Vom Monolith zu Microfrontends: Übertragbare Patterns aus Sicht eines Backend-Entwicklers

Bachelorarbeit zur Erlangung des Akademischen Grades
Bachelor of Science
im Studiengang Code & Context
der Technischen Hochschule Köln

von Enrico Retz

Kurzfassung

Durch Microservices hat die Modularisierung von Software in den letzten Jahren erhebliche Fortschritte gemacht. Die dadurch gewonnene Team-Autonomie wird weitgehend noch durch ein monolithisches Frontend eingeschränkt. In den letzten Jahren hat sich dafür eine Lösung ergeben: Microfrontends. Unternehmen stehen nun nach der Migration ihrer Backends erneut vor der Aufgabe, einen Monolithen zu zerteilen.

In dieser Arbeit wird die Frage beantwortet, in welchem Umfang sich aus der Migration von Backends bewährte Patterns auf Microfrontends übertragen lassen. Es werden die drei Patterns „Strangler-Fig“, „Branch-By-Abstraction“ und „Anti-Corruption-Layer“ anhand ihrer Resilienz, Dauer eines Releasezyklus und Förderung der Team-Autonomie bewertet, um Unternehmen eine fundierte Entscheidungsgrundlage zu bieten, um eine Migration des Frontends anzugehen.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das Gliedern von Softwarearchitekturen in kleine Module hat durch die Entwicklungen im Bereich der Microservices in den letzten Jahren erhebliche Fortschritte gemacht. Besonders die steigende Beliebtheit von Cloud-Services hat Microservices zu größer werdender Popularität verholfen, da Cloud-Native-Microservices besonders von der „Skalierbarkeit, Resilienz und Flexibilität“ der Cloud-Dienste profitieren [1]. Auch die Adaption von CI/CD-Pipelines welche an sich schon für mehr Stabilität und Flexibilität sorgen und die Releasezyklen deutlich verkürzen [2].

Betrachtet man die Google-Suchen zu dem Suchbegriff „Microservices“, haben diese in dem letzten Jahrzehnt ständig zugenommen und sind seit 2020 ungefähr auf dem gleichen Level geblieben. Daraus lässt sich schließen, dass das Interesse an Microservices gestiegen ist. Während das Interesse an Microservices nach wie vor hoch ist, ist das an Microfrontends noch nicht wirklich gestiegen, obwohl ThoughtWorks es bereits seit 2019 als „Adopt“-Technologie einstuft [3]. Sam Newman argumentiert in seinem Standardwerk zu Microservices [4; Seite 31], dass die organisatorische Autonomie von Entwicklungsteams ein Hauptvorteil von Microservices ist. Diese Autonomie kann durch den Einsatz von Microfrontends auch in das UI übertragen werden und die Teams vollständig autonom in ihrer Domäne arbeiten lassen.

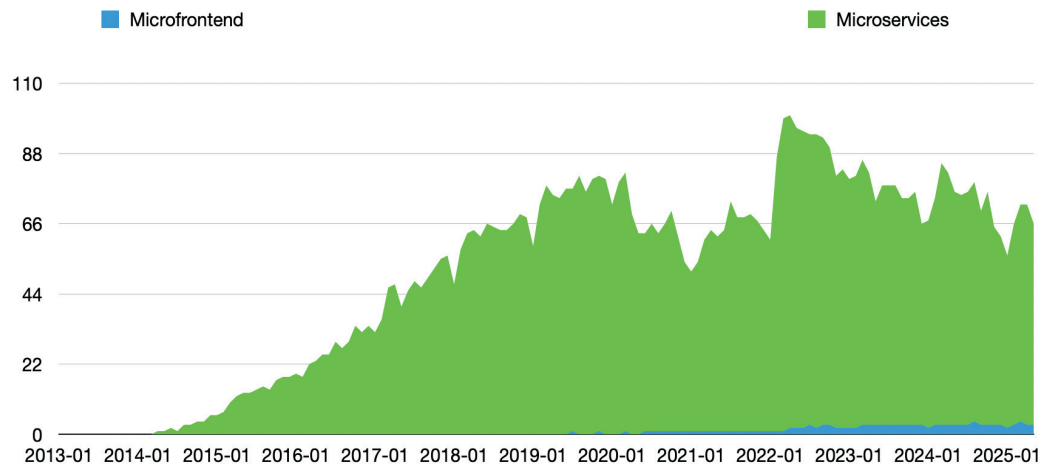


Abb. 1: Google Trends zu „Microservices“ und „Microfrontend“

Um diese Lücke zu schließen und dabei zu unterstützen, aus vorhandenen Microservices eine komplette, isolierte Funktionseinheit zu machen, geht diese Arbeit der Frage nach, inwieweit bewährte Migrationsmuster in der Frontend-Entwicklung angewendet werden können, wenn ein monolithisches Frontend in Microfrontends migriert werden soll. Um den Rahmen einer Bachelorarbeit nicht zu überschreiten, werden drei etablierte Migrationsmuster untersucht: das Strangler-Fig-Pattern [5] und Branch-By-Abstraction-Pattern [6] nach Martin Fowler und das Anti-Corruption-Layer-Pattern nach Eric Evans [7; Seite 235]. Das Strangler-Fig-Pattern wurde untersucht, weil es von Newman auch in seinem Werk zur Migration von Backends erwähnt [8; Seite 81] wird. Das Branch-By-Abstraction-Pattern wurde untersucht, weil es als ein Pattern beschrieben wird, welches „große Änderungen an Systemen so ermöglicht, dass während des

Prozesses ständig veröffentlicht werden kann“ [6]. Zu guter Letzt wurde sich auch für das Anti-Corruption-Layer-Pattern entschieden, es wird in Eric Evans‘ Standardwerk zu Domain Driven Design erwähnt [7; Seite 235] und dort als effektives Werkzeug beschrieben, um ein System umzubauen, ohne direkt alles Alte als unbrauchbar abzutun. Da so eine Migration in großen Projekten sehr lange dauern kann, macht es Sinn, alles was funktioniert, so lange es geht, zu behalten.

Um dem Rahmen einer Bachelorarbeit nicht zu überschreiten, wurde die Migration des Projekts mithilfe dieser drei Patterns anhand von drei Kriterien mit einem Ampelsystem bewertet. Das Ampelsystem bietet die Möglichkeit, die Entwurfsmuster möglichst minimal mit gut/grün, mittel/gelb und schlecht/rot zu bewerten und so übersichtlich miteinander zu vergleichen. Die Kriterien, anhand die Muster bewertet werden, sind die Resilienz, Releasezyklus und Team-Autonomie und bieten Teams, die vor der Aufgabe einer Migration stehen, eine fundierte Entscheidungsgrundlage. Besonders die Team-Autonomie ist dabei ein wichtiges Kriterium, da genau diese eine besondere Stärke der Microservice-Architektur ist und zu „einer schnelleren Umsetzung von Code in der Produktion führt“ [9; Seite 14].

Als experimentelle Grundlage dient die vom Autor entwickelte Anwendung „Jukefy“, welche vereinfacht die Situation eines Microservice-Backends und monolthischen Frontends darstellt. In dieser Umgebung lassen sich die Muster vergleichbar testen, ohne viele Ressourcen auf komplizierte Algorithmen oder Infrastruktur zu verwenden [10]. Es ist ein sehr überschaubares Projekt mit lediglich fünf Services und damit gut für den Rahmen dieser Bachelorarbeit geeignet.

1.1 Motivation für die Migration

Man stelle sich ein Unternehmen vor, welches vier Entwicklungsteams beschäftigt. Jedes arbeitet bereits an einem bestimmten Microservice. Es gibt ein dediziertes Frontend-Team welches die Webanwendung baut und mit allen Backends gekoppelt ist. In dem Fall, dass es nun Änderungen an den Backend-APIs gibt, so muss das Frontend-Team auch aktiv werden und sich mit dem entsprechenden Backend-Team absprechen. Das steht einem unkompliziertem Release im Weg und verzögert die Einführung einer neuen Funktion.

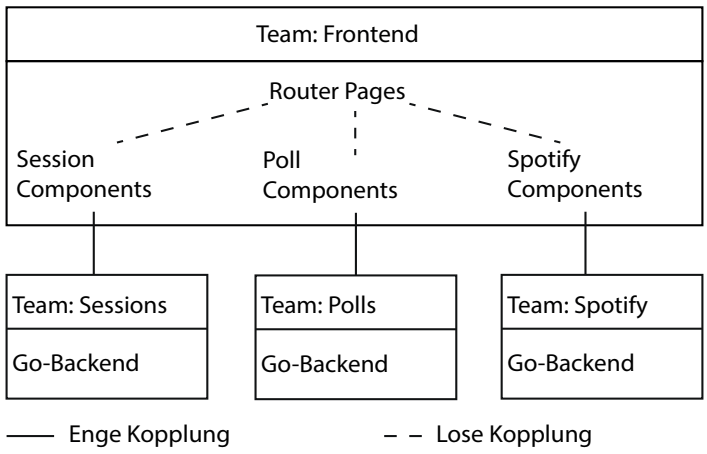


Abb. 2: Organisationsdiagram des fiktiven Unternehmens vor der Migration

Ziel der Migration ist es, die Kopplung des Frontend-Teams zu den Microservice-Teams zu lösen und die spezifischen Teile der Domänen dem spezifischen Team zu geben. Das Unternehmen könnte die Anwendung nachher in diese Struktur teilen:

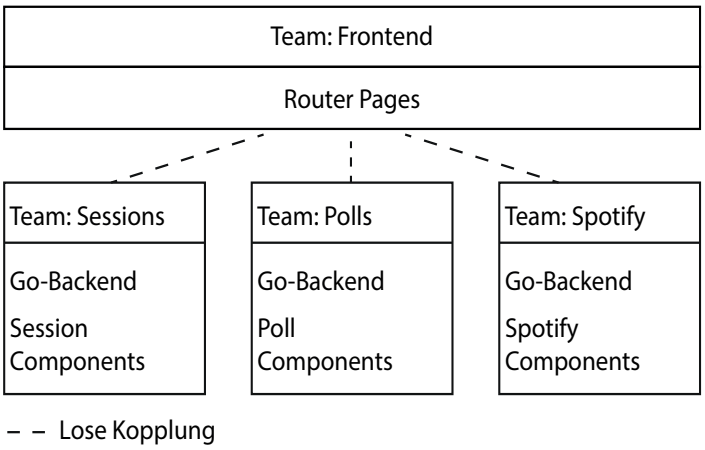


Abb. 3: Organisationsdiagram des fiktiven Unternehmens nach der Migration

Martin Fowler beschreibt in seinem Blog viele Vorteile von Microfrontends [11]. Einige Unternehmen stecken bei der Entwicklung neuer Funktionen fest. Sie haben einen Monolithen, der auf alten Technologien basiert, und neue Funktionen lassen sich dort nur schwergängig implementieren. Eine „vollständige Neuentwicklung ist sehr verführerisch“ [11] aber mit einem hohen Aufwand verbunden. Das Unterteilen des Frontends in kleine, simple, gut wartbare, schnell und unabhängig veröffentlichbare Microfrontends ist ein guter Weg, ein komplettes Neuschreiben zu umgehen.

In dieser Arbeit soll ein effektiver und schneller Weg gefunden werden, um das Frontend in Microfrontends aufzuteilen. Es ist sinnvoll, sich dabei an etablierte Patterns zu halten, da diese als aus Expertenwissen abgeleitete Baupläne für bekannte Probleme anzusehen sind [12]. Bei der Auswahl des richtigen Pattern für diese Aufgabe zu unterstützen ist das Ziel dieser Arbeit.

1.2 Definition des „Pattern“ Begriffs

Pattern können dabei helfen, die Produktivität beim Programmieren und die Qualität des Programms zu steigern. Sie helfen Neulingen dabei, ihre Fähigkeiten zu verbessern, und Erfahrenen dabei, sich an gute Praktiken zu halten. Außerdem erleichtern sie die Kommunikation zwischen Entwickler:innen und denjenigen, die das Programm warten [13]. In seinem Standardwerk zur Migration von einem Monolithen zu Microservices beschreibt Sam Newman etliche Pattern, die bei der Migration unterstützen [8].

Ein „Pattern“ beschreibt nach Christopher Alexander „ein Problem, das in unserer Umgebung immer wieder auftritt, und dann den Kern der Lösung für dieses Problem, und zwar so, dass man diese Lösung millionenfach anwenden kann, ohne sie jemals zweimal auf die gleiche Weise zu machen.“[14; Seite X]

Diese Definition beschreibt Patterns als universell einsetzbare und wiederverwendbare Lösungen für häufig auftretende Probleme. Wobei diese jedoch so flexibel sind,

dass sie sich an verschiedene Kontexte anpassen lassen. Dass Christopher Alexander nichts mit Softwareentwicklung zu tun hat, sich seine Definition aber trotzdem so gut auf diese übertragen lässt, verdeutlicht auch noch mal stark die Allgemeingültigkeit des Patternbegriffs.

Nach Erich Gamma [15; Seite 2ff] besteht ein Pattern aus vier grundlegenden Elementen:

- **Name**
  Kurze Beschreibung des Problems, der Lösung oder der Konsequenzen
- **Problem**
  Wann wird das Pattern angewendet?
- **Lösung**
  Der Bauplan, um das Problem zu lösen
- **Konsequenzen**
  Wenn du das Pattern andere Wege verbaut werden oder andere Konsequenzen nach sich zieht

Patterns sind nicht nur theoretische Konzepte. Sie helfen in der Softwareentwicklung dabei, bewährte Lösungen zu nutzen und fördern dabei Konsistenz und Verständlichkeit [13].

2 Herangehensweise

Um die Pattern zu testen, wird ein monolithisches, mit dem React-Framework entwickeltes Frontend als Basis herangezogen. Es wurde ein Git-Repository erstellt, welches eine Docker-Compose-Datei und den Monolith in einem eigenen Verzeichnis enthält. Auf diese Weise können die verschiedenen Pattern in separaten Branches getestet und entsprechende Microfrontends implementiert werden. Um die Pattern zu testen, wird zu einem Service ein entsprechendes Microfrontend entwickelt und die starke Kopplung zwischen dem monolithischen Frontend und den Microservices zu lösen.

2.1 Das Programm „Jukefy“

Die Basis für die Experimente ist das Programm „Jukefy“. Es ist eine Anwendung für die demokratische Erstellung von Spotify-Warteschlangen. Es basiert auf einer minimalen Microservice-Architektur und zeichnet sich durch die Einfachheit der einzelnen Komponenten aus, wodurch es sich hervorragend als Anschauungs-Anwendung für Microfrontends eignet. Es besteht aus einem monolithischen Frontend mit mehreren Komponenten. Die Patterns werden auf den „PollService“ angewendet und es wird versucht, ein allein stehendes Programm für Umfragen auszugliedern, welches sowohl mit einem Backend als auch mit einem Frontend daher kommt.

Die gemeinsame Erstellung von Spotify-Warteschlangen geschieht über Umfragen. Jede:r auf der Seite kann durch die Suchleiste ein Lied suchen und durch einen Klick eine Umfrage starten. Alle anderen können über das Lied abstimmen. Bei Erfolg kommt es in die Warteschlange.

Die Anwendung besteht aus Diensten für die Geschäftslogik, den App Services, welche alle über NATS Messages kommunizieren. Jeder App Service schreibt in eine eigene Datenbank der gleichen Redis-Instanz. Theoretisch könnte jeder aber auch seine eigene haben. Frontend und Backend sind über Traefik miteinander verbunden. Der Reverse Proxy leitet sowohl die HTTP- als auch Websocket-Anfragen an die Backend-Services weiter.

Die drei App Services sind:

- „SessionService“
  Erstellt und verwaltet das Session-Objekt. Durch dessen ID werden alle anderen Objekte miteinander verbunden und können zugeordnet werden.
- „SpotifyService“
  Enthält alles, was mit Spotify zu tun hat. Also die Suche und das Einreihen der Songs in die Warteschlange.
- „PollService“
  Ist für die Umfragen, das Abstimmen und Zeit stoppen zuständig

In den nachfolgenden Abbildungen ist die aktuelle Architektur abgebildet.

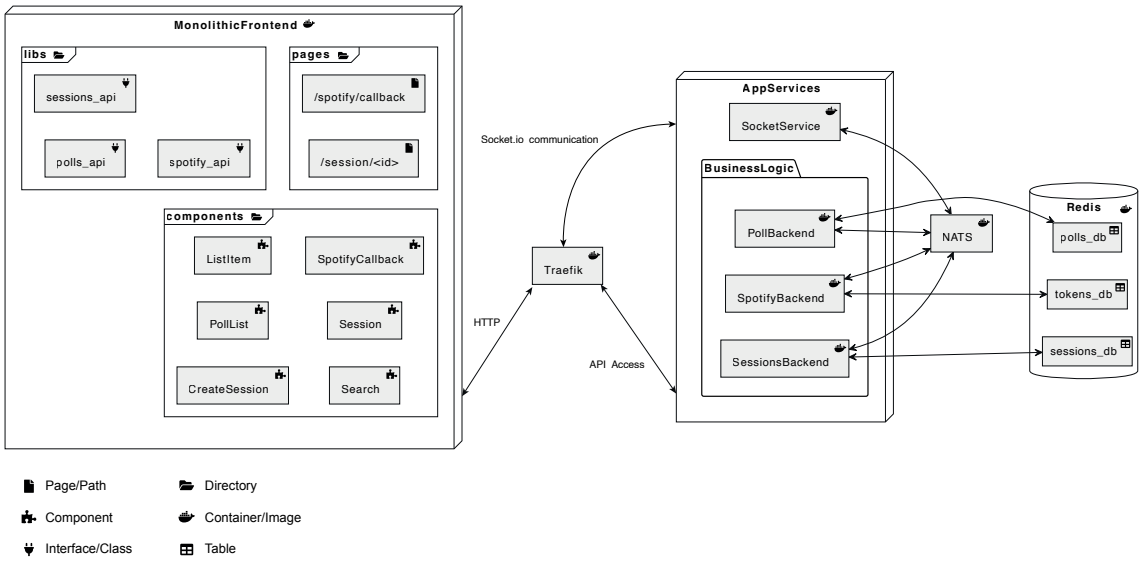


Abb. 4: „Jukefy“ Ausgangsarchitektur Diagramm

Der Fokus liegt dabei auch auf der internen Architektur des monolithischen Frontends, da es in dieser Arbeit hauptsächlich um dieses geht.
Dieses besteht aus drei Einheiten:

- „libs“
  Hier sind alle API-Interfaces und TypeScript-Dateien für die Interaktion mit dem Backend oder extra Frontend-Skripte (etwa die Verwaltung des „LocalStorage“).
- „pages“
  Sind die Seiten, die der React Router aufruft. In der Abbildung sind nur die, die spezifisch für eine Session sind. Es gibt auch noch eine Startseite, wo eine neue Session erstellt werden kann und das Impressum zu lesen ist.

- „components“
  Hier sind alle Frontend-React-Components die von den Seiten benutzt werden. Es sind die kleinsten Funktionseinheiten, aus denen sich das Frontend zusammen setzt.

Das Backend wurde hauptsächlich in Go programmiert. In dieser Arbeit werden die Komponenten, die mit dem „PollBackend“ interagieren, in ein eigenes Frontend-Modul ausgelagert. Damit wird die komplette Domäne „Umfrage“ im vollständigen Technologiestack von einem Team bedient, also Frontend, Backend und Datenspeicherung.

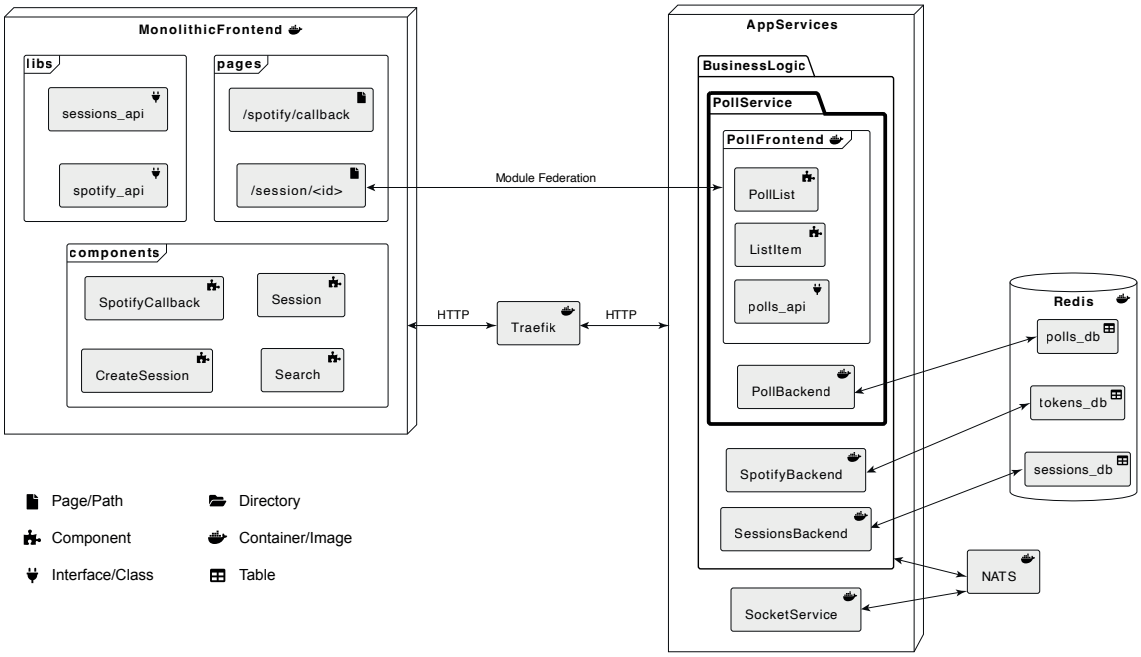


Abb. 5: „Jukefy“ Ziel Architektur

2.2 Module Federation

Martin Fowler hat in seinem Blog auch schon einige Implementierungsansätze dokumentiert. [11] Er beschreibt drei grundlegende Herangehensweisen an die Thematik:

- „Server-side template composition“
  Bei dieser Herangehensweise werden Aufrufe zu Seiten durch bspw. Nginx an verschiede Server umgeleitet. So kann jedes Team eine eigene Seite entwickeln.
- „Build-time integration“
  Jedes Team erstellt ein eigenes „package“, welches von der Hauptanwendung als Dependency geladen und eingebunden wird.
- „Run-time integration“
  Diese Herangehensweise basiert auf der Prämisse, dass die Microfrontends während der Laufzeit integriert werden. Martin Fowler beschreibt dort verschiedene Möglichkeiten, dies zu verwirklichen, etwa durch „iframes“, JavaScript-Einbindungen oder „WebComponents“.

In Martin Fowlers Ansatz wird von jedem Microfrontend eine ganze Seite ausgeliefert. Wenn nun aber selbst innerhalb einer Seite verschiedene Komponenten genutzt werden sollen, muss man kreativ werden.

Der Erfolg des Ansatzes der „Server-side template composition“ ist dann stark abhängig von der gewählten Technologie, bzw. zwingt alle Teams die gleiche, oder kompatible Template-Engine zu verwenden. Die meisten Template-Engines wie etwa Jinja2 erlauben das Einbinden von Modulen [16]. In dem Haupttemplate könnten dann etwa Module eingebunden werden, die von anderen Teams entwickelt wurden und beim Deployment auf den Eltern-Server hochgeladen werden, wo sie dann verfügbar sind. Bei diesem Ansatz sind die Teams sehr stark aneinander gekoppelt, da sie die gleiche Template-Engine verwenden müssen und es lässt sich nur schwer skalieren, da die Module für alle Instanzen verfügbar sein müssen.

Der Vorteil von „Build-time integration“ liegt darin, dass die Komponenten von den Teams im alten Frontend wie native Module genutzt werden können. Das Styling und Komponentenübergreifende Funktionen können zwischen diesen problemlos geteilt werden. Der große Nachteil liegt in der Abhängigkeit beim Deployment. Wenn ein Team eine Komponente aktualisiert, wird diese Änderung erst bei einem Update von dem Anwendungs-Frontend für die Benutzer:innen sichtbar. Dies verlangsamt den Entwicklungsprozess und bindet die Teams aneinander. Dies erkennt auch Martin Fowler und rät davon ab [11].

Ein Ansatz, der viele Vorteile einer Microservice-Architektur enthält, ist die „Run-time integration“. Hier kommt es allerdings darauf, an welchen Ansatz man wählt. „iframes“ sind zwar schnell und ermöglichen unabhängiges Arbeiten der Teams, sind jedoch zu isoliert. Sollen bspw. gemeinsames Styling verwendet werden, so muss jedes Team Zugriff auf die gleichen CSS-Dateien haben. Das gleiche Problem haben auch „Web-Components“, also selbst definierte HTML-Elemente die über eine JavaScript-Datei geladen werden.

Um die Microfrontends zu realisieren, wurde auf „Module Federation“ gesetzt. Dieses ist ein ursprünglich von Webpack entwickeltes Plugin, um JavaScript-Code dynamisch und während der Laufzeit von einer anderen Quelle zu laden. Während diese Arbeit geschrieben wurde, erschien bereits Version 2.0 von Module Federation, welches einige Punkte verbessert. Besonders die Unterstützung für „Remote Type Hints“ [17] ist hier hervorzuheben, weil die das manuelle Übertragen der Types in die Host-Anwendung überflüssig macht. Diese Funktion ist bisher leider nur mit dem Webpack-Plugin verfügbar, nicht aber mit dem hier verwendeten Plugin für „Vite“.

„Module Federation“ ist dabei ein Weg zwischen „Build-time integration“ und „Run-time integration“. Die Komponenten können wie native Elemente im JavaScript-Framework genutzt werden. Das Styling kann ganz einfach vom monolithischen Team überschrieben und angepasst werden. Dabei sind die Komponenten aber unabhängig veröffentlicht und werden während der Laufzeit in die Seite integriert. Module Federation bietet auch eine breite Technologie Offenheit. Alle Module, die mit Vite oder Webpack gebaut werden, sind kompatibel. Ein Team könnte ihre Komponenten in Svelte bauen und ein anderes in Vue.js. Mit Modern.js gibt es auch ein Framework das Server-Side-Rendering unterstützt.

Eine große Stärke von „Module Federation“ ist, dass geteilte Abhängigkeiten nicht mehrfach geladen werden. Benötigen mehrere Komponenten bspw. „React“, so wird es auch nur einmal geladen.

3 Patternauswahl

Um im Rahmen einer Bachelorarbeit zu bleiben, wurden drei Patterns ausgewählt, die bereits bewährte Strategien für eine schrittweise und risikoarme Migration von monolithischen Backends zu Microservices bieten. Sie ermöglichen es, einzelne Komponenten aus dem Monolith herauszulösen, unabhängig weiterzuentwickeln und ohne die restlichen Module zu sehr zu beeinflussen, zu deployen. Durch die parallele Koexistenz von alten und neuen Systemteilen lässt sich die Architektur umbauen, ohne den laufenden Betrieb zu unterbrechen oder lange Entwicklungszyklen zu benötigen.

In Sam Newmans Werk zur Migration von Backends [8] werden auch noch zahlreiche andere Patterns vorgestellt, welche sich für die Aufgabe eignen, einen Monolithen in kleinere Services zu teilen. Das „UI Composition Pattern“ [8; Seite 100] teilt das Backend vom Frontend ausgehend ein. Dort wird eine Funktion aus dem Frontend ausgegliedert und dazu ein Microservice erstellt. Dieses Pattern eignet sich hervorragend, wenn auf beiden Seiten der Anwendung noch eine monolithische Architektur eingesetzt wird, da dort auch direkt der Umstieg auf Microfrontends mit vollzogen wird. Da in diesem Fall aber schon ein Backend mit der Microservice-Architektur vorlag, wurde es nicht extra untersucht.

Das „Parallel Run Pattern“ [8; Seite 114] ist eine Möglichkeit, die Funktion der Migration zu validieren. Dabei werden beide Systeme parallel betrieben, aber anstelle immer nur eines aufzurufen, werden die Daten von beiden Systemen (altes und neues) geholt und miteinander verglichen. Dies funktioniert im Backend-Bereich wo die Antworten einfach miteinander verglichen werden können, erweist sich im Frontend-Bereich allerdings schwierig, wenn DOM-Elemente und Funktionen miteinander verglichen werden müssen. Um die Funktionalität des neuen Systems zu gewährleisten, sollte eher auf Techniken wie Frontend-Tests gesetzt werden.

3.1 Strangler-Fig-Pattern

Das Strangler-Fig-Pattern setzt auf einen inkrementellen Ansatz, um Software fundamental zu ändern [18]. Anstatt alles direkt neu zu schreiben und User somit lange auf neue Funktionen warten zu lassen, werden neue Funktionen mit der neuen Architektur/ Technologie implementiert und die alte Code-Basis langsam mit einer neuen ersetzt. Hierbei leben der alte und der neue Code lange nebeneinander weiter, jedoch wird der alte nicht mehr geändert. Neue Funktionen können bereits deployt werden, sind aber über das UI noch nicht aufrufbar. Das ermöglicht ein intensives Testing in der Produktiv-Umgebung, aber auch ein schnelles Umschalten zwischen den neuen und alten Komponenten.

3.2 Branch-by-Abstraction-Pattern

Martin Fowler beschreibt das Branch-By-Abstraction-Pattern als eine Technik, um umfangreiche Änderungen an einem Softwaresystem sukzessive durchzuführen [19]. Das zu ersetzende System wird in einer Abstraktionsschicht gekapselt und die Clients

nacheinander so umgebaut, dass sie die Abstraktionsschicht aufrufen. Anschließend können wir das System ersetzen und die Abstraktionsschicht anpassen, um das neue System zu benutzen. Der Trick hierbei ist, dass die Abstraktionsschicht kompatibel mit dem alten System ist, sodass ein unkompliziertes „Umschalten“ auf diese von den Clients gewährleistet ist. Innerhalb dieser Schicht kann dann die Kompatibilität zu dem neuen System sukzessiv implementiert werden.

3.3 Anti-Corruption-Layer-Pattern

Das Anti-Corruption-Layer-Pattern kommt aus der Welt des Domain-Driven-Designs und wird sowohl von Eric Evans [6] als auch von Vaughn Vernon [20] in ihren Standardwerken erwähnt. Dieses Pattern ist zwar mehr ein Integrations- als ein Migrationsmuster, hilft aber bei der Migration von Architekturen, durch eine Antikorruptionsschicht zwei Komponenten voneinander zu isolieren. Diese Schicht übersetzt dann entsprechend zwischen den beiden Komponenten, wo zwischen sie steht. Im Gegensatz zum Branch-By-Abstraction-Pattern, wo die Abstraktionsschicht nur als eine vorübergehende Lösung während der Migration gedacht ist, ist die Antikorruptionsschicht als permanente Lösung gedacht, die eine bi-direktionale Übersetzung zwischen den Modellen der Komponenten ermöglicht.

Im Kontext der Microfrontends bedeutet das, dass die Eigenschaften der Komponenten und die Abhängigkeiten vom Hostsystem in eine Anti-Korruptionsschicht verpackt werden.

4 Bewertungskriterien

Um die ausgewählten Patterns zu vergleichen und zu bewerten, wurden vergleichbare Kriterien herangezogen. Diese Kriterien basieren auf etablierten Bewertungsmethoden, die auch innerhalb des Unternehmens verwendet werden können, um eine Migration hin zu einer Microservice-Architektur zu analysieren und zu bewerten. Da sich viele der zugrunde liegenden Herausforderungen von Microservices im Backend auch auf das Frontend übertragen lassen, wurden ähnliche Kriterien herangezogen.

Bei der Auswahl der Kriterien wurde versucht, nicht nur technische Aspekte zu berücksichtigen, sondern auch die Zusammenarbeit im Team und die agile Entwicklung zu berücksichtigen. Teamautonomie ist ein entscheidender Faktor, der für die Microservice-Architektur spricht [21].

4.1 Resilienz

Die Resilienz beschreibt die Standfestigkeit und ist eine wichtige Eigenschaft von Systemen mit Microservice-Architektur [22; Seite 1]. In einer Architektur, die von so viel mehr abhängt als der eigentlichen Geschäftslogik (bspw. Reverse-Proxy, Message Broker etc.), ist es oft schwer, die Fehlerquelle ausfindig zu machen. Auftretende Fehler können sich kaskadenartig durch viele Services weitertragen. Backend-Microservices können durch zahlreiche etablierte Mechanismen abgesichert werden. Ein Ausfall des Frontends stellt ein besonderes Risiko dar, da es unmittelbar die Nutzerinteraktion unterbricht und der Ruf des gesamten Produkts auf dem Spiel steht.

Die gewählte Implementierung der Microfrontends durch Module Federation verfügt über die Möglichkeit, eine sog. „ErrorBoundary“-Komponente zu verwenden. Diese fängt einen Fehler im Microfrontend-Modul ab und bringt den Rest der Anwendung nicht zum Absturz. Außerdem kann eine andere Komponente als Fallback benutzt werden, was dafür sorgt, dass die Anwendung lauffähig bleibt.

Die Bewertung und Definition der Resilienz eines Softwaresystems ist eine eigene Arbeit für sich, da es zahlreiche Studien, Definitionen und Methoden gibt. Jiaxin Pan (und weitere) haben in ihrer Arbeit verschiedene Methoden gegeneinander gestellt und miteinander verglichen [23]. Viele dieser Modelle erfordern dezidiertes Testing und Analysen, welche nicht in den Rahmen dieser Arbeit passen. Das Berechnen der Resilienzmetriken macht Sinn beim Endzustand, aber der Fokus dieser Arbeit liegt auf der Zeit zwischen den Zuständen. Für diese Zeit hat sich nach Ansicht des Autors vor allem die „Recoverability“ als gutes Mittel der Bewertung ergeben.

Die Bewertung der „Recoverability“ erfolgt nach folgendem Schema:

grün	gelb	rot
automatischer Rollback <2 min	manueller Rollback <15 min	kein definierter Rollback

Tabelle 1: Bewertungsschema Resilienz

Dabei wird lediglich der Arbeitsaufwand gemessen, nicht die tatsächliche Deployment-Zeit, da diese von der Pipeline abhängig ist.

4.2 Releasezyklus

Ein zentraler Vorteil von Microservices ist die Verkürzung der Releasezyklen [24], daher ist es wichtig, die Patterns dahingehend zu bewerten. Durch die Aufteilung in kleinere, unabhängige Einheiten können Teams/Entwickler:innen schneller und autonomer Änderungen vornehmen und deployen. Da am Ende des Weges alle Patterns dazu geführt haben, dass jeder Service sein eigenes Microfrontend hat und von da an die Releasezyklen wieder ähnlich kurz sind, wurde nur der anfängliche Zeitaufwand bewertet. Dieser initiale Aufwand ist besonders relevant, da er eine erhebliche Investition darstellt und den Zeitgewinn, der nach der Migration gewonnen wird, nicht um ein erhebliches übertreffen sollte.

Martin Lehmann hat in seiner Arbeit ein Framework zur Messung von „Continuous Delivery“-Strategien geschaffen [25]. Er bringt folgende Punkte an, um eine Pipeline zu bewerten:

- „Testability“
- „Deployment Abstraction“
- „Environment Parity“
- „Time to deploy“
- Availability adequacy“

Dieses Framework eignet sich hervorragend, um die CI/CD Pipelines standardisiert zu vergleichen, allerdings liegt der Fokus dieser Arbeit bei dem Prozess der Migration und nicht, wie die Pipeline danach aussieht. Auch im Buch von Len Bass (und weiteren) zum Thema Softwarearchitektur werden drei Metriken genannt, um die Qualität einer Pipeline zu messen, dort sind es: „Cycle Time“, „Traceability“ und „Repeatability“ [26].

Im Beispiel dieser Arbeit ist die Pipeline allerdings schon vorgegeben und wird von keinem Pattern berührt. Es macht in diesem Zusammenhang keinen Sinn, die Pipelines miteinander zu vergleichen. Viel mehr ist es wichtiger zu unterscheiden, wie viel Aufwand betrieben werden muss, damit ein Ergebnis der Migration auf Production deployt werden kann. Vor diesem Hintergrund werden hier die Zeilen Code gemessen, die geändert werden müssen, bis etwas deployt werden kann, und ob überhaupt während der Migration etwas deployt werden kann, oder diese erst abgeschlossen sein muss, bevor es ein brauchbares Ergebnis gibt. Daraus ergibt sich folgendes Bewertungsschema:

grün	gelb	rot
<10% Zeilen geändert und/ oder viele Deployments während Migration möglich	10%-50% Zeilen geändert; Deployments erst nach	<50% Zeilen geändert und/ oder keine Deployments während Migration möglich

Tabelle 2: Bewertungsschema Releasezyklus

Code, der von dem Monolithen kopiert wurde, zählt in dieser Arbeit nicht als geänderter Code. Lediglich Code, der spezifisch für das Pattern hinzugefügt oder geändert wurde. Auch spezifischer Code für das Setup, der für alle Patterns gleich ist, wurde nicht berücksichtigt (etwa die package.json, vite config etc.). Jede Zeile enthält nur einen Befehl. Die Skala orientiert sich an der Größe des Services (436 LoC).

4.3 Teamautonomie

Microservices erfordern ein Unternehmen mit autonomen Teams [27] und unterstützen diese Arbeitsweise. Die Autonomie liegt in der Sache der Microservice-Architektur und sollte daher auch bei der Migration des Frontends besonders berücksichtigt werden. Eine hohe Teamautonomie bringt viele Vorteile mit sich und spielt auch bei den anderen Kriterien eine große Rolle. Die explizite Bewertung der Autonomie ermöglicht auch eine Einordnung der anderen Kriterien. Sind durch ein Pattern bspw. Services noch stark aneinander gekoppelt, verlängert sich durch diese Abhängigkeit auch der Release-Zyklus oder die Resilienz kann darunter leiden.

Die Autonomie der Teams misst sich daran, wie stark die Services aneinander gekoppelt sind und über wie viele Dinge während der Entwicklung kommuniziert werden muss. Für die Kopplung von Microservices haben sich Chenxing Zhong (et al.) ein ausgeklügeltes System überlegt [28], um einen „Microservice Coupling Index“ (MCI) zu berechnen. Dabei wird berücksichtig, wie viele Schnittstellen jeweils von einem Service genutzt werden und wie viele Entitäten innerhalb eines Services von einem anderen abhängen. Die Werte sind dabei gerichtet, dass heißt, die Kopplung von A zu B ist nicht gleich die umgekehrte.

Weitergehend wird auch der „aMCI“ und „eMCI“ entwickelt, was für „afferent“ und „efferrent“ steht. „Afferent“ bezeichnet dabei die Verantwortung, die ein Service trägt, weil viele andere von ihm abhängig sind. „Efferent“ hingegen ist die Verwundbarkeit eines Services, der von vielen abhängig ist.

Im Rahmen dieser Arbeit wurde die Bewertung vereinfacht und beschränkt sich oberflächlich auf die Anzahl der Abhängigkeiten während der Migration. Da das Ziel der Migration ist, einen Service komplett in ein Team zu geben und diese Aufteilung unter allen Patterns gleich ist, wird nicht etwa die Anzahl der Aufrufe der Schnittstellen zu anderen Services gezählt, sondern generelle Absprachen, welche die Teams treffen müssen, um die Architektur zu migrieren. Dazu zählen etwa Technologieentscheidungen (bspw. Socket.io oder React), aber auch Eigenschaften der Komponenten, welche zwischen den Teams kommuniziert werden müssen. Bei Klassen und Interfaces zählt jede Funktion oder Eigenschaft als eigene Abhängigkeit mit dem Faktor 0,5. Alles, was in beiden Systemen vorkommt, wird als Abhängigkeit betrachtet.
Die Skala ist an die Größe und Komplexität des Projekts angepasst.

grün	gelb	rot
<3 Abhängigkeiten	3-7 Abhängigkeiten	>7 Abhängigkeiten

Tabelle 3: Bewertungsschema Teamautonomie

5 Anwendung & Bewertung der Patterns

Die in Punkt 3 ausgewählten Patterns wurden auf die Codebasis von „Jukefy“ angewendet und der Prozess sowie das Ergebnis der Migration mit den in Punkt 4 definierten Kriterien bewertet. Aus der Anwendung wird das Frontend für den „PollService“ ausgegliedert. Dieses besteht aus drei Hauptkomponenten:

- ListItem Component
- PollList Component (inkl. LocalStorage Management Dateien)
- Dateien für den Backend-Zugriff (HTTP API Interface, Types, Error Types)

Die „PollList“-Komponente ist dabei die einzige, die von einer servicefremden Komponente geladen wird und zwingend über Module Federation bereitgestellt werden muss. Die Dateien für den Backend-Zugriff sollen nur innerhalb des Poll-Frontends verwendet werden, damit die Domäne „Umfrage“ von allen anderen komplett getrennt wird.

Die Dateien für den Backend-Zugriff bleiben bei allen Patterns unverändert. Diese enthalten die Definition und Logik für den Aufruf des Backend Services. Die einzigen Dateien, die relevant sind und verändert werden, sind die für die React-Komponenten.

5.1 Strangler-Fig Pattern

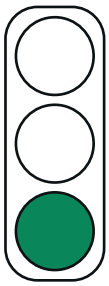
Das Strangler-Fig Pattern lässt sich sehr gut im Bereich der Microfrontends anwenden. Es kann jede Komponente nacheinander übertragen werden und mithilfe eines Feature-Toggles freigeschaltet werden. Um das Strangler-Fig Pattern anzuwenden, wurde

ein neues React-Projekt eingerichtet und das Module Federation Plugin für Vite installiert und eingerichtet.

Ist das Projekt eingerichtet, kann direkt mit der Entwicklung begonnen werden. Hierzu kann der Code Modul für Modul von dem Monolithen rüberkopiert werden. Ist ein Modul kopiert und lauffähig, kann es in der Vite-Konfiguration für Module Federation eingerichtet und dem Monolithen so zur Verfügung gestellt werden. Auf Seite des Monolithen kann das neue Modul dann wie das alte in den Code eingefügt werden. Zu Testzwecken wurde auch noch eine Checkbox eingebaut, um zwischen den Komponenten zu wechseln, besser wäre hier allerdings die Möglichkeit, über eine Umgebungsvariable zwischen dem Microfrontend und dem alten zu wechseln.

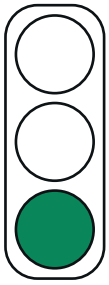
5.1.1 Bewertung

Resilienz:



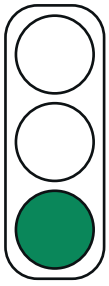
Da die Migration mit einer kleinen Teilkomponente beginnt, lässt sich ein eventueller Fehler sehr schnell eingrenzen. Da sich zu dem Zeitpunkt die alte Komponente noch im Monolithen enthalten ist, kann sie als Fallback in der „ErrorBoundary“ benutzt werden. Im schlimmsten Fall kann durch einen einfachen Schalter auch wieder die alte Komponente genutzt werden. Sollte also das Microfrontend nicht erreichbar sein, können sowohl das Team, das an dem Monolithen arbeitet, wie auch das Microfrontend-Team an sich aktiv werden und den Fehler beheben. Wenn der Schalter zum Umschalten auf das Microfrontend durch eine Umgebungsvariable implementiert ist, so kann diese während des Deployments nach entsprechendem automatischen Tests gesetzt werden. Außerdem kann so, wenn ein Fehler auftritt, automatisch zurückgeschaltet werden.

Releasezyklus:



Durch den inkrementellen Ansatz des Strangler-Fig-Pattern kann während der Migration sehr schnell und oft deployt werden. Es kann jede Teilkomponente nacheinander migriert und als Microfrontend eingebunden werden. Es wird kaum neuer Code benötigt, da Komponente für Komponente kopiert werden kann. Lediglich die Module Federation Konfiguration muss nach jeder Komponente auf der Sender-Seite angepasst werden, um die neu migrierte Komponente auch bereitzustellen. Auf der Konsumenten-Seite muss nach der anfänglichen Konfiguration nur der Name bzw. der Pfad des importierten Moduls geändert werden. Es mussten nur zwölf Zeilen geändert werden.

Team-Autonomie:



Nach dem Kopieren des Codes aus dem Monolithen ist das Team sofort autonom und kann weitgehend unabhängig von dem Monolithen weiterentwickeln. Neben dem Namen und der Adresse der bereitgestellten Komponente müssen auch die Eigenschaften, in diesem Fall zwei, kommuniziert werden, damit es im Monolithen über Module Federation geladen werden kann. Wenn die Migration durch Kopieren der Komponenten stattfindet, sind die Eigen-

schaften dieser gleich und die Komponenten an sich austauschbar. Die Kleinschrittigkeit der Migration erfordert aber ein ständiges Deployment von beiden Seiten.

5.2 Branch-By-Abstraction-Pattern

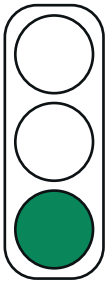
Das Branch-by-Abstraction-Pattern hat bei der Migration von den Komponenten aus dem Monolithen unterstützt. Das Pattern überzeugt dabei mit der Möglichkeit der schnellen Deployments. Es wird ein Proxy im Monolithen eingerichtet, der zwischen der lokalen und der entfernten Komponente umschalten kann. Dieser wird in der Eltern-Komponente im Monolithen auch direkt verwendet, selbst wenn der Microfrontend-Service noch nicht läuft. Von hier an kann das Microfrontend dann unabhängig entwickelt werden. Sobald es läuft, kann im Proxy umgeschaltet werden. Bei der hier gewählten Implementierung passiert das durch eine „ErrorBoundary“-Komponente automatisch. Läuft diese Änderung stabil, können größere Änderungen vorgenommen werden.

Um die beiden Dienste noch weiter voneinander zu entkoppeln, kann nun im Microfrontend eine Abstraktionsschicht implementiert werden, welche die Kompatibilität zu der alten Komponente gewährleistet. Diese Abstraktions-Komponente würde dann im Monolithen in der Proxy-Komponente anstelle der „PollList“-Komponente verwendet. Nach diesem Schritt können nun die meisten Änderungen komplett unabhängig vom Monolithen getätigt werden. Um das zu überprüfen, wurden die Eigenschaften der „PollList“-Komponente geändert. Diese Änderung wurde in der Abstraktionsschicht aufgefangen.

Diese Herangehensweise ermöglicht Teams intern schnellere Änderungen umzusetzen, ohne groß Rücksicht auf den Monolithen nehmen zu müssen.

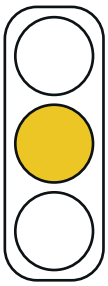
5.2.1 Bewertung

Resilienz:



Durch die Proxy-Komponente während der Migration ist der Monolith sehr schnell wieder in einen funktionierenden Zustand zurückversetzt. Auch hier bleibt die alte Komponente für die Dauer der Migration erhalten und kann in der „ErrorBoundary“ als Fallback genutzt werden. Durch die zusätzliche Abstraktionsschicht wird allerdings eine neue Fehlerquelle eingeführt. Diese ist zwar nur für die Dauer der Migration dort, sorgt allerdings innnerhalb eines Teams für erweiterte Komplexität. Sollte ein Fehler auftreten, kann das Team, das am Monolith arbeitet, durch die Proxy-Komponente sehr schnell wieder in den alten Zustand zurückkehren. Hier kann mit einer Umgebungsvariable und automatischen Tests gearbeitet werden, welche einen schnellen automatischen Rollback ermöglichen. Ein Fehler bei der Kommunikation der Eigenschaften, wodurch die beiden Ebenen verbunden werden, ist schnell gefunden und behoben.

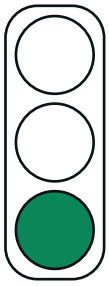
Releasezyklus:



Bei der Migration mithilfe dieses Patterns kann ständig deployed werden. Durch die inkrementellen Änderungen und die Einführung des Proxy und der Abstraktionsschicht ist jeder Schritt an sich lauffähig. Die Migration war nicht aufwendig. Die Erstellung des Proxy ging sehr schnell, da dieser überwiegend aus einer If-Bedingung besteht. Insgesamt wurde die Migration der

Komponenten in wenigen Stunden abgeschlossen. Das Löschen der Abstraktionsschicht nach der Migration verdoppelt allerdings nahezu die Anzahl der veränderten Zeilen Code.

Team-Autonomie:



Die Autonomie des Teams während der Migration ist sehr hoch. Nach der Einrichtung des Proxy im Monolithen und der Abstraktionsschicht auf der Microfrontend-Seite kann das Microfrontend unabhängig weiterentwickelt bzw. migriert werden. Ist die Migration abgeschlossen, sind die Teams aber wieder auf Abstimmungen bezüglich der Eigenschaften der Komponenten angewiesen. Gerade große Änderungen erfordern dann wieder Planung oder die erneute Einführung einer Abstraktionsschicht. In diesem Fall müssen für die Komponente aber nur zwei Variablen gesetzt werden.

5.3 Anti-Corruption-Layer-Pattern

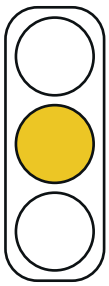
Das Anti-Corruption-Layer-Pattern ist eine hervorragende Wahl, um bei der Migration zu Microfrontends zu unterstützen. Nach der Einrichtung des Microfrontends und dem Kopieren der Komponenten aus dem Monolithen wurde das Anti-Corruption-Layer als Interface implementiert. Dieses Interface ersetzt die Eigenschaften der „PollList“-Komponente. Diese Komponente hat verschiedene Abhängigkeiten von der Elternkomponente, welche in diesem Fall nicht mehr in der Domäne der Umfragen liegen.

Einerseits hängen die Umfragen an den „Sessions“ und benötigen eine „session_id“, um die Umfragen für die richtige Session zu laden. Andererseits reagiert und sendet die Liste mehrere Events. In diesem Fall sind das zum einen Socket.io-Events und zum anderen Browser-Events. Diese Technologiewahl wird durch den Monolithen vorgegeben und koppelt die beiden Systeme nun aneinander.

Durch das Anti-Corruption-Layer wird diese Abhängigkeit gelöst, da in der „PollList“-Komponente selbst nur die generischen Handlerfunktionen für die Events aufgerufen werden, die Handlerfunktionen aber in der Elternkomponente (in dem Fall im Monolithen) definiert werden.

5.3.1 Bewertung

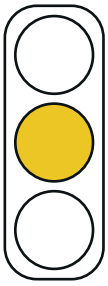
Resilienz:



Trotz der Anti-Korruptionsschicht können Fehler auf beiden Seiten dieser auftreten. Die Fehler sind zwar gut isoliert, sollte allerdings etwas mit der Konfiguration oder Implementierung der Schicht nicht stimmen, wird es für ein einzelnes Team schwer, den Fehler auf der jeweils anderen Seite zu finden, obwohl ein Fehler auf der jeweils anderen Seite auch die eigene beeinträchtigen kann. Da die Schicht allerdings bestehen bleibt, könnte so etwas nur während der Migration passieren und im laufenden Betrieb nicht mehr. Dadurch, dass die Schicht als permanente Lösung entwickelt ist, kann die

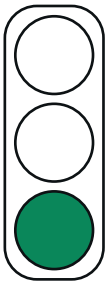
alte Komponente als Fallback für die „Error-Boundary“ auch länger im alten System bestehen bleiben. Ein automatischer Rollback ist allerdings nicht möglich, somit muss ein Team aktiv werden und manuell auf die alte Version zurückstellen.

Releasezyklus:



Der initiale Aufwand besteht hier in der Definition und Extraktion von wichtigen Eigenschaften und Funktionen in die Anti-Korruptionsschicht. Dies resultiert in der Deklaration eines Interfaces pro Komponente. In diesem Projekt wurde für eine Komponente, die in den Monolithen eingebunden wird, ein Interface deklariert, das aus elf Zeilen Code besteht. In der Komponente selbst wurde nicht viel verändert, was zu einem schnellen Release geführt hat. Da allerdings auch die konsumierende Seite den Code auf die Schicht umbauen musste, war der Releasezyklus länger und von Dritten abhängig. Insgesamt wurden 60 Zeilen Code bearbeitet.

Team-Autonomie:



Nach der Änderung auf beiden Seiten hin zur Nutzung der Anti-Korruptions-Schicht sind beide Teams deutlich autonomer. Dieses Pattern eignet sich hervorragend, um die Domänen zu trennen. Besonders das Zurückgeben der Verantwortung für das An- und Abmelden der Events an das konsumierende System ist hier hervorzuheben. Dadurch wird auf der Seite des bereitstellenden Dienstes ein einheitliches Schema für Events geschaffen. Die einzige Abhängigkeit ist die Definition des Anti-Corruption Layer Interfaces.

6 Fazit

Die Migration von einem monolithischen Frontend zu Microfrontends gestaltet sich als nicht so aufwendig, da auf die bereits von den Microservices im Backend vorgegebenen Domänen zurückgegriffen werden kann. Die Vorteile der gewonnenen Teamautonomie, kürzeren Releasezyklen und der verbesserten Resilienz überzeugen im Frontend genauso, wie sich schon im Backendbereich überzeugt haben.

Das Strangler-Fig-Pattern eignet sich besonders für eine schnelle und inkrementelle Migration, da dort kein neuer Code benötigt wird, sondern die alten Komponenten lediglich in ein neues Projekt übertragen werden können und nach einander im alten System ersetzt werden können. Somit ist die Migration zwar unkompliziert, jedoch sind die Systeme noch über die Eigenschaften der Komponenten und die Verwaltung der Events bzw. der Technologie Vorgabe aneinander gekoppelt. Diese Kopplung hält sich allerdings in Grenzen.

Der Vorteil vom Branch-By-Abstraction-Pattern liegt in der Proxy-Komponente. In dieser kann während der Migration zwischen den alten und neuen hin und her gewechselt werden. Theoretisch ließe sich dort auch ein A-B-Testing realisieren oder eine Komponente, welche die neuen Module erst für eine Hand von Nutzenden ausrollt. Für den Preis der Implementierungszeit für den Proxy/die Abstraktionsschicht bekommt man allerdings mehr Sicherheit, um bei Fehlern wieder zum alten System zurückzukehren.

Das Anti-Corruption-Layer-Pattern arbeitet ähnlich wie das Branch-By-Abstraction-Layer-Pattern mit einer neuen Komponente, welche als Bindeglied zwischen dem alten und neuen System steht. Anders als die vorherigen Pattern ist dieses aus der Welt des Domain-Driven-Designs und soll eine permanente Brücke zwischen zwei Domänen schlagen. Auf das Frontend übertragen, schlagen wir hier die Brücke zwischen dem Monolithen und dem Microfrontend. Dieses Pattern überzeugt durch die Entkopplung aller Services. Aufgaben, die mehrere Dienste betreffen, werden aus dem Microfrontend herausgezogen und in dem Monolithen vorgenommen. Besonders hervorzuheben ist in diesem Fall die Einrichtung der Events über Socket.io oder Browser-Events. Das Microfrontend setzt nur voraus, dass Events gesendet und empfangen werden können, auf die mit einem Handler reagiert wird. Wie das Ganze geschieht, ob es jetzt über Socket.io, WebSockets, Browser-Events oder einer ganz anderen Lösung passiert, ist für das Microfrontend an sich nicht relevant und fällt deshalb in den Aufgabenbereich des Monolithen.

	Resilienz	Releasezyklus	Team-Autonomie
Strangler-Fig	🟢	🟢	🟢
Branch-By-Abstraction	🟢	🟡	🟢
Anti-Corruption-Layer	🟡	🟡	🟢

Tabelle 4: Vergleichstabelle Pattern Bewertung

An der vorliegenden Anwendung bot das Strangler-Fig-Pattern alles in allem den geringsten Aufwand bei der Migration, da kein neuer Code hinzugefügt werden musste. Bei anderen Code-Basen, wo der Code im Frontend noch nicht so strukturiert ist und welcher über die Zeit gewachsen ist, könnten Branch-By-Abstraction oder das Anti-Corruption-Layer ihre Vorteile entfalten und hilfreicher sein.

6.1 Ausblick

Im Rahmen von „Jukefy“ lief die Migration sehr unkompliziert und schnell. Überträgt man die Pattern auf ein großes Projekt, an dem viele Teams unabhängig arbeiten, so steht man vor anderen Herausforderungen. Beginnend muss natürlich jedes Microservice-Team auch Fähigkeiten im Frontendbereich haben. Ein Unternehmen muss dahingehend die Teams umstrukturieren und eventuell Kapazitäten aus dem monolithischen-Frontend-Team in die Microservice-Teams verteilen oder neue Mitarbeitende anwerben. Die Technologie-Offenheit von Microfrontends ist bei dieser Herausforderung allerdings sehr unterstützend. Sollten neue Talente rekrutiert werden müssen, so ist das Unternehmen dort flexibler aufgestellt.

Was die Autonomie der Teams betrifft, so sind diese, was den Frontend-Bereich angeht, auf jeden Fall unabhängiger von anderen, da jede Einheit ja auch nur das Frontend zu dem Backend im eigenen Zuständigkeitsbereich bereitstellen muss. Daher wird die meiste Kommunikation zwischen dem Anbieter der Komponenten und dem Konsumenten der Komponenten stattfinden müssen.

Im Bereich der Infrastruktur sind keine großen Hürden zu erwarten. Die von ModuleFederation bereitgestellten Komponenten müssen lediglich für die Clients erreichbar sein. Der neue Frontend-Service fügt sich wie andere Services in die vorhandene Infrastruktur ein.

Um das Problem von standardisierten und wiederholten Tests zu umgehen, lohnt es sich, bestimmte Pipeline-Schritte in denen Tests laufen, die über den eigenen Service hinausgehen, in ein Template zu verpacken. In GitLab bspw. ist dies ohne weiteres möglich. So können verschiedene Teams gemeinsame Tests verwenden und die Funktion der Anwendung als ganzes Testen.

Quellenverzeichnis

1 OYENIRAN, Oyekunle Claudius, et al. Microservices architecture in cloud-native applications: Design patterns and scalability. International Journal of Advanced Research and Interdisciplinary Scientific Endeavours, 2024, 1. Jg., Nr. 2, S. 92-106.

2 UGWUEZE, Vincent Uchenna und CHUKWUNWEIKE, Joseph Nnaemeka, 2025. Continuous Integration and Deployment Strategies for Streamlined DevOps in Software Engineering and Application Delivery. In: International Journal of Computer Applications Technology and Research, 14(1), 1–24. ISSN 2319–8656. DOI 10.7753/IJCATR1401.1001.

3 https://www.thoughtworks.com/radar/techniques/micro-frontends; abgerufen am 14.04.2025

4 NEWMAN, Sam, 2021. Building Microservices Designing fine grained system. Second Edition; O'Reilly. ISBN 9781492034025

5 FOWLER, Martin; https://martinfowler.com/bliki/StranglerFigApplication.html; abgerufen am 03.04.2025

6 FOWLER, Martin; https://martinfowler.com/bliki/BranchByAbstraction.html; abgerufen am 03.04.2025

7 EVANS, Eric, 2004. Domain-driven design: tackling complexity in the heart of software. Boston [u.a.]: Addison-Wesley. ISBN 9780321125217

8 NEWMAN, Sam und Thomas DEMMIG, 2020. Vom Monolithen zu Microservices: Patterns, um bestehende Systeme Schritt für Schritt umzugestalten. Heidelberg: O'Reilly Verlag. ISBN 9783960104230

9 NADAREISHVILI, Irakli, et al. Microservice architecture: aligning principles, practices, and culture. „ O'Reilly Media, Inc.", 2016.

10 BOLLEN, Enrico; Git-Repository: Jukefy; https://gitlab.com/jukefy/jukefy; abgerufen am 22.04.2025

11 FOWLER, Martin; https://martinfowler.com/articles/micro-frontends.html; abgerufen am 14.04.2025

12 ZHANG, Cheng; BUDGEN, David. What do we know about the effectiveness of software design patterns?. IEEE Transactions on Software Engineering, 2011, 38. Jg., Nr. 5, S. 1213-1231.

13 PRECHELT, Lutz, et al. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. IEEE Transactions on Software Engineering, 2002, 28. Jg., Nr. 6, S. 595-606.

14 ALEXANDER, Christopher, 1977. A pattern language: towns, buildings, construction. New York, NY: Oxford. Univ. Pr. ISBN 9780195019193; [übersetzt]

15 GAMMA, Erich, 1995. Design Patterns: Elements of Reusable Object Oriented Software. Reading, MA: Addison Wesley. ISBN 0201633612; [übersetzt]

16 https://jinja.palletsprojects.com/en/stable/templates/#child-template; abgerufen am 25.04.2025

17 XIAO, Zhou und JACKSON, Zack; https://module-federation.io/blog/announcement.html#type-hints; abgerufen am 14.04.2025

18 FOWLER, Martin; https://martinfowler.com/bliki/StranglerFigApplication.html; abgerufen am 03.04.2025

19 FOWLER, Martin; https://martinfowler.com/bliki/BranchByAbstraction.html; abgerufen am 03.04.2025

20 VERNON, Vaughn, [2016]. Domain-driven design distilled. Boston ; Columbus ; Indianapolis: Addison-Wesley. ISBN 9780134434421; E-BOOK; Chapter 4. Strategic Design with Context Mapping

21 https://learn.microsoft.com/de-de/azure/architecture/guide/technology-choices/microservices-assessment#record-architectural-decisionst; abgerufen am 27.03.2025

22 Kanglin Yin, Qingfeng Du 2020. On Representing Resilience Requirements of Microservice Architecture Systems

23 PAN, Jiaxin, et al. An empirical study of software architecture resilience evaluation methods. Journal of Systems and Software, 2023, 202. Jg., S. 111726.

24 https://camunda.com/blog/2023/02/benefits-of-microservices-advantages-disadvantages/; abgerufen am 31.03.2025

25 LEHMANN, Martin; SANDNES, Frode Eika. A framework for evaluating continuous microservice delivery strategies. In: Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing. 2017. S. 1-9.

26 BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. Software architecture in practice. Addison-Wesley Professional, 2021; auf O‘Riley online gelesen am 21.04.2025

27 FOWLER,Martin; https://martinfowler.com/articles/break-monolith-into-microservices.html; abgerufen am 31.03.2025

28 ZHONG, Chenxing, et al. On measuring coupling between microservices. Journal of Systems and Software, 2023, 200. Jg., S. 111670.