleich zu Frontend Frameworks viel leichtgewichtiger und langlebiger[20],[21],[22]. Programmiersprachen wie Golang eignen sich durch ihre gut sortierte Standardbibliothek[23] auch für Backend-Entwicklungen ganz ohne Frameworks und sind im Zusammenhang mit einem “lightweight router” kaum Grenzen bei der API-Entwicklung gesetzt.

Abgesehen von der großen Vielfalt an Frameworks, Libraries und Tools rund um die Webentwicklung, unterscheidet man moderne Frontend Applikation in lediglich 2 Kategorien. Wenn man heute als CTO, Architekt/-in, Tech-Lead oder Entwickler/-in ein neues Projekt startet, hat man grundsätzlich diese 2 Möglichkeiten: Eine Single-Page-Application (SPA) oder eine universale bzw. isomorphe Anwendung.
Letzteres läuft auf beiden Seiten, am Server und am Client und kann zusätzlich auch mit statischen Seiten kombiniert werden, welche in der Cloud oder vor Ort (on the Edge) gehostet werden.

2.2 Zwei Typen von Webapplikationen

2.2.1 Single-Page-Application (SPA)

Single Page Applikationen (SPAs) bestehen aus einer oder mehreren JavaScript Dateien, welche die gesamte Frontend Applikation abkapseln. Der größte Unterschied zu traditionellen Webseiten ist, dass kein stetiges neu laden des Browsers mehr notwendig ist, da die komplette Präsentationslogik auf dem Client geschieht (Scott, 2015).

Damit der User das Ergebnis solch einer SPA in seinem Browser sieht, werden im Hintergrund verschiedene Schritte durchgeführt, welche jedoch von der Art des Renderings abhängig sind. Abgesehen von der Art des Renderings muss nicht nur das geladene JavaScript, sondern auch CSS und alle weiteren für die Anwendung benötigten Dateien geladen werden, damit daraus schlussendlich eine interaktive Anwendung entsteht, mit die der User mittels Browser interagieren kann.
Da die komplette Präsentationslogik in ein oder mehreren JavaScript Dateien liegt, können Server Transaktionen lediglich Daten beinhalten, abhängig von den Präferenzen für Daten-Rendering. In den meisten Fällen kommunizieren SPA mit APIs, indem sie Daten mit der persistenten Schicht des Backends austauschen, auch bekannt als serverseitig (server-side). Sie vermeiden auch mehrere Rundgänge zum Server zum Laden zusätzlicher Anwendungslogik und rendern alle Ansichten während des Anwendungslebenszyklus.

Diese Funktionen verbessern die Benutzererfahrung und simulieren, was wir normalerweise haben, wenn wir mit einer nativen Anwendung für mobile Geräte oder Desktops interagieren, wo wir von einem Teil unserer Anwendung zum anderen springen können, ohne zu lange zu warten. Bei traditionellen Webseiten muss man hingegen noch nach jeder Interaktion mit der Seite ein neues HTML-Dokument vom Server anfragen.

[20] https://github.com/neiesc/awesome-minimalist
[21] github.com/expressjs/express, github.com/fastify/fastify, github.com/hapijs/hapi
[22] github.com/gin-gonic/gin, github.com/go-chi/chi, github.com/labstack/echo
[23] https://pkg.go.dev/std, https://pkg.go.dev/net@go1.18.3

Das Verwenden von SPA hat viele Vorteile, wie z.B. dass der Browser den Code für die Applikation nur einmal laden muss und daraufhin ist die komplette Anwendungslogik für den User verfügbar und das für die gesamte Dauer der Benutzer Sitzung.

Darüber hinaus verwaltet eine SPA den Routing-Mechanismus vollständig auf der Client-Seite. Dies bedeutet, dass die Anwendung jedes Mal, wenn sie eine Ansicht ändert, die URL auf eine sinnvolle Weise ändert, sodass Anwender/-innen die URL einfach teilen oder als Lesezeichen speichern können, um die Navigation von einer bestimmten Seite aus zu starten.

SPAs ermöglichen Entwickler/-innen auch selbst zu entscheiden, wie man die Anwendungslogik verteilt. Er kann sich für einen “fat client” und einem “thin server” entscheiden wo der client zum Großteil die Logik speichert und der Server als persistente Schicht verwendet dient, oder auch für einen “thin client” und “fat server” wo die Logik an das Backend delegiert wird und der Client keine smarte Logik ausführt, sondern lediglich auf die Statusveränderungen der API reagiert.[24]
Beide Ansätze haben jedoch ihre Vor- und Nachteile. Die beste Wahl hängt immer von der Art der Anwendung ab.

Mezzalira (2019c) fand im Zuge der Entwicklung einer „cross-platform“ Applikation heraus, dass in solch einem Fall ein “thin-client” die bessere Wahl ist. Dieser Ansatz ermöglichte ihm, ein Feature einmalig zu entwerfen und auf mehreren Geräten gleichzeitig serverseitig auf den Anwendungsstatus zu reagieren.
Bei Desktop Anwendungen, wo das Speichern von offline Dateien eine wesentliche Funktion darstellt, verwendete Mezzalira (2019c) stattdessen lieber einen “fat client” und “thin server”. Anstatt den Zustand an beiden Stellen zu managen, bevorzugte er diesen lediglich an einer Stelle zu managen und den Server nur zur Datensynchronisation zu benutzen.[25]

Wie alles in der IT sind auch SPA kein Wundermittel und bringen Nachteile mit sich.
Ein großer Nachteil ist die Performance des allerersten Seitenaufrufes, auch “first contentful paint”[26] genannt. Im Grunde sind SPA sehr schnell in der Navigation, jedoch kann das initiale Laden der ersten Seite länger dauern als mit anderen Architekturen. Dies liegt daran, dass die HTML Seite, die den Browser anfangs erreicht, nur eine Index Seite ist, welche noch über keine präsentativen Inhalte verfügt, sondern lediglich über Links zu all den benötigten Ressourcen verfügt. Das bedeutet, dass zuerst alle möglichen Inhalte geladen werden, anstatt nur das herunterzuladen, was der User sehen muss. Wenn die Anwendung dann auch noch schlecht umgesetzt wurde, kann die Downloadzeit zu einem großen Problem werden, insbesondere wenn sie mit einer instabilen oder unzuverlässigen Verbindung auf mobilen Geräten wie Smartphones und Tablets geladen wird.
Heutzutage kann man den Inhalt auf verschiedene Art und Weise direkt auf dem Client zwischenspeichern, um das Problem zu entschärfen. Neben dem caching[27] gibt es noch weitere Techniken, wie “Code-Splitting” oder “Lazy-Loading” von JavaScript-Bundles, um die Performance von Webseiten zu erhöhen.

[24] Building Micro-Frontends S151
[25] Building Micro-Frontends S151
[26] https://developer.mozilla.org/en-US/docs/Glossary/First_contentful_paint
[27] https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

Auch das Verwenden von progressiven Web-Apps[28] ist eine erwähnenswerte Technik. Progressive Web-Apps bieten eine Reihe neuer Funktionen, die auf Servicemitarbeitern basieren. Ein sogenannter “Service Worker” stellt somit ein Skript dar, das getrennt vom Browser im Hintergrund ausgeführt wird, um Funktionen wie Offline-Erfahrung oder Push-Benachrichtigungen bereitzustellen.

Ein weiterer Nachteil von SPA stellt die Suchmaschinenoptimierung (SEO) dar. Wenn ein Crawler, ein Programm, das das World Wide Web durchsucht, um einen Daten Index zu erstellen, an eine Webseite gerät, wo nicht sofort ein aussagekräftiges HTML vorliegt und erst einmal ein Chunk von JavaScript Dateien ausgeführt werden muss, kann dies für Probleme sorgen. Das Resultat ist dann oft eine Webseite, die in einem Suchmaschinenergebnis ganz weit hinten gereiht wird, weil es dem Crawler schwerfällt alle von der SPA bereitgestellten Inhalte zu indizieren, es sei denn, wir bereiten alternative Wege zum Abrufen vor.

Wichtig zu erwähnen ist, dass die genannten Nachteile der SPA auf deren Render-Methode Client-Side-Rendering (CSR) zurückzuführen sind. Um jedoch die Ladezeit bis zur ersten Interaktion erheblich zu verkürzen und auch die bestmögliche SEO-Wertung zu erreichen, ist man auf Server-Side-Rendering (SSR) gebunden.
Im Zusammenhang mit SPA spricht man in den meisten Fällen von Client-Side gerenderten Applikationen, jedoch besteht die Möglichkeit, Seiten am Server zu rendern und dem Client nur mehr das fertige HTML zu übergeben. Sobald der Code jedoch zwischen Server und Client geteilt wird, spricht man von universellen Anwendungen oder auch isomorphen Anwendungen, über die ich im nächsten Schritt etwas genauer eingehen möchte.

2.2.2 Universelle (isomorphe) Applikationen

Universelle Applikation stellen im Grunde eine Kombination aus zwei oder mehreren Render Methoden dar. Dabei stehen client-side-rendering (CSR), server-side-rendering (SSR) und static-site-generation (SSG) als Render Methode zur Verfügung. Da SPA in der Regel client-side gerendert werden, können diese auch Teil einer universellen Applikation sein.

In der ersten Version des Webs (Web 1.0) waren Webseiten ausschließlich statische HTML Seiten welche bereits am Server zusammengebaut und dem Client (Browser) fix fertig übergeben werden, was Abbildung 2.3 veranschaulicht.

[28] https://en.wikipedia.org/wiki/Progressive_web_application

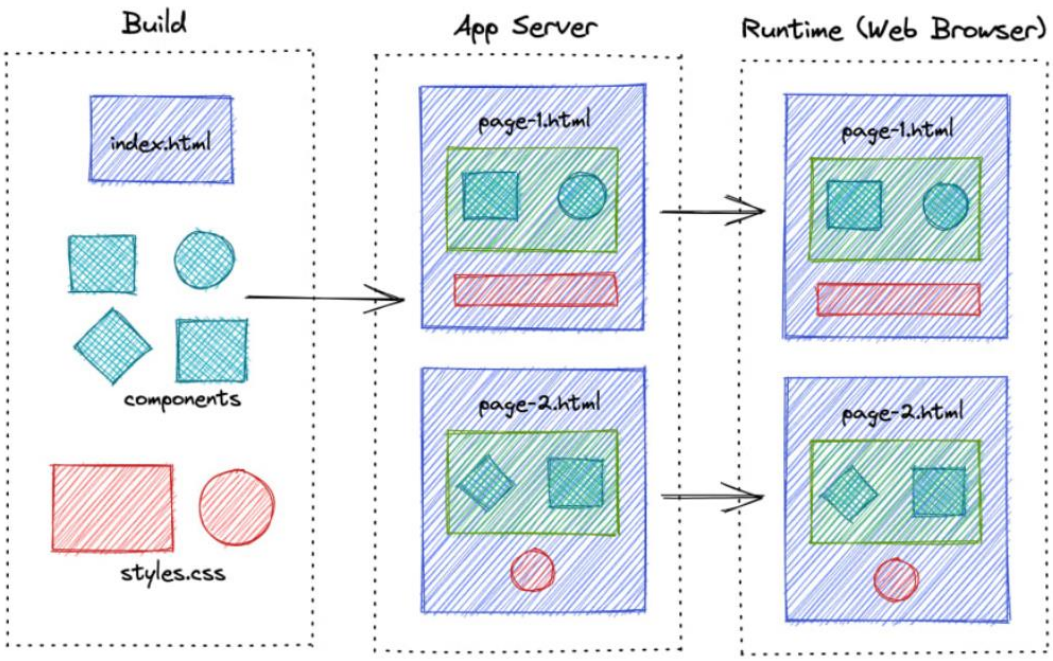


Abbildung 2.3: Server Side Rendering (Perera, 2022)

Erst mit Web 2.0 begann sich das Web von einer rein statischen Lösung zu entfernen, was die Client-Side gerenderte SPA so beliebt machte. Dank CSR war es nun möglich dem Nutzer auch dynamische Inhalte anzuzeigen, was dem Nutzer das Gefühl nach einer interaktiven Anwendung gab. Bei dieser Render-Methode erhält der Webbrowser lediglich eine leere HTML-Datei vom Server. Diese HTML Seite verfügt lediglich über Links zu all den benötigten Inhalten wie JavaScript, CSS, Bilder etc., die vom Webbrowser (Client) nachgeladen und verarbeitet werden. Somit ist der Client selbst für die vollständige und benutzerfreundliche Darstellung der Seite im Webbrowser verantwortlich, was Abbildung 2.4 nun visuell verdeutlicht.

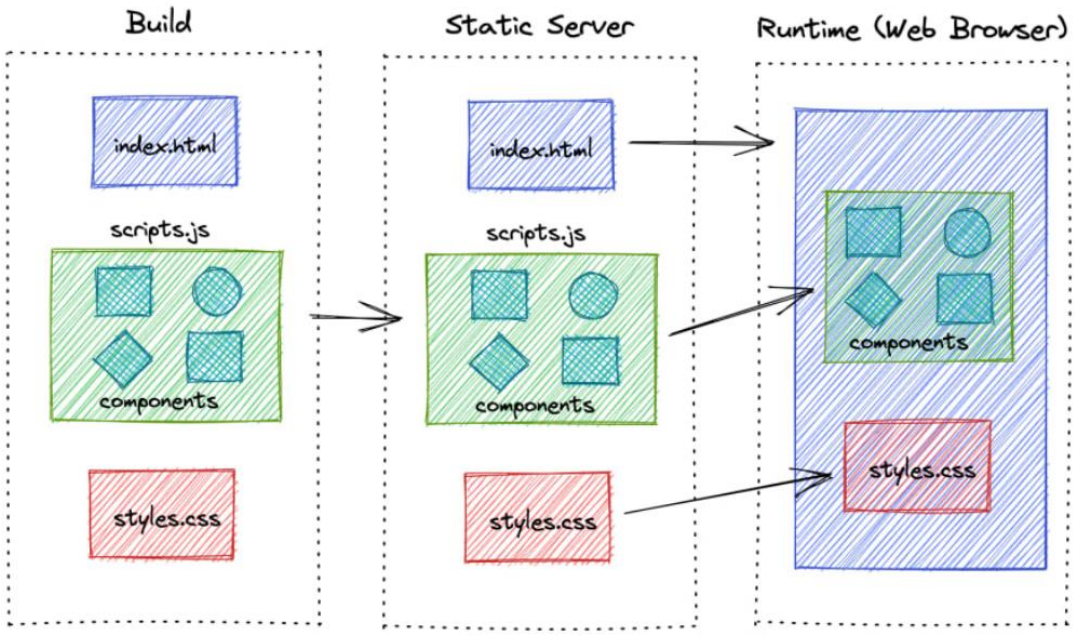


Abbildung 2.4: Client Side Rendering (Perera, 2022)

Mit zunehmender Beliebtheit von SPA, wurde SSR immer uninteressanter. Die Client-Side gerenderte Neuheit namens SPA führte die User in eine interaktive Welt von Web-Anwendungen, fern von statischem Server-side gerenderten HTML.

Da das Schlagwort statisch, genau das Gegenteil von dynamischem Content darstellt, ist dies der ideale Übergang um die dritte und letzte Methode des Rendering SSG vorzustellen: Bei SSG wird, wie bei SSR der Client mit einer fix fertigen Seite versorgt mit dem einzigen Unterschied das diese Seite bereits statisch am Server liegt und dort gar nicht erst gerendert werden muss, weil die Anwendung in einem zuvor passierenden “Buildprozess” zusammengebaut wird, sodass sie wie jede andere statische Datei in einem Content Delivery Network[29] (CDN) von einem Client konsumiert werden kann.

An Abbildung 2.5 kann man erkennen, dass der Zustand des Inhaltes (in dem Fall eine statische HTML Seite) bereits am Ende des Build Prozesses erreicht wird, welcher problemlos auf einem statischen Server bzw. CDN zum Konsum bereitgestellt werden kann.

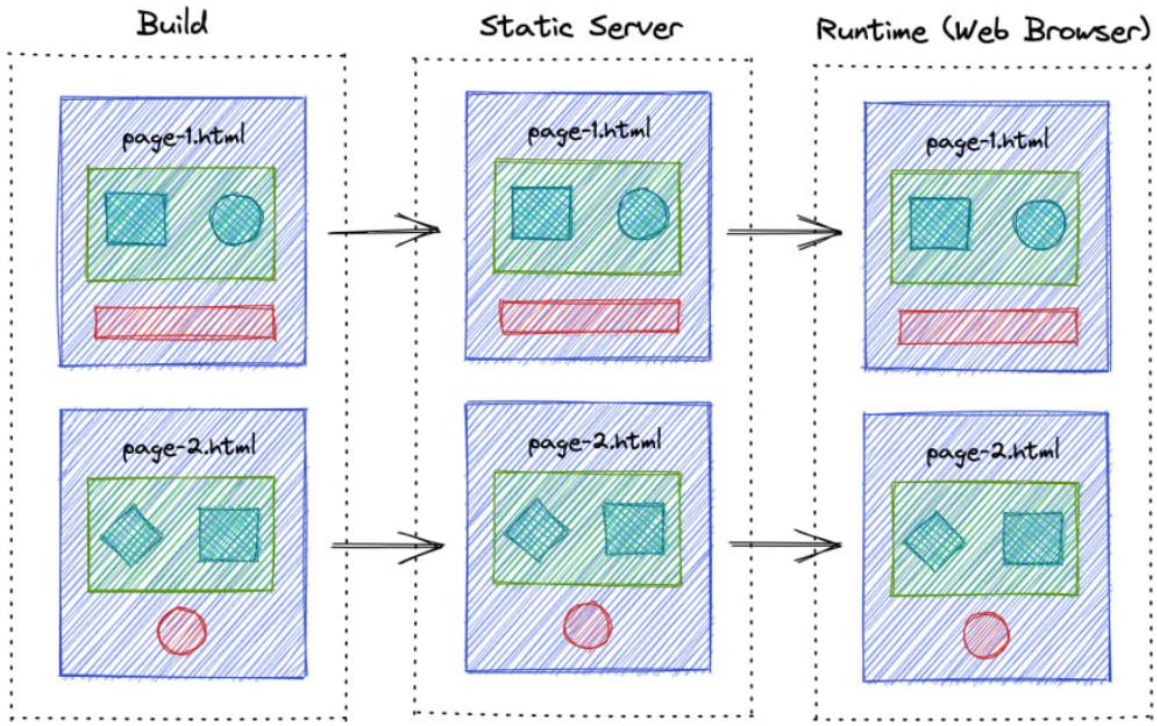


Abbildung 2.5: Static Site Generation (SSG) (Perera, 2022)

Diese Statischen Seiten werden auch als “pre-rendered sites” bezeichnet, da diese bereits vorgerendert am Server landen. Diese Methode hat den Vorteil, dass Inhalte sehr schnell im Browser geladen werden können. Das bedeutet, dass die jeweiligen Seiten bereits zur “build time” generiert werden und ihre Inhalte sich nicht ändern, es sei denn, man fügt neue Inhalte oder Komponenten hinzu und erstellt diese dann neu, was bedeutet, dass die Webseite neu erstellt werden muss, wenn man die Inhalte aktualisieren möchte. Diese Variante eignet sich somit besonders gut für Inhalte, die sich kaum ändern, wie z.B. Artikel und Blogposts.

[29] https://en.wikipedia.org/wiki/Content_delivery_network

Um den Verwendungszweck der jeweiligen Render Methoden besser einordnen zu können, gibt ein Vergleich dieser 3 Methoden in Abbildung 2.6 einen groben Überblick über deren Stärken und Schwächen anhand wichtiger Performance Metriken.

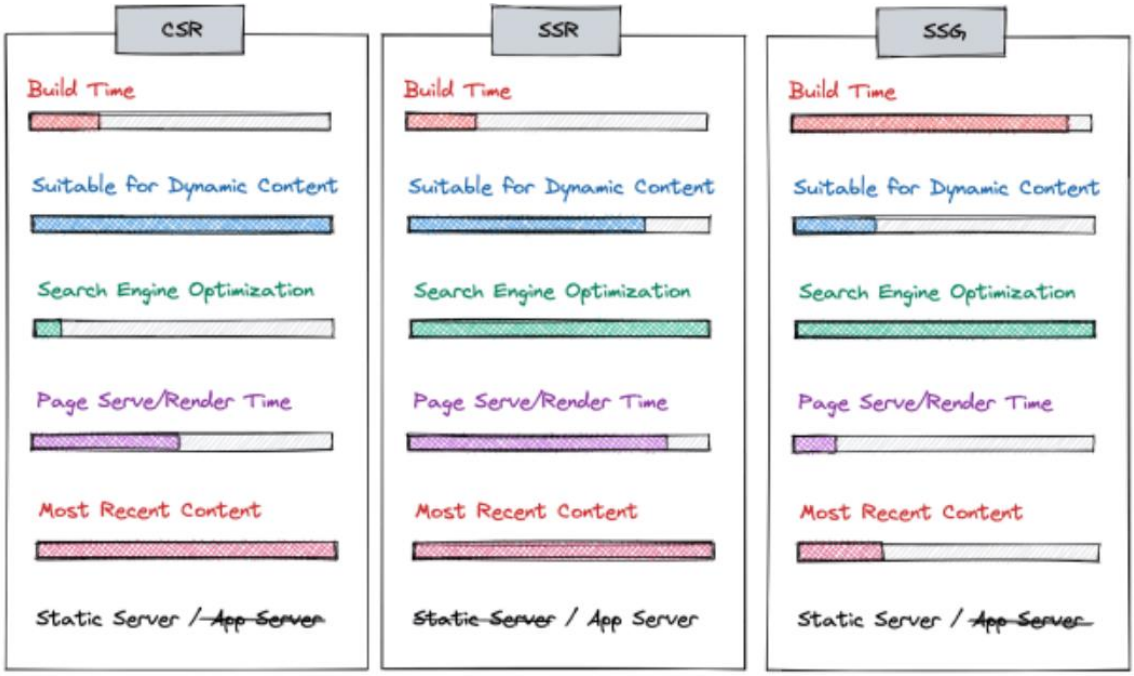


Abbildung 2.6: Render Methoden Vergleich aufgrund gängiger Web Metriken (Perera, 2022)

Streng genommen ließe sich SSG noch durch die Methode “Incremental Static Regeneration” erweitern wodurch die Inhalte nur für eine bestimmte Zeit „gecached“ werden und nach Ablauf eines selbst bestimmten Zeitfensters die Seite automatisch im Hintergrund neu erstellt wird, sodass diese am Ende die in der Zwischenzeit geänderten Inhalte aktualisiert zur Verfügung stellt.

Zusammengefasst sind CSR und SSR am besten für hochdynamische Webanwendungen geeignet, während SSG und ISR am besten für statische Inhalte geeignet sind. ISR ist zwar fortschrittlicher und optimierter als SSG, erfordert jedoch bestimmte Plattformen, um zu funktionieren.
Die Trennung zwischen CSR, SSR und SSG fand so lange statt, bis die ersten Frontend-Frameworks anfingen, alle 3 Arten des Renderings zu unterstützen. Ein Beispiel hierfür wäre wie das bereits in Kapitel 2.1 erwähnte React Framework Next.js, welches bereits 2016 in der Version 1.0 erschien und Client-Side gerenderte Anwendungen um SSR erweiterte.
Erst mit 2020 fing Nextjs an auch SSG zu unterstützen und vereinte somit als erstes Framework alle 3 Render Methoden miteinander. Heute gibt es neben Next.js einige andere sehr beliebte Frameworks, welche SSR oder SSG anbieten wie Gatsby, Hugo oder Remix, wobei Remix als einziger nur SSR anbietet. Remix verfolgt verglichen mit einem Riesen wie Next.js einen viel simpleren und zugleich hoch innovativen Ansatz, der viel hochgradig von den nativen Web-Funktionen eines Browsers profitiert, anstatt diese wie viele andere zu umgehen.

Remix ist das beste Beispiel, welches zeigt, dass man auch moderne Webseiten bauen kann, indem man sich voll auf Web Standards setzt, anstatt das Rad stets neu erfinden zu wollen und geht mit großem Schritt voran, Webentwicklung wieder einfacher zu machen.

Zu guter Letzt muss im Zuge des Mottos “Make the web simple again” noch ein sehr beliebter Technologie Stack erwähnt werden, denn viele Entwickler/-innen schätzen den einfachen Umgang damit. Dieser wurde ursprünglich als „JAMstack“ bezeichnet, wobei „JAM“ für JavaScript, API & Markup stand. Mathias Billmann CEO und Co-founder von Netlify beschreibt Jamstack[30] als "Eine moderne Webentwicklung Architektur basierend auf clientseitigem JavaScript, wiederverwendbaren APIs und vorgefertigtem Markup".
Im Grunde bietet der Jamstack 2 große Vorteile. Zum einen versorgt dieser den User mit schnell verfügbaren Inhalten, da dieser zum Großteil vorbereitet werden kann. Zum anderen steckt dahinter viel mehr ein Versuch, das Web wieder einfacher zu machen, was wiederum den Entwickler/-innen zugutekommt. Passend dazu stieß ich auf ein interessantes Zitat, was hervorragend zu einer immer komplizierter werdenden Frontend-Welt voller Frameworks passt.

“For any technology, the hardest part is not establishing simplicity, but protecting it over time.” - Matt Biilmann, CEO of Netlify[31]

2.3 Vom Monolithen zu Microservices

Ein fehlendes Puzzlestück fehlt jedoch noch in diesem Ökosystem, eine Lösung, die es ermöglicht, Projekte, an denen zehn oder gar hundert Entwickler/-innen beteiligt sind, zu skalieren.

Frederick Brooks (1975), ein amerikanischer Computer Architekt, hat bereits 1975 erkannt, dass an einem gewissen Punkt, das Hinzufügen von Entwickler/-innen das Projekt nicht schneller voranbringt, sondern dieses sogar verzögert. Dies beschreibt er in seinem bereits 1975 erschienenen Buch “The Mythical Man-Month: Essays on Software Engineering“, welches sich neben der Software-Entwicklung auch dem Projekt Management widmet.

Wenn es also einen Weg gäbe, ein großes Projekt in mehrere kleinere Projekte zu verwandeln, würde das nicht bedeuten, dass man damit Brooks (1975) Law umgehen könnte? So ganz nach dem Ansatz: Wenn du vor einem komplexen Problem stehst, beginne mit einer kleinen Version dieses Problems.
Um der Antwort auf diese Frage näher zu kommen, sollte man erst einen Schritt zurück machen und einen Blick auf Software-Architekturstile wie jene des Monolithen und Microservices (MS) werfen.

Während sich besonders in Bezug auf MF auch der Blick in Microservice Prinzipien lohnt, sollte man um MF richtig einordnen zu können erst verstehen, was ein Monolith ist, und warum daraus der Bedarf von Microservices entstanden ist.

[30] https://jamstack.org/
[31] https://www.netlify.com/blog/2021/04/14/distributed-persistent-rendering-a-new-jamstack-approach-for-faster-builds/

2.3.1 Monolith

Wenn alle Funktionalitäten eines Projekts in einer einzigen Codebasis vorhanden sind, wird diese Anwendung als monolithische Anwendung bezeichnet.

Gall (1975), erwähnt in seinem Buch “Systemantics: How Systems work and especially how they fail” sehr ausführlich wie schwer komplexe Systeme zu designen sind und ist der Meinung, dass die Greifbarste und wirkungsvollste Methode ein Projekt zu starten die Beste ist.

A complex system that works is invariably found to have evolved from a simple system that works. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work. – John Gall (1975, p.47)

Galls Gesetz hilft es uns zu verstehen, dass man immer zuerst mit einem kleinen funktionierenden System starten sollte, auf welches man kontinuierlich aufbaut. Wenn das System komplex ist, sagt uns Gall (1975), dass es nicht möglich ist, eine perfekte Lösung auf Anhieb zu entwickeln. Je mehr man also Galls Theorien Glauben schenkt, umso sinnvoller mag ein Monolith erscheinen. Aus meiner Erfahrung neigen ohnehin viele Entwickler/-innen zu “Over Engineering”. Ich bin der Überzeugung, dass selbst das ausgeklügeltste System vor keinem “rewrite” sicher ist, da Software erst durch die stetige Iteration und Reflexion an Robustheit und Qualität gelangt.

Auch wenn ein Projekt bereits Microservices im Backend verwendet, steht dieser vorwiegend einem Monolithen im Frontend gegenüber, was Abbildung 2.7 verdeutlicht.

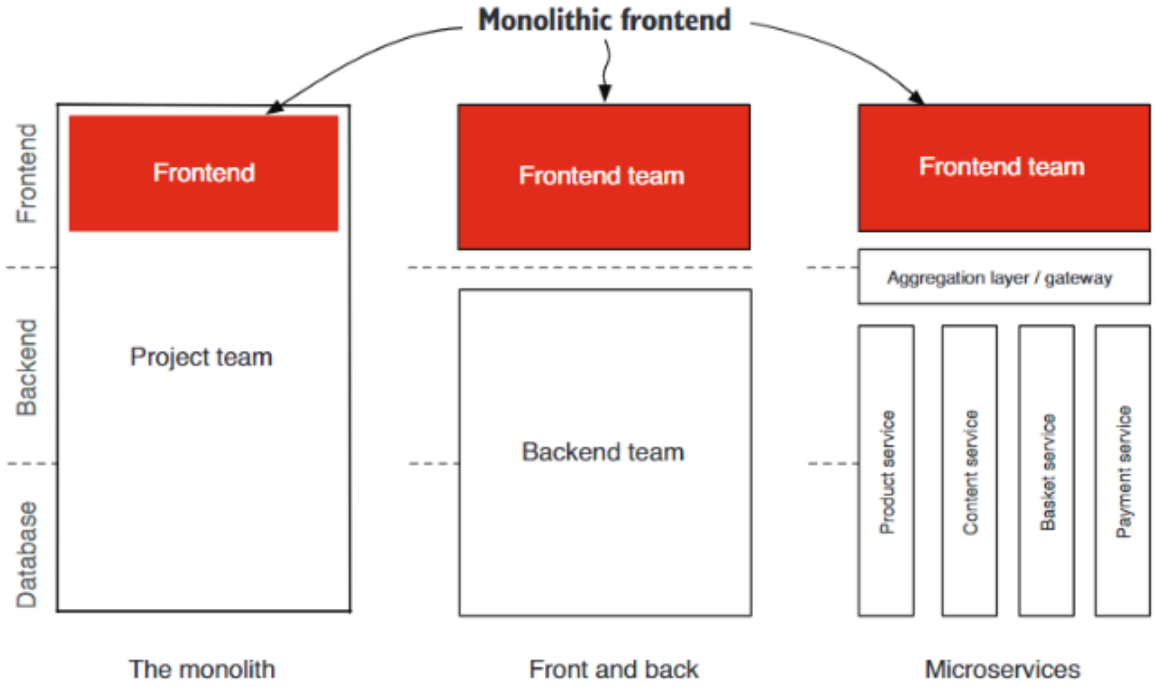


Abbildung 2.7: In den meisten Architekturen ist das Frontend ein monolithisches System (Geers, 2020)

Von einer Norm, welche Microservice ähnliche Strukturen auch im Frontend vorsieht, sind wir, was Abbildung 2.7 verdeutlicht, noch sehr weit entfernt. Selbst wenn die MSA in der Datenbank sowohl als auch im Backend Layer praktiziert wird, bedeutet das noch lange nicht, dass auch das Frontend in solch einer Architekturstil anzufinden ist. Einige große Softwareunternehmen wie Amazon, Netflix, DAZN, Microsoft, Zalando, Sony, Alibaba, Tencent, Reddit und Co.(Havro IT Solutions, 2022) verwenden zwar bereits MF, einige davon sogar Webpacks Module Federation (Github, 2022), trotzdem besteht der Großteil der Frontends auch heute noch aus einem monolithischen System. Das liegt zum Teil daran, dass sich der microservice Ansatz noch nicht ansatzweise so stark durchgesetzt hat wie im Backend. Ein Grund dafür ist vermutlich die große Zeitspanne, die zwischen der Einführung der beiden Terminologien liegt. Dabei sind aber durchaus auch einige Prinzipien aus der MSA für das MF von Bedeutung. Auch wenn die Wissensübertragung zwischen Backend und Frontend sehr schlecht sein mag, gibt es in Bezug auf MF einiges von der MSA zu lernen. Man könnte vermuten, dass es ohne Microservices heute gar keine Micro Frontends gäbe.

Trotz allem Lob, welches Microservices die vergangenen Jahre erhalten haben, hat der Monolith vor allem am Frontend seine Daseinsberechtigung. Oft macht dieser, vor allem am Anfang eines Projektes, mehr Sinn, vor allem wirtschaftlich und zeittechnisch betrachtet. Aus wirtschaftlicher Sicht sollte meiner Meinung nach ein „minimum viable product“ (MVP) keine perfekt ausgeklügelte Architektur beinhalten. In vielen Software-Büchern liest man auch, dass vorzeitige Abstraktionen oder Optimierungen ein Fehler sind, besonders weil sich das Geschäftsmodell vor allem bei neuen Unternehmen schnell ändern kann.

Was die Skalierbarkeit eines Systems betrifft, bringt ein Monolith durchaus Nachteile mit sich. Obwohl Cloud-Systeme es ermöglichten, Projekte einfacher als zuvor zu skalieren, fordern Monolithen, dass man nicht nur einen Teil des Systems, sondern das gesamte System skalieren muss.

Auch die Arbeit großer Entwicklerteams an einer gemeinsamen monolithischen Codebasis kann schnell zur Herausforderung werden, insbesondere nach dem Erreichen mittlerer oder großer Teamgrößen. In Frederick Brooks Buch “The mythical man month” (Brooks, 1975) findet man eine Formel, mit der man ganz einfach die Kommunikationskosten eines Teams berechnen kann, wobei n für die Anzahl der jeweiligen Teammitglieder steht:

Brooks (1975) Gruppen-Interkommunikationsformel: $k = n\,(n - 1)\,/2$.

Um eine ungefähre Idee zu bekommen wie schnell sich das hinzufügen weiterer Teammitglieder negativ auf den Kommunikationsaufwand auswirkt gebe ich ein paar kurze Beispiele:

5 Personen	5*(5–1) /2	=	10
15 Personen	15*(15–1) /2	=	105
50 Personen	50*(50–1) /2	=	1,225
150 Personen	150*(150–1) /2	=	11,175

Diese Formel veranschaulicht sehr gut, wie sich der Kommunikationsaufwand exponentiell erhöht. Womöglich war diese Formel auch der Grund, wieso Jeff Bezos, die in der Softwareindustrie sehr weit verbreitete “pizza rule” eingeführt hat:

“Every internal team should be small enough that it can be fed with two pizzas” – Jeff Bezos, CEO of Amazon

Auf lange Sicht verlangsamen Organisationen mit großen Monolithen normalerweise alle Vorgänge, die für die Weiterentwicklung neuer Features erforderlich sind, und verlieren den großen Schwung, den sie zu Beginn eines Projekts hatten, als alles einfacher und kleiner war, mit wenigen Komplikationen und Risiken.

Bereitstellungen (Deployments) eines Monolithen bergen wesentlich mehr Risiken, da diese mit höheren Wahrscheinlichkeiten einhergehen, die APIs in der Produktion zu beschädigen, insbesondere wenn die Codebasis nicht absolut solide oder ausgiebig getestet ist.

Zusammengefasst sind die Nachteile eines Monolithen:

- Infrastruktur Skalierbarkeit (das gesamte System muss skaliert werden)
- Eine riesige Codebasis
- Hoher Kommunikationsaufwand. Steigt exponentiell mit der Anzahl d. Teammitglieder
- Unflexibel und schwergängig (Langsame Feature Entwicklung)
- Das Projekt verliert schnell an Schwung und sorgt für mehr Komplikationen und Risiken.
- Riskante Deployments (Es droht die Gefahr das gesamte System lahmzulegen)

Um diese und viele andere Herausforderungen zu meistern, mit denen Mitarbeiter konfrontiert sind, könnte ein Unternehmen von komplexen monolithischen Codebasen zu mehreren kleineren Codebasen und bereichsbezogenen Domänen, den sogenannten Microservices, wechseln.

2.3.2 Microservices

“Do one thing and do it well” – M.D. McIlroy 1975 (Haigh, 2017, S.47)

Schon M.D.McIlroy, der Erfinder von Unix Pipelines, hat erkannt, dass es viele Vorteile hat, ein Programm als Teil einer “Pipeline” zu betrachten, dessen Output man problemlos für andere Programme als Input verwenden kann. Auf diese Weise konnten viele kleine Programme zu erstaunlich funktionalen Prozessen verkettet werden. Eine sehr ähnliche Philosophie verfolgen auch Microservices.

Es gibt zahlreiche Behauptungen über die Herkunft des Begriffs Microservices. Auch wenn Peter Rodgers im Jahr 2005 das erste Mal von “Micro-Web-Services” sprach, gibt es vergleichbare Ansätze, die schon viel früher entstanden sind. So entstand bereits 1997 im Hause IBM eines der ersten Bemühungen, kleine unabhängige Services zur Verfügung zu stellen, betitelt mit “Enterprise Java Beans to Service Oriented Architecture” (EJB) (Foote, 2021).
2011 einigte sich eine Gruppe von Software-Architekten in der Nähe von Venedig auf den Begriff "Microservices" (Microservices, 2022). 2012 in Krakow präsentierte James Lewis ein paar dieser Ideen in seiner Fallstudie “Microservices - Java, the Unix Way”(Lewis,2012).

Laut Martin Fowler, dem Software-Guru, der ausführlich über das Thema geschrieben hat, besteht eine Microservice-Architektur (MSA) aus Garnituren unabhängig bereitstellbarer Dienste. Microservices stellen somit eine bestimmte Art des Entwerfens von Software-Anwendungen dar. Kurz gesagt, stellt der Microservice-Architekturstil einen Ansatz zur Entwicklung einer einzelnen Anwendung als eine Garnitur kleiner Dienste dar, welche jeweils in einem eigenen Prozess ausgeführt werden und mit leichtgewichtigen Mechanismen, häufig einer HTTP-Ressourcen-API, kommunizieren (Lewis & Fowler, 2015).

Durch unabhängig bereitstellbare Dienste ist es einem Team möglich, vollständige Eigentümerschaft für einen Dienst zu übernehmen. Da im Vergleich zu einer monolithischen Architektur alle Teile unabhängig voneinander sind, ermöglicht es dem Team, unabhängig von anderen zu entwickeln. Die Codebasis ist bei einem MSA wesentlich überschaubarer, da ein Projekt in kleinere Dienste zerlegt wird. Diese sorgfältige Trennung der einzelnen Geschäftslogik in unabhängige Dienste wird von Entwickler/-innen breitflächig angenommen, da es viel einfacher ist als sich tausende Zeilen von Code anzusehen, was mit einer Verringerung der kognitiven Belastung des Teams einhergeht.
Mezzalira beschreibt einen weiteren wesentlichen Vorteil von Microservices im Bereich der Skalierung. Somit kann nur ein Teil der Anwendung skaliert werden anstatt eines „One-Size-Fits-All“ Ansatzes ähnlich einem Monolithen (Mezzalira, 2021).

Wenn man alle Prinzipien einer MSA in einem Bild zusammenfassen eignet sich hierfür sehr gut die Definition von Newman (2021)

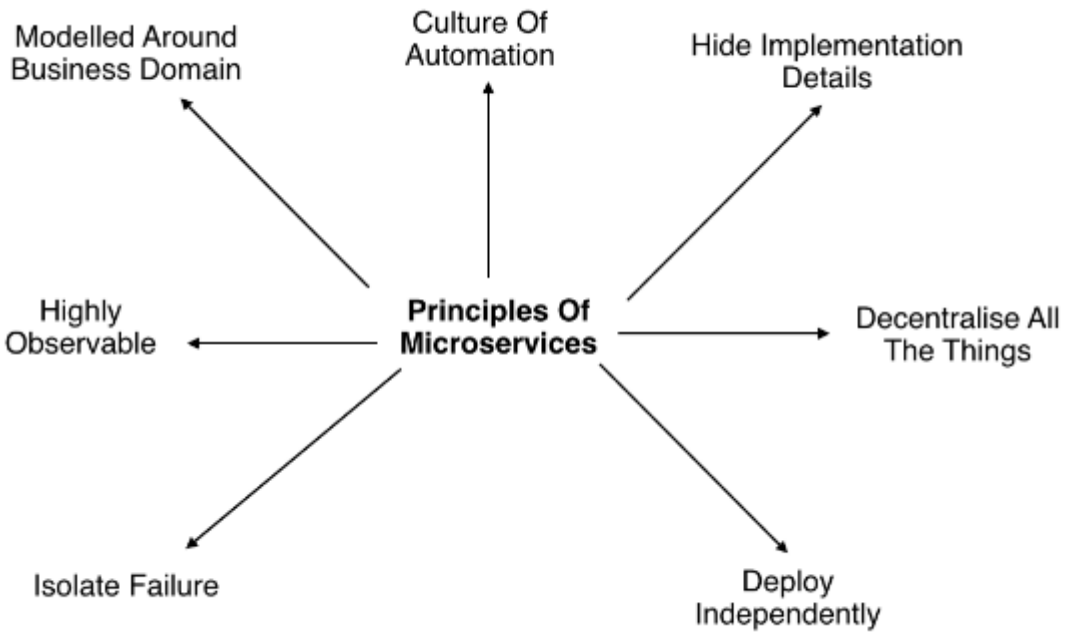


Abbildung 2.8: Principles of Microservices (Newman, 2021)

Je eher diese Prinzipien eingehalten werden, desto robuster und flexibler ist das geschaffene System dahinter und desto mehr entfalten sich die enormen Vorteile der MSA. Da dieses Kapitel die MSA nur grob behandelt, wird hier nicht genauer auf die einzelnen Prinzipien eingegangen.

Mezzalira (2021) erwähnt auch Fallstricke der MSA wie die Investition in Automatisierung, Beobachtbarkeit und Überwachung, welche abgeschlossen sein muss, um ein verteiltes System steuern zu können.Ein weiterer Fallstrick sei die falsche Definition der Grenzen eines Microservices, zum Beispiel ein Microservice zu haben, das zu klein ist, um eine Aktion innerhalb ihres Systems auszuführen, das es sich auf andere Microservices stützt, wodurch eine starke Kopplung zwischen Diensten verursacht wird und sie jedes Mal zusammen bereitgestellt werden müssen. Wenn sich dieses Phänomen auf mehrere Dienste ausdehnt, riskiert man sein Projekt in ein ineinander verwobenes Geflecht mutieren zu lassen, welches aufgrund seiner Komplexität schwer zu erweitern ist.
Wie bereits erwähnt, bringen Microservices nicht nur Vorteile, sondern auch Nachteile mit sich. Deshalb ist es immer ratsam, eine Architektur begründet einzusetzen und sollte nicht leichtsinnig ausgewählt werden, nur weil es der neueste und beliebteste Ansatz ist. Ganz gleich verhält es sich mit Micro Frontends. Obwohl Microservices ursprünglich für Backend Services gedacht waren, möchte ich nun über die Adaptierung der MSA im Präsentations-Layer (Frontend) sprechen.

2.4 Microservices Adaptierung im Frontend

Wie bereits in der Einleitung kurz erwähnt, besteht eine Webanwendung im Grunde aus 3 Schichten: Datenbank (DB), Backend und Frontend. Wenn man sich für eine Architektur der beiden oberen Schichten entschieden hat, muss man sich für eine Architektur und Technologien entscheiden, mit der man die Daten präsentieren möchte. Die folgende Abbildung zeigt den aktuellen Stand einer Web-Applikation mit Microservices und SPA.

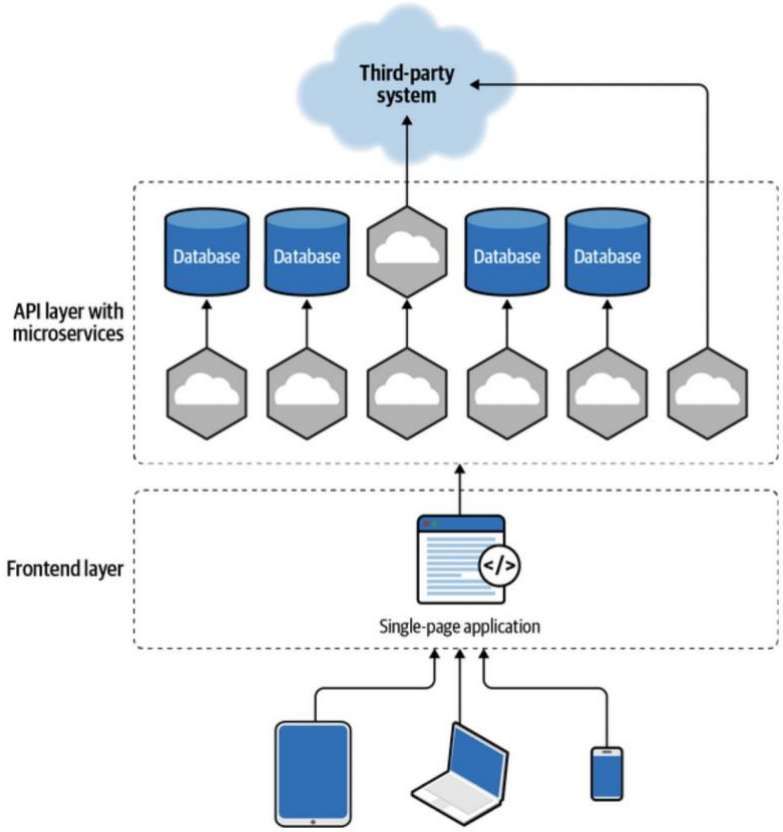


Abbildung 2.9: Current state-of-the-art Webapplikation mit Microservices & SPA (Mezzalira, 2021)

Im API Layer ist jedes Team für sein Set von Microservices verantwortlich. Teammitglieder können so selbständig Entscheidungen treffen, was die Wahl der eingesetzten Tools, Programmiersprachen oder Datenbankprovider betrifft. Des Weiteren ermöglichen Microservices dem Team, eigene Schemas zu definieren, oder den Server mittels Caching-Strategien noch schneller zu machen. Im Grunde bewegen wir uns in einer Welt, wo jedes Team dazu berechtigt ist, mit Eigenverantwortung ihre eigenen Entscheidungen zu treffen. Die Verantwortungsbereiche reichen von der Entwicklung bis zur Produktion und Wartung dieser Services. Wenn man nun einen Blick auf den Frontend Layer wirft, merkt man sofort, dass der Microservice-Ansatz nicht bis dahin durchgezogen wurde.
Nun stelle man sich vor, die Marketingabteilung hat die Idee für die kommende Weihnachtssaison, pro verkauftem Premium Produkt XY ein Tablet mit 1-monatiges Sport-Abonnement zu verschenken. Das Marketing-Team spricht mit dem Fachbereich über die Erweiterung der bestehenden Daten. Der Fachbereich spricht mit den Backend Entwickler/-innen über die Erweiterung ihrer API. Das Backend Team informiert das Frontend Team über die veränderten API-Schnittstellenparameter und die Kette setzt sich fort bis zu dem Testteam welches schlussendlich das Feature abnimmt. Falls nicht, werden erneute Meetings einberufen. Diese Kette an Kommunikationskanälen wird irgendwann so groß, dass sich das Unternehmen nur mehr sehr langsam Änderungen einstellen und neue Features entwickeln kann. Die Kopplung der Teams ist, wenn man die erste mit der zweiten Grafik vergleicht, um einiges loser geworden, ein Glied in dieser Kette scheint jedoch noch zu fehlen, das Frontend.

“We need to break up the frontend monolith, like the backend, to eliminate the friction that impedes innovation.” - (Gilbert, 2021, S.47)

Wie Mikroarchitekturen kombiniert in allen 3 Schichten aussehen können, zeigt nun folgende Abbildung.

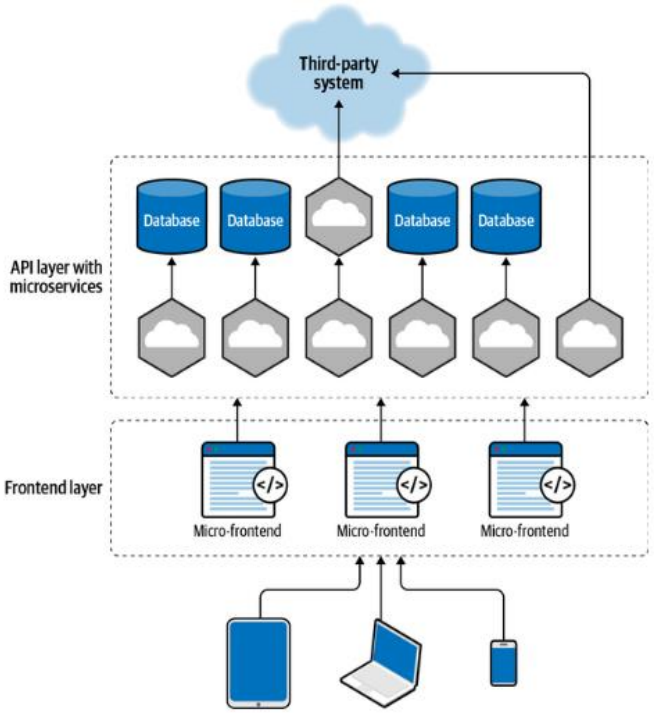


Abbildung 2.10: Microservice Architektur mit Micro Frontends (Mezzalira, 2021)

Das nächste Kapitel sieht sich das Aufbrechen des Monolithens etwas genauer an und erklärt anhand eines Beispiels, wie man eine SPA Monolithen im Frontend in mehrere Micro Frontends aufteilen kann.

2.4.1 Den Frontend Monolithen aufbrechen

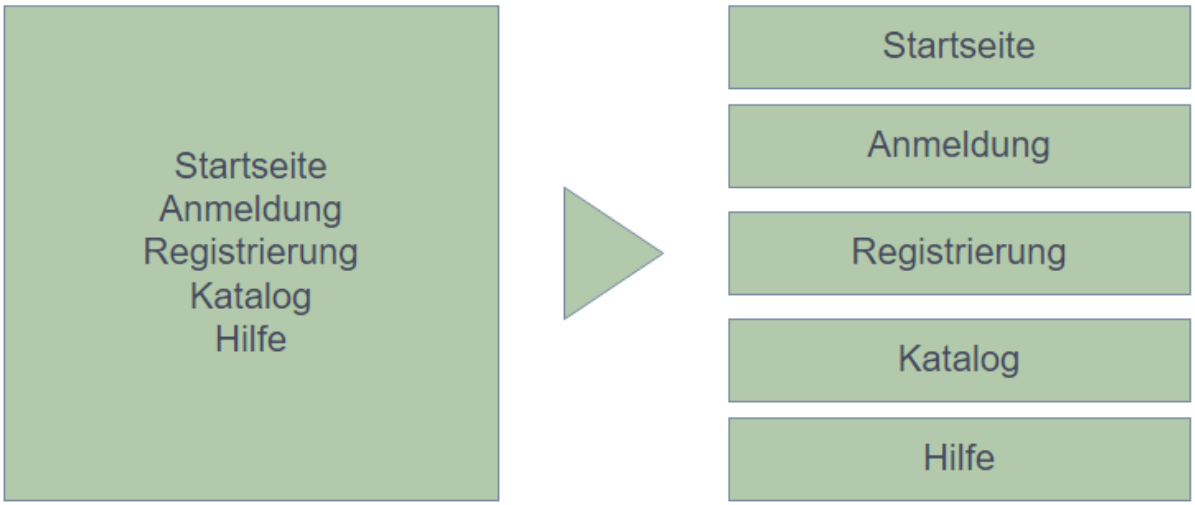


Abbildung 2.11: Aufbrechen eines Frontend Monolithen (eigene Abbildung)

Gilbert (2021) ist davon überzeugt, dass man alle Schichten der technischen Architektur auf derselben Granularitätsebene zerlegen muss. Das Frontend, das Backend und die Datenbank müssen alle als geschlossene Einheit zusammenarbeiten. Erst danach kann man den Teams auch die Kontrolle über einen Teil des gesamten Stapels geben, damit sie unabhängig vorankommen können.

Interessanterweise hat es jedoch sehr lange gedauert, bis man sich im Frontend Gedanken über Dinge wie lose Kopplung und Granularität gemacht hat. Lange Zeit wurden im Frontend keine Bemühungen angestellt, den Microservice-Ansatz zu verfolgen. Berechtigterweise war dies auch lange Zeit nicht notwendig, da der Server den Großteil der Businesslogik beinhaltet. Heute in Zeiten von interaktiven und dynamischen Webapplikationen trifft dies keineswegs mehr zu, ganz im Gegenteil, stattdessen verschieben sich die erforderlichen Aufwände einer Webapplikation immer mehr in Richtung Frontend.

Rappl (2021) schreibt in seinem Buch “The Art of Micro Frontend", dass sich Microservices nicht einfach 1 zu 1 ins Frontend übertragen lassen. Man könne nicht einfach microservices HTML anstatt JSON liefern lassen. Das alleinige Zusammenstellen von HTML wäre zwar nicht so schwierig, jedoch reiche das nicht. Man benötigt auch JavaScript, CSS und Bilddateien. Wie lassen sich JavaScript-Inhalte ineinander verschachteln? Falls unterschiedliche Micro Frontends denselben globalen Variable Namen verwenden, welche gewinnt dann?
Hier gäbe es mehr Fragen als Antworten. Selbst wenn das Anwenden einer MSA 1 zu 1 ins Frontend übertragbar wäre, dann würde man die Trennung zwischen Frontend und Backend verlieren, die, wie Rappl (2021) erwähnt, ganz grundlegend für die heutige Denkweise der Webentwicklung ist.

Einer der Gründe, warum sich die Aufwände eines Projekts immer mehr in Richtung Client verschieben, ist, dass die Nutzer nach einer besseren Erfahrung beim Navigieren der Webinhalte suchen. Dies verwandelte die damals noch zum Großteil statischen Webseiten in interaktive Web-Anwendungen. Als dann die ersten SPA Frameworks am Open Source Markt auftauchten, sorgte dies regelrecht für eine Neudefinition von Frontend Entwicklung. Der enorme Fortschritt, den SPA mit sich brachten war viel zu groß, sodass sich eine Zeit lang niemand so richtig Gedanken über Skalierungsprobleme machte. Diese Probleme entstanden erst mit wachsendem Projekt und Teamgrößen.

In der Vergangenheit hat man solche Probleme gelöst, indem man begonnen hat Teile der Anwendungslogik in externe Module zu packen oder versucht gängige Komponenten in Komponenten Bibliotheken zu verschieben, welche dann Anwendungsübergreifend importiert werden konnten. Im Grunde versuchte man so viel Code wie möglich wieder zu verwenden. Laut Mezzalira (2021) sind all diese Methoden auch aus heutiger Sicht noch eine gute und gängige Methode, solange man es mit Abstraktionen nicht übertreibt, da diese schnell die Komplexität und Leserlichkeit negativ beeinflussen können.

Auch wenn das Gesetz von Gall (1975) eine greifbare und funktionierende Lösung befürwortet wie z.B. mit einem Monolithen zu starten, so ist Mezzalira (2019c) der Überzeugung das man es sich auf lange Sicht mit dieser Architektur nicht verbessern kann. Auch bei Teams, welche in verschiedenen Zeitzonen arbeiten, spricht sich Mezzalira (2019c) klar gegen den Einsatz eines Monolithen aus. Des Weiteren spricht er von einer großen Investition, die notwendig sei, um bestehende Technologien umfassend zu überarbeiten, bevor diese erkennbare Ergebnisse liefern. Micro Frontends bieten hingegen, schnell neue Funktionen hinzuzufügen, sich mit dem Unternehmen weiterzuentwickeln und einen Teil der Anwendung autonom ohne Big-Bang-Releases bereitzustellen.

Eines der ersten Micro Frontend Buch Autoren, Geers (2020), sieht die Vorteile von MF in optimierter Feature Entwicklung, steigerten Flexibilität durch lokale Entscheidungen, und den Gewinn an Unabhängigkeit. Vor allem die erhöhte Skalierbarkeit, welche durch die Autonomie der Teams erreicht wird, hebt Geers (2020) als einen der größten Vorteile von MF heraus. Bei Scrum sollte ein Team zwischen 3 und max. 9 Leuten haben.
Er spricht von sehr positiven Erfahrungen über Konstellationen, in denen 2 bis 6 Teams mit bis zu 50 Mitarbeitern innerhalb eines Projektes arbeiten. MF seien jedoch nicht auf diese Zahl begrenzt. Das Aufteilen einer Anwendung in kleine autonome Systeme sei jedoch mit anfänglichem Zusatzaufwand verbunden. Bei einem Neustart müssen erst passende Grenzen der Teams gefunden werden, das System aufgesetzt und eine Integrationsstrategie entworfen werden, so Geers (2020). Man müsse erst gemeinsame Regeln definieren, mit denen alle Teams einverstanden sind. Des Weiteren weist er darauf hin, dass es wichtig sei, Wege zu finden, um Wissen teamübergreifend teilen zu können.

Die Vorteile für MF könnte man nun wie folgt zusammenfassen:

- Skalierung auf mehrere Teams
- Unabhängige Entwicklung (Schnellere Feature Entwicklung)
- Unabhängige Bereitstellung (Höhere Stabilität durch weniger riskante Deployments)
- Bessere Wartbarkeit der Anwendung (updaten, upgraden oder neu schreiben nur Teile der Anwendung)
- Inkrementelle Updates
- Isolierte Laufzeit der einzelnen MF
- Technologie-unabhängig (Kombination mit unterschiedlichen Frameworks möglich)

MF bringen auch einige Nachteile mit sich:

- Nur für mehrere Teams geeignet (Aufwand lohnt sich bei kleinen Teams nicht)
- Redundanzen oder Inkonsistenzen (Code Duplikation, abweichende Designs)
- Informationssilos (Durch viele kleine Teams könnten Wissensbarrieren entstehen, welche zu Zeitverschwendung und Ineffizienz führen können)
- Konfiguration einer MF Orchestrierungs-Werkzeuges kann schnell kompliziert werden
- Vielzahl der Standards mit denen man Schritt halten muss
- Komplexes CI/CD, Deployment Setup notwendig
- Nicht für Junior Teams geeignet

2.5 Software Architektur Micro Frontend

“Software architecture is those decisions which are both important and hard to change”
- Martin Fowler

Diese Definition zur Software-Architektur von Martin Fowler, impliziert, wie wichtig es ist, ein System von Grund auf mit Sorgfalt zu wählen, da diese später nur schwer zu ändern seien.

Um ein Rahmenwerk für MF zu erstellen, gibt es viele Möglichkeiten. Das nun folgende Kapitel 2.5 orientiert sich an Mezzaliras (2021) “Micro Frontends Decisions Framework” welches besagt das jede MFA von vier maßgeblichen Entscheidungen abhängt:
Wie man ein Mikro-Frontend definiert, wie man die verschiedenen Ansichten orchestriert, wie man die endgültige Ansicht für den Benutzer zusammenstellt und wie Mikro-Frontends kommunizieren bzw. Daten austauschen.

1. Definition
2. Zusammenstellung (Komposition)
3. Route
4. Kommunikation

Mezzalira (2021) betont dabei, dass jede dieser Entscheidungen eine fundamentale Rolle für das Ergebnis eines MF Projekts spielt. Neben diesen 4 Säulen gibt es zwar noch weitere Entscheidungen, die man treffen muss, mit diesen Eckpfeilern sollte man jedoch in der Lage sein, die überwiegende Mehrheit dieser Entscheidungen zu bündeln, ohne die Architektur neu bewerten zu müssen, so Mezzalira.

2.5.1 Säule 1 - Definition von Micro Frontends

Die erste Entscheidung, die man treffen muss, hier sind sich Mezzalira (2021), Geers (2020), Gilbert (2021) und Rappl (2021) einig, ist es sich einig zu werden, wie man ein MF aus technischer Sicht betrachten will. Hier gibt es zwei Möglichkeiten, entweder teilen sich mehrere MF eine Seite oder es bekommt jedes MF seine eigene Seite zugeteilt.

Bei der Per-Ansicht Methode ist ein Team für eine komplette Seite oder Ansicht zuständig, hingegen orientiert sich die zweite Methode nach Komposition, in der sich mehrere MF eine gemeinsame Ansicht (View) teilen. Geers (2020), Mezzalira (2021) und Rappl (2021) nennen diese 2 Methoden horizontale oder vertikale Aufteilung.

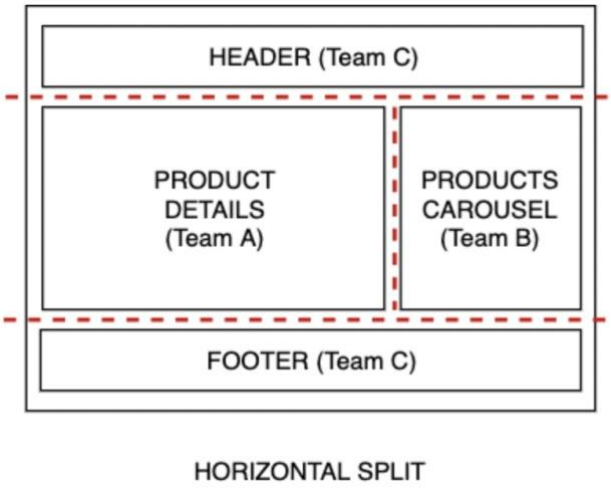


Abbildung 2.12: Horizontale Teilung (Mezzalira, 2019c)

Bei einem horizontalen Ansatz werden Mikro-Frontends normalerweise von Personen entwickelt, die aus verschiedenen Subdomains kommen. Die folgende Grafik illustriert einen typischen horizontalen Ansatz: Eine horizontale Aufteilung bedeutet, dass man MF innerhalb derselben Ansicht definiert. Die einzelnen Teams müssen sich um die Komposition der einzelnen MF absprechen, damit am Schluss die gewünschte Gesamtansicht für den Nutzer entsteht.

Während man sich bei der horizontalen Ansicht noch eine Ansicht teilt, ist bei der vertikalen Ansicht ein Team für die komplette Seite verantwortlich, wie folgende Grafik zeigt.

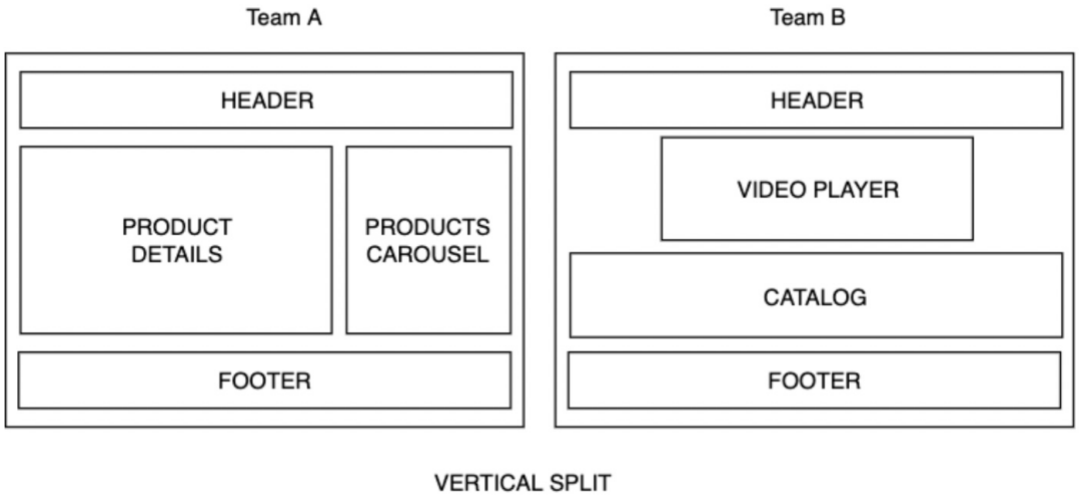


Abbildung 2.13: Vertikale Teilung (Mezzalira, 2019c)

Diese Ansicht vereinfacht viele Entscheidungen, da dies eher der Art und Weise entspricht, wie Frontend-Entwickler/-innen arbeiten, so dass viele bereits erlernte Praktiken leicht auf diesen Ansatz angewendet werden können, so Mezzalira (2019c).
In Sachen Koordination ist der vertikale Ansatz auch einfacher zu handhaben, da jedes Team einen vertikalen Teil der Anwendung besitzt und nicht mehrere Teile, die über die Anwendung verteilt sind (Mezzalira, 2019c).

Ein Nachteil ist jedoch, dass es vorkommen kann, dass gewisse Elemente wie hier z.B. Header und Footer von mehreren Teams benötigt werden und man sich auf eine gemeinsame Lösung einigen muss, ansonsten besteht die Gefahr von abweichenden Designs.

2.5.1.1 Domain Driven Design (DDD) und deren Nutzen zur Definition von MF

Abgesehen von dem horizontalen oder vertikalen Ansatz bin ich während meiner Recherche über ein weiteres Konzept gestoßen, das insbesondere für die Definition von MF äußerst nützlich erscheint, Domain-Driven Design (DDD).
DDD stellt eine Herangehensweise für die Modellierung komplexer Software dar. Der Begriff DDD wurde von Evans (2003) in seinem gleichnamigen Buch geprägt, welches von vielen auch als das “Blue Book” bezeichnet wird. DDD kann man auch als eine Kollektion von Prinzipien und Mustern sehen, welchen es Entwickler/-innen hilft, elegante Objektbezogene Systeme zu erstellen.

Da das Konzept von DDD sehr objektbezogen ist, bekommt man diesen Begriff im Frontend nur sehr selten zu hören. Aus meiner Sicht ist es generell selten, dass Frontend Entwickler/-innen über fundiertes Fachwissen aus anderen Technologie Bereichen wie z.B. Microservices verfügen. Auch Mezzalira kritisierte diese Erkenntnis in einem seiner Blogposts (Mezzalira, 2019a) und ist ebenfalls der Meinung, dass es so gut wie keine Wissensübertragung innerhalb von unterschiedlichen Technologie Communities gibt. Dabei

findet sich so viel Stoff zum Nachdenken, wenn man die Prinzipien hinter Microservices untersucht, anstatt sich auf die bloße Implementierung zu konzentrieren (Mezzalira, 2019a).

Während Evans (2003) mit seinem “Blue Book" den Grundstein zu DDD liefert, hat sich Khononov (2018) das Ziel gesetzt die inhärente Einfachheit des Domänen gesteuerten Designs einzufangen, woraus sein Buch “Learning Domain Driven Design” (Khononov, 2018) entstand.

Khononov (2018) beschreibt den Begriff Domain als die Definition für den gesamten Geschäftsbereich eines Unternehmens. Um ein erstes Beispiel zu nennen, so wäre Netflixs Domäne diejenige der Unterhaltung. Natürlich kann ein Unternehmen in verschiedenen übergreifenden Domänen arbeiten. Zum Beispiel hat Amazon im Einzelhandel begonnen, aber derzeit ist Cloud Computing eine weitere Domäne von Amazon. Um jedoch die Ziele einer Geschäftsdomäne zu erreichen, so ist Khononov (2018) sowie Evans (2003) überzeugt, muss ein Unternehmen in mehreren Subdomains operieren.

Aus der Sicht von MF Architekten und Architektinnen, beschreibt Mezzalira DDD im Grunde als Methode, die versucht, das Geschäft mit der technischen Seite in Einklang zu bringen und spricht in dem Zusammenhang von zwei Hauptbereichen, die ein Unternehmen versucht zu vereinen: Produkt und Technologie. Des Weiteren beschreibt Mezzalira DDD beginnend mit der Idee, Teile der Anwendung zu identifizieren, die eine Subdomain der endgültigen Anwendung darstellen (Mezzalira, 2019a).

Evans (2003) unterteilt dieser Subdomains in folgende 3 Kategorien:

Core (Sub)domains

Die Core Subdomains sind die geheime Formel eines Unternehmens. Meist sind sie das Alleinstellungsmerkmal, welches dem Unternehmen einen unternehmerischen Vorteil bietet. Unternehmen wollen Wettbewerbsvorteile mit hohen Eintrittsbarrieren, welche die Konkurrenten nicht schnell kopieren können. Niemand wird in etwas investieren, das seine Konkurrenten schnell kopieren könnten. Daher sind Core-Subdomains naturgemäß komplex und ihre Geschäftslogik ändert sich normalerweise oft, da sie im Laufe der Zeit optimiert wird (Khononov, 2018).

Generic Subdomains

Generische Subdomains umfassen Bereiche, die alle Unternehmen auf die gleiche Weise tun wie zum Beispiel Authentifizierung, Marketing oder Analytics. Man kann diese auch als bereits gelöste Probleme bezeichnen. Wenn man sich z.B. die Buchhaltung ansieht, dann gibt es gesetzliche und regulatorische Anforderungen, an die sich alle Unternehmen halten müssen. Aus diesem Grund werden hierzu bereits Lösungen in Form von Standardprodukten angeboten, sodass man diese nicht alle selbst implementieren muss. Auch zahlreiche Open-Source-Projekte sind vorhanden, welche sich oft sehr gut in Projekte einbauen lassen (Khononov, 2018).

Supporting Subdomains

Unterstützende Subdomains bieten wie generische Subdomains zwar keinen Wettbewerbsvorteil für das Unternehmen, sind jedoch für die Implementierung einer Core-Subdomain erforderlich. Da es hierfür keine Lösungen von der Stange gibt, müssen

Unternehmen diese selbst implementieren. In Bezug auf das Beispiel von Netflix könnten persönliche Filmvorschläge oder das Film Bewertungssystem Teil der Supporting Subdomain sein, während das Streamen von Filmen von überall und jederzeit zur Core-Subdomain gehört (Khononov, 2018).

Die folgende Abbildung zeigt das Beispiel einer Domain mit mehreren Subdomains und begrenzten Kontexten.

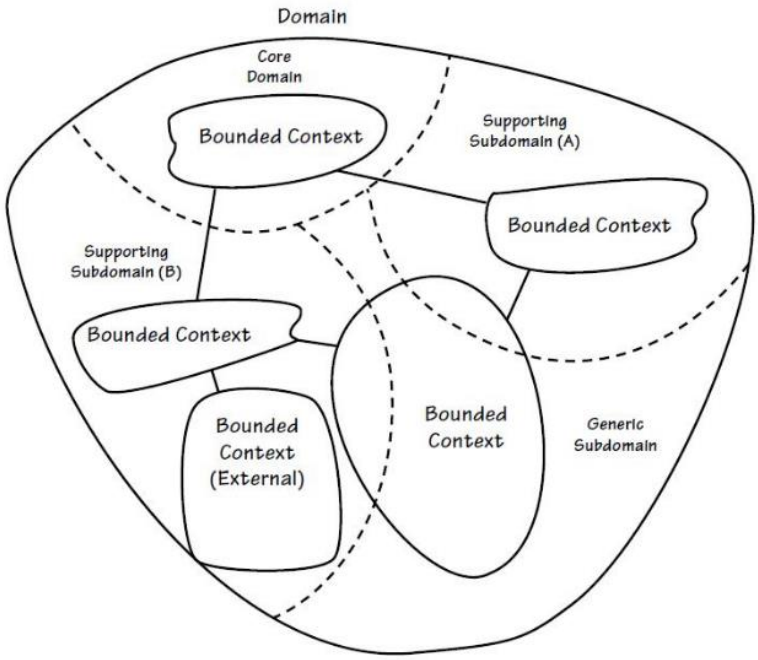


Abbildung 2.14 - Domain mit mehreren Subdomains und begrenzten Kontexten
Evans (2003)

2.5.1.2 Domain Driven Design und der Bezug auf Micro Frontends

Da Evans (2003) das Frontend innerhalb von DDD nicht berücksichtigt, findet man dessen Ansätze hauptsächlich in der Backend-Schicht und nur selten in Frontend-Schicht. Dabei ermöglicht eine Ausweitung dieser Konzepte auf das Frontend, diese leichter zu identifizieren.
Während DDD das Wort Module verwendet, was damals ein Alias für Pakete war, könnte man Module auch als Mikro-Frontend bezeichnen (Rappl, 2021).

In Bezug auf MF sind sich Mezzalira (2019a, 2021), sowie Rappl (2021) einig, dass einige der Schlüsselkonzepte von DDD auch für die MFA sehr nützlich sind. Des Weiteren meint Mezzalira (2019a, 2021), dass es nicht oft vorkommt, dass wir DDD-Prinzipien auf Frontend-Architekturen anwenden, aber in Fall von MF haben wir einen guten Grund, diese zu untersuchen.

Dabei ordnen Rappl (2021) und Mezzalira (2019a, 2021) den Konzepten von DDD unterschiedliche Wichtigkeit zu. Während Mezzalira (2019a, 2021) auch beim Konzept von “ubiquitous language” etwas für MF abgewinnen kann, meint Rappl (2021), dass nur ein Teil

der DDD Toolsets für MF nützlich sind. Stattdessen nimmt Rappl (2021) fast ausschließlich DDD als Blaupause, um sich bei den folgenden zwei Dingen zu helfen:

- Definieren von Mikro-Frontends mit klaren Grenzen
- Entwicklung einer Strategie, um diese Grenzen festzulegen

In Bezug auf MF, so sind sich Mezzalira (2019a, 2021) und Rappl (2021) einig, ist das Definieren eines bounded context auch für MF ein ganz grundlegender Schritt ist, um deren Grenzen abzustecken.

Ein Punkt den Mezzalira (2019a, 2021) zusätzlich erwähnt ist das innerhalb jeder Subdomain, Technologie- und Produktteams eine allgegenwärtige Sprache identifizieren sollten um Funktionalitäten, Objekte, aber mehr im Allgemeinen das Domänenmodell zu identifizieren.
Zum Beispiel ist die Core-Domain von Netflix das Video-Streaming. Die Subdomains innerhalb dieser Kerndomain sind der Katalog, die Anmeldefunktion und der Videoplayer.

Wenn man nun die Subdomain Katalog betrachtet, kann man zusätzlich innerhalb dieser Subdomain mehrere Bereiche mit spezifischen Funktionalitäten identifizieren, welche direkt mit Backend-APIs verbunden sind, wie folgende Grafik zeigt.

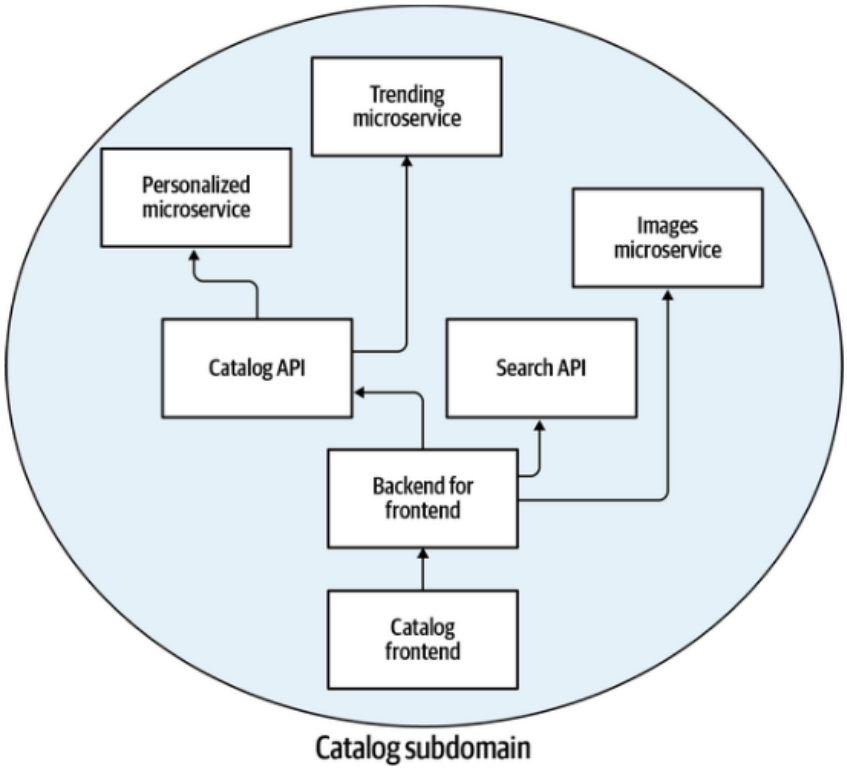


Abbildung 2.15: Catalog Subdomain (Mezzalira 2019a)

Wenn man nun den unteren Teil der Abbildung näher betrachtet, sieht man das im Falle von Netflix das „Backend For Frontend“ (BFF) Muster verwendet wird, die Prinzipien bleiben jedoch gleich. Trotz der technischen Integration, die über Backend für Frontend, GraphQL, Server Side Rendering usw. erfolgen kann, ist das Wichtigste zu verstehen, dass diese Bereiche alle mit derselben Subdomain verknüpft sind. Daher sollten diese Microservices

sowie das Frontend in einer einzigartigen Subdomain mit einer eigenen allgegenwärtigen Sprache gekapselt werden, so Mezzalira (2019a).

Das Definieren eines „bounded context“ ist dabei ein ganz grundlegender Schritt und hilft dabei aufzuzeigen, wie man eine Anwendung isoliert organisieren kann.
Das Verständnis, dass das Frontend Teil der Subdomain ist, ermöglicht es, ganzheitlich über die Webanwendung nachzudenken (Mezzalira 2019a).

2.5.1.3 Identifizieren eines begrenzten Kontextes (bounded-context) für MF

Mezzalira (2019a, 2021) unterscheidet bei der Identifizierung von MF von 2 Hauptszenarien: Greenfield-Projekte und Legacy-Projekten. Bei Greenfield-Projekten hat man zwar ein freies Feld, was für die meisten Entwickler/-innen viel aufregender ist, jedoch auch um einiges komplizierter, da man keine Informationen über die Nutzerbasis hat und wie diese die Inhalte konsumieren würden. Bei Legacy Projekten ist das Begrenzen des Kontextes um einiges einfacher da man viele Informationen über Nutzerverhalten hat (z.B. durch Google-Analytics) welche eine logische Identifizierung des begrenzten Kontextes auf der gesamten Plattform nach dem Verhalten des Benutzers möglich macht. Deshalb empfiehlt Mezzalira dringend die Aufteilung von Mikro-Frontends basierend auf Nutzerdaten.
Folgende Grafik zeigt das Nutzerverhalten basierend auf dem Nutzer Verkehr:

Abbildung 2.16: Nutzerverhalten basierend auf Nutzer Verkehr (Mezzalira 2019a)

Angelehnt an Mezzaliras (2019a) Methode, könnte man nun aus dem Bereich Landing-Page und Authentication eigene MF definieren. Der authentifizierte Bereich ist jedoch etwas schwieriger zu trennen, da diese Domain sehr vom jeweiligen Business abhängig ist. Wenn wir bei dem Beispiel Netflix bleiben, könnte man sich aus dem authentifizierten Bereich in Katalog, User Account und Support trennen. Da der Katalog über viele Funktionen verfügt wie der dahinter liegende Abspielbereich und die Filmbeschreibung, kann man diese ebenfalls als eigene Subdomains uns so auch MF herausziehen. Dass sich nun 2 Subdomains innerhalb der Katalog Subdomain befinden, hindert einen nicht daran, alle 3 als eigenständige MF zu betrachten.

Zusammen ergäbe diese Auftrennung folgende sieben MF:

- Startseite
- Authentifizierung
- Katalog
- Abspielbereich (Teil des Katalogs)
- Filmbeschreibung (Teil des Katalogs)
- Benutzerkonto
- Support

Ob man den 2 untergeordneten Domänen des Katalogs die Filmbeschreibung und den Abspielbereich nun einem eigenen Team zuordnen will, kann man anhand des Arbeitsaufwandes dieser MF entscheiden. Eine Alternative wäre auch, dass sich ein Team um beide MF kümmert. Nur weil 7 MF entstehen, bedeutet das nicht, dass man auch 7 Teams benötigt.

Laut Evans (2003) gibt es was die Teamaufstellung und Subdomains gibt eine wichtige Regel. So darf ein Team durchaus an mehreren Subdomains arbeiten, es sollten jedoch niemals mehrere Teams an ein und derselben Subdomain arbeiten. In diesem Fall stellen die Subdomains MF dar.

Mezzalira (2021) führt in seinem Buch ein sehr interessantes Beispiel an, bei dem es um 3 Teams geht, die alle in 3 unterschiedlichen Standorten an einer gemeinsamen Codebasis arbeiten.
Diese Teams entscheiden sich für eine horizontale Aufteilung unter Verwendung von “Iframes” oder "Web Components" für ihre Mikro-Frontends. Nach einer Weile erkannten sie, dass Mikro-Frontends in derselben Ansicht irgendwie kommunizieren müssen und sie immer mehr Arbeit dafür aufwenden mussten. Dies kann gepaart mit Problemen wie verschiedene Zeitzonen, Abhängigkeiten, Wissensverteilung oder verteilte Teamstruktur schnell zu Frustration führen, da die Teams mehr Zeit damit verbringen, verschiedene Mikro-Frontends in derselben Ansicht zusammenzufassen und zu debuggen, um sicherzustellen, dass alles ordnungsgemäß funktioniert, (Mezzalira, 2021).

Nun gibt Mezzalira (2021) einen alternativen Ansatz an, indem man das Projekt aus geschäftlicher Sicht angeht, so könnte man ein unabhängiges Mikro-Frontend erstellen, bei dem weniger Kommunikation über mehrere Subdomains erforderlich ist. Sein ursprüngliches Beispiel vergleicht er nun mit einem neuen Ansatz, in dem das Team anstatt Iframes nun mit Hilfe von SPA und einzelnen Seiten arbeitet. Dieser Ansatz ermöglicht es einem Team, alle APIs zu entwerfen, die zum Erstellen einer Ansicht (View) erforderlich sind. Gleichzeitig kann dieses Team auch die Infrastruktur bereitstellen, welche zum Skalieren der Dienste, abhängig vom Datenverkehr, erforderlich ist (Mezzalira, 2021).

Auch wenn DDD mit Methoden wie jene des begrenzten Kontexts beim Entwerfen unserer Systeme hilft, so weist Mezzalira (2021) daraufhin das ein begrenzter Kontext nicht in Stein gemeißelt sei, da sich Unternehmen mit der Zeit verändern und man auf diese Änderungen früher oder später reagieren müsse. Von einer zu frühen Definierung dieser Kontexte ratet er deshalb ab, vor allem wenn man noch nicht die benötigten Daten dafür hat.

Obwohl begrenzte Kontexte Modellgrenzen sind, kann es immer noch Fälle geben, in denen dasselbe Modell einer Subdomain oder ein Teil davon in mehreren Begrenzungen implementiert wird. Khononov (2018) betont hier die Wichtigkeit, das gemeinsam genutzte Modell über alle begrenzten Kontexte, die es verwenden, konsistent zu halten. Evans nennt solch einen geteilten Kontext einen geteilten Kern (shared kernel), welcher oft Teil der Core-Domain, einer Generic-Domain oder beidem ist, was nun folgende Abbildung verdeutlicht.

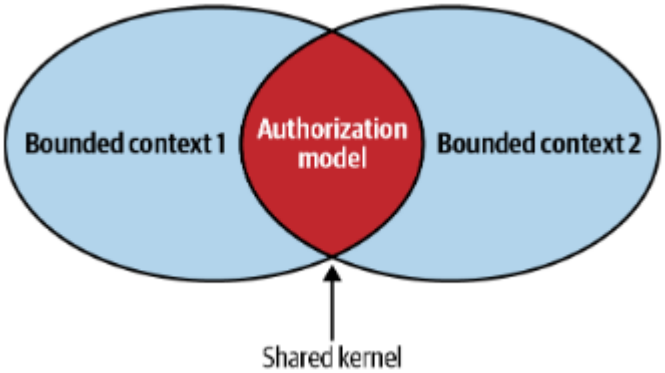


Abbildung: 2.17: Shared Kernel (Evans, 2003)

Neben Autorisierung könnten dies weitere Beispiele für geteilte Subdomains sein: Logging, Navigation, grundlegende Benutzerinformationen, User, Berechtigungen oder Analytics (Khononov, 2018).

Mezzalira (2019a) hebt dabei ein weiteres wichtiges Konzept hervor, welches besagt, dass eine Subdomain nicht als Komponente einer Seite erkannt werden sollte. Es stimmt, dass in jeder Benutzeroberfläche Links oder grafische Elemente zu finden sind, man müsse jedoch verstehen, dass diese keine eigenständige Subdomain darstellen und dass Teams eine Subdomain durchgehend besitzen müssen.

Wenn das Konzept des Bounded Kontextes auch auf die Infrastruktur ausgedehnt wird, merkt man, dass es möglich wird ein Team beginnend von Frontend, Backend bis hin zur Infrastruktur, ohne allzu viele externe Abhängigkeiten zu bilden. Die Kombination aus Mikroarchitekturen, Microservices und Mikro-Frontends bietet eine unabhängige Bereitstellung (Deployment) ohne hohe Risiken, das gesamte System für die Freigabe in der Produktion zu gefährden (Mezzalira, 2021).

2.5.2 Säule 2 - Zusammenstellung (Komposition) von Micro Frontends

Wenn man sich nun weiter nach den 4 Säulen von Mezzalira (2019c, 2021) orientiert, dann ist die zweite Entscheidung zu verstehen, wo wir Mikro-Frontends zusammenstellen möchten, hier haben wir 3 Optionen zur Auswahl:

Client-side:
Client-side bedeutet die Implementierung von Techniken wie einer App Shell, welche das Laden von MF übernimmt. Die MF werden dabei vom Ursprung (origin) oder CDN (falls bereits zwischengespeichert) abgerufen. Die endgültige Ansicht jedoch erst wird vom Client erstellt.
Ein paar Beispiele wären Webpacks-Module-Federation[32], das Single-Spa[33] Framework, das Luigi-Framework[34] aus dem Hause SAP welches iframes benutzt oder über eine Bibliothek, die eine clientseitige Transklusionstechnik verwendet[35].

Edge-side:
Kanten Seitige Implementierung würde bedeuten, CDN-Funktionen wie Edge-side-Includes (ESI) zu nutzen oder über Berechnungen auf Kanten (edges) in der Nähe zurückzugreifen. Hier wird die endgültige Ansicht bereits auf CDN Level erstellt, die MF werden dabei vom Ursprung (origin) abgerufen, damit das Endresultat dem Client übergeben werden kann.

Server-side:
Hier werden die MF bereits auf der Ursprungsebene (origin level) zusammengesetzt auf CDN-Ebene zwischengespeichert und schließlich dem Client bereitgestellt.
Auf der Serverseite gibt es auch ähnlich wie clientseitig einige Frameworks, die man verwenden kann wie z.B. Ara Framework, Open Components, Piral[36] oder Zalandos Tailor.js[37] und viele weitere.

Folgende Grafik veranschaulicht diese 3 Varianten:

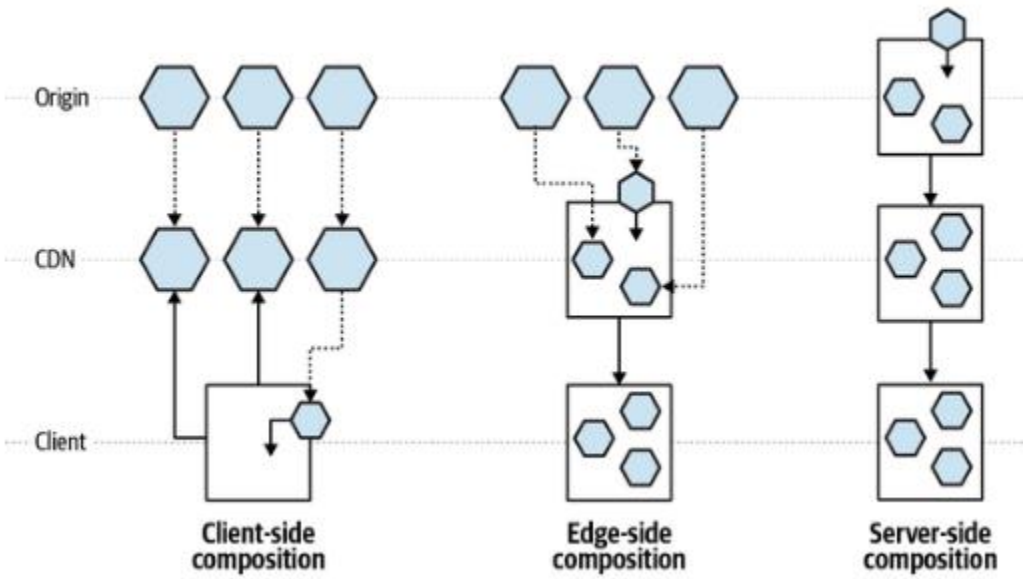


Abbildung 2.18: Micro-Frontend Kompositionsdiagramm (Mezzalira, 2021)

[32] https://webpack.js.org/concepts/module-federation/
[33] https://single-spa.js.org/
[34] https://luigi-project.io/
[35] https://gustafnk.github.io/microservice-websites/#hinclude-and-h-include
[36] https://piral.io/
[37] https://github.com/zalando/tailor

Mezzalira (2019a, 2019c, 2021), welcher mitunter für die Video-on-Demand-Sportplattform DAZN[38] gearbeitet hat gibt was die Zusammenstellung von MF betrifft, auch ein paar Empfehlungen ab (Mezzalira, 2019c): Wenn man eine Client-seitige Zusammensetzung anstrebt welche ähnlich zu “single-spa” mittels Bootstrap Methode eine Kombination aus SPA lädt, dann wäre die vertikale Aufteilung sehr gut dafür geeignet, da sie es bei DAZN ähnlich gemacht haben und gute Erfahrungen damit gemacht hatten. Wenn man sich jedoch entscheiden sollte, MF als einzelnen Teil einer Ansicht zu identifizieren, stehen einem alle Optionen zur Verfügung. (clientseitig, randseitig und serverseitig).

2.5.3 Säule 3 - Routing von Micro Frontends

Beim Routing hat man dieselben Optionen wie bei der Komposition und kann die 3 Varianten auch miteinander kombinieren wie z.B. clientseitig zu routen und Logik auf der Edge hinzuzufügen oder Routing-Logik nur auf dem Client oder Server zu haben. Mezzalira empfiehlt jedoch, mit dem Kompositionsteil kohärent zu sein (Mezzalira, 2019c).

Client-side routing

Hier übernimmt die App-Shell die Zuordnung der Routing-Logik abhängig vom Benutzer-Status (user-state). Das bedeutet zum Beispiel, dass die Anwendung im Falle eines authentifizierten Benutzers den authentifizierten Bereich lädt und bei Nutzern, die zum ersten Mal die Seite besuchen wird, die Landing-Page geladen. Diese Variante eignet sich perfekt für komplexes Routing, z.B. wenn MF auf Authentifizierung, Geolokalisierung oder anderen ausgefeilten Logiken basieren (Mezzalira, 2021).

Edge-side routing

Das Routing basiert auf der Seiten URL und das CDN stellt die Seiten zur Verfügung, indem es die unterschiedlichen MF mittels Transklusion, auf Kanten Ebene (Edge-level) zusammensetzt, was wenig Spielraum für intelligentes Routing zulässt und somit eher für „multipage-websites“ geeignet ist.

Server-side routing

Falls man sich entscheidet, MF am Ursprung (origin) zusammenzustellen dann ist man bei einer serverseitigen Komposition gezwungen alle Anforderungen (inkl Routing) am Ursprung handzuhaben da die gesamte Anwendungslogik auf dem Anwendungsservern (App-Server) lebt (Mezzalira, 2021).

Wie bereits zu Anfang erwähnt, schließen sich diese Routing-Ansätze nicht gegenseitig aus. Diese können kombiniert werden, indem man CDN & Origin oder Client & CDN zusammen verwendet.

[38] https://www.dazn.com/

2.5.4 Säule 4 - Kommunikation zwischen Micro Frontends

Auch wenn man die Kommunikation so gering wie möglich halten sollte, um die Selbständigkeit der MF zu garantieren, ist das in der Praxis nicht immer möglich. Es ist dabei wichtig nicht zu vergessen, dass ein einzelnes MF nichts über die Existenz anderer MF wissen darf, da man ansonsten Prinzipien wie jene des “independent deployment” breche. Trotzdem muss man andere MF über gewisse User Interaktion informieren. Bei mehreren MF auf derselben Seite (horizontal-split) ist dieser Kommunikationsaufwand oft höher als wenn jedes MF eine komplette Seite einnimmt (vertical-split).

Bei einer horizontalen Trennung oder MF welcher Herr ihrer eigenen Kommunikation sein wollen empfiehlt sich der Einsatz eines Eventbuses oder custom events. Hiermit kann jedes MF auf Events horchen und sobald es an einem davon interessiert ist, dementsprechend darauf reagieren. Um den Eventbus zu injizieren, benötigen man einen übergeordneten “Container”, um den Eventbus dort zu instanziieren, damit alle anderen Micro-Frontends auf diesen Zugriff haben. Bei custom events verhält es sich ähnlich, nur das man hier Events in einem globalen Objekt wie dem “window object” speichert (Mezzalira, 2021).

Die folgende Abbildung zeigt einen Event Emitter oder custom events Diagramm.

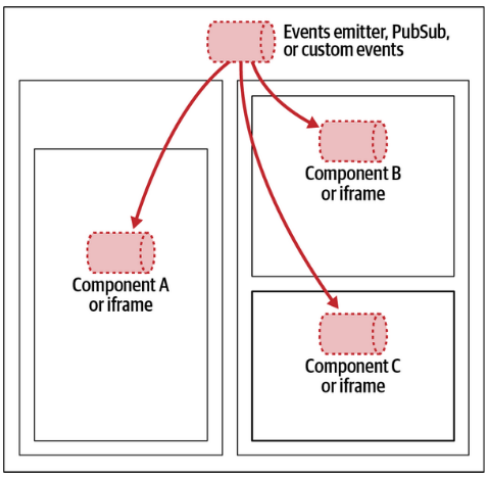


Abbildung 2.19: Event Emitter / Custom Events (Mezzalira, 2021)

Bei einer vertikalen Aufteilung kommt man meist ohne Events aus, da der Kommunikationsbedarf viel geringer ist und man sich hauptsächlich um Seitenwechsel (Routing) oder Authentifizierungsstatus kümmern muss. Ein Seitenwechsel kann dabei über URL-Routing implementiert werden und das Weiterleiten von Authentifizierungs-Tokens kann mittels local-storage, session-storage oder cookies implementiert werden (Mezzalira, 2021).

Die nächste Abbildung zeigt das Teilen von Daten zwischen zwei unterschiedlichen “views”

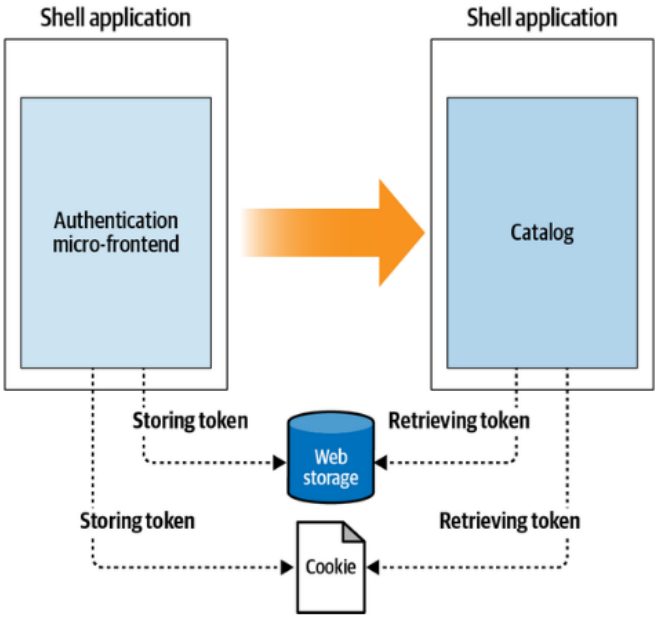


Abbildung 2.20: Teilen von Daten zwischen zwei unterschiedlichen “views” (Mezzalira, 2021)

Eine weitere Option, Daten zu teilen, wäre via „query strings“, indem man z.B. Produkt IDs über die URL handhabt, wie https://www.shop.com/products?id=123
Diese Option hat viele Anwendungsgebiete, auf welche jedoch nicht näher eingegangen wird, da diese nicht im Fokus dieser Arbeit liegen. Ein wichtiger Punkt wäre noch zu erwähnen, und zwar dass sich „query strings“ nicht für sensible Daten wie Passwörter oder Benutzer IDs eignen. Für sensible Daten eignen sich bessere Methoden wie JWT (JavaScript Web Tokens) gespeichert im Web-Storage, welche oft für Authentifizierungs-Zwecke oder das Konsumieren von authentifizierten APIs verwendet werden.

Da wir nun alle 4 Entscheidungen für eine gute MFA getroffen haben, möchte ich diese im nächsten Kapitel noch einmal kurz zusammenfassen.

2.5.5 Zusammenfassung – 4 Säulen einer Micro Frontend Architektur

Wie man sieht, gibt es einige Entscheidungen zu treffen, wenn man sich einer MFA nähern will. Es gibt zwar noch viele weitere Entscheidungen wie z.B. eine nahtlose Benutzererfahrung zu erschaffen, CI/CD Pipelines bereitzustellen, Leistungsoptimierungen, Bundle-Size Reduzierung, Code-Sharing, Design Konsistenzen, Technologie Unabhängigkeit, Abhängigkeiten Management, organisatorische Auswirkungen und viele mehr. Diese 4 im Voraus getroffenen Entscheidungen bieten jedoch eine starke Richtlinie für die Bewältigung aller anderen Herausforderungen und schränken die Anzahl der zur Auswahl stehenden Optionen ein (Mezzalira, 2019b).

Noch einmal zusammengefasst hat man nun über 4 der Schlüsselfunktionen entschieden

- **Definition** - Wie man ein Mikro-Frontend definiert, vertikale oder horizontale Aufteilung, Domain Grenzen erkennen mittels DDD.
- **Komposition**- Wie und wo man die verschiedenen Ansichten orchestriert (client, server, edge) und für welche Render-Methode(n) man sich entscheidet. Der Weg von Origin über CDN zum Client
- **Routing** - Wo und wie soll das Routing stattfinden (edge, server oder client). Wie will man den User von einer Seite zur nächsten weiterleiten.
- **Kommunikation** - Wie kommuniziert man zwischen MF

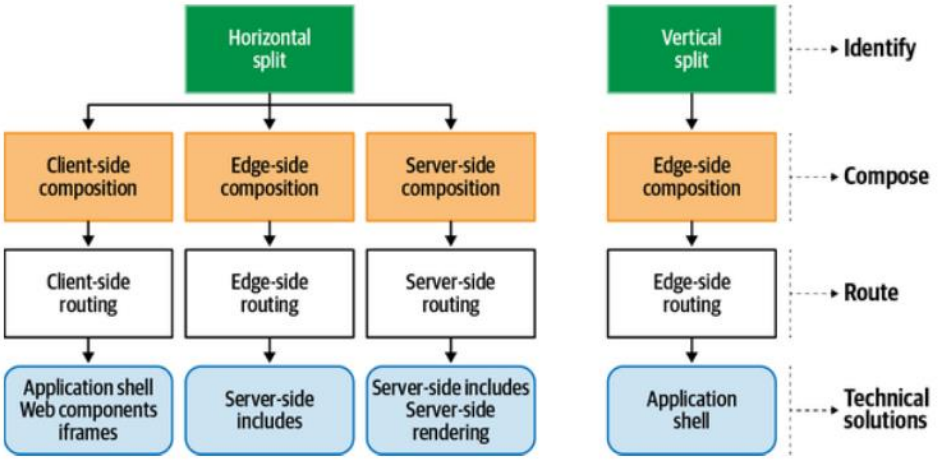


Abbildung 2.21: Micro-Frontends-Decisions-Framework (Mezzalira, 2021)

Wenn man nun erneut einen Blick auf diese 4 Säulen wirft, merkt man, dass bis zu diesem Zeitpunkt keine Frameworks, Programmiersprachen oder spezielle Werkzeuge erwähnt wurden. Dies sorgt für ein solides Fundament, auf dem man nun die Implementierung aufbauen kann.

Beim vierten Schritt, erkennt man, dass dieser nicht mehr mit “Communication”, sondern mit „Technical Solutions“ betitelt wird. Dies liegt daran das man ab den vierten Schritt nicht nur überlegen muss wie die einzelnen MF miteinander kommunizieren sollen, sondern müssen von nun an auch technische Entscheidungen getroffen werden wie z.B. ob man ein MF-Framework für die Verwaltung von MF verwenden möchte und wenn ja, welches.
Da die Auswahl für MF-Frameworks sehr überschaubar ist, fällt der nun folgende Teil sehr kurz aus.

2.5.6 Verfügbare Frameworks zur Implementation von MF

Der aktuelle Stand von MF-Frameworks ist sehr überschaubar. An und für sich ist die Auswahl sehr groß, jedoch werden nur wenige davon aktiv weiterentwickelt. Das liegt womöglich daran, dass einige Unternehmen ihre eigenen MF-Frameworks entwickeln, um den eigenen Anforderungen gerecht zu werden und diese nur selten Open Source zur Verfügung stellen. Die meisten MF-Frameworks stellen eine sogenannte Application-Shell zur Verfügung, welche die einzelnen MF-Laden, das Seiten-Routing übernehmen, oder die einzelnen MF- wie Lego-Bausteine zu einer gemeinsamen Ansicht komponieren. Im Grunde übernimmt die Shell die komplette Orchestrierung einer MF Landschaft. Manche MF-Frameworks bieten auch weitere Funktionen wie “Dependency” Management oder einen bereits vor implementierten “message bus”. Auch wenn das Erstellen einer Anwendungs-Shell nicht allzu viel Aufwand erfordert, gibt es MF-Frameworks, die diesen Teil übernehmen. Dabei muss man erwähnen, dass das Konzept einer App-Shell optional für MF ist, jedoch verwenden die meisten der MF-Frameworks diesen Ansatz.

Die Auswahl an MF-Frameworks ist im Jahr 2022 um einiges größer als noch 2016, wo man neben single-spa kaum Alternativen hatte. Heute gibt es MF-Frameworks wie Bit, Luigi, Piral, Single-Spa, Qiankun, Module Federation, SystemJS, Podium, Open Components und viele mehr[39]. Die Open Source Community hinter MF ist, zwar nicht annähernd mit Größen wie React, Vue oder Angular vergleichbar, was leider bedeutet das manche MF-Frameworks nicht mehr aktiv weiterentwickelt werden wie z.B. das einstig sehr beliebte Framework Single-Spa.

Eine große Änderung die MF regelrecht revolutioniert haben passierte jedoch 2020 als Webpack mit Module Federation[40] ein Plugin zur Verfügung stellte, mit denen man nun ab sofort MF entwickeln konnte. Nachdem ich mich schon seit 2 Jahren auch beruflich mit MF beschäftige, würde ich Module Federation heute als die “solideste” Wahl unter allen MF-Frameworks bezeichnen. Der in meinen Augen größte Vorteil von Webpacks Module Federation ist, wie einfach man Code zwischen unabhängigen Projekten teilen kann. Dabei ist der Umgang damit nicht nur leicht, sondern ermöglicht Module Federation das Teilen von Packages (node_modules) und Feature/Application Code auf einem viel höheren Niveau als andere. Das bedeutet, dass man dynamisch Code von externen Applikationen verwenden kann, was dafür sorgt, dass sich der Umgang damit so anfühlt, als würde man lokal auf eine externe Applikation zugreifen und trotzdem eine lose Kopplung bewahren.

Die vielen Vorteile von Module Federation machen es sehr schwer, sich für andere MF zu entscheiden. Da Module Federation durch den leichtgewichtigen Aufbau für manche vielleicht zu wenig Funktionen bietet und aktuell nur CSR unterstützt (SSR Support befindet sich in der Beta) lohnt sich der Blick auch auf andere aktiv weiterentwickelte MF Frameworks. Dafür gibt es unter “github.com/rajasegar/awesome-micro-frontends” eine Seite, welche alle Ressourcen rund um MF zusammenträgt und “up to date” haltet.

[39] https://github.com/rajasegar/awesome-micro-frontends
[40] https://webpack.js.org/concepts/module-federation/

3. Konzeptioneller Vorgehens- und Lösungsansatz

Dieses Kapitel gibt einen groben Überblick über die eingesetzten Methoden, das Konzept hinter dem Prototyp, einen Einblick darüber wie man organisatorische Probleme lösen will, und schließt mit einer Begründung der Methodenwahl ab

Mithilfe der Information über den aktuellen Stand der Wissenschaft und Technik möchte ich nun in die Implementierung übergehen. Das 4 Säulen Modell bietet dabei ein gutes Fundament für die MFA, auf wessen technische Implementierung nun aufgebaut werden kann. Das Micro-Frontends-Decisions-Framework von Mezzalira hilft bei den ersten großen Entscheidungen für die Architektur, worauf nun jedoch weitere technische Entscheidungen folgen, wie die Wahl der Frameworks, Tools und Programmiersprachen.

3.1 Prototyp

Um den Prototypen auch an die Forschungsfrage zu richten sind für eine Implementierung im Wesentlichen 4 Schritte notwendig:

3.1.1 Erstellen eines Problem Pools

Da die Forschungsfrage darauf abzielt, gängige Probleme aufzuzeigen und Lösungswege zu erkunden, muss zuerst eine Problemsammlung erstellt werden, welche im Zuge der Implementierung berücksichtigt wird. Diese Probleme können Nachteile von MF aus der Literatur sein oder andere Schilderungen über Herausforderungen im Zusammenhang mit MF, die entweder aus der Community oder aus eigenen Erfahrungen im Umgang mit MF stammen.

3.1.2 Technologische Entscheidungen

Dieser Schritt stellt eine Vielzahl an technischen Entscheidungen dar, die man treffen muss, um das Projekt technisch umsetzen zu können, wie z.B. die Wahl der Programmiersprache, Frameworks, Bibliotheken und weitere Tools.

Ein wichtiger Punkt, bei dem man hier unterscheiden muss, sind 2 Arten von Frameworks. Zum einen benötigt man für eine SPA eines der JavaScript Frameworks wie z.B. React, Vue, Angular, Svelte etc. Zum anderen muss man sich entscheiden, ob man sich für oder gegen ein MF-Framework entscheidet und wenn ja, welches z.B. single-spa, module-federation usw.

3.1.3 Design einer Testdomain (Definition von MF)

Da MF immer in der Mehrzahl anzufinden sind, müssen nicht nur mehrere definiert werden, sondern auch festgelegt werden, wie diese schlussendlich eine gemeinsame Anwendung (Domain) abbilden. Die Größe der MF Landschaft sollte mindestens so groß sein, dass die Webanwendung mindestens 2 Seiten aufweist und genügend MF zur Verfügung stehen, damit man einen horizontalen und vertikalen Split simulieren kann.

3.1.4 Umfangreicher Funktionstest

Um zu erfahren, wie ein Prototyp in den unterschiedlichen Disziplinen abschneidet, muss der Prototyp einen ausreichenden Umfang bieten. Dabei ist es nicht wichtig, schöne User Interfaces zu bauen, jedoch sollte ein Prototyp auf ausreichend und vor allem unterschiedliche Funktionen wie MF-Komposition, Routing, Dependency Sharing, Error Handling getestet werden.

3.2 Weighted Scoring Modell

Die Weighted Scoring Methode dient dazu Technologien, in diesem Fall 5 MF-Frameworks (Module-Federation, Single-spa, Piral, Podium oder Mashroom-Server) und 2 verschiedene Code-Repository-Typen (Monorepo & Multi-Repo) zu vergleichen. Da es sich bei MF-Frameworks und Code-Repositories um 2 völlig verschiedene Technologien handelt, werden 2 Weighted Scoring Tabellen erstellt.

3.3 Begründung zur Wahl der Methoden

Ich bin der Meinung, dass selbst die beste Software Architektur spätestens bei der Entwicklung an seine Grenzen stößt, da sich theoretische Aspekte nicht immer direkt in die Praxis umsetzen lassen. Ein Prototyp stellt aus meiner Sicht nicht nur die ideale Methode dar, um eine Software-Architektur zu testen, sondern eignet sich auch gut, um Probleme aufzuzeigen oder neue Lösungswege zu erkunden.

Da es den Rahmen dieser Bachelorarbeit sprengen würde, für jede Technologie einen eigenen Prototypen zu schreiben, wird für einen Vergleich mit anderen Technologien die Weighted Scoring Methode angewandt.
Auch wenn ein Prototyp aus meiner Sicht schon eine sehr gute Methode darstellt, hat dieser gewisse Limitierungen. Ein Nachteil des Prototyps ist , dass dadurch eine Art künstliche Situation hergestellt wird, welche von einem realen Softwareprojekt manchmal zu weit entfernt ist. Um diesem Problem etwas entgegenzuwirken, werde ich auf Probleme eines bestehenden Projekts aus dem privaten Umfeld eingehen und dieses bei der Implementierung berücksichtigen. Zusätzlich wird diese Arbeit auch bereits bestehende Probleme aus der Community adressieren, was dabei helfen soll, die Aussagekraft des Prototyps zu stärken.

4. Definition eines Rahmenwerks

Dieses Kapitel widmet sich der Testprojekt-Vorbereitung und definiert 3 Schritte, in denen der Prototyp für die Implementation vorbereitet wird. Nach dem Erstellen eines Problem Pools werden die technischen Entscheidungen für das Testprojekt getroffen. Zum Schluss werden noch Funktionen definiert, die das Testprojekt erfüllen muss, damit die Implementierung in Kapitel 5 erfolgen kann.

4.1 Erstellen eines Problem Pools

Um den Erfolg des Prototyps beurteilen zu können, wird eine Liste von Problemen und Herausforderungen im Zusammenhang mit MF erstellt.
In weiterer Folge wird ein Prototyp entwickelt, der es sich zum Ziel setzt, möglichst viele Probleme auf dieser Liste zu lösen.
Die Implementation des Prototyps wird in einer Form beschrieben, sodass dieser einfach nachgebaut werden kann.

Um einen “Problem Pool” erstellen zu können, analysiert diese Arbeit ausgewählte Probleme aus der Literatur und lässt zusätzlich eigene Erfahrungen im Umgang mit MF mit einfließen. Im beruflichen Umfeld konnten bereits 2 Jahre Erfahrung im Umgang mit MF innerhalb eines Enterprise Umfeldes gesammelt werden. Da große Unternehmen oft dutzende MF im Einsatz haben und die Fähigkeit, schnell auf Änderungen zu reagieren, sehr gering ist, ist ein Prototyp ein gutes Mittel, neue Praktiken zu testen.

Dabei wurden folgende Herausforderungen und Probleme im Umgang mit MF festgellt:

1. Aufwendige Infrastruktur
Für jedes einzelne MF muss ein CI/CD Prozesse erstellt werden (Mezzalira, 2021)

2. Ausfallrisiko
Bei mehreren MF gibt es zwar keinen Single Point of Failure mehr, wenn eine Anwendung jedoch aus angenommen 100 MF besteht, reicht ein Fehler oder eine Versionierungs Ungereimtheit in einem dieser MF aus, um die gesamte Anwendung zum Stehen zu bringen.

3. Schlechtes Code sharing
Schlechtes *Dependency Sharing* (Teilen von externen Abhängigkeiten):
Wenn eine externe Bibliothek von mehreren MF verwendet wird, kann es unter Umständen vorkommen, dass diese *Dependencies* nicht geteilt werden und stattdessen mehrfach geladen werden, was die *Bundle-Size* erhöht.[41]
Schlechtes allgemeines Code Sharing:

- Teilen von Bibliotheken (common-libs)
- Teilen von MF
- Teilen von Businesslogik

[41] https://www.sitepoint.com/micro-frontend-architecture-pitfalls/

4. Performance Einbußen
Durch zu große *Bundle-Sizes* (Schlechtes dependency management als häufiger Auslöser) Unerwünschte *Page-Reloads* beim Wechsel von einem MF zum Anderen.

5. MF-Overhead
Dieses Problem taucht in der Community sehr häufig auf und wird oft unterschiedlich beschrieben. Manche sprechen von einem störenden Boilerplate, den manche MF-Frameworks benötigen, Mezzalira spricht wiederum von einer unnötig hohen Lernkurve, wenn man sich zu weit von einer typischen SPA wegbewegt. Je weniger MF-Framework-Spezifischer Code benötigt wird, umso einfacher ist es, neue Entwickler/-innen zu rekrutieren.

6. Developer Experience
Die Arbeit mit MF, welche oft in unterschiedlichen Repositories untergebracht sind, erfordern einiges an Zusatzaufwand:

- Mühsames ändern von Code entlang der MF Landschaft (Bibliotheken, Frameworks, Designs und andere geteilter Code)
- Mehrere MF müssen lokal gestartet werden
- Mehrere Entwickler/-innen Umgebungen/Code Editor (IDE) gleichzeitig geöffnet
- Mehrere Pull Requests für MF übergreifende Features
- Schlechter Überblick über die gesamte Softwarelandschaft. Wo verlaufen die Grenzen zwischen MF? Wo verlaufen die Grenzen zwischen MF und einem UI-Framework? Was macht was? Wie passen die Teile zusammen? Was macht dieses Repo (oder jenes Repo)? Was ist das mentale Modell für das gesamte System? (Mezzalira, 2021)

7. Code Duplication
Einige Daten und Funktionen müssen für jedes einzelne MF verfügbar gemacht werden, was häufig zu Code-Duplikation führt.

8. Schechter Knowledge Transfer
Auch wenn das Splitten einer großen Anwendung viele Vorteile hat, kann der Knowledge-Transfer entlang der einzelnen Teams stark darunter leiden. Es entstehen oft Informationssilos innerhalb der einzelnen Teams.

9. Kommunikationsprobleme
Funktionsübergreifende Teams sind eine gängige Art, Projekte zu verwalten, aber es kann zu Zeitverschwendung und Ineffizienz führen. Das Problem entsteht, wenn verschiedene Gruppen an verschiedenen Codebasen arbeiten, ohne ihre Arbeitsbelastung mit anderen Abteilungen zu teilen.
Dies kann zur Duplizierung spezifischer Implementierungsmethoden führen und wertvolle Zeit für das Unternehmen verschwenden[42].

Kommunikationsprobleme zwischen Teams – Kommunikation in einem großen Team kann ein Problem sein, aber nichts ist schlimmer als die Kommunikation zwischen mehreren

[42] https://www.adservio.fr/post/the-dark-side-of-micro-frontends

Teams. Dies liegt daran, dass mehrere Teams, die an unterschiedlichen Codebasen arbeiten, es schwieriger machen, wiederverwendbare Features, Funktionen und Dienstprogramme zu finden. Dies ist im Hinblick auf die Auffindbarkeit des Codes und damit die Wiederverwendbarkeit schlecht. Mit anderen Worten, Sie können am Ende leicht doppelte Implementierungen derselben Komponenten über verschiedene Mikro-Frontends hinweg haben[43].

10. Überlappende Designs (stylesheets)
Da jedes MF eigene Designs definiert, kann das Zusammenführen von *Stylesheets* bzw. *classnames* zu Konflikten führen. Wenn 2 MF dieselben *classnames* verwenden, wird eine davon verworfen, was zu unerwarteten Ergebnissen führt.

11. Inkonsistente Designs
Design Abweichungen entlang der Anwendung sind in einem MF Setup ein häufiges Problem.

4.2 Wahl der Technologien für das Testprojekt

Bei der Wahl der Technologien spielen Frameworks eine große Rolle, welche bereits in Kapitel 2.1 (Frontend Frameworks) und Kapitel 2.5.6 (MF-Framework) vorgestellt wurden.

Frontend Framework: Reactjs
Innerhalb eines MF Umfelds ist es zwar nicht zwingend notwendig, sich auf ein FE-Framework zu einigen, jedoch wird in der Literatur an vielen Stellen dazu geraten.
Die Wahl fiel hier auf Reactjs aus folgenden Gründen:

- Am besten vertreten am Arbeitsmarkt
- Größte Open Source-community
- Herausragender Micro Frontend Support (Entwickler von Module Federation bevorzugt Reactjs)
- “Simple & Powerful” (Gute Ausgewogenheit zwischen Funktionsumfang und Lernkurve)

MF-Framework: Module Federation
Bei der Wahl eines MF-Frameworks gibt es einige Optionen, zwecks mangelndem Support scheiden hier jedoch viele bereits im Vorfeld aus. Module Federation konnte vor allem durch folgende Punkte überzeugen :

- Aktiver Support (Viele MF-Framework werden nicht mehr weiterentwickelt)
- Wirkt im Vergleich zur Konkurrenz technisch stark überlegen
- Simplicity (setzt den geringsten "Boilerplate Code" vorraus)
- Geringer Vendor Lock-In (Setzt lediglich Webpack 5 vorraus)
- Modularität - Kann gut mit anderen MF-Frameworks kombiniert werden

Module Bundler: Webpack
Auch heute muss jedes Frontend-Projekt dessen Code in "deployable" JavaScript “chunks” verwandeln, was bedeutet, dass sich schließlich jedes Team für einen Module Bundler entscheiden muss. Webpack war hierzu jahrelang der einzige Module Bundler am Markt und

[43] https://www.sitepoint.com/micro-frontend-architecture-pitfalls/

ist auch heute noch am weitesten verbreitet. Seit ein paar Jahren gibt es jedoch auch andere ernst zu nehmende Alternativen am Markt. Diese sind zwar noch nicht annähernd so weit verbreitet wie Webpack, erreichen durch ihre hohe Performance jedoch kürzere “build” Zeiten.
Da Module Federation Webpack 5 voraussetzt, muss hier im Vorhinein eine Entscheidung getroffen werden, da diese nachher nicht mehr einfach rückgängig zu machen ist. Da Webpack seit vielen Jahren der beliebteste “Module Bundler" ist, stellt diese Abhängigkeit für viele Projekte gar keine richtige Abhängigkeit dar, da diese seit jeher nichts anderes verwenden, um deren Code-Blöcke für die Produktionsumgebung zu bündeln.

Die folgende Grafik zeigt den Vergleich der beliebtesten Modul-Bundler: browserify, esbuild, parcel, rollup und webpack aus den letzten 2 Jahren.

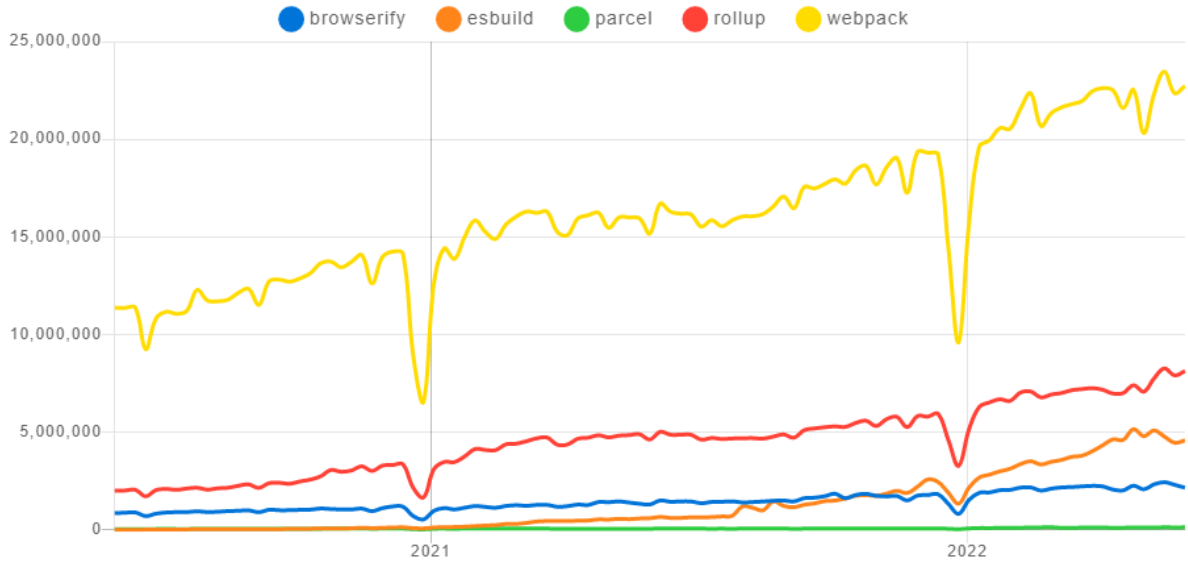


Abbildung 5.1: Module Bundler Downloads via NPM (Eigenrecherche via npm-trends[44])

Auch wenn diese Abbildung verdeutlicht, wie weit Webpack mit den Downloadzahlen vorne liegt, sollte das nicht automatisch bedeuten, dass dies auch die beste Wahl ist. Alternativ könnte man auch “rollup” oder “esbuild” als “module bundler” verwenden, da die Community dahinter viel aktiver ist als bei Riesen wie Webpack.
Auch der Blick auf die Versionierungshistorie kann bei Open Source Projekten Aufschluss geben, wie stabil und aktiv jene Projekte sind. Im Falle von Webpack sieht man, dass der erste “stable release” bereits 8 Jahre zurückliegt (v1.0). Da Webpack seit vielen Jahren unzählige Produktionsumgebungen mit gebündelten files versorgt, stellt die Entscheidung auf Webpack kein Risiko dar.

Code Repository
Ein weiteres wichtiges Detail beim Erstellen eines Softwareprojektes ist die Struktur der einzelnen Code-Repositories. Ursprünglich wurden die ersten MF Anwendungen in “multi-repositories” erstellt, was bedeutet, dass jedes MF in einem eigenen Repository liegt.
Eine weitere Variante wäre das sogenannte Monorepo, welches ähnlich wie ein “multi-repo” Verbindungen zwischen den unterschiedlichen Anwendungen abbilde, jedoch anstatt

[44] https://www.npmtrends.com/esbuild-vs-parcel-vs-rollup-vs-webpack-vs-browserify

mehrere “repositorys” miteinander zu verbinden, bleiben bei einem Monorepo alle isolierten Code-Teile in einem Repository.
Isoliert bedeutet, dass Monorepos nichts mit monolithischen Anwendungen gemeinsam hat.
Die folgende Grafik veranschaulicht die Varianten eines Code-Repositories.

Abbildung 5.2: Code-Repository-Types (Noel, 2019)

Auch wenn ein “multi-repo” die klassische und weit verbreitete Variante seinen Code zu organisieren darstellt, sind Monorepos in der Webentwicklung weit verbreitet.
Gerade im Enterprise-Umfeld treten Monorepos oft im kolossalen Ausmaß auf. Es wird spekuliert, dass Google das theoretisch größte Code-Repository aller Zeiten besitzt, welches mit tausenden von täglichen “commits” bereits 2015 eine Größe von über 80 TB aufwies.
Rachel Potvin und Josh Levenberg haben hierzu bereits 2016 einen sehr interessanten wissenschaftlichen Artikel mit dem Titel “Why Google Stores Billions of Lines of Code in a Single Repository” herausgebracht. Folgende Grafik veranschaulicht die Zahlen aus dem oben besagten Artikel genauer:

Google repository statistics, January 2015.

Total number of files	1 billion
Number of source files	9 million
Lines of source code	2 billion
Depth of history	35 million commits
Size of content	86TB
Commits per workday	40,000

Abbildung 5.3: “Why Google stores billions of lines of code in a single repository” (Potvin, 2016)

Andere Unternehmen, von denen bekannt ist, dass sie große Monorepos betreiben, sind Microsoft, Facebook und Twitter.[45]

[45] What is monorepo? 1.Juni 2022, Tomas Fernandez

Natürlich hat ein Prototyp nichts mit Google oder anderen Software-Riesen gemein, jedoch scheinen Monorepo und MF gut miteinander zu harmonieren, da beide Technologien dazu entwickelt wurden Skalierungsprobleme zu lösen.

Auch wenn es wie ein Widerspruch klingt, bietet die Kombination von Micro Frontends und Monorepos einige Vorteile:

- Keine Versionskonflikte per Design,
- einfaches Code-Sharing
- optimierte Bundles.

Außerdem können Mikro-Frontends weiterhin separat bereitgestellt und voneinander isoliert werden.[46]

Zusätzlich muss erwähnt werden, dass in Anbetracht des “Problem Pools” aus Kapitel 4.1 ein Monorepo hohes Problemlöse-Potential bietet, dadurch wurde die Wahl auf ein Monorepo begründet.

4.3 Erstellen von Bedingungen für das Testprojekt

Neben den technischen Entscheidungen wie “React, Module Federation und Monorepos” sind weitere Rahmenbedingungen notwendig, um die Funktionen von “Module Federation” als Micro Frontend Framework zu untersuchen und im späteren Verlauf bewerten zu können.
Um den Funktionsumfang einer Micro Frontend Technologie beurteilen zu können müssen folgende Rahmenbedingungen vom Prototyp erfüllt werden:

- Die Webanwendung muss mehrere “routes” bzw. Seiten aufweisen
- Die jeweiligen MF müssen auf horizontale (Komposition innerhalb einer Seite, “horizontal split”) und vertikale (seitenbasiert, “vertical-split”) Anordnung getestet werden.
- Das *Routing* und die Orchestrierung der einzelnen MF (Remotes) durch eine Application-Shell muss genau untersucht werden. Dabei muss untersucht werden, welche Abhängigkeiten zwischen Host/Shell und Remote(s) möglich bzw. erforderlich sind.
- Die Simulation eines Deployments ist wichtig, um zu verstehen, wie die einzelnen “JavaScript Bundles" innerhalb einer MFA angeordnet sind.
- Optional: Testen von unterschiedlichen “Code sharing” Funktionen, z.B. durch externe Bibliotheken (libs), “Host to Remote", “Remote to Remote" .

[46] https://www.angulararchitects.io/en/aktuelles/using-module-federation-with-monorepos-and-angular/

5. Implementierung

In diesem Kapitel wird der Prototyp realisiert. Zu Beginn wird die Software Architektur und Seitenstruktur definiert. Danach erfolgt eine kurze Einführung zur Erstellung eines NX-Monorepos, gefolgt von einer detaillierten Implementationsbeschreibung in dem alle CLI-Befehle aufgeführt und erklärt werden um den Prototyp bis hin zum Deployment fertig zu stellen. Neben dem Deployment werden noch einige Vorzüge des gewählten Entwicklungs-Stacks erläutert, wie Ausfallsicherheit, Developer Experience und Dependency Sharing.

5.1 Software Voraussetzungen

Um das Testprojekt nachzubauen, wird neben *node* und *npm* keine spezielle Software vorausgesetzt.

Eine spezielle Version von *node* oder *npm* wird nicht vorausgesetzt. Die Auflistung der verwendeten Versionen, mit denen der Prototyp entwickelt wurde, kann jedoch für den Debugging Fall hilfreich sein.

Tabelle 5.1 Software Versionen aus denen der Prototyp erstellt wurde (eigene Abbildung)

Bezeichnung	Version	Befehl
Betriebssystem	Linux - Ubuntu 20.04 LTS	lsb_release -a
node-Version	16.17.0	node -v
npm-Version	8.15.0	npm -v

Hinweis: Der Prototyp kann natürlich auch unter Windows oder MacOS nachgebaut werden da *node* und *npm* auf allen Systemen verfügbar ist (Node enthält bereits npm als Packagemanager)

5.2 Prototyp Architektur

Mit Module Federation, wird eine Anwendung aufgeteilt in:

1. Eine **Host** Anwendung, welche externe Anwendungen referenziert.
2. **Remote** Anwendungen, die eine einzelne Domain oder ein Feature abbilden. Ob die Domain nun eine ganze Seite abbildet oder nur eine kleine Subdomain, bleibt dem Team überlassen.

Host und Remote sind technisch gesehen beides MF, der Unterschied ist lediglich die unterschiedliche Rolle.

Im folgenden Teil sieht man ein Beispiel einer Anwendung, welche aus einer Host App (host) und vier Remotes (shop, payment, about, search) besteht.

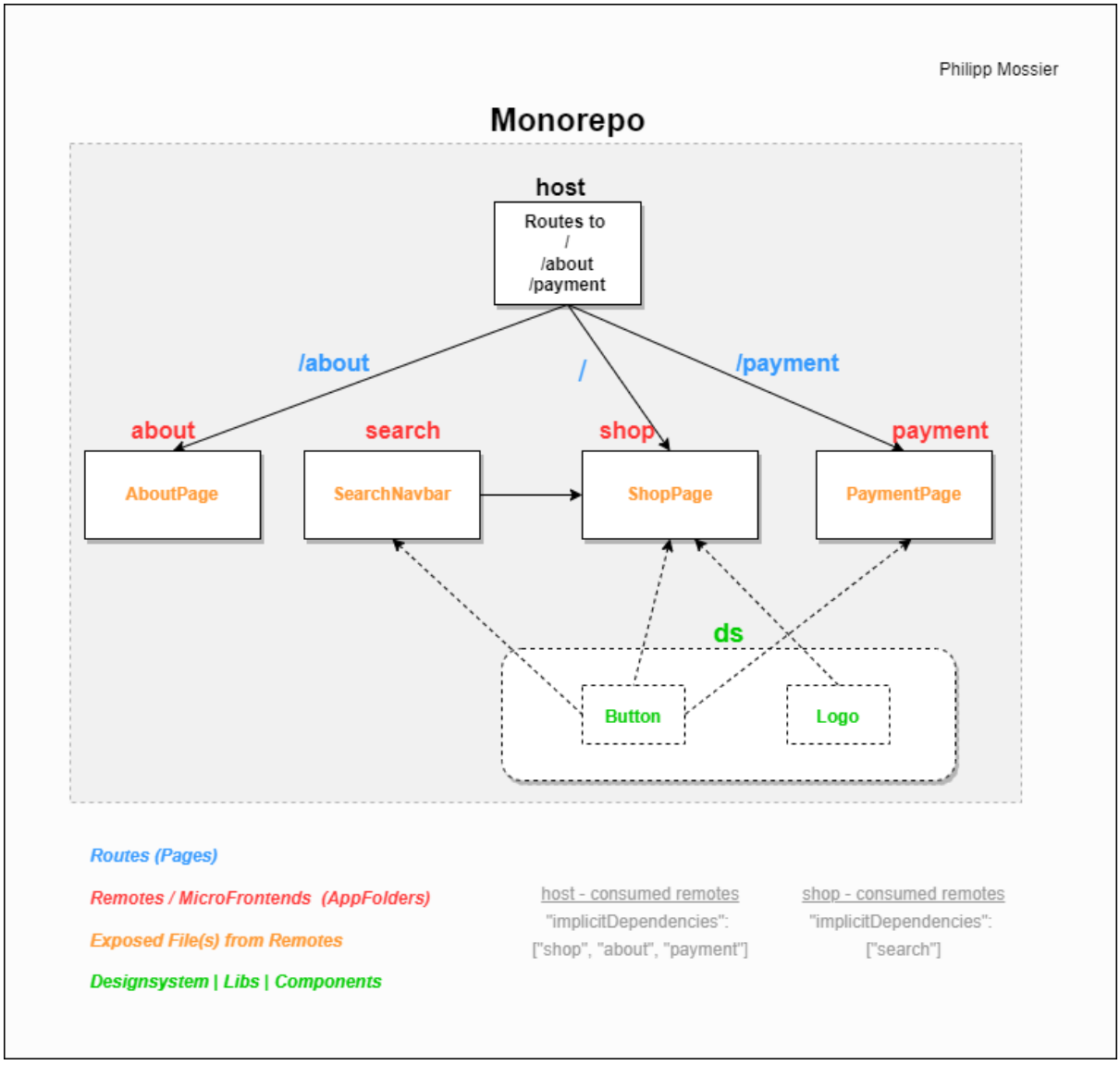


Abbildung 5.4: Prototyp Architektur (eigene Abbildung)

Hinweis: Um eine bessere Übersicht zu gewährleisten, werden alle Bezeichnungen von Abbildung 5.4 in kursiv geschrieben.

Obwohl alle Anwendungen unabhängig voneinander erstellt werden und daher keine Abhängigkeit zwischen ihnen bestehen, kann man sich diese konzeptionell in der folgenden Hierarchie vorstellen.
Dadurch, dass alle Anwendungen unabhängig voneinander erstellt wurden, können diese auch beliebig miteinander kombiniert werden. Das bedeutet dass sich mehrere MF eine

Seite teilen können, wie es hier bei der ShopPage der Fall ist. Die ShopPage ist mittels Basispfad "/" erreichbar und rendert in diesem Fall das search MF und das shop MF. Damit die Host Anwendung weiß wo sie die einzelnen MF aggregieren soll, müssen implizite Abhängigkeiten definiert werden. Dabei ist zu betonen, dass es technisch gesehen gar keine Abhängigkeiten zwischen den einzelnen MF gibt und diese an jeder beliebigen Stelle in der Anwendung platziert werden können.
Im Quellcode bedeutet das, dass man bei Host und Remote die impliziten Abhängigkeiten so definieren muss, dass am Ende die gewünschte Anordnung wie auf dem Bild zu sehen entsteht.

Hinweis: Da der Code einer Codebase typischerweise komplett in Englisch geschrieben wird, verzichtet die nun folgende Implementationsbeschreibung darauf, jeden einzelnen Begriff ins Deutsche zu übersetzen.

5.3 Seitenstruktur

Dieses Kapitel zeigt Schritt für Schritt, wie man eine Host Anwendung mit vier Remotes erstellt. Die *host* App orchestriert dabei drei Remotes und definiert dabei folgende routes:

1. /
2. /payment
3. /about

Bei dem *shop* Remote wurde auf das Prefix "shop" verzichtet, somit ist dieses über den Basispfad "/" erreichbar.

Das vierte noch fehlende MF *search*, bekommt keine eigene Seite zugeteilt, da die Suchleiste auf der Shopseite integriert werden soll.

Diese Konstellation wurde bewusst gewählt, um eine "vertical-split" (Seiten basierte) und eine "horizontal-split" (Komposition basierte) Darstellung zu testen.

Abgesehen davon, kann die Suche und das Filtern von Produkten sehr komplex sein, womit das Heraustrennen eines eigenen *search* MF durchaus Sinn macht.

Wenn man sich nun über die Anordnung der einzelnen MF und die daraus resultierende Architektur im Klaren ist, kann es in die Implementation übergehen.

5.4 NX und das erstellen eines Monorepos

Der Source Code, der aus der nun folgenden Anleitung entsteht, kann öffentlich über dieses Github Repository https://github.com/philippmossier/micro-frontend-shop-demo eingesehen werden.

Der erste Schritt ist das Erstellen eines leeren Arbeitsplatzes. In diesem Fall spricht man von Workspace oder Monorepo. Eine Monorepo stellt eine Sammlung von Apps dar, die auch als MF oder Remotes bezeichnet werden können.
Technisch gesehen handelt es sich bei der Host App auch um ein MF, jedoch hat dieses etwas unterschiedliche Aufgaben und wird eher als Container von unterschiedlichen MF angesehen.

Ein Remote zeichnet sich dadurch aus, dass dieses immer von einem anderen Teil der

Anwendung konsumiert wird (in diesem Fall vom Host).
Der Host wiederum konsumiert/lädt und orchestriert die einzelnen Remote. Das bedeutet nicht, dass ein Remote kein anderes Remotes konsumieren kann. Lediglich das Bereitstellen von Code (expose) ist eine Voraussetzung für ein Remote. In vielen Codebases wird die Host Anwendung auch Shell oder Application-Shell genannt, bedeutet jedoch dasselbe.
Die jeweiligen Kommandozeilen Befehle die zum erstellen des Prototyps verwendet werden, sind in folgende Gruppen unterteilt: - Linux Befehle (cd, mkdir) - Node Package Manager Befehle (npm, npx) - NX-CLI Befehle (nx generate, nx serve)

Da Node und Linux Befehle jedem Frontend-Entwickler und jeder Frontend-Entwicklerin vertraut sein sollten, werde ich lediglich auf die NX-CLI Befehle näher eingehen.

Bei NX (https://nx.dev/) handelt es sich um ein Build System, welches von Frontend Experten entwickelt wurde. NX ist ein Produkt des Unternehmens nrwl (https://nrwl.io/) welches von Angular-Teammitgliedern und ehemaligen Google Mitarbeitern gegründet wurde. NX betitelt deren Produkt als “Smart, Fast and Extensible Build System. Next generation build system with first class monorepo support and powerful integrations”. Da es sich bei NX um ein “Build System & Monorepo Tool” handelt, wird die NX-CLI vorwiegend zum Konfigurieren und Verwalten des Monrepos verwendet.

5.5 Umsetzung - CLI Befehle

Mittels NX-CLI lassen sich viele Befehle ausführen, die man zum Entwickeln einer Frontend Anwendung benötigt. Der eigentliche Quellcode wird durch NX nicht beeinflusst, lediglich das Aufsetzen, Verwalten und Arbeiten innerhalb einer Codebasis wird dadurch erleichtert.

Um die NX-CLI zu verwenden, kann man entweder nx oder npx nx verwenden. Die kürzere Version nx funktioniert jedoch nur, wenn man nx global auf seinen Rechner mittels npm install --global nx installiert. Mit dem prefix npx (was für node package execute steht) kann man auch nicht installierte node packages ausführen. Es ist also ganz egal, ob man npx, nx oder nx verwendet.

Mithilfe von NX lässt sich eine Codebase in verschiedene Projekte aufteilen und das alles in einem zentral verwalteten Monorepo. Die NX-CLI bietet Befehle für den Betrieb und die Verwaltung der verschiedenen Teile der Codebasis. Diese Befehle fallen in drei Kategorien:

- Auf Code reagieren bzw. Code ausführen (build, serve, test, run, run-many…)
- Code ändern/generieren (generate, create-nx-workspace…)
- Verständnis der Codebasis (graph, list..)

Im ersten Schritt muss zuerst ein Monorepo erstellt werden. Mithilfe der NX-CLI Option *--preset* kann man ein Monorepo erstellen, welches bereits auf ein bestimmtes Framework vorkonfiguriert ist, oder einfach ein leeres Monorepo erstellen. In unserem Fall erstellen wir ein leeres Monorepo und bauen uns schrittweise eine Micro-Frontend-Landschaft auf.

```
npx create-nx-workspace@14.2.4 microfrontends-monorepo --
preset=empty --nxCloud=true
```

```
cd microfrontends-monorepo
```

Hinweis: Die Version 14.2.1 war zum Zeitpunkt der Entwicklung die neueste stabile Version. NxCloud kann, muss aber nicht verwendet werden, da diese lediglich die "Build"-Zeiten durch Caching verkürzt. Da nxCloud jedoch umsonst ist und keine Registrierung benötigt, kann diese problemlos ausgewählt werden.
Dieser Befehl erstellt ein minimal vorkonfiguriertes Monorepo setup mit folgender Ordnerstruktur:

```
├── apps
├── libs
├── node_modules
└── tools
```

Nutzer des Frameworks React müssen noch das Plugin @nrwl/react installieren.

```
npm install --save-dev @nrwl/react@14.2.4
```

Mittels @nrwl/react lassen sich nun Host- und Remote-Applikationen generieren.

```
npx nx generate @nrwl/react:host host --
remotes=shop,payment,about,search --style=@emotion/styled
```

Nun sollten folgende 5 Anwendungen im “apps” Ordner zu finden sein. (Exklusive der Auto generierten “e2e” Ordner, welche nur für End-to-End Tests relevant sind.)

```
├── about
├── host
├── payment
├── search
├── shop
```

Hinweis: Wenn man die Option “–style” weglässt, dann bekommt man von der CLI eine Reihe von Stylesheet-Formaten zur Auswahl. Mit welchen Tools man sein APP “stylen” möchte, bleibt jedoch den EntwicklerInnen überlassen. NX bietet dabei eine umfangreiche Auswahl an, angefangen vom klassischen “CSS” bis hin zu einigen “CSS in JS” Lösungen.

Mit folgendem Befehl kann nun die host Anwendung gestartet werden:

```
npx nx serve host --open
```

Der obige Befehl serviert den Host im Entwicklungsmodus, während die Remotes statisch erstellt und bereitgestellt werden. Das heißt, Änderungen am Host aktualisieren sein Bundle, aber Änderungen an Remotes werden nicht aktualisiert.

Wenn man auch die Remotes dynamisch im Entwicklermodus starten möchte kann man dies mit der option --devRemotes tun:

```
npx nx serve host --open --devRemotes=shop,cart
```

Hinweis: Beide Befehle servieren das ganze System. Mittels "–devRemotes" teilt man mit, welche Teile sich davon ändern werden. Dies bedeutet, dass bei Befehl "npx nx serve host –open" die Remotes nicht automatisch auf Änderungen im Code reagieren und diese nicht sofort im Browser sichtbar machen. Nur ein "browser page refresh" macht die Code Änderungen im Browser sichtbar. (deswegen die Bezeichnung statisch und nicht dynamisch)

Was wurde generiert ?

Um zu verstehen, wie Module Federation mit NX funktioniert, sind drei Dateien für die Kontrolle dieses Features verantwortlich:

1. apps/host/project.json
2. apps/host/webpack.config.js
3. apps/about/module-federation.config.js (dasselbe gilt für die restlichen Remotes)

1. project.json

In dieser Datei sieht man die zugeordneten Tools der jeweiligen NX-CLI-Befehle. Bei dem *build* Befehl sieht man, dass Webpack als Executor eingetragen ist. Somit ist auch hier Webpack für das Bündeln des Codes zuständig, ganz gleich wie auch im Großteil aller heutigen SPA.
Semantisch gesehen, bilden der Host und die jeweiligen Remotes eine Anwendung, sodass man den Host nicht ohne Remotes erstellen kann. Um nun der Architektur aus Abbildung 5.4 zu folgen, müssen in der project.json File des Hosts die zugehörigen Remotes als implicitDependencies angegeben werden.

```
// apps/host/project.json
{
  //...
  "implicitDependencies": ["shop","payment","about"]
}
```

2. webpack.config.js

Um Webpack zu konfigurieren stellt NX eine Helferfunktion namens *withModuleFederation* zur Verfügung.

```
// apps/host/webpack.config.js
const withModuleFederation = require('@nrwl/react/module-federation');
const moduleFederationConfig = require('./module-federation.config');

module.exports = withModuleFederation({
  ...moduleFederationConfig,
});
```

Auf den ersten Blick passiert hier noch nicht viel, da die eigentliche MF Konfiguration aus der Datei *module-federation.config.js importiert wird.*

Wenn man sich diese Datei nun genauer ansieht, erkennt man sofort, wie es um die Abhängigkeiten von Host und Remotes steht.

```
// apps/host/module-federation.config.js
module.exports = {
  name: 'host',
  remotes: ['shop','payment','about'],
};
```

Der erforderliche Wert “name” ist die Magie, um den Host und die Remotes miteinander zu verbinden. Die *host* Anwendung verweist auf drei Remotes mit ihrem Namen.

Was bedeutet die durch NX bereitgestellte Funktion *withModuleFederation* ?

Bei dieser Funktion handelt es sich um eine Abstraktion von Webpacks Module Federation Plugin. Somit vereinfacht NX die Webpack Konfiguration für Module Federation. Im Vergleich dazu würde eine Standard Module Federation Konfiguration ohne NX so aussehen:

```
// Standard webpack.config.js without NX
new ModuleFederationPlugin({
  name: 'host',
  filename: 'hostRemoteEntry.js',
  remotes: {
    shop: 'shop@http://localhost:4201/shopRemoteEntry.js',
    payment: 'payment@http://localhost:4202/paymentRemoteEntry.js',
    about: 'about@http://localhost:4203/aboutRemoteEntry.js',
  },
  shared: {
    react: {
      singleton: true,
      requiredVersion: PACKAGE.dependencies.react
      },
    "react-dom": {
      singleton: true,
      requiredVersion: PACKAGE.dependencies.react
      },
```

```
    "react-router-dom": {
      singleton: true,
      requiredVersion: PACKAGE.dependencies.react
    },
   }
})
```

Hier sieht man, dass ohne NX zusätzlich “filename” und “shared” definiert werden muss. Während “filename” nur auf den “bundle” Namen referenziert, handelt es sich bei “shared” um eine ganz wichtige Funktion, die Module Federation bietet, welche im nächsten Kapitel genauer ausgeführt wird.
Wenn man nun die Standard “webpack.config.js” mit jener von NX vergleicht, sieht man das NX einige Zusatzfunktionen im Umgang mit MF zur Verfügung stellt wie:

- Alle Bibliotheken (npm und Arbeitsbereich) sind standardmäßig gemeinsam genutzte “singletons”, sodass man diese nicht manuell mittels “requiredVersion” konfigurieren muss.
- Bei Remotes wird nur auf den Namen verwiesen, da Nx weiß, auf welchen Ports jedes Remote läuft (im Entwicklungsmodus).

3. module-federation.config.js

Im Vergleich zu der “module-federation.config” der host Anwendung, definiert man bei einem Remote immer den Wert “exposes”, welcher den Pfad zu den geteilten Dateien definiert. Dies kann ein Pfad sein der die kompletten Anwendung teilt z.B. durch “src/App.tsx” oder auch nur einen kleinen Teil vom Remote teilt wie z.B. eine Button Komponente mittels “src/components/Button.tsx”.

```
// apps/about/module-federation.config.js
module.exports = {
  name: 'about',
  exposes: {
    './Module': './src/remote-entry.ts',
  },
};
```

Im Grunde bedeuten die Werte “remotes” (welche MF konsumiert werden) und “exposes” (was vom eigenen MF geteilt wird). Dabei schließt das eine das andere nicht aus, was bedeutet, dass ein MF seinen Code mittels “exposes” zur Verfügung stellen und gleichzeitig beliebig viele MF mittels “remotes” konsumieren kann. Das Konsumieren von MF ist somit nicht auf den Host beschränkt. Schlussendlich bestimmen diese 2 Werte die Struktur einer “Module-Federation-Anwendung”.

Angenommen, es gäbe ein Team, welches sich um die Suche und das Filtern von Produkten kümmert, und ein MF bzw. Remote mit dem Namen search erstellt. Dieses search remote soll dann im weiteren Verlauf auf der “/” Route dargestellt werden. Wenn man sich nun noch einmal die Abbildung 5.4 ansieht, erkennt man das auf der “/” Route die ShopPage gerendert wird. Nun soll sich also das shop-remote mit dem search remote eine Seite teilen.

Das bedeutet, dass man nun nicht mehr von einer vertikalen Aufteilung von MF spricht (“vertical-split”), sondern von einer horizontalen Komposition (“horizontal-split”).

Dieses Ergebnis könnte nun ganz einfach mit folgender “module-federation.config.js” erreicht werden:

```
// apps/about/module-federation.config.js
module.exports = {
  name: 'shop',
  exposes: {
    './Module': './src/ShopPage.tsx',
  },
  remotes: ["search"]
};
```

Dies setzt natürlich voraus, dass man innerhalb der Datei ShopPage.tsx auch das search remote an der gewünschten Stelle im Code platziert. Anhand dieses Beispiels erkennt man sofort die hohe Flexibilität, die einem Module Federation bietet.

5.6 *sharing dependencies* via Module Federation

Wenn man noch einmal einen Blick auf die Funktion withModuleFederation wirft, sieht man das in der “webpack.config.js” die Option “shared” definiert wurde. Bei dieser Funktion handelt es sich um Webpacks Alleinstellungsmerkmal Abhängigkeiten innerhalb der Micro Frontend Landschaft zu teilen.

Während manche MF-Frameworks keine Möglichkeiten bieten, "dependencies" zu teilen, bieten andere MF-Frameworks wie “single-spa” oder “piral” nur bedingt Möglichkeiten an. Bei “single-spa” könnte man diese über sogenannte “import-maps” teilen, welche von einigen Browsern jedoch nicht unterstützt werden.

Dabei ist das Teilen von “dependencies” von sehr großer Bedeutung, da dies großen Einfluss auf die “bundle-size” einer Webseite hat. Wenn eine Seite z.B. aus mehreren MF besteht (“horizontal-split”) und auf dieser mehrmals dasselbe Framework, eventuell noch mit unterschiedlichen Versionen geladen wird, dann kann das drastische Folgen für die Performance einer Webapplikation haben.

Module Federation, bietet mittels “shared” eine sehr einfache und umfangreiche Funktion, "dependencies" zu teilen. Zusätzlich hat man durch die Optionen “singleton”, “requiredVersion” und “eager” die vollständige Kontrolle über das Teilen von Abhängigkeiten. Mittels “singleton” lässt sich steuern, dass die Abhängigkeit nur einmal geladen werden darf.

Hinweis: Für den Produktionsmodus muss man *url* und *port* manuell in “webpack.config.prod” angeben, da NX nicht wissen kann, auf welchen externen Container oder Server die “remote bundle files” abgelegt wurden. Mehr Details zu Deployment gibt es in Kapitel 5.8

5.7 Code Sharing Möglichkeiten mit NX-Monorepos

Das Teilen von Code, speziell innerhalb eines Module Federation Monorepos, ist nicht nur auf MF (remotes) limitiert. Es könnte auch eine Authentifizierungs-Bibliothek (auth-lib) zwischen mehreren Remotes geteilt werden. Diese *auth-lib* würde man in so einem Fall auch als *common-library* bezeichnen, dessen Code ebenfalls im Monorepos untergebracht ist. Alle anfallenden common libraries könnte man so in einem lib Ordner unterbringen, welcher bei einem Deployment isoliert behandelt wird.

In einem NX monorepo werden diese libraries typischerweise in path mappings definiert, welche entweder innerhalb tsconfig.json oder tsconfig.base.json definiert werden (abhängig vom Projekt Setup)

```
// tsconfig.json or tsconfig.base.json
"paths": {
    "@common/auth-lib": [
        "libs/auth-lib/src/index.ts"
    ]
},
```

Zusätzlich zu den MF, welche im “app” Ordner untergebracht sind, finden “common libraries” im “lib” Ordner Platz, was eine weitere Möglichkeit bietet, Code-Teile voneinander zu isolieren.

Diese Form von Ordnerstruktur macht es einfach, Code entlang der Anwendung bzw. der MF zu teilen. Zusätzlich gibt ein Monorepo immer einen guten Überblick über die gesamte Softwarelandschaft.

Oft werden Monorepos kritisiert, weil man denkt, dass die enorm wichtige Funktion von separaten Builds dadurch verloren geht, jedoch ist dies bei modernen Monorepos durch die richtige Konfiguration kein Problem mehr.
Separate Builds sind bei Softwareprojekten, an denen unterschiedliche Teams arbeiten, von hohem Wert, da die einzelnen Teams damit unabhängige “release cycles” fahren können. Im nächsten Kapitel werden Deployments innerhalb eines NX Monorepos getestet.

Durch die Verwendung von Module Federation wird der Anwendungs Erstellungsprozess (application build process) im Wesentlichen vertikal aufgeteilt. Auch eine horizontale Aufteilung ist möglich, indem man einige Bibliotheken (libraries) “buildable” macht. NX empfiehlt es nicht, alle Bibliotheken im Arbeitsbereich “buildable” zu machen, aber in einigen Szenarien kann es Continuous Integration (CI) Prozesse beschleunigen, wenn einige große Bibliotheken “buildable” gemacht werden.

Da die Distributed Tasks Execution von Nx Cloud mit jedem Task-Graph funktioniert, wird das Vorhandensein von “buildable libraries” automatisch gehandhabt. Wenn man z.B. eine Komponentenbibliothek von denen alle MF abhängen, erstellt NX Cloud zuerst die Bibliothek, bevor die einzelnen MF erstellt werden.

5.8 Deployment

Da Bibliotheken normalerweise keine Versionen in einem Monorepo haben, sollten immer alle geänderten MF zusammen erneut deployed werden. Nx nimmt einem viel Arbeit ab, indem es dabei hilft die geänderten Code-Teile aufzuzeigen damit man sofort weiß welche Teile neu deployed werden müssen und welche unverändert blieben.

Mit folgendem Befehl kann man den aktuellen “feature branch” gegen den “main branch” vergleichen.

```
nx print-affected --base=main --head=HEAD
```

Für das Deployment sind folgende 2 Dateien von großer Bedeutung. webpack.config.prod.js und “project.json”

Innerhalb von webpack.config.prod.js ist es wichtig, die jeweiligen remotes und deren URLs anzugeben.

```
// apps/host/webpack.config.prod.js
const withModuleFederation = require('@nrwl/react/module-
federation');
const moduleFederationConfig = require('./module-
federation.config');

module.exports = withModuleFederation({
  ...moduleFederationConfig,
  remotes: [
    ['shop', 'http://localhost:3000/shop'],
    ['payment', 'http://localhost:3000/payment'],
    ['about', 'http://localhost:3000/about'],
  ],
});
```

Da dieses Test Deployment lokal simuliert wird, werden auch nur localhost URLs verwendet. In einer Produktionsumgebung müssen diese gegen echte Pfade ausgetauscht werden, wie z.B. example.com/path/to/app1/remoteEntry.js

Als nächstes müssen die impliziten Abhängigkeiten definiert werden. Hierzu müssen die beiden project.json Dateien angepasst werden (*host* und *shop* remote)

```
// apps/host/project.json
{
  //...
  "implicitDependencies": ["shop","payment","about"]
}
```

Jetzt wo die impliziten Abhängigkeiten des *hosts* geklärt sind, fehlt noch das *shop* remote, welches das *search* remote als implizite Abhängigkeit aufweist.

```
// apps/shop/project.json
{
  // ...
  "implicitDependencies": ["search"],
```

Um die Verlinkung des Hosts und der Remotes zu überprüfen, kann folgenden Befehl ausgeführt werden:

```
nx graph
```

Wenn bei der Konfiguration alles richtig gemacht wurde dann sollte die daraus generierte Bilddatei wie folgt aussehen:

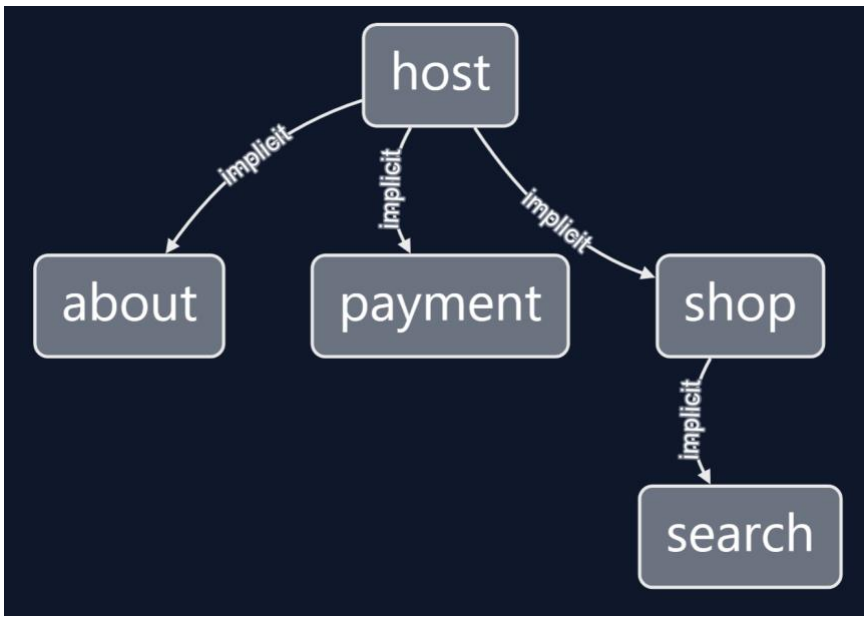


Abbildung 5.5: Generierte Bilddatei aus dem Befehl “nx graph” (zu finden im Github Repository des Prototyps)

Jetzt wo man sichergehen kann, dass die Projekte richtig verlinkt sind, kann der Befehl:

```
npx build host
```

ausgeführt werden, um alle impliziten Abhängigkeiten im Produktionsmodus zu bauen.

Im “dist” Ordner sollte nun folgende Ordnerstruktur zu finden sein :

```
dist/apps
├── about
├── host
├── payment
├── search
└── shop
```

Da nun alle Datenpakete für einen "production build" vorhanden sind, kann ein Deployment mit folgenden Befehl simuliert werden:

```
npx nx g @nrwl/workspace:run-command deploy --project=host \
--command="rm -rf production && mkdir production && \
cp -r dist/apps/host/* production && \
cp -r dist/apps/shop production && \
cp -r dist/apps/payment production && \
cp -r dist/apps/search production && \
cp -r dist/apps/about production && \
http-server -p 3000 -a localhost production"
```

Zusätzlich sorgt NX dafür, dass der obige Befehl dem Kürzel “nx deploy host” zugewiesen wird, indem es diesen unter “apps/host/project.json” speichert. Dadurch wird der Befehl nicht nur im richtigen Projektordner gespeichert, sondern kann auch in Zukunft mit dem Kürzel “nx deploy host” ausgeführt werden. Die lokal bereitgestellte Anwendung kann nun unter *http://localhost:3000* im Browser aufgerufen werden. Wenn man sich nun den “production” Ordner ansieht, dann sollten alle 4 Remotes die Datei *remoteEntry.js* enthalten. Die Host Anwendung ist dadurch erkennbar, dass diese keine *remoteEntry.js* Datei beinhaltet und deren Dateien den anderen Ordnern übergeordnet sind, was folgender “filetree” noch einmal verdeutlicht.

```
production/
├── about
│   ├── remoteEntry.js
│   └── (...)
├── assets
│   └── .gitkeep
├── payment
│   ├── remoteEntry.js
│   └── (...)
├── search
│   ├── remoteEntry.js
│   └── (...)
├── shop
│   ├── remoteEntry.js
│   └── (...)
├── index.html
└── (...)
```

Anhand dieses Test Deployments kann man gut erkennen, wie NX separate Builds handhabt, die man normalerweise nur aus getrennten Repositories kennt.

5.9 Developer Experience mit React & Module-Federation

Da es sich bei Module Federation um ein sehr leichtgewichtiges Plugin von Webpack handelt, gibt das den Entwickler/-innen die Flexibilität, ihre Projekte so zu bauen, wie sie es wollen, ohne sich an ein starres Framework richten zu müssen.

Da Module Federation Komponenten zur Laufzeit integriert, entstehen dadurch einige Vorteile gegenüber anderen MF Lösungen:

- Es wird nicht das gesamte von einer anderen App generierte Bundle heruntergeladen, sondern nur die benötigte Komponente.
- Niedrige Lernkurve, da so gut wie kein Boilerplate notwendig ist (nur webpack.config.js muss richtig konfiguriert werden. Bei Verwendung eines Monorepos wie NX muss zusätzlich auch die Datei project.json konfiguriert werden)
- Importieren von externen Komponenten zur Laufzeit fühlt sich so an, als wären diese lokal verfügbar.
- Durch das importieren von externen MF zur Laufzeit, kann man gut auf Fehler reagieren und Fallbacks einbauen (Siehe Beispiel aus Kapitel 5.10)

Verglichen mit einem anderen MF-Framework, wie z.B. single spa, werden dort MF Framework Diagnostisch auf html Ebene importiert. Dies ermöglicht zwar, MF aus verschiedenen FE-Frameworks in die Shell zu laden (bei single-spa spricht man anstatt von einer shell, oft von einer root-config welche eine index.ejs mit dem erwähnten html enthält) jedoch bringt single-spa's Variante MF zu orchestrieren einige Nachteile mit sich:

- MF können nicht als Module importiert werden (fühlt sich nicht wie eine SPA an, erschwert error handling und Integration mit anderen MF)
- Viel Boilerplate in der shell bzw. root-config
- Das Verwenden von single-spa macht nur Sinn, wenn unterschiedliche FE-Frameworks im Einsatz sind.
- Höhere Lernkurve (Weiter entfernt von einer typischen SPA)

Warum das Importieren von MF auf Modulebene so nützlich ist, sieht man im nächsten Kapitel, wenn es um die Stabilität von federated MF geht.

5.10 Hohe Stabilität der Anwendung durch separate Builds und MF-Versionierungs *Fallbacks*

Da Module Federation es ermöglicht, externe Module bzw. MF zur Laufzeit (runtime) zu laden, kann auch zur Laufzeit auf Fehler von externen MF reagiert werden. Sogenannte Version Rollbacks könnten somit die Downtime Risiken einer Anwendung drastisch reduzieren. Um zu demonstrieren wie wertvoll solch ein Feature für eine Produktionsumgebung sein könnte, stelle man sich folgendes Szenario vor:

Man nehme an, jedes MF *(shop, about, search, payment, host)* wäre einem eigenen Team zugeordnet. Aktuell ist jedes MF mit der Version 1.02 live, jedoch möchte das Payment Team ein Update ihres *payment* remotes bereitstellen, welches morgen mit der Version 1.03 live geht. Dabei ist dem *payment* Team nicht bewusst, dass Version 1.03 einen Fehler

enthält, welcher die *PaymenPage* nicht nur unbenutzbar macht, sondern den User auch daran hindert, einen Produktkauf abzuschließen.

Um nun zu verhindern, dass der User vor einer fehlerhaften *PaymentPage* steht, müsste man der Anwendung beibringen, im Falle eines Fehlers, einen Fallback auf Version 1.02 des *payment* MF zu verwenden.

Da Module Federation dazu in der Lage ist, Code bzw. MF zur Laufzeit (runtime) zu laden, kann solch ein Fallback relativ mühelos eingebaut werden. Eine Implementation solch eines Fallbacks ist im Quellcode unter *apps/host/src/app/app.tsx* zu finden.

```
<Route
   path="/payment"
   element={
     <Payment
       delayed={<div>Loading payment MF...</div>}
       error={<PaymentOld/>}
     />
   }
 />
```

Um ein MF im Fehlerfall mit dessen Vorversion abzusichern, muss man lediglich 2 Versionen des MF (remoteEntry.js) bereitstellen. Da es sich hierbei um eine statische Datei handelt, muss nicht zwingend ein Cloud Speicher wie z.B. Amazon S3 verwendet werden, sondern es reicht auch ein CDN völlig dafür aus.

6. Analyse und Auswertung der Ergebnisse

In diesem Kapitel werden 3 Bereiche ausgewertet. Als erstes wird der Erfolg des Prototyps in Hinblick auf den Problempool aus Kapitel 4.1 ausgewertet. Der zweite und dritte Schritt umfasst 2 Vergleiche mittels Weighted Scoring Modell, in dem einmal das Monorepo mit einem Multi-Repo verglichen wird und zum zweiten 5 unterschiedliche MF-Frameworks verglichen werden.

6.1 Auswertung der MF-Problemanalyse

Der aus Kapitel 4.1 erstellte MF-Problempool wird nun anhand des gewählten Entwicklungs-Stacks (React, Module Federation und & Monorepo) ausgewertet:

Tabelle 6.1 MF-Problemanalyse in Hinblick auf den gewählten Entwicklungs-Stack

		React, Module Federation & NX-Monorepo - Entwicklungsstack Problemanalyse
1	Infrastruktur	Aufwandsminiminierung durch Monorepo
2	Ausfallrisiko	MF-Versions-Fallbacks auf Komponentenebene durch Module Federation & React
3	Code Sharing	Federated Modules, common-libs, Laufzeit & Buildzeit Sharing
4	Performance	Auto-Code-Splitting,Nested-Routes,Dependency-Management,React-Suspense
5	Overhead	SPA-ähnlich durch Moduel Federation
6	Developer Experience	Monorepo Vorzüge: Single-PR, Single-IDE, Easy-Startup, CLI, Überblick
7	Code Duplication	Einfacher über Code Duplikation zu stolpern
8	Knowledge Transfer	Gemeinsame Code Basis verringert Barrieren
9	Kommunikation	Erleichtertes Teilen von Code, ansonsten kein Effekt
10	Design Konflikte	Hängt von der CSS Lösung ab - CSS in JS hat keine Konflikte
11	Inkonstistente Designs	Hängt vom Design System ab - Design Tokens könnten eine Lösung sein

6.1.1 Aufwendige Infrastruktur

Auch wenn das Erstellen eines CI/CD Prozesses für ein Monorepo komplexer ist, muss dieser verglichen mit multiplen Repositories nur einmal erstellt werden, da sich alle MF in einem Repository befinden. Wie man die Bundles schlussendlich aufteilen möchte, bleibt den Entwickler/-innen überlassen, da NX und Module Federation hier größtmöglichen Freiraum bieten. Mit NX-Cloud hat man noch zusätzlich die Möglichkeit, Build-Zeiten drastisch durch Caching zu reduzieren.

6.1.2. Ausfallrisiko

Um hohe Stabilität und gute Code Sharing Eigenschaften gewährleisten zu können, benötigt man eine skalierbare Lösung für die gemeinsame Nutzung von Knoten Modulen und Funktionen-/Anwendungscode. Dies muss zur Laufzeit geschehen, um adaptiv und dynamisch zu sein. Externals[47] leisten keine effiziente oder flexible Arbeit. Import-Maps[48] lösen keine Skalierungsprobleme. Man möchte nicht nur Code herunterzuladen, um

[47] https://webpack.js.org/configuration/externals/
[48] https://www.honeybadger.io/blog/import-maps/

Abhängigkeiten zu teilen, es braucht eine Orchestrierungsebene, die Module dynamisch zur Laufzeit mit Fallbacks teilt.[49]
Module Federation bietet all das. In Kapitel 5.10 wurde solch eine Orchestrierungsebene geschaffen (Anwendungscode der Host-MF), welche MF mit Fallbacks lädt.

6.1.3. Schlechtes Code Sharing

Die verbessertes Code Sharing Eigenschaften eines Monorepos sind enorm. Abhängigkeiten wie Bibliotheken, Frameworks etc. werden an einer zentralen Stelle gehalten. Das Updaten dieser Dependencies entlang der kompletten Softwarelandschaft wird dadurch immens erleichtert. Auch das Teilen von common-libs ist in einem Monorepo einfacher, da man sich die mühsame Versionierung spart, die man von node-packages kennt. Der CLI-Befehl “nx graph” hilft dabei, die Relationen der einzelnen Packages, MF und Dependencies im Blick zu behalten.
Wie souverän NX das Erstellen von separaten Builds meisterte, konnte man bereits in Kapitel 5.8 sehen. Ob MF oder common-lib, NX hat für beides bereits Guidelines, an die man sich halten kann und glänzt mit einer außerordentlich gut strukturierten und mit unzähligen Beispielen gefüllten Dokumentation. NX, gepaart mit der hohen Flexibilität von Module Federation, macht dieses Setup zu einer Code-Sharing Kombination, die ihresgleichen sucht. Die Möglichkeit, Dependencies zu sharen, funktionierte bei Module Federation auf Anhieb, ohne großen Aufwand. Bei anderen MF-Frameworks gab es entweder gar keine Möglichkeit, Dependencies zu sharen oder man erhielt nur bedingte Unterstützung durch weniger performante *import-maps* oder *externals*.

6.1.4. Performance Einbußen

Module Federation stellt sicher, dass Remotes von den Host dependencies abhängen. Wenn der Host keine Abhängigkeiten hat, lädt das Remote seine eigenen. Das bedeutet kein doppelter Code, sondern integrierte Redundanz. Dies sorgt für kleinere Bundles und bessere Performance. Dadurch dass Module Federation sehr leichtgewichtig ist, ermöglicht das auch höchste Flexibilität in Sachen Routing. So könne man die Routing Strategie selbst wählen und könnte man im Fall von React, den äußerst beliebten react-router verwenden, welche mit “nested-routes” einen hohen Performance Boost bietet, da dieser erst bei aufrufen “genesteder” routes Code oder Requests aufruft. Die bessere Performance durch “Nested-routes” machen sich vor allem bei den requests bemerkbar da sie den “Waterfall” nicht beim ersten laden einer Website mit “requests” verstopft die eventuell gar nicht, oder erst zu einem viel späteren Zeitpunkt benötigt werden.

Auch die Fähigkeit, zwischen MF und verschiedenen Seiten zu springen, ohne die Seite neu laden zu müssen, ist ein Feature von Module Federation. Automatisches Code-Splitting sorgt dafür, dass das Bundle in mehreren Etappen und ebenfalls nur bei Bedarf geladen wird.
Selbst wenn jede Seite einer Website unabhängig, bereitgestellt und kompiliert wird, integriert Webpack diese Anwendung am Ende so, als wäre diese ein großer Webpack-Build ohne jegliche Seiten-Neuladungen beim Ändern der Route.

[49] https://indepth.dev/posts/1173/webpack-5-module-federation-a-game-changer-in-javascript-architecture

6.1.5. MF-Overhead

Nach einem Vergleich von unterschiedlichen MF-Frameworks benötigte Module Federation den geringsten Zusatzaufwand. Man spürte sofort die Leichtgewichtigkeit des Webpack Plugins, verglichen zu einem MF-Framework wie single-spa, podium oder MF-Plattformen wie mashroom-portal oder piral.
Das dynamische Laden von Modulen zur Laufzeit sorgte dafür, dass sich das Arbeiten mit Federated MF sich wie eine herkömmliche SPA anfühlt.

Micro Frontends sind ursprünglich entstanden um organisatorische Probleme mittels erhöhter technischer Komplexität zu lösen, jedoch ist im Fall von Module Federation diese erhöhte Komplexität so niedrig gehalten das man abgesehen von der “webpack-config” gar keinen MF spezifischen “glue-code” benötigt.

6.1.6. Developer Experience

Während der Entwicklung des Prototyps konnten verglichen mit einem multi-repo single-spa Projekt folgende Vorteile erkannt werden.

- MF übergreifende Features benötigten nur einen Pull Request
- Eine Entwicklungsumgebung (IDE) reicht aus um alle MF zu bearbeiten
- Die komplette Anwendung (mehrere MF) lässt sich einfacher starten als in einem multi-repo setup (NX stellt einige MF CLI Befehle zur Verfügung)
- Verbessertes Code-Sharing durch Monorepo und Module Federation. common-libs lassen sich viel einfacher erstellen und teilen als herkömmliche npm-packages.
- Das Monorepo Setup gab den besten Überblick über die gesamte Softwarelandschaft.
- Die Grenzen den einzelnen MF und common-libs (wie z.B. ein Design-System) ließen sich wunderbar mit dem Befehl *nx graph* aufzeigen

Anmerkungen zu nicht oder nur zum Teil gelöster Probleme

6.1.7. Code Duplication

Die Chance über Code zu stolpern, den man in anderen Teilen der Anwendung wiederverwenden könnte, ist zwar in einem Monorepo höher, jedoch bleibt dieses Problem ansonsten ungelöst. Da es jedoch viel einfacher ist, Code entlang eines Monorepos zu teilen, kommt man seltener in die Versuchung, Code zu duplizieren.

6.1.8. Schechter Knowledge Transfer

Eine gemeinsame Codebasis macht es einfacher, wiederverwendbare Features, Funktionen und Dienstprogramme zu finden. Ansonsten konnten durch das Monorepo keine Vorteile zu diesem Punkt erkannt werden.

6.1.9. Kommunikationsprobleme

Ähnlich wie Punkt 8 erleichtert ein Monorepo das Teilen von Code und erleichtert eventuell auch das Teilen von Wissen und Informationen, jedoch sind dies lediglich Vermutungen, welche schwer zu überprüfen sind.

6.1.10. Überlappende Designs (stylesheets)

Innerhalb eines Monorepos möge es vielleicht einfacher sein, mehrere Stylesheets zusammenzufassen. React, Monorepos oder Module Federation ändern jedoch nichts an diesem Problem.

Während der Implementation konnten jedoch unabhängig vom gewählten Entwicklungs-Stack folgende Erkenntnisse zu diesem Thema gemacht werden:

- CSS-in-JS[50] Lösungen haben kein Problem mit “className Konflikten”, da anstatt Stylesheets JavaScript zum Designen der Anwendung verwendet wird.
- Bei multiplen stylesheets können *unique hash prefixes* das Problem von doppelten classnames lösen (jedes MF versieht ihre classNames mit einem hash prefix z.B. “.className-67f89dd87sf89ds”)[51]

Dieser Vergleich hat zwar nichts mit dem gewählten Entwicklungs-Stack zu tun, jedoch bietet CSS-in-JS eine einfache Lösung zu diesem Problem.

6.1.11. Inkonsistente Designs

Auch hier bot der Prototyp keine Vorteile, da dieses Problem aus meiner Sicht nur mit globalem Theming zu lösen ist. Es müsse vielmehr ein eigener Design-Layer geschaffen werden, der den anderen MF sogenannte Design-Tokens zur Verfügung stellt. Einige Unternehmen stecken auch sehr viel Zeit in ein Design System, um Komponenten im eigenen Unternehmen wiederverwendbar zu machen.

Aus meiner Sicht ist ein Design System viel zu aufwändig, da es zu spezifisch und wartungs hungrig ist. Hingegen dazu, stellen Design-Tokens eine weitaus flexiblere und weniger aufwändige Lösung dar. Hier gibt es bereits weit verbreitete Open Source Lösungen wie z.B. tailwindcss[52], welche ein hervorragendes *Theming* bieten, ohne dabei Entwickler/-innen einzuschränken. Bei tailwindcss innerhalb einer MF Umgebung gibt es jedoch ebenfalls das Problem von überlappenden classnames. Auch hier gilt: In Kombination mit CSS-in-JS treten diese Design-Konflikte nicht mehr auf (tailwindcss in Kombination mit twin-macro[53])

[50] https://en.wikipedia.org/wiki/CSS-in-JS
[51] https://single-spa.js.org/docs/ecosystem-css/#scoped-css
[52] https://tailwindcss.com/
[53] https://github.com/ben-rogerson/twin.macro

6.2 Monorepo & Multi-Repo Vergleich mittels Weighted Scoring Modell

Die größten Unterschiede zwischen Monorepo und Multirepo bestehen im Bereich Developer Experience, Code Sharing und Feature Angebot. In Kapitel 6.1 wurden die Vorzüge im Bereich DX und Code Sharing bereits angesprochen. Beim Feature Angebot bietet das Monorepo durch zahlreiche Plugins und CLI Unterstützung einige Vorteile wie Startup-Scripts, Code Generation (nx generate) oder andere Hilfsfunktionen wie (nx graph).

Im Bereich Performance und Lernkurve kann das herkömmliche Multirepo glänzen, da das Laden von Millionen von Codezeilen bei riesigen Monorepos zum Bottleneck werden kann. Da NX jedoch ausgeklügelte Caching-Strategien bietet, fällt der Unterschied nicht ganz so dramatisch aus. In Sachen Lernkurve fällt zwar das Onboarding innerhalb eines Monorepos leichter, jedoch müssen die Monrepo-spezifischen Abläufe erst einmal gelernt werden.

Da NX ebenfalls separate Builds ermöglicht, konnten im Bereich Infrastruktur keine großen Unterschiede erkannt werden. Das Erstellen der Infrastruktur fällt zwar bei einem Monorepo komplexer aus, jedoch erspart man sich wiederholte Arbeitsschritte, da man im Vergleich zum Multirepo nur ein Repository betreuen muss.

In folgender Tabelle sieht man einen Vergleich eines herkömmlichen Multi/Poly Repositories mit einem NX-Monorepo:

Tabelle 6.2 Weighted Scoring Modell - Vergleich von NX-Monorepo & Multi/Poly repo

Kategorie	Performance	Lernkurve	DX	Code Sharing	Infrastruktur	Features	SCORE
Gewichtung	15	15	15	25	25	5	
Monorepo (NX)	3	3	5	5	3	5	390
Multi/Poly repo	5	5	1	1	3	1	270

Das Monorepo zeigte sehr viel Potential bei der Lösung der Probleme. Die verbesserte Developer Experience und das einfache Teilen von Code waren die größten Vorteile verglichen mit einem Multirepo.
Separate Deployments, der oft erwähnte Vorteil von multiplen Repositories, waren auch unter NX kein Problem. Nun bleibt die Frage nach dem Haken. Einzig der Nachteil bleibt, dass die Teams sich auf eine Version von Abhängigkeiten einigen müssen. Am Ende tauscht man bei einem Monorepo etwas Freiheit ein, um Versionskonflikte und erhöhte Bundle-Größen zu vermeiden[54].

Auch wenn es im Netz unzählige Diskussionen und Vergleiche gibt, welche dem Monorepo sehr kritisch gegenüberstehen, muss beachtet werden, dass sich Monorepos stetig weiterentwickeln. Diesen stetigen Fortschritt kann man am besten erkennen, wenn man das erste Monorepo Lerna als Vergleich nimmt, welches keine Funktionen wie “tree diffing” oder Caching bietet.

[54] https://www.angulararchitects.io/en/aktuelles/using-module-federation-with-monorepos-and-angular/

In Hinblick auf die Performance verglichen mit anderen Monorepo Anbietern hat auch hier NX klar die Nase vorne. Durch sogenanntes “tree diffing” und einer ausgeklügelten Caching Strategie, erreicht NX erheblich schnellere Build-Zeiten. Ein Performance Vergleich der 3 bekanntesten Monorepo Anbieter (NX, Lerna & Turborepo) kann auf “https://github.com/vsavkin/large-monorepo” eingesehen werden.

6.3 Weighted Scoring Modell - Vergleich von 5 MF-Frameworks

Ein Vergleich von MF-Frameworks gestaltete sich durchaus schwieriger, da die Auswahl und die Anzahl zu berücksichtigender Faktoren viel größer ist. Da es zahlreiche Möglichkeiten gibt, MF zu bauen, musste sich diese Arbeit auf eine Handvoll Frameworks beschränken, da dies ansonsten den Umfang dieser Arbeit sprengen würde. Dass sich manche Frameworks wegen mangelndem Support oder schlechter oder gar fehlender Dokumentation gar nicht vergleichen ließen, erschwerten dieses Vorhaben zusätzlich.

Nach zahlreichen Tests der unterschiedlichsten MF-Frameworks wurde die Wahl am Ende auf 5 MF-Frameworks begrenzt. Bei der Auswahl wurde darauf geachtet, dass es sich um produktions-erprobte Frameworks handelt, welche sich in der Funktion möglichst stark unterscheiden.

Als Methode kommt das Weighted Scoring Modell zum Einsatz, was es ermöglicht, die Frameworks: module-federation, single-spa, piral, podium und mushroom-server anhand von 10 unterschiedlichen Kriterien zu vergleichen. Das Ergebnis wird in der nun folgenden Tabelle gezeigt.

Tabelle 6.3 MF-Framework Vergleich (Module-Federation, Single-spa, Piral, Podium & Mashroom-Server) via Weighted Scoring Methode

Kategorie	Komplexität			Performance			Funktionen				SCORE
Kriterien	Overhead, Boilerplat	Docs	Lernkurve	Auto Code-	Dependency Sharing	Nested Routes	SPA ähnlich	Rendering	Ausfall-sicherheit	Adaptierbarkeit,	
Varianten	viel, mittel, wenig	schlecht, gut, sehr gut	schlecht, gut, SPA Level	ja, nein	keine, import-maps/externals, build-	ja, nein	ja, nein	ssr, csr, beides	gering, mittel, hoch	schlecht, mutli-framework, lightweight	
Möglichkeiten	3	3	3	2	2	2	2	3	2	3	
Gewichtung	10	10	10	10	10	10	20	10	5	5	
module-federation	3	3	3	3	3	3	3	3	3	3	300
single-spa	2	3	2	1	2	1	1	2	1	2	165
piral	1	3	1	1	2	1	1	2	1	1	140
podium	1	2	1	3	3	1	1	1	2	1	155
mashroom-server	1	1	1	1	1	1	1	3	1	2	130

Dieser Vergleich zeigt das sich die Behauptung von Zack Jackson:

“Webpack 5 Module Federation: A game-changer in JavaScript architecture” - Zack Jackson (Entwickler von Module Federation) [55]

bewahrheitet hat. Auch eigene Tests konnten keine nennenswerten Schwächen erkennen. Einzig der Zwang Webpack 5 als Module Bundler zu benutzen, stellt einen Nachteil gegenüber Single-spa und Piral dar. Da es mir bei podium jedoch nicht gelungen ist herauszufinden, ob man dort freie Bundler Wahl hat, wurde dieses Kriterium weggelassen. Übrig blieben 10 Kriterien, welche sich nahezu gleichmäßig auf die Kriterien Performance, Komplexität und Funktionsumfang aufteilen ließen.

Die ausgezeichneten Code Sharing Möglichkeiten, die Leichtgewichtigkeit, das Laden von Modulen zur Laufzeit, das Single Page Applikation ähnliche Verhalten und die “Out of the box Features wie Dependency Management und Code splitting” machen Module Federation zum klaren Gewinner dieses Vergleichs.

6.3.1 Erkenntnisse im Umgang mit den anderen 4 MF-Frameworks:

Bevor die Entscheidung auf das Testprojekt fiel, wurden diese 5 Frameworks genauer beleuchtet. Mit der MF-Plattform mashroom-server ergaben sich große Startschwierigkeiten da der erforderliche Boilerplate sehr hoch und schlecht dokumentiert war. Mashroom’s Funktion hinter *sharedDependencies* war ebenfalls unzureichend und zeigte kein vernünftiges Beispiel wie man *npm-packages* entlang MF teilen konnte. Lediglich das Teilen von ganzen Bundles wurde in der Dokumentation beschrieben.

Die MF-Plattform Piral hatte keine Probleme mit schlechter Dokumentation oder fehlendem Dependency-Management. Da Piral jedoch Webpack externals für das Teilen von Abhängigkeiten verwendet, hat hier module-federation durch ihre kluge Bundle Strategie klar die Nase vorn. Der benötigte Boilerplate ist bei Piral größer, da für jedes MF sogenannte Pilets erstellt werden, welche in einer Piral-Instance miteinander kommunizieren. Im Großen und Ganzen machte Piral einen sehr guten Eindruck, jedoch hätte ein ausführlicher Test den Rahmen dieser Bachelorarbeit gesprengt. Beim SSR-MF-Framework Podium verhält es sich ähnlich wie bei Piral, mit der Ausnahme, dass Podium die jeweiligen MF auf der Serverseite orchestriert.

Auch Single-spa*, welches* sich lange Zeit als Alleinstellungsmerkmal innerhalb der MF-Community, wurde einem kurzen Test unterzogen. Im Grunde eignet sich Single-spa sehr gut für einen Multi-Framework Einsatz, jedoch wirkte hier die Art mit anderen MF zu interagieren etwas veraltet, da dies auf HTML Ebene mittels “script tags” und “import maps” erfolgte. Bei einem Multi-Framework Setup bietet Single-spa durchaus Vorteile und kann sogar mit Module Federation kombiniert werden. Bei einer Softwarelandschaft, welche jedoch ausschließlich aus React-SPA besteht, bietet Module Federation eine bessere Integration auf Modul-Ebene, was für ein SPA identisches Verhalten sorgt. Auch die

[55] https://indepth.dev/posts/1173/webpack-5-module-federation-a-game-changer-in-javascript-architecture

Fähigkeit, Abhängigkeiten zu teilen, ist bei Single-spa durch die Verwendung von import-maps keine ideale Lösung.

6.3.2 Module Federation in Kombination mit anderen Frontend-Frameworks

Module Federation setzt kein bestimmtes Frontend Framework voraus, somit ist die Wahl auf React, eigenen Präferenzen zuzuordnen. Gegenüber Angular bietet Vue und React durch Funktionen wie Suspense und Nested-Routes einige Performance Vorteile. Ähnliche Effekte sind bei Angular nur über Umwege zu erreichen, jedoch muss erwähnt werden, dass das Team hinter *angulararchitects.io* gute Pionierarbeit leistet, was die Kombination NX, Module-Federation und Angular betrifft.

Die Idee zur Verwendung unterschiedlicher Frameworks ist aus meiner Sicht zwar hilfreich für Migrationsprojekte, um die Bundle Größe und die Komplexität einer Micro Frontend Landschaft nicht unnötig zu erhöhen, würde ich stark von dieser Vorgehensweise abraten und bin ein starker Befürworter der “1 JS-Framework” Philosophie.

Da Zack Jackson der Entwickler von Module Federation zu Beginn dieser Arbeit React als Framework empfahl,verzichtete diese Arbeit auf einen detaillierteren Vergleich.

6.3.3 Zusammenfassung Module Federation Vorteile

Zusammengefasst bietet Module Federation folgende Vorteile gegenüber single-spa, piral, podium und mashroom-server:

- MF lassen sich wie herkömmliche Module importieren, was dazu führt, dass sich die ganze Anwendung wie eine herkömmliche SPA anfühlt.
- Orchestrierungsebene, die Module dynamisch zur Laufzeit mit Fallbacks teilt
- Einfaches und effektives Dependency Management
- First Class Webpack Support (Da es sich um ein Plugin aus dem Hause Webpack handelt)
- Lightweight und flexibel (piral und mashroom-server sind als komplette MF Plattformen sehr aufzwingend und unflexibel).
- Modularität. Module Federation kann bei Bedarf mühelos mit anderen MF-Frameworks wie single-spa kombiniert werden.
- Gute Dokumentation. Die kurz gehaltene Dokumentation und deren "One-Click-live-Preview" erlauben einen sehr schnellen Einstieg.
- Niedrigste Lernkurve vor allem im Vergleich mit piral und mashroom-server. Besonders mashroom-server erschlägt neue User regelrecht mit deren Funktionen und unübersichtlicher “One Page Documentation” [56]

[56] https://www.mashroom-server.com/documentation/docs/html/

7. Schlussfolgerungen

Dieses Kapitel widmet sich der Beantwortung der Forschungsfrage, gefolgt von einer Hypothesenbewertung. Dabei wird auch über die Limitationen dieser Bachelorarbeit eingegangen und zum Schluss werden noch Vorschläge für weiterführende Forschungen abgegeben.

7.1 Beantwortung der Forschungsfrage und Hypothesen Bewertung

Um die Übersicht zu erleichtern, werden die in Kapitel 1 erwähnten Forschungsfragen und Hypothesen hier noch einmal angeführt.

Die Bewertung der Ergebnisse und das Anwenden der Methoden aus Kapitel 6. zeigt, dass die Hypothese bestätigt werden kann.

Forschungsfragen:

Welcher Code-Repository-Type (Monorepo oder Multi-Repo) ist für Micro Frontends besser geeignet?

Welches Micro-Frontend-Framework (Module-Federation, Single-Spa, Piral, Podium oder Mashroom-Server) eignet sich besser für Micro Frontends?

Hypothesen:

Ein Monrepo eignet sich besser für Micro Frontends als ein Multi-Repo.

Module-Federation ist als Micro-Frontend-Framework besser geeignet als Single-Spa, Piral, Podium oder Mashroom-Server.

Hypothesen Bewertung;

Die Bewertung der Ergebnisse aus Kapitel 6 hat gezeigt, dass beide Hypothesen verifiziert werden können.

7.2 Limitationen

Trotz des großen Erfolgs der ausgewählten Technologien in Bezug auf die Problemstellungen, konnten nicht alle Technologien getestet werden. Zusätzlich muss darauf hingewiesen werden, dass der Weighted Scoring Vergleich und die Definition der einzelnen Kriterien nur vom Verfasser dieser Bachelorarbeit durchgeführt wurden.

Hinzu kommt, dass einige Unternehmen eigene Frameworks (nicht immer Open-Source) entwickeln, was zwar die aufwändigere Methode darstellt, aber oftmals viele Vorteile bietet, da man so auf individuelle Bedürfnisse des Unternehmens eingehen kann.

Micro Frontends sind keine Antwort auf alle Probleme im Frontend. Im Grunde lösen Mikro-Frontends organisatorische Probleme, indem sie technische Komplexität einführen. Deshalb ist eine individuelle Lösung, die auf die eigenen Unternehmensbedürfnisse zugeschnitten ist, meist die beste Lösung.

Da es sich bei dem Prototyp nur um ein Testprojekt handelt, konnten Probleme, die eventuell in einem Enterprise Umfeld “at scale” entstehen, nicht untersucht werden. Bei der Auswahl der Technologien wurde jedoch stets darauf geachtet, ausschließlich produktionstaugliche Tools einzusetzen. Ein Prototyp stellt somit immer eine künstliche Situation her, welche von einem realen Softwareprojekt manchmal zu weit entfernt ist. Um diesem Effekt entgegenzuwirken, wurden auch “Real World Enterprise Probleme” aus eigener Erfahrung mit einbezogen, jedoch können sie sich von Unternehmen zu Unternehmen sehr unterscheiden.

In der IT ist man ständig mit Trade Offs konfrontiert, was ein stetes Abwägen von Vor- und Nachteilen einer Technologie voraussetzt. Was für ein Unternehmen unwichtig erscheint, kann für ein anderes wiederum höchste Priorität haben.

MF sind keine Antwort auf alle Probleme im Frontend. Im Grunde lösen MF organisatorische Probleme, indem sie technische Komplexität einführen. Wer keine organisatorischen Probleme hat, benötigt aus meiner Sicht auch keine MF.

7.3 Empfehlung für weiterführende Forschungen

Die vorliegende Bachelorarbeit hat erste Erkenntnisse in Bezug auf das Lösen gängiger Probleme im Umgang mit MF geliefert. Diese Forschung kann durch Experteninterviews fortgeführt werden, um weitere Ergebnisse zu erreichen. Besonders Personen aus dem Enterprise Umfeld können wertvolle Einblicke zu weiteren Problemstellungen oder Best Practices liefern.

8. Fazit

Durch den sehr lösungsorientierten Ansatz dieser Bachelorarbeit konnten mit Hilfe des Prototyps 6 von 11 häufig unter MF auftretende Probleme gelöst werden. Allein dieses Ergebnis zeigt nicht nur das hohe Potential der verwendeten Technologien, sondern zeigt dies auch, wie viel unerkundetes Potential in der MFA steckt.
Der unter der Weighted Scoring Methode durchgeführte Framework Vergleich zeigte, wie sehr sich Webpacks Module-Federation von anderen Alternativen abhebt.
Zusätzlich hat diese Arbeit gezeigt, wie die Verwendung eines Monorepos die Nachteile von MF mildert, ohne die Vorteile zu beeinträchtigen.

9. Quellen

9.1 Literaturverzeichnis

Brooks, F. P. (1975). The Mythical Man-Month: Essays on Software Engineering. Addison Wesley Longman Publishing Co.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.

Foote, K. D. (2021, 2. April). *A Brief History of Microservices*. Dataversity. Abgerufen am 12. April 2022, von https://www.dataversity.net/a-brief-history-of-microservices/#

Gall, J. (1975). *General Systemantics: An Essay on how Systems Work, and Especially how They Fail.* General Systemantics Press.

Geers, M. (2020). *Micro Frontends in Action.* Manning Publications.

Gilbert, J. (2021). *Software Architecture Patterns for Serverless Systems: Architecting for innovation with events, autonomous services, and micro frontends.* Packt Publishing.

Github. (2022). *GitHub - module-federation/module-federation-examples: Implementation examples of module federation , by the creators of module federation.* Abgerufen am 20. Juni 2022, von https://github.com/module-federation/module-federation-examples

Haigh, T. (2017, Oktober). *The History of Unix in the History of Software.* University of Wisconsin.

Havro IT Solutions. (2022, 11. Februar). *7 Successful companies using micro frontends* Thehavro. Abgerufen am 12. April 2022, von https://thehavro.com/2022/02/11/companies-using-micro-frontends/

Khononov, V. (2018, 26. Januar). *Revisiting the Basics of Domain-Driven Design.* Vladikk. https://vladikk.com/2018/01/26/revisiting-the-basics-of-ddd/

Khononov, V. (2021). *Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy.* O'Reilly UK Ltd.

Lewis, J. (2012, 19. März). *Conference for Java Masters - Micro services - Java, the Unix Way.* 33rd Degree. http://2012.33degree.org/talk/show/67

Lewis, J. & Fowler, M. (2015, 1. Juli). *Microservices.* martinfowler.com. Abgerufen am 16. Mai 2022, von https://martinfowler.com/articles/microservices.html

Mezzalira, L. (2019a, Mai 21). *Identifying micro-frontends in our applications - DAZN Engineering.* Medium. Abgerufen am 16. Mai 2022, von https://medium.com/dazn-tech/identifying-micro-frontends-in-our-applications-4b4995f39257

Mezzalira, L. (2019b, Mai 27). *I don't understand micro-frontends.* Medium. Abgerufen am 16. Mai 2022, von https://medium.com/@lucamezzalira/i-dont-understand-micro-frontends-88f7304799a9

Mezzalira, L. (2019c, Dezember 22). *Micro-frontends decisions framework.* Medium. Abgerufen am 16. Mai 2022, von https://medium.com/@lucamezzalira/micro-frontends-decisions-framework-ebcd22256513

Mezzalira, L. (2021). *Building Micro-Frontends: Scaling Teams and Projects Empowering Developers.* O'Reilly UK Ltd.

Microservices. (2022, 14. Juni). In *Wikipedia*. Abgerufen am 14.Juni 2022, von https://en.wikipedia.org/wiki/Microservices

Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems (2nd edition).* O'Reilly UK Ltd.

Noel, A. (2019, 13. Februar). Guide to Monorepos for Front-end Code. Toptal Engineering Blog. Abgerufen am 2. Juni 2022, von https://www.toptal.com/front-end/guide-to-monorepos

Perera, P. (2022, 4. Januar). *Visual Explanation and Comparison of CSR, SSR, SSG and ISR.* DEV Community. Abgerufen am 2. Juni 2022, von https://dev.to/pahanperera/visual-explanation-and-comparison-of-csr-ssr-ssg-and-isr-34ea

Potvin, R. (2016). Why Google Stores Billions of Lines of Code in a Single Repository. Google Research Publications. Abgerufen am 2. Juni 2022, von https://research.google/pubs/pub45424/

Rappl, F. (2021). *The Art of Micro Frontends: Build websites using compositional UIs that grow naturally as your application scales.* Packt Publishing.

Scott, E. A., Jr. (2015). *SPA Design and Architecture: Understanding Single Page Web Applications.* Manning Publications.

Stack Overflow. (2021, Mai). *Stack Overflow Developer Survey 2021.* Abgerufen am 2. Juni 2022, von https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-webframe-love-dread

Statista. (2022, 23. Februar). *Most popular web frameworks among developers worldwide 2021.* Abgerufen am 14. April 2022, von https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/

Statistics and Data. (2021, 21. Juni). *The Most Popular JavaScript Frameworks - 2011/2021.* Abgerufen am 12. April 2022, von https://statisticsanddata.org/data/the-most-popular-javascript-frameworks-2011-2021/

The Software House. (2022, 25. Mai). *The State of Frontend 2022.* Abgerufen am 28. Mai 2022, von https://tsh.io/state-of-frontend/

Thoughtworks. (2016). *Micro frontends | Technology Radar.* Abgerufen am 12. Jänner 2022, von https://www.thoughtworks.com/de-de/radar/techniques/micro-frontends

Zandt, F. (2021, 17. September). *How Big Tech Contributes to Open Source.* Statista Infographics. Abgerufen am 22. Jänner 2022, von https://www.statista.com/chart/25795/active-github-contributors-by-employer/

9.2 Abbildungsverzeichnis

9.3 Tabellenverzeichnis

9.4 Anhang