

OBJEKTSpektrum

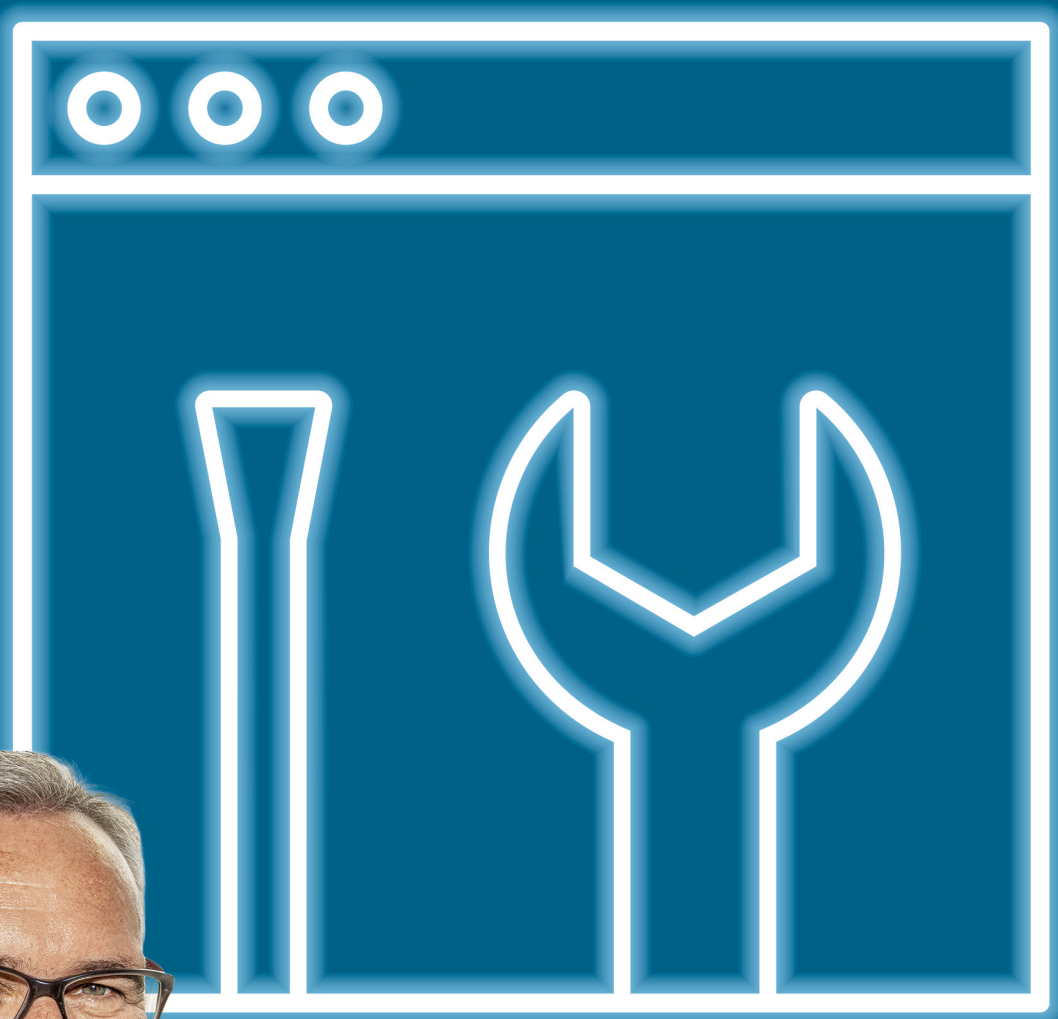
Software-Architektur und IT für Profis

Wie man humane Frontends baut und den Erfolg von Softwareprojekten mitbestimmt

www.OBJEKTSpektrum.de

Wie man humane Frontends baut und den Erfolg von Softwareprojekten mitbestimmt

G 6540F



Interview mit Dirk Fassott

Der IT-Vorstand der WWK Versicherungen erklärt, warum COBOL nicht tot und selbst BS2000 noch lebendig scheint

- Erfahrungsbericht: Von Wasserfall bis agil
- Security-Anforderungen für verteilte Umgebungen modellieren
- Was agile Teams in Corona-Zeiten leisten können

Frontends mit Microservices

Warum die Auswahl der Frontend-Architektur auch Auswirkung auf die Teamzusammenarbeit hat

Microservices sind über die letzten Jahre zu einem etablierten Architekturansatz herangewachsen. Sie sind mittlerweile soweit etabliert und erprobt, dass der Hype vorüber ist, und es liegen jetzt genügend Erfahrungen über Nutzen, Grenzen, Vor- und Nachteile vor. Dieser Artikel beleuchtet, welche Vorteile oder Versprechen der Microservice-Ansätze sich erfüllen lassen, wenn Frontends mitbetrachtet werden.

Für die Übertragung der Microservice-Konzepte auf Frontends hat sich der Begriff Micro-Frontend etabliert. Wir vergleichen die aus unserer Sicht wichtigsten Ansätze und Lösungsmuster wie SSI (Server Side Includes), JavaScript mit Ajax sowie Web Components. Dabei berichten wir über unsere Erfahrungen aus der Praxis. Wie bei allen grundlegenden Architekturentscheidungen sollte man sich vor dem Einsatz von Micro-Frontends über deren Vor- und Nachteile im Klaren sein. Eine Korrektur dieser grundlegenden Entscheidung ist später nur mit hohem Aufwand möglich. Neben den architektonischen und technologischen Aspekten gehen wir auch darauf ein, was der Einsatz von Micro-Frontends in der teamübergreifenden Zusammenarbeit bedeutet.

Businessszenarien

Nach unseren Erfahrungen lassen sich Frontends aus fachlicher Sicht betrachtet in drei grobe Businessszenarien einteilen:

- **Inhouse:** Expertenapplikationen, die mir als Anwender zur Erledigung der eigenen Arbeit zur Verfügung gestellt werden.
- **Business-to-Consumer (B2C):** Web-Anwendungen wie zum Beispiel Online-

Shops oder On-demand-Angebote, in denen ich als Anwender Waren oder Dienstleistungen erkaufen oder abonnieren kann.

- **Business-to-Business (B2B):** Web-Anwendungen in einem Geschäftskontext, mit denen ich Geld Erlösen kann, wie zum Beispiel Zugänge zu Marktplätzen, auf denen eigene Waren oder Dienstleistungen angeboten werden können.

Jedes Businessszenario hat zudem unterschiedliche Schwerpunkte in seinen nicht-funktionalen Anforderungen (siehe Tabelle 1). Dabei lässt sich feststellen, dass die Wichtigkeit der meisten nicht-funktionalen Anforderungen zunimmt, je näher die Software am Endkunden liegt und je weniger wir den Endkunden persönlich kennen oder ansprechen können. Nehmen wir die Zuverlässigkeit als Beispiel, werden Inhouse-Anwender potenziell mehr Verständnis für Wartungsarbeiten oder Ausfälle aufbringen können als ein anonymer Endanwender, welcher ungeplant und spontan auf einen angebotenen Service zugreifen möchte.

Betrachten wir nun speziell Anwendungen innerhalb dieser Szenarien, welche auf Basis von Microservices entwickelt wurden, so sind einige dieser Anforderungen (z. B. Skalierbarkeit) bereits vom Grundgedanken her abgedeckt und müssen nicht

extra im Lösungsdesign berücksichtigt werden. Für Microservice übergreifende Frontends und deren Look & Feel oder Benutzbarkeit gilt dies leider nicht. Wir müssen uns dabei selbst neue Lösungsansätze überlegen. Im Folgenden wollen wir diese näher betrachten und dabei den jeweiligen Lösungsansatz im Zusammenspiel mit dem Businessszenario bewerten.

Microservices

In den letzten Jahren hat sich in vielen Projekten der Architekturstil der Microservices etabliert. Dabei steht als oberster Gedanke die Zerlegung des Gesamtsystems in kleinere gut wartbare Einheiten. Das Ziel dabei ist es, innerhalb der jeweiligen Microservices einen fachlich engen Zusammenhalt und untereinander eine möglichst geringe Kopplung zu erreichen. So können Bestandteile des Gesamtsystems unabhängig voneinander weiterentwickelt werden. Es sei angemerkt, die fachliche Zerlegung selbst ist auch der kritische Teil. Diese entscheidet, ob das Ziel kleiner unabhängiger Einheiten erreicht wird oder man einen „Spaghetti-Code-Monolithen“ nur mittels verteilter Services umgesetzt hat.

Richtig umgesetzt versprechen Microservices die in Tabelle 2 zusammengefassten Vorteile.

Anforderungen (allgemein)	Inhouse	B2B	B2C
Leistung und Skalierbarkeit Anzahl paralleler Benutzer	–	0	+
Zuverlässigkeit Zero Downtime	0	0	+
Benutzbarkeit Antwortzeiten, Ladezeiten	0	0	+
Look & Feel Usability, unterstützte Geräte	0	0	+
Sicherheitsanforderungen Security, SSO/FIDO2	+	+	+

Anforderungen (speziell)	Inhouse	B2B	B2C
Auffindbarkeit Suchmaschinenoptimierung	–	0	+
Repräsentation Corporate Identity	–	+	+
Entwicklung Anzahl Entwickler, Schnitt der Teams	0	0	+
Rollout Frequenz, CI/CD	–	0	+

Nach Wichtigkeit: + hoch, 0 mittel, – niedrig

Tabelle 1: Einige nicht-funktionale Anforderungen in den Businessszenarien Inhouse, B2B und B2C. Bewertung nach Wichtigkeit im jeweiligen Szenario und gemittelt auf Basis unserer Projekterfahrung. Ausreißer in speziellen Anwendungsfällen möglich

Parallele Entwicklung	Teams können unabhängig voneinander entwickeln
Unabhängige Deployments	Teams können einzelne Microservices unabhängig voneinander und unabhängig von anderen Teams ausliefern
Unabhängige Technologiewahl	Jeder Microservice kann in einer zum Problem passenden Technologie entwickelt werden
Schmale Schnittstellen	Es werden nur die notwendigen Schnittstellen nach außen hin geöffnet
Modelliert um Business Domains	Die Microservices sind passend zu einer abgeschlossenen Fachlichkeit („Business Domain“) aufgebaut
Unabhängiger Betrieb	Microservices können unabhängig voneinander betrieben und skaliert werden
Keine technologische Abhängigkeit	zwischen den einzelnen Teams z. B. aufgrund gemeinsam selbstentwickelter Libraries oder geteilten Environments
Basiert auf Standards	Die Implementierung basiert, speziell wenn es um die Interaktion mit anderen Systemen und Microservices geht, auf Standards

Tabelle 2: Vorteile der Microservices

Gerade für Services, die in erster Linie über APIs miteinander verbunden sind oder diese als Public-API bereitstellen, erscheinen die Vorteile sehr erstrebenswert. Aber kann man diese noch einhalten, wenn eine Oberfläche für den Nutzer bereitgestellt werden muss, dessen Anwendungsfall das Zusammenspiel mehrerer Microservices benötigt? Können wir in diesem Fall noch die Vorteile der Microservices gelten? Oder ergeben sich durch die Erweiterung signifikante Nachteile? Zur Beantwortung dieser Fragen gilt es, zunächst zu unterscheiden, welchen Architekturansatz man für das Frontend ausgewählt hat. Ergänzend zur eingangs fachlich motivierten Sicht, sprechen wir hier von der technischen Sicht auf Micro-Frontends.

Frontend-Architektur

Grundlegend unterscheidet man zwischen Frontend-Monolith und Frontend-Integration.

Der Frontend-Monolith umspannt dabei alle Microservices mit einem zentralen Frontend. Bei der Frontend-Integration wiederum ist jeder Microservice für sein eigenes Frontend verantwortlich und die Herausforderung liegt darin, diese miteinander für den Nutzer sinnvoll zu integrieren. Für die Frontend-Integration hat sich in den letzten Jahren ebenfalls der Begriff Micro-Frontends etabliert. Dieser wurde 2016 das erste Mal von Thoughtworks in deren Technology Radar verwendet [Tho16].

Beide Ansätze haben ihre Daseinsberechtigung für bestimmte Anwendungsfälle und damit verbundene Stärken und Schwächen. Wir vergleichen die unterschiedlichen Ansätze miteinander, indem wir prüfen, ob die oben genannten Vorteile von Microservices weiter eingehalten werden können. Zusätzlich möchten wir aus unserer Erfahrung berichten, ob gegebenenfalls entstehende Nachteile sich im jeweiligen Businesszenario auch negativ auswirken.

Frontend-Monolith

Stellen alle Microservices APIs für ihre Businessfunktionen bereit, erscheint es naheliegend, diese über eine zentrale Frontend-Komponente anzusprechen und so dem Nutzer einheitlich zugänglich zu machen.

In diesem Szenario wird das Frontend als eigenständige Einheit entwickelt und kommuniziert mit den unterschiedlichen Services über definierte APIs. Backend-Teams stellen Features nur noch über (versionierte) APIs bereit, und ein dediziertes Team erstellt das Frontend für alle anderen Teams (siehe Abbildung 1). Dadurch ergibt sich eine horizontale Frontend-Schicht, von der alle anderen Teams abhängig sind. Im Sinne einer Microservice-Architektur sind die Teams nicht mehr unabhängig. Die Featurebereitstellung für den Nutzer erfolgt erst, sobald ein neues Frontend-Release ausgeliefert wurde.

Mittels einer Gateway-Schicht kann eine Vereinfachung des API-Zugriffs stattfinden. Zusätzlich kann von der Gateway-Schicht auch die Authentifizierung oder ein Caching übernommen werden. Wenn mehrere Frontend-Monolithen (Native-App, Web) eingesetzt werden sollen, können diese über mehrere spezialisierte Gateways angebunden werden [MSB17]. Dennoch sollte man zur Sicherstellung von Wartbarkeit immer klare Regeln definieren, um zu verhindern, dass neben der Datentransformation Teile der Business-Logik in das Gateway diffundiert. Durch eine einheitliche Umsetzung von UI und UX, sowohl innerhalb der vertikalen Microservices als auch in übergreifenden Use-Cases über mehrere Microservices hinweg, wirkt die Anwendung wie ein klassischer Monolith auf den Endnutzer. Tabelle 3 zeigt, welche Auswirkung der Frontend-Monolith auf die Vorteile von Microservices hat.

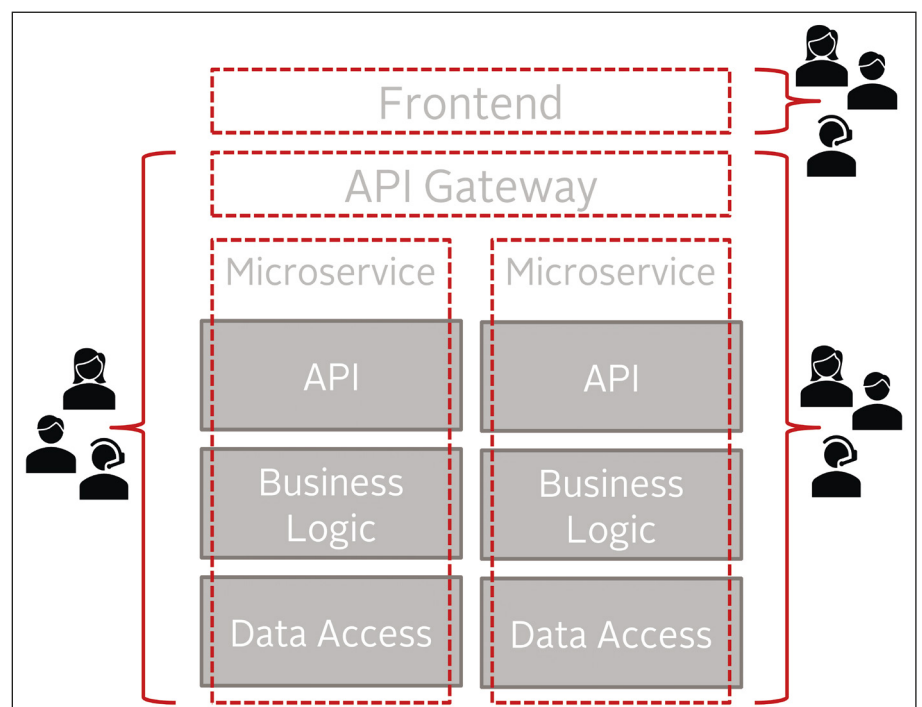


Abb. 1: Frontend-Monolith: Separates Frontend, welches mehrere Backendsysteme für den Nutzer zusammenführt. Verschiedene Teams können für unterschiedliche Teile zuständig sein

Parallele Entwicklung	weiterhin gegeben mit den üblichen Abstimmungen zu Schnittstellen zwischen Front- und Backend
Unabhängige Deployments	Sollen neue Features für den Nutzer ausgeliefert werden, entstehen Abhängigkeiten zwischen dem Frontend-Monolithen und den Backend-Services, welche organisatorisch gelöst werden müssen
Unabhängige Technologiewahl	Alle Backend-Services haben die gleiche Frontend-Technologie
Schmale Schnittstellen	zusätzliche Schnittstellen neben der eigentlichen Fachlichkeit für die Interaktion mit der UI
Modelliert um Business Domains	Die Fachlichkeit über alle Microservices wird im Frontend-Monolithen wieder zusammengebracht
Unabhängiger Betrieb	Zwischen dem Frontend-Monolithen und Backend besteht immer eine Abhängigkeit. Einzelne nur auf einen Backend-Service zeigende Frontend-Module können nicht separat skaliert werden
Keine technologische Abhängigkeit	Das komplette Frontend in einer Technologie
Basiert auf Standards	Kommunikation zwischen Frontend, Backend und ggf. Gateway wird individuell verhandelt

Tabelle 3: Bewertung des Frontend-Monolith-Musters in Bezug auf die Vorteile von Microservices

Habe ich nur wenige Teams oder wenig Fachlichkeit abzudecken, kann ein Frontend-Monolith, oder Layer beziehungsweise Schicht, eine gute Lösung sein. Habe ich dagegen eine größere Zahl an Teams oder eine sehr breit aufgeteilte Fachlichkeit, das heißt, Domänen im Sinne des Domain-Driven Design (DDD), so sind das erste Indikatoren, warum eine Micro-Frontend-Architektur in Betracht kommt und damit eines der in den folgenden Abschnitten beschriebenen Lösungsmuster für Frontend-Integration auf Verwendbarkeit geprüft werden sollte.

Wenn zum Zeitpunkt der Architektur-entscheidung aber wiederum nicht klar ist, wie sich die Domäne unterteilt oder wie viele Teams entstehen werden, sollte zunächst eine einfache Lösung gewählt werden, die nicht zu viele Entscheidungen vorwegnimmt und viel Ramp-up-Aufwand erzeugt. Daher spricht nichts dagegen, sich erst einmal für eine Frontend-Schicht zu entscheiden. Was nicht davon befreit, auch diese Schicht sehr sauber zu designen und zu entwickeln. Auch in der Frontend-Schicht sollten sich klare Domänen finden und Domäne und Technik voneinander getrennt werden. Wenn es

dann später Gründe gibt, die Architektur in Richtung Micro-Frontends weiterzuentwickeln, lässt sich dies mit deutlich weniger Aufwand durchführen.

Bezogen auf die Business-szenarien gibt es keine klare Präferenz beim Frontend-Monolithen. Prüfen sollte man jedoch für den Bereich B2C im Speziellen die Anforderungen in Bezug auf Entwicklungsgeschwindigkeit (z. B. durch unabhängige Deployments) und ob die notwendige Skalierbarkeit der Anwendung erreicht werden kann. Insbesondere ob einzelne Business-Komponenten im Frontend stärker frequentiert sein könnten und damit separat skalierbar bleiben müssen.

Frontend-Integration

Von Frontend-Integration sprechen wir, wenn das Frontend Bestandteil des jeweiligen Microservice ist und von dessen Team mitentwickelt wird (siehe Abbildung 2). Für die Integration mit anderen Microservices stehen mehrere Möglichkeiten zur Verfügung.

Im einfachsten Falle sind die fachlichen Kontexte der Microservices so klar geschnitten, dass eine triviale Integration

mittels einer gemeinsam genutzten Navigation hergestellt werden kann. HTTP-Links zwischen den Kontexten erleichtern es dem Nutzer, zwischen verschiedenen Kontexten zu springen, sofern nötig. Diese Variante stößt aber schnell an Grenzen, sobald immer gleichzeitig Inhalte von verschiedenen Kontexten dem Nutzer parallel präsentiert werden müssen. Eine erweiterte Integration kann über verschiedene Wege realisiert werden: Integration während des Builds, Integration am Server und Integration auf dem Client.

Integration während des Builds

Die Build-Time Integration basiert darauf, die Frontends über Packages bereitzustellen. Diese werden dann von einer Container-App als Dependency angezogen. Das bedeutet aber, dass ich bei einer Änderung in einem Micro-Frontend ein Rebuild und ein komplettes Deployment durchführen muss.

Von diesem Ansatz ist dringend abzuraten. Bei jeder Änderung eines UI-Teils müssen alle Teile des Produkts neu gebaut werden. Auf Codebasis sind die Services gut voneinander getrennt, damit diese unabhängig entwickelt und getestet werden können. Durch die Build-Time Integration hat man aber wieder eine enge Kopplung zwischen den Services. Daher ist unbedingt eine Architektur zu wählen, welche das Frontend erst zur Laufzeit integriert. Es gibt mit Webpack5 und Module Federation einen sehr spannenden Ansatz. Zur Build-Time müssen die Abhängigkeiten nur definiert sein. Die Auslieferung der einzelnen Module erfolgt dabei unabhängig in einzelnen Deployment-Einheiten. Mehr dazu ist bei [Mod20] zu finden. Dieser Ansatz ist unabhängig vom UI-Framework und funktioniert mindestens mit Angular und React robust, auch wenn Webpack5 erst in der Beta- beziehungsweise in der RC2-Version verfügbar ist. Diese Lösung hat den Vorteil,

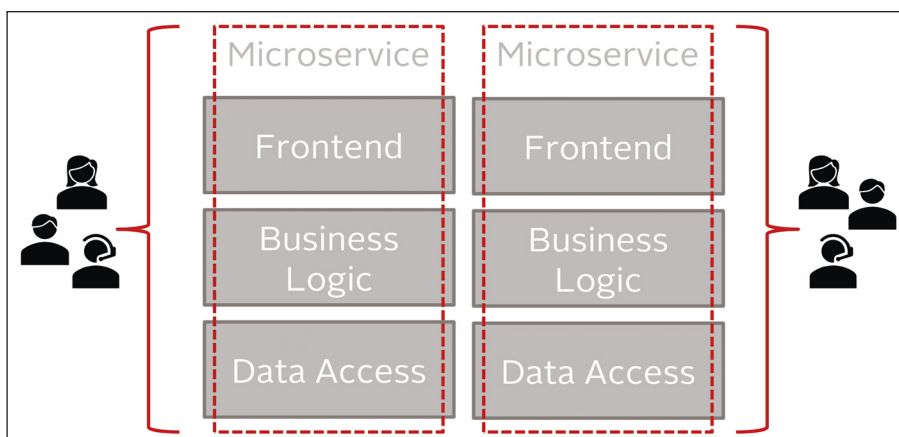


Abb. 2: Frontend-Integration: Das Frontend ist Bestandteil für der jeweiligen Microservices. Diese müssen separat verknüpft werden, um für den Nutzer eine einheitliche UI/UX bereitzustellen

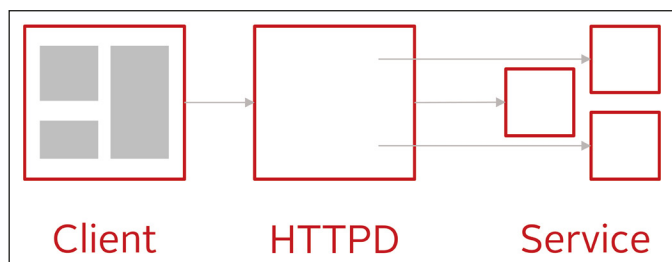


Abb. 3: Komponenten und Aufrufe bei der Integration zur Laufzeit am Server

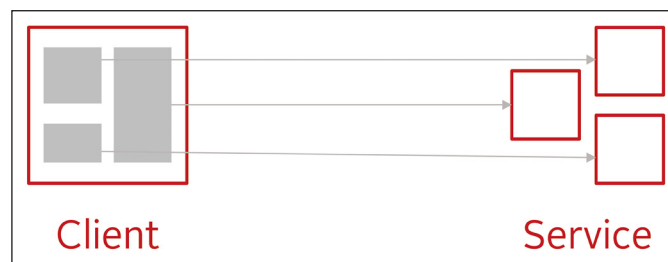


Abb. 4: Komponenten und Aufrufe bei der Integration zur Laufzeit am Client

dass nicht alle Nachteile einer Build-Time Integration zum Tragen kommen und man eine Validierung der Abhängigkeiten bekommt. Dafür müssen alle Teams jedoch Webpack5 einsetzen.

Integration zur Laufzeit am Server

Eine der einfachsten Lösungen ist der Zusammenbau am Server. Frontends, welche aus mehreren Bestandteilen unterschiedlicher Backend-Microservices bestehen, werden vom Server aufgelöst und gebündelt an den Client zurückgegeben. Für den Client ist nur ein Request notwendig (siehe Abbildung 3). Dies spielt seine Stärken insbesondere in einem Umfeld aus, in dem mit höheren Latenzzeiten gerechnet werden muss oder Verbindungen nicht immer stabil sind.

Die Technologien für die Realisierung sind auf Ebene des Webserver beziehungsweise Cache mit Server Side Includes (oder Edge Side Includes, ESI) verfügbar. Auch möglich ist die Realisierung mittels separater Frameworks (z. B. Tailor).

Server Side Includes (SSI)

SSI ist eine etablierte Technologie, die ihren Ursprung bereits in den 90er-Jahren hat und weiterhin von den gängigen Webservern (z. B. nginx, Apache, IIS) unterstützt wird. Ob SSI oder ESI ist für die Betrachtung des Architekturmusters nicht relevant. Wir nutzen daher zur Betrachtung eine Lösung mithilfe von SSI.

Befinden sich in der vom Client angeforderten Seite weitere Include-Anweisungen, so werden diese vom Webserver oder Cache aufgelöst und die referenzierte Seite wird bereits in die Response integriert. Ein wesentlicher Nachteil ist, dass die Antwortzeit vom langsamsten Service bestimmt wird.

Das Ganze kann resilient ausgelegt werden, damit ich schnell auf einen Ausfall reagieren kann. Der Webserver oder Proxy ist hier jedoch der Single Point of Failure und sollte daher schmal und redundant gehalten werden. Ebenfalls im Proxy können auch Querschnittsfunktionen wie Autorisierung und Authentifikation stattfinden.

Auch wenn aktuell nicht mehr kritisch, sei angemerkt, dass die Integration der einzelnen Frontends auch ohne aktiviertes JavaScript im Browser gegeben ist. Dinge wie eine einheitliche UX/UI sind mit dieser Lösung jedoch nicht abgedeckt und müssen weiterhin organisatorisch berücksichtigt werden (siehe Tabelle 4).

Für Szenarien, bei denen es wichtig ist, die Zeit zu minimieren, bis der Endnutzer etwas im Browser sieht, spielt diese Variante deutliche Vorteile aus. Allerdings nur sofern alle Services hinter dem Proxy schnell antworten. Aus diesem Grund ist diese Variante insbesondere für B2C- und B2B-Businessszenarien, bei denen Antwortzeiten kritisch sind, unserer Meinung nach einer Betrachtung wert.

Integration zur Laufzeit am Client

In dieser Variante stellt jeder Microservice seine Daten mittels eines separaten Requests bereit (siehe Abbildung 4). Dies kann jedoch, je nach Verbindung, negative Auswirkungen darauf haben, wann alle UI-Komponenten beim Nutzer sichtbar werden. Die einfachste Variante der Frontend-Integration am Client erreicht man bereits über iFrames. Sind iFrames keine Option oder denkt man gleichzeitig darüber nach, durch asynchrones Nachladen und Austauschen von Informationen im DOM eine höhere UX zu erreichen, sind JavaScript und Ajax beziehungsweise Frontend-Frameworks auf dieser Technologie eine weitere Wahl. Darüber hinaus werden mittlerweile Web Components (Custom Elements) durch die immer breitere Browser-Unterstützung populärer, mit denen die einzelnen Microservices einzelne Widgets bereitstellen können. Diese können dann von anderen Microservices in der GUI eingebettet werden.

Single-Page Application (SPA)

Eine SPA ist eine Webanwendung, die aus einer einzigen HTML-Seite besteht. Alle notwendigen Ressourcen und Libraries werden beim ersten Aufruf ausgeliefert. Veränderbare Inhalte werden dynamisch nachgeladen. Diese werden auf dem Client im Browser verarbeitet und die Darstellung wird aktualisiert.

Parallele Entwicklung	in den Microservice-Teams gegeben. Anpassungen am Proxy mit Abstimmung
Unabhängige Deployments	Gut zu erfüllen, nur eine Stelle, an der die Microservices zusammengebracht werden. Nur wenige Abhängigkeiten untereinander, die sorgfältig bedient werden müssen
Unabhängige Technologiewahl	Ausreichend erfüllt, da nur Einigung auf SSI/ESI und verwendete Frontend-Frameworks, welche technisch miteinander harmonisieren müssen
Schmale Schnittstellen	gegeben, da UI und Fachlichkeit an einer Stelle im Microservice
Modelliert um Business Domains	Abstimmung notwendig, wie die Schnittstellen aussehen, wenn ich von einem Anwendungsteil zum nächsten springe
Unabhängiger Betrieb	Alle Microservices können unabhängig betrieben werden
Keine technologische Abhängigkeit	Die Teams sind weitgehend unabhängig voneinander. Nur das Zusammenspiel mit dem Proxy und ggf. Autorisierung/Authentifikation muss berücksichtigt werden
Basiert auf Standards	SSI/ESI sind als W3C-Standard definiert

Tabelle 4: Bewertung von Server Side Includes in Bezug auf die Vorteile von Microservices

Parallele Entwicklung	ist gegeben, da die jeweiligen SPA in den Teams der jeweiligen Microservices liegen
Unabhängige Deployments	sind möglich, solange es nicht um Entwicklungen an den Schnittstellen oder dem Überhang zwischen zwei Frontends geht
Unabhängige Technologiewahl	gegeben, jedoch ist die Anzahl von verschiedenen Frontend-Technologien begrenzt und sollte zwecks gemeinsamer Integration abgestimmt sein
Schmale Schnittstellen	Notwendig ist die Bereitstellung der fachlichen Kommunikation zwischen den Schnittstellen sowie eine Abstimmung, wie ein fachlicher Kontextwechsel für den Nutzer zwischen den SPA realisiert wird
Modelliert um Business Domains	erfüllt, da die Fachlichkeit innerhalb der Teams sowohl im Front- als auch im Backend übernommen wird
Unabhängiger Betrieb	erfüllt. Jeder Microservice muss für Redundanz und Resilienz selbst sorgen
Keine technologische Abhängigkeit	teilweise erfüllt. Verwendete Frontend-Frameworks müssen untereinander auf Kompatibilität abgestimmt werden. Es sollten nicht mehr als zwei Frameworks parallel zum Einsatz kommen, da sich die Ladezeit für den Nutzer sonst verlängert
Basiert auf Standards	Basiert auf im Projekt vereinbarten Kommunikationsstandards und den eingesetzten Frameworks

Tabelle 5: Bewertung des SPA-Musters in Bezug auf die Vorteile von Microservices

Dies bedeutet, dass am Anfang viel Payload übertragen wird. Danach werden über Events weitere Requests ausgelöst. Das kann zum Beispiel auf Basis von JSON-Kommunikation erfolgen. Die Antwort wird verarbeitet und die entsprechende UI angezeigt. Es lassen sich existierende APIs gut integrieren.

Mit dem Framework single-spa [Spa20] hat man die Möglichkeit, mehrere JavaScript-Frameworks parallel zu benutzen. Über eine Konfigurationsdatei als Integrationsschicht werden die unterschiedlichen SPA in eine Gesamtapplikation integriert. Das Ganze funktioniert jedoch nicht mehr, wenn JavaScript deaktiviert ist. Unserer Meinung nach mittlerweile ein vernachlässigbarer Nachteil. Zudem bestehen weitere Vorteile, wie Skalierung einzelner Teile bei hohen Benutzerzahlen, Möglichkeiten für Offlinefähigkeit und hohe UX aufgrund der Reaktionsfähigkeit (siehe Tabelle 5).

Wenn mehrere Frontend-Frameworks genutzt werden sollen und es auf Ladezeitverhalten der Gesamtanwendung nicht unbedingt ankommt sowie die Verbindung aufgrund einer höheren Anzahl von Requests gesichert ist, können Nachteile gut kompensiert werden. Diese Eigenschaften haben wir meist in Inhouse-Umgebungen gesehen, für welche wir dieses Muster eher präferieren würden. Aber auch im B2B-

Umfeld wiegen diese Nachteile je nach Art der Kunden nicht ganz so stark.

Web Components

Web Components haben das Ziel, auf einem Standard basierende und wiederverwendbare Komponenten zu bauen. Es handelt sich um wiederverwendbare HTML-Komponenten, die Browser übergreifend funktionieren. Web Components wurden vom W3C bereits 2012 als Standard beschrieben und werden mittlerweile in allen gängigen Browsern unterstützt. Web Components setzen sich im Wesentlichen aus den Teilen Custom Elements, dem Shadow DOM und HTML Templates zusammen.

Web Components haben den Vorteil, dass sie unabhängig von eingesetzten Bibliotheken und Frameworks (wie Angular oder jQuery) implementiert werden können. Die Schnittstellen zu anderen Teams bilden dann UI-Komponenten, also der Verbund von Darstellung, Verhalten und Daten und nicht nur eine API und Datenobjekte. Für die Integration werden die Bereitstellung und der Zugriff auf die Implementierung benötigt. Die Kommunikation in Richtung Backend übernimmt die Web Component selbst.

In der Architektur benötige ich eine Composition-Schicht, die eine Startseite darstellt und die einzelnen Web Components

lädt und anzeigt. Ferner wird noch ein Paketmanager benötigt, wie zum Beispiel node. Der sollte aber mittlerweile in jedem Webprojekt ein De-facto-Standard sein.

Voraussetzung ist die Nutzung von JavaScript. Aber auch das ist ein mittlerweile zu vernachlässigbarer Rahmenbedingung, die keinen Nachteil darstellt (siehe Tabelle 6).

Bezogen auf das Businessszenario lassen sich keine klaren Vor- und Nachteile bei B2B, B2C oder Inhouse ausmachen. Einzig die gestiegene Anzahl an Server-Roundtrips sollte bei B2B- und B2C-Zielszenario im Auge behalten werden.

Weitere Erfahrungen

Weitere Erfahrungen, welche wir zusätzlich zur verwendeten Variante der Frontend-Integration gesammelt haben, sind hier zusammengefasst.

Integration von Micro-Frontends in die bestehende IT-Landschaft

Wenn ein neues System entsteht, gibt es immer im Kontext bereits existierende Bestandssysteme. Entweder soll ein Ersatz für ein existierendes System geschaffen werden oder die neue Funktionalität wird benötigt.

Die Muster für die Integration von Bestandssystemen in eine Micro-Frontend-

Parallele Entwicklung	ja, keine Einschränkung
Unabhängige Deployments	ja, keine Einschränkung, eventuell kommt es zu Darstellungsproblemen, wenn Auflösungen nicht gut berücksichtigt werden
Unabhängige Technologiewahl	Es muss ein Browser vorausgesetzt werden, der den Web Components-Standard ausreichend unterstützt. Dies kann bedeuten, dass nicht alle Browser unterstützt werden
Schmale Schnittstellen	ja, UI wird Teil der Schnittstelle, eher als neutral zu betrachten und kein Nachteil
Modelliert um Business Domains	ja, Darstellung, Daten, Verhalten liegen in der Hand eines Teams
Unabhängiger Betrieb	ja, analog Parallel-Entwicklung und Deployments
Keine technologische Abhängigkeit	Zwischen den Teams ergeben sich keine wesentlichen Abhängigkeiten
Basiert auf Standards	ja, standardisiert durch W3C

Tabelle 6: Bewertung von Web Components in Bezug auf die Vorteile von Microservices

Landschaft ist oft abhängig vom Alter der zu integrierenden Anwendungen beziehungsweise der Architektur und dem Technologie-Stack. Es muss keine komplette Neuentwicklung aller Frontend-Funktionalitäten auf den Bestandssystemen durchgeführt werden. Oft genügt bereits die Bereitstellung kleinerer fachlicher Kontexte für einen Mehrwert im Gesamten.

Mögliche Szenarien für die Integration und Nutzung sind die sukzessive Ablösung und die Adaption. Die schrittweise *Ablösung* der Bestandssysteme wird gesteuert über den Business-Value, den eine Neugestaltung mit sich bringt. Das erfordert eine parallele Entwicklung und den parallelen Betrieb in beiden Systemen. Für die Ablösung wird das Bestandssystem zeitweise in das neue System integriert, zum Beispiel indem eine API-Fassade geschaffen wird. Diese Fassade versteckt das Bestandssystem und bindet zeitgleich die Services des neuen Systems ein. Dabei werden Funktionen aus dem Bestandssystem genutzt und in den neuen Microservice und die Micro-Frontend-Architektur über die Fassade integriert. Nach und nach werden immer weniger Funktionen des Bestandssystems genutzt. Für den User ist es nicht ersichtlich, ob er mit den neuen Systemen arbeitet oder mit dem Bestandssystem. Dieses Muster wird als Strangler-Pattern bezeichnet. Wer mehr zum Ablösungsszenario erfahren möchte, kann dies bei [MSS17] oder [CRI18] nachlesen.

Bei der *Adaption* geht es um die Erweiterung des Funktionsumfangs der Bestandssysteme. Das können technische Erweiterungen sein, indem zum Beispiel Zugriffe über Native Apps mobiler Endgeräte benötigt werden. Oder neue Fachlichkeit wird benötigt und diese muss mit vorhandenen Bestandssystemen interagieren. Hier bieten sich als Muster spezialisierte API-Schichten an, um generalistische und umfangreiche API-Schichten zu vermeiden. Diese spezialisierten Schichten werden als „Backend for Frontend“ bezeichnet. Sie sind auch geeignet, wenn die Bestandssysteme in die neue Micro-Frontend-Landschaft integriert werden sollen. Das Bestandssystem ist dann parallel über zwei Frontends nutzbar. Mehr zum Thema Backend for Frontend ist bei [MSB17] oder [New20] zu finden.

Bootup Template

In jeder Microservice-Architektur gibt es sich wiederholende Anforderungen, die jedes Team erfüllen muss. Dies gilt auch für Micro-Frontends. Diese sollten nicht komplett unabhängig gelöst, sondern zwischen den Teams abgestimmt werden.

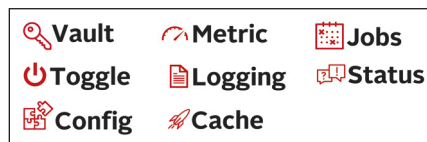


Abb. 5: Querschnittsfunktionen, für die eine abgestimmte Lösung zwischen den Microservices erarbeitet werden sollte

Unserer Erfahrung nach sind folgende Funktionalitäten im Gesamtkontext zu betrachten und zu vereinbaren (vgl. Abbildung 5):

- Ablagen und Auslesen von Secrets in einem Vault,
- Anbindung von Persistenzen,
- Funktionen, um zeitgesteuerte Jobs aufzusetzen,
- Caching von statischem Content und unveränderlicher Daten,
- fachliches und technisches Logging über Microservice-Grenzen hinweg,
- Monitoring,
- Konfiguration von Komponenten und
- Nutzbarkeit von Feature-Toggles.

Wir bevorzugen die Bereitstellung von Lösungsmustern, die einmal entwickelt werden und dann von anderen Teams genutzt werden können. Dabei ist nicht ein Team für diese Funktionalitäten verantwortlich. Die Muster sind als Vorlage zu verstehen, jedes Team kann nach Belieben den Umfang erweitern, die Muster verbessern oder Fehler beseitigen. Die anderen Teams können dann die angepassten Muster übernehmen.

Die Pflege durch ein zentrales Team muss genau überlegt und abgewogen werden. Dadurch ergibt sich eine horizontale Abhängigkeit aller Teams, die es im Sinne der Ziele von Microservices zu vermeiden gilt.

Pattern Library

Um ein gemeinsames Look & Feel herzustellen, sollte eine gemeinsame Pattern Library [PaLi] vereinbart werden. Eine zentrale Bereitstellung beschleunigt die Ergebnisse der Microservice-Teams. Es gilt, einen Modus zu finden, in dem diese Verwendung findet. Eine zentrale Kontrolle und Abnahme sind wenig hilfreich, die Teams sollten in die Lage versetzt werden, das Frontend entsprechend definierter Rahmenbedingungen mithilfe der Pattern Library selbst aufzubauen. Es kann angebracht sein, eine Gruppe an Usability- und UX-Experten aufzubauen. Dieses Team kann sich um Themen kümmern, wie User-Befragung, Vergleichsstudien, Usability-Analyse und Usability-Test, und um ganzheitliche Betrachtungen mithilfe von User Journey Maps. Aber auch diese

Skalierung ist eine Frage der Größe des Gesamtsystems.

Was ist sonst noch wichtig?

Neben dem Erreichen der Ziele, die sich aus den nicht-funktionalen Anforderungen an deren technologischen Rahmenbedingungen ableiten, ergeben sich Lösungsmuster. Diese haben sich nach unserer Erfahrung als hilfreich erwiesen und förderlich zum Erreichen der vorgenannten Ziele:

- Die Teams sollten als Cross-Functional Teams besetzt werden, um alle Tätigkeiten innerhalb des Teams zu halten. Und egal in welchem Businessszenario man unterwegs ist, der Customer Focus muss unbedingt in die Teams gebracht werden. Wenn die Teams eine Inhouse-Anwendung entwickeln, so hat sich die Business-Value-Chain als hilfreiches Modell erwiesen [DHI16].
- Es werden auch Spezialisten für bestimmte Themenfelder gebraucht, die Experten auf ihrem Themenfeld sind und anderen Teams in diesem Feld helfen können und Wissen vermitteln.
- Der Fokus sollte auf der Auslieferung liegen, dies bedeutet – Auslieferungsgeschwindigkeit geht vor Synergie-Effekten. Damit kann erreicht werden, dass jedes Team in seinem Tempo Releases ausliefert und Business-Value generieren kann. Die Teams können also sehr schnell Feedback zu ihren Features bekommen.
- Jedes Feature sollte sowohl im Fachlichen als auch im Technischen möglichst einfach gebaut werden.
- Bei Backend-Implementierungen ist es angebracht, erst sehr spät Laufzeit und Ressourcenbedarf zu optimieren. Moderne Cloud-Umgebungen skalieren sehr gut und es gibt genügend Muster und Erfahrungen, um Optimierungen durchzuführen.
- Dagegen sollte im Frontend schon von Anfang an optimiert werden. Speziell was Anzahl und Dauer der Request/Response-Zyklen angeht.
- Breaking-Changes, also Änderungen, die dazu führen, dass Anwendungen nicht mehr funktionieren, lassen sich nicht vermeiden oder vorausplanen. Je mehr technologische Abhängigkeiten zwischen den Teams existieren, umso wahrscheinlicher werden Breaking-Changes und umso aufwendiger sind sie. Abhängigkeiten zwischen Teams sind manchmal sinnvoll, sofern die Entscheidungen bewusst getroffen werden.

Literatur & Links

- [CRI18] Ch. Richardson, *Microservice Patterns*, Manning Publications, 2018
- [DHI16] Dark Horse Innovation, *Digital Innovation Playbook*, Murmann Publishers, 2016
- [GSt15] G. Steinacker, *Von Monolithen und Microservices*, in: *Informatik Aktuell*, siehe: <https://www.informatik-aktuell.de/entwicklung/methoden/von-monolithen-und-microservices.html>
- [Jac19] C. Jackson, *Micro Frontends*, siehe: <https://martinfowler.com/articles/micro-frontends.html>
- [Mod20] Z. Johnson, *Webpack 5 Module Federation*, siehe: <https://indepth.dev/webpack-5-module-federation-a-game-changer-in-javascript-architecture/>
- [MSS17] Strangler Pattern (Einschnürungsmuster), siehe: <https://docs.microsoft.com/azure/architecture/patterns/strangler>
- [MSB17] Muster „Back-Ends für Front-Ends“, siehe: <https://docs.microsoft.com/azure/architecture/patterns/backends-for-frontends>
- [New15] S. Newmann, *Building Microservices*, O'Reilly, 2015
- [New20] S. Newmann, *Monolith to Microservices*, O'Reilly, 2020
- [PaLi] Otto (GmbH & Co KG), otto.de Pattern Library, siehe: <http://patterns.otto.de>
- [Spa20] single-spa – A javascript router for front-end microservices, siehe: <https://single-spa.js.org/>
- [Spr19] S. Sprinter, *Micro Frontends bringen Microservices ins Frontend*, entwickler.de, siehe: <https://entwickler.de/leseproben/micro-frontends-monolith-579890021.html>
- [Tho16] Thoughtworks Technology Radar, siehe: <https://www.thoughtworks.com/radar/techniques/micro-frontends>

Fazit

Aufgrund seiner Einfachheit und Robustheit ist die serverseitige Implementierung auf Basis von Server Side Include sehr häufig eine gut anwendbare Lösung. Es gibt wenig Einschränkungen und Voraussetzungen, die durch alle Teams erfüllt werden müssen. Wichtig ist, darauf zu achten, die Integrationsschicht der SSI möglichst klein zu halten und nur die notwendigen Cross Cutting Concerns an einer Stelle zu implementieren. Wird die Integrationsschicht zu mächtig, besteht die Gefahr technischer Abhängigkeiten und die Gefahr für teuer Breaking-Changes steigt erheblich.

Es bleibt abzuwarten, ob Webpack5 sich etabliert und eine robuste Lösung anbietet. Das aktuelle Release sieht vielsprechend aus.

Web Components werden durch immer mehr Browser unterstützt und sind durch einen Standard spezifiziert. Insgesamt sind Web Components eine gute Lösung, wenn UI-Komponenten als Abstraktionsebene im Kontext der individuellen Anforderungen geeignet sind.

Ist die fachliche Domäne überschaubar umfangreich und nicht sehr breit, so kann dies ein Indiz sein, dass ich auf ein Micro-Frontend verzichten kann. Es könnte ausreichen, eine gute Microservice-Architektur für die Fachlichkeit zu entwerfen und beim Frontend eher eine übergreifende Frontend-Schicht einzusetzen. Es besteht in diesem Szenario die Gefahr, dass mit der Implementierung eines Micro-Frontends zu viel Overhead und nicht notwendige Komplexität aufgebaut wird.

Entwickeln sehr viele Teams am Produkt und ist die Domäne stark fragmentiert, so

kann es sehr vorteilhaft sein, eine Micro-Frontend-Lösung zu bauen.

Erst bei großem Funktionsumfang können Micro-Frontends ihre Vorteile ausspielen. Die getrennten Entwicklungsteams können jede für sich arbeiten und die Versprechen der Microservices erfüllen sich. Wichtig ist, dabei die technologische Vielfalt bei den UI-Frameworks im Auge zu behalten und zu minimieren. Es entsteht ansonsten ein nicht mehr beherrschbarer Technologiezoo.

Es ist davon auszugehen, dass Anforderungen an Frontends sowohl von fachlicher als auch von nicht-funktionaler Seite

immer umfassender werden. Daher ist eine skalierbare Architektur notwendig. Eine klare Trennung zwischen Domänen ist notwendig, um eine gute Balance zwischen Kopplung und Kohäsion zu erreichen.

Gerade weil die Prinzipien und Praktiken für Microservices etabliert und bekannt sind, ist es ratsam, diese auf die Frontends zu übertragen. Aber immer in einem Maße, wie es notwendig und angebracht ist. Und dafür ist es zwingend notwendig, seine fachlichen und nicht-funktionalen Anforderungen nicht nur zu kennen, sondern diese sehr genau zu verstehen. ||

Die Autoren



André Fleischer

(andre.fleischer@maibornwolff.de)
ist Executive IT Consultant bei MaibornWolff in Hamburg. Davor war er lange Zeit bei OTTO als Manager und bei Lufthansa Systems beschäftigt. Dort hat er gelernt, was Refactoring von Organisation und Systemen bedeutet. Sein aktueller Interessenschwerpunkt liegt auf agiler Produktentwicklung, selbstorganisierten Teams und coolen IT-Lösungen. André liebt Brettspiele, Clean Code und langlebige Softwarearchitekturen!"



Heiko Schröder

(heiko.schroeder@otto.de)
hat als Kind der 80er seine ersten Zeilen Code auf dem Amiga geschrieben. Findet seitdem an allem Gefallen, was programmierbar ist. Seit 2009 ist er in diversen Stationen auf verschiedenen Technologie-Stacks unterwegs. Seine Leidenschaft liegt darin, Softwarelösungen fachlich und technologisch mitzugestalten und für sinnvolle Architektur innerhalb und außerhalb einer Anwendung zu sorgen.