

Frontends und Clients in großen, verteilten Systemen am Fallbeispiel eines Videostreamingdienstes

In der heutigen digitalen Ära sind große, verteilte Systeme ein integraler Bestandteil der globalen IT-Infrastruktur sowie unseres alltäglichen Lebens. Diese sogenannten *Ultra Large Scale Systems* zeichnen sich durch eine komplexe Architektur mit einer Vielzahl von Komponenten aus, die miteinander interagieren müssen, um für die großen Nutzerzahlen weltweit verfügbar zu sein und diesen ein nahtloses Erlebnis bieten zu können. Neben sozialen Netzwerken, Dienstleistungsunternehmen oder E-Commerce-Anwendungen zählen hierzu auch Videostreamingplattformen wie Netflix, Twitch oder YouTube, die diese Anforderungen erfüllen müssen.

In diesem Paper wird der Schwerpunkt auf die Frontends und Clients gelegt und am Fallbeispiel des Videoportals "YouTube" die spezifischen Herausforderungen und Lösungen dieses Teilbereichs im Kontext der großen, verteilten Systeme genauer betrachtet. Der erste Abschnitt befasst sich dabei mit den verschiedenen Arten von Clients und Endgeräten und welche Anforderungen von diesen an das Gesamtsystem gestellt werden. Anschließend werden einige ausgewählte Aspekte hinsichtlich der User Experience vorgestellt und erläutert, die bei einem Videostreamingdienst entscheidend sind, um für die Endnutzer positive Erlebnisse zu schaffen und diese bestmöglich an den Service binden zu können. Im abschließenden Teil der Arbeit wird das API-Design in einer solchen Anwendung untersucht sowie verschiedene Architekturmöglichkeiten beleuchtet und diskutiert, die bestehen, um die Frontends mit dem Backend und Services zu verknüpfen und deren Zusammenspiel zu optimieren.

1. Überblick über die Clients und deren Anforderungen

User haben mittlerweile den Anspruch, Dienste und Anwendungen nicht nur auf einem einzigen Client, sondern auch geräteübergreifend nutzen zu können. Dies spielt insbesondere für Unternehmen hinter den Videostreamingservices eine Rolle da diese den Fokus nicht nur auf ein mobiles, natives Frontend und eine Webanwendung legen können, sondern aufgrund des großen Konkurrenzangebots in diesem Markt und den Erwartungen der Nutzer sicherstellen müssen, dass ihre Inhalte problemlos auf einer

Vielzahl von verschiedenen Gerätetypen, Plattformen oder Betriebssystemen verfügbar sind.

Einer der wichtigsten Clients für YouTube ist weiterhin deren Webanwendung. Bei der Entwicklung und Umsetzung müssen jedoch die Besonderheiten und Unterschiede zwischen den verschiedenen Browsern, vor allem im Hinblick auf die Unterstützung bestimmter Videoformate, beachtet werden. Der Codec H.264 bietet die breiteste Unterstützung und kann von allen gängigen Browsern wie Chrome, Edge, Safari oder Firefox im HTML5-Video player abgespielt werden. [1] Dessen Nachfolger H.265 kann zwar vor allem hochauflösende Videos deutlich effizienter komprimieren und so bei gleichbleibender Qualität die Bitrate um bis zu 50% senken, die Einschränkungen dieses modernen Codes liegen jedoch in seiner Geräteunterstützung. [2] So ist Safari aktuell der einzige Browser, der den Codec nativ abspielt. Zwar können auch aktuelle Versionen von Chrome und Edge das Format verarbeiten, der Support hängt jedoch auch von der Hardware des jeweiligen Endgerätes ab. Neben den höheren Hardwareanforderungen zum Decodieren und Streamen der Inhalte sind hier auch die teuren Patentlizenzen für die Nutzung des Codecs ein Grund für die aktuell eingeschränkte Verbreitung von H.265. [3]

Eine deutlich breitere Unterstützung erreichen dabei der lizenzgebührenfreie VP9-Codec sowie dessen ebenfalls offener Nachfolger AV1, der als Konkurrent zum H.265 und speziell für das Abspielen von Videoinhalten im Internet von der Alliance for Open Media entwickelt wurde. Sein moderner Algorithmus ermöglicht Anwendern eine hervorragende Streaming-Qualität bei gleichzeitig geringen Bandbreitenkosten. Mit zunehmender Verbreitung in den letzten Jahren wird das Format mittlerweile von den neuesten Versionen der Browser verarbeitet. Aufgrund des komplexen Dekodierungsprozesses werden jedoch auch hier entsprechende Hardwarebeschleuniger der Endgeräte vorausgesetzt, um ein flüssiges Abspielen sicherzustellen. [4]

	H.264 (MP4)	HEVC/H.265 (MP4)	VP9 (WebM)	AV1 (WebM)
Chrome	Alle Versionen	Ab V. 107 [2022], hängt von Hardware- support und OS ab	Ab Version 25 [2013]	Ab Version 70 [2018]
Edge	Alle Versionen	Ab V. 12 [2015], hängt von Hardwaresupport und OS ab	Ab Version 79 [2020]	Ab Version 121 [2024]
Safari	Ab Version 3.2 [2008]	Ab Version 13 [2019]	Ab Version 16 [2022]	Ab V. 17 [2022], jedoch nur auf Devices mit Hardware-Decoder
Firefox	Ab Version 23 [2013]	Kein Support	Ab Version 28 [2014]	Version 67 [2019]

Abbildung 1: Überblick über die Browserunterstützung ausgewählter Codecs

YouTube setzt daher nicht nur auf einen, sondern auf mehrere verschiedene Videocodecs, um eine optimale Balance zwischen Qualität, Kompression und Kompatibilität zu erreichen. Der am häufigsten verwendete Codec ist H.264, der aufgrund seiner breiten Unterstützung grundlegend für alle Videos eingesetzt wird. Für höhere Qualität, insbesondere bei 4K- oder HDR-Inhalten, nutzt YouTube VP9, während der neueste Codec AV1 für besonders populäre Videos mit sehr hohen Abrufzahlen zum Einsatz kommt. Die Plattform kodiert so die Videos in mehreren Formaten und wählt beim Streaming dynamisch, basierend auf Faktoren wie Geräteleistung, Netzwerkbedingungen und Browserunterstützung. [5]

Neben dem Web als wichtiges Frontend für viele stationäre Clients spielen vor allem mobile Endgeräte, die mit über 60% den größten Teil aller Videoaufrufe ausmachen, für YouTube eine enorm bedeutende Rolle. [6] Um den Nutzern der Plattform hier ein bestmögliches Streamingerlebnis zu bieten, stellt YouTube sowohl für iOS als auch für Android eine native Anwendung in den jeweiligen App Stores zur Verfügung. Heruntergeladen werden die Applikationen ab iOS Version 14 und Android 8.0, wodurch so 96,1% bzw. 97,2% aller potentiellen Nutzer erreicht werden. [7] [8] Auch die im vorherigen Abschnitt beschriebenen und von YouTube eingesetzten Codecs werden mit Einschränkungen im AV1-Support sowohl von Android- als auch Apple-Geräten abgespielt. Während Android im März 2024 allen Geräten ab Android 12 den "dav1d"-Decoder als Softwarelösung für das Streamen von AV1-Inhalten zur Verfügung stellt, setzt Apple noch keinen eigenen Software-Decoder ein, sodass hier

Hardwaredecoder ab dem iPhone 15 das Abspielen des modernen Kompressionsformats ermöglichen. [9] [10]

Während für YouTube der Fokus weiterhin auf dem Streaming im Browser und auf mobilen Geräten liegt und beispielsweise im Vergleich zu Netflix deutlich weniger Nutzer die Inhalte auch auf Smart TVs konsumieren, so hat Google mit wachsender Größe seines Services trotzdem den Anspruch, auch diese Clients abzudecken. Die Landschaft der Smart-TVs und Streaming-Devices ist jedoch geprägt von einer großen Herstellervielfalt mit unterschiedlichen Betriebssystemen, was die Entwicklung der Anwendungen komplex gestaltet. Während Samsung auf Tizen und LG auf WebOS setzt, beide mit JavaScript als primäre Entwicklungssprache, nutzen Sony und Philips AndroidTV mit Kotlin. [11] Applikationen für Apples TV-Box werden hingegen mit Swift implementiert. [12] Um trotz der Größe und den verfügbaren Ressourcen von YouTube den Entwicklungsaufwand der TV-Anwendungen im Verhältnis zu den Nutzerzahlen nicht zu groß werden zu lassen, ist es hier durchaus sinnvoll im Rahmen einer Anforderungsanalyse die Kernfunktionen der Plattform zu identifizieren und für Smart TVs ungeeignete Features wie einen Videoupload oder Shorts außen vor zu lassen.

In Bezug auf die Codec-Unterstützung bieten hier die meisten TV-Plattformen eine breite Kompatibilität mit H.264 und H.265, während aktuell das AV1-Format noch Einschränkungen in der Unterstützung aufweist. [9]

2. Aspekte der User Experience in Video-Streamingdiensten

Mit zunehmender Konkurrenz an digitalen Angeboten und Services, die für heutige Nutzer mit einem Klick auf einer Vielzahl an Clients abrufbar sind, ist neben dem Angebot an den Inhalten selbst eine gute User Experience entscheidend, ob und wie häufig der Service tatsächlich genutzt wird. So wird dem interdisziplinären Feld der UX in der Praxis eine immer wichtigere Rolle zugeschrieben. Neben einem durchdachten und modernen Design der Oberfläche muss YouTube hier auch immer wieder Faktoren aus Psychologie, Marketing, Technik und Soziologie berücksichtigen, um die Bedürfnisse der Nutzer zu verstehen und ein optimiertes, intuitives Streamingerlebnis über die verschiedenen Clients hinweg zu bieten. [13]

Mit über 500 Stunden an neuem Videomaterial, das jede Minute auf YouTube hochgeladen wird, kann es für den Nutzer schnell unübersichtlich und schwer werden, interessante und für ihn selbst relevante Inhalte zu entdecken. [8] Eine gute Personalisierung mit sinnvollem und angepasstem Content auf dem Feed der Startseite oder den weiteren Vorschlägen neben einem Video ist daher essenziell für eine gute User Experience. Neben einem reduzierten Informationsoverload führt so eine Darstellung der auf den Nutzer zugeschnittenen Inhalte nachweislich zu einer gesteigerten Conversion-Rate im Hinblick auf das Anklicken von Videos und einem insgesamt deutlich höheren User Engagement auf der Plattform. Nutzer fühlen sich durch eine Personalisierung dabei besser verstanden und wertgeschätzt, was häufiger positive Emotionen hervorruft und langfristig in einer engeren Bindung zur Brand bzw. dem Service resultieren kann. [14] [15]

Auch bei einer guten Personalisierung mit passenden Empfehlungen bleibt bei der großen Menge an immer neuen, nutzergenerierten Content die Herausforderung, die Videos oder Shorts in bestmöglicher Weise im User Interface der Applikation darzustellen. Im App- und Webdesign gibt es mit der sogenannten *Pagination* und dem *Infinite-Scrolling* zwei grundlegende Möglichkeiten, um Nutzern eine größere Auswahl an verschiedenen, unstrukturierten Inhalten zu präsentieren. Gerade in Anwendungen und Services mit Social-Media-Aspekten setzen mittlerweile nahezu alle Unternehmen auf einen "Infini"-Ansatz, bei dem der Nutzer, ohne ein Ende der Seite zu erreichen immer weiter nach unten scrollen kann und dynamisch neue Inhalte nachgeladen werden. [16] So ermöglicht dieses Prinzip den Nutzern, sich durch den Feed mit neuen Inhalten treiben zu lassen und ohne zusätzliche Interaktion wie bei einem Klick eines Seitenwechsels neue Videos oder Shorts zu entdecken. Gerade auf den für YouTube relevanten mobilen Endgeräten kann so eine bessere Usability erreicht werden, da das Scrollen nach unten für den Nutzer intuitiv die leichtere Bewegung darstellt. [17] Die Engagement-Rate wird gesteigert und die User verweilen dadurch durchschnittlich deutlich länger auf dem Feed oder YouTubes Shorts.

Aus Sicht des Unternehmens ist dieser Effekt erwünscht und als positiv anzusehen, es mehren sich jedoch die Stimmen, die diesen Designansatz aufgrund der negativen Effekte auf unser Gehirn als fragwürdig ansehen. Infinite Scroll nutzt dabei ein variables Belohnungssystem. Ähnlich wie beispielsweise bei Spielautomaten erhalten Nutzer unvorhersehbare Belohnungen in Form interessanter Videos und Shorts, was

sie dazu verleitet, in der Hoffnung auf die nächste Dopaminausschüttung, immer weiter zu scrollen und das Prinzip daher potenziell süchtig machen kann. [18] Der psychologische Effekt des *Unit Bias* beschreibt dabei die Tendenz unseres Gehirns, eine vorgegebene Einheit oder Portion als angemessen wahrzunehmen. Mit dem Einsatz eines unendlichen Feeds wird dieses natürliche Gefühl der Vollendung außer Kraft gesetzt. Ohne einen tatsächlichen visuellen Hinweis, wie das Erreichen des Seitenendes, fällt es dem User schwerer abzuschätzen, wie lange er sich tatsächlich schon auf der Anwendung befindet, sodass dieser tendenziell mehr Inhalte konsumiert. [19]

Um trotz des Einsatzes des Infinite-Scrollings im YouTube-Feed auch im Hinblick auf die Ladeperformance eine gute User Experience sicherzustellen, müssen die Inhalte sinnvoll beim Scrollen nach unten immer weiter nachgeladen werden. Neben dem *Lazy Loading* zunächst noch nicht sichtbarer Inhalte versucht YouTube die Nutzererfahrung dabei insbesondere für das initiale Laden der Anwendung zu verbessern. Aufgrund des Bedürfnisses unseres Gehirns, eine direkte Bestätigung oder Antwort auf eine Aktion zu erhalten, fühlt sich für uns jegliche Art von Wartezeit und Leerlauf immer verschwendet an und wirkt sich daher auch dementsprechend negativ auf die UX einer Anwendung aus. Entscheidend ist jedoch nicht die tatsächliche Zeit, die wir warten müssen, sondern wie lange sich diese Zeit für uns anfühlt. Die subjektiv wahrgenommene Wartezeit wirkt dabei für einen User immer deutlich länger, wenn dieser nach dem Öffnen der Seite einige Sekunden nur einen leeren Screen sieht. Statische Ladehinweise oder Loading Spinner bieten zwar eine erste Lösung, sind jedoch für eine User Experience noch nicht optimal. [20]

YouTube setzt daher gezielt sogenannte Skeleton-Screens ein, die als Platzhalter zunächst das grundlegende Layout der Seite repräsentieren und dann nach und nach mit den eigentlichen Inhalten ersetzt werden. Diese geben dem Nutzer sofort ein visuelles Feedback, reduzieren für ihn die Unsicherheit, was gleich passiert und bieten einen konsistenten Übergang vom ungeladenen zum vollständig geladenen Zustand. Indem die Platzhalter immer weiter ausgetauscht werden, fokussiert sich der Nutzer eher auf den Fortschritt als auf das eigentliche Warten, was die subjektiv wahrgenommene Ladezeit verkürzt. [20] [21]

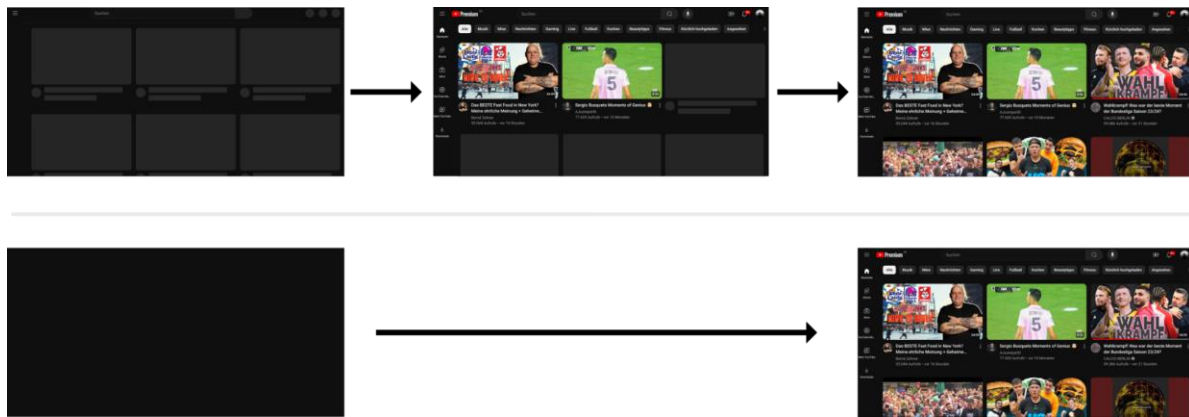


Abbildung 2: Ladeprozess mit Skeleton-Screen im Vergleich

Auf der Startseite nutzt YouTube mit dem *Zeigarnik-Effekt* einen weiteren psychologischen Mechanismus, um das Nutzerengagement zu steigern und die Verweildauer auf der Plattform zu erhöhen. Dieser Effekt basiert auf der Tendenz des menschlichen Gehirns, sich an unterbrochene oder unvollendete Aufgaben besser zu erinnern als abgeschlossene, was YouTube geschickt in sein Interface-Design integriert. So werden im Feed mit den Recommendations nicht nur neue, sondern auch angefangene, aber noch nicht fertig gesehene Videos vorgeschlagen. Fortschrittsbalken signalisieren dabei visuell den unvollendeten Status eines Inhalts und regen den User zum Weiterschauen an. [22]

3. API-Pattern in verteilten Microservice-Architekturen

Während die Nutzer im Gegensatz zum User Experience- und Interface Design von der API-Architektur nicht direkt betroffen sind, so ist die darunterliegende Schnittstelle zur Verknüpfung der Frontends mit den Services der Anwendung im Backend dennoch ein weiterer wichtiger Aspekt hinsichtlich der Entwicklung und bestmöglichen Unterstützung der im ersten Abschnitt vorgestellten Clients. In kleineren, monolithischen Systemen sind APIs leicht zu implementieren und zu verwalten. Verteilte, komplexe Microservice-Anwendungen wie YouTube erfordern hingegen spezifische und gut durchdachte API-Patterns, um Herausforderungen wie Skalierbarkeit und Performance zu bewältigen sowie die unterschiedlichen Anforderungen der verschiedenen Endgeräte zu erfüllen.

3.1 Direkte Client-to-Service Kommunikation und API-Gateways

Ein erster naheliegender Ansatz die entkoppelten und verteilten Services auf der Backendseite von YouTube mit den einzelnen Frontends zu verknüpfen ist die direkte Client-to-Microservice-Kommunikation, bei der Clients direkt mit den einzelnen Microservices interagieren. Dies ist zwar eine Möglichkeit, die in kleineren Microservice-Strukturen eine durchaus valide Lösung bieten kann, gerade in Ultra Large Scale Systems mit einer Vielzahl an Microservices birgt die direkte Interaktion jedoch einige Herausforderungen und Nachteile. Da jeder Microservice seine eigene Schnittstelle hat, führt dies dazu, dass Clients zahlreiche Endpunkte kennen und korrekt aufrufen müssen. Diese fehlende Abstraktion erschwert die Entwicklung und Wartung der Frontends erheblich. Zudem benötigen einzelne UI-Views meistens Daten von verschiedenen Microservices, was zu mehreren Netzwerk-Roundtrips zwischen Client und Server führt. Dies kann besonders kritisch auf mobilen Endgeräten mit möglicherweise eingeschränkter Internetverbindung sein, wo die vielen API-Aufrufe die Performance der Anwendung deutlich beeinträchtigen können. Da außerdem die einzelnen Endpunkte direkt für die Clients öffentlich zugänglich sein müssen, bietet dieses Pattern eine größere Angriffsfläche für potenzielle Sicherheitsrisiken. [23]

Um die Probleme der direkten Kommunikation zu lösen, kann als Ansatz ein API-Gateway implementiert werden. Dies stellt einen einheitlichen, zentralen Endpunkt für alle Clients bereit, wodurch die Frontends von den Services im Backend entkoppelt werden. Das Gateway fungiert dabei als *Reverse-Proxy*, der alle Anfragen entgegennimmt, diese an die entsprechenden Services weiterleitet und deren Antworten für den Client aggregiert. Die verschiedenen Frontends müssen so nur einen einzigen Endpunkt kennen und aufrufen, wodurch diese nicht mehr von den Eigenheiten oder Änderungen der einzelnen Services abhängig sind und die Implementierung vereinfacht wird. So können zudem Aspekte hinsichtlich Sicherheit, Authentifizierung, Load-Balancing oder Caching zentral im Gateway verwaltet werden. Durch die reduzierte Anzahl an API-Calls der Clients kann auch die Performance insbesondere bei einer schlechten Netzwerkverbindung optimiert werden. [23]

Trotzdem bestehen bei der Umsetzung einer Schnittstelle mit einem einzigen Gateway Herausforderungen, die beachtet werden müssen. So muss die Schnittstelle gut

skalierbar und zuverlässig verfügbar sein, damit der Endpunkt nicht zu einem Bottleneck der Gesamtapplikation wird und mit diesem Pattern ein Single Point of Failure geschaffen wird. [23] Da gerade bei YouTube die Inhalte von einer Vielzahl an verschiedenen Endgeräten abgerufen werden, kann die einheitliche, zentrale API nicht auf die teilweise unterschiedlichen Bedürfnisse der Clients zugeschnitten werden. Dies resultiert typischerweise in *Overfetching* und *Underfetching*, bei denen die Clients entweder zu viele oder zu wenige der benötigten Daten erhalten und diese dann erst herausfiltern bzw. erneute Anfragen stellen müssen. [24]

3.2 BFFs und GraphQL als Lösung für Anwendungen mit verschiedenen Clients

Ein möglicher Lösungsansatz Over- und Underfetching zu vermeiden und die Schnittstelle besser an die Bedürfnisse der unterschiedlichen Clients anpassen zu können ist das Backend-for-Frontend-Pattern, kurz BFF. Das Grundprinzip ähnelt dabei der gerade beschriebenen Gateway-Architektur, mit dem Unterschied, dass die Client-Requests nicht über eine einheitliche Schnittstelle verarbeitet werden, sondern für jedes Frontend eine eigene Backend-API implementiert wird. Jedes BFF kann die Daten genauso aufbereiten und filtern, wie es für den jeweiligen Client-Typ optimal ist. [25] Außer den effizienten API-Requests und der Möglichkeit, auch client-spezifische Businesslogik beispielsweise hinsichtlich Caching oder Sicherheit einfacher umzusetzen, bietet der Ansatz auch Vorteile hinsichtlich der Organisation und Arbeitsweise der Entwicklerteams. Da typischerweise die Teams des jeweiligen Frontends auch die zugehörigen BFF-Gateways umsetzen und warten, sind diese so unabhängiger voneinander und können in ihrer eigenen Entwicklungsgeschwindigkeit arbeiten, wodurch weniger Development-Bottlenecks entstehen. Zudem sind die Teams auch flexibel in der Technologiewahl des Gateways und können die spezifischen Schnittstellen in der für den Client oder die Entwickler am besten geeigneten Programmiersprache implementieren. [26]

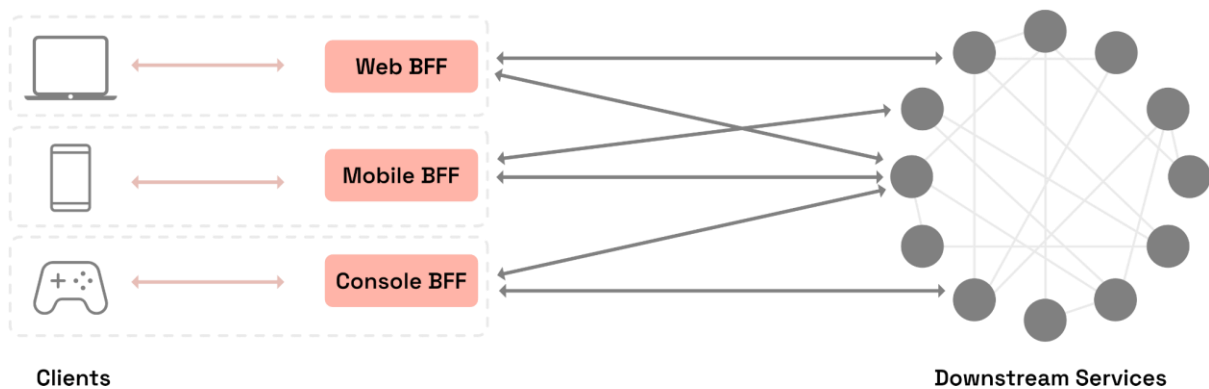


Abbildung 3: Verknüpfung der Clients mit den Microservices über ein BFF-Pattern

Neben dem beliebten Backend-for-Frontend-Ansatz, der so von einigen Unternehmen wie Soundcloud oder Uber umgesetzt wird, kann die Herausforderung der optimierten Datenbereitstellung für viele unterschiedliche Clients nicht nur auf architektonischer, sondern mit Hilfe der Datenabfragesprache GraphQL auch auf technischer Ebene gelöst werden. [27] Die ursprünglich von Facebook entwickelte Open-Source-Technologie ist dabei grundlegend Sprachen-, Plattform-, und sogar Protokollunabhängig. Ähnlich wie bei einer REST-API werden für den Datenaustausch zwischen Client und Server meistens HTTP-GET oder POST-Requests genutzt. Während bei REST jedoch die einzelnen Ressourcen unter den verschiedenen URLs der Endpunkte liegen und auf eine Anfrage immer der komplette Datensatz übertragen wird, bestimmt bei einem GraphQL-Request der Client welche Informationen er genau benötigt. Over- und Underfetching wird so auch bei einem einzigen Gateway für alle Client-Typen vermieden. Darüber hinaus bietet GraphQL eine Typsicherheit, gute Selbstdokumentation und ist deutlich flexibler erweiterbar als eine REST-API. [28]

Die Grundlage einer solchen Schnittstelle bildet dabei immer das *GraphQL-Schema*, über das definiert wird, welche verfügbaren Datentypen und Beziehungen zwischen diesen vorhanden sind und über welche *Queries* die Daten vom Client abgefragt werden können. Um die Anfragen dann entsprechend zu verarbeiten, aufzulösen und die benötigten Informationen bereitzustellen, werden in GraphQL sogenannte *Resolver* benötigt. Diese Funktionen bilden somit eine Brücke zwischen den GraphQL-Typen im Schema und den zugrunde liegenden Datenquellen, wie Datenbanken, externen REST-APIs oder anderen Speichermöglichkeiten. [29] In dem vereinfachten Beispiel (siehe Abb. 4) wird über den Query "GetVideos" angegeben, welche genauen Daten der Client vom Typ Video benötigt. Der Resolver stellt nun

exakt die Daten bereit, die das Frontend benötigt, wodurch Over- und Underfetching vermieden werden kann.

Schema Definition	Client Query	Data & Resolver
<pre># Definiert den Videotypen mit verschiedenen Feldern type Video { id: ID! title: String! views: Int! creator: User! }</pre>	<pre>#Abfrage der id und der views aller Videos query GetVideos { videos { id views } }</pre>	<pre>const videos = [{ id: "1", title: "GraphQL Basics", views: 300, creator: "John Doe" }, { id: "2", title: "Advanced GraphQL", views: 600, creator: "Jane Smith" }, // Weitere Videos...]; const resolvers = { Query: { // Resolver für die videos Query videos: () => { return videos; } } };</pre>

Abbildung 4: Beispiel einer GraphQL-Anfrage

3.3 Federated GraphQL-APIs in skalierten Applikationen

Die zuvor beschriebenen API-Ansätze wie das BFF-Pattern oder ein GraphQL-Gateway sind beides gute Lösungen für Applikationen, die viele verschiedene Clients mit teilweise unterschiedlichen Anforderungen unterstützen müssen, jedoch können diese in weltweit verfügbaren Ultra Large Scale Anwendungen an ihre Grenzen geraten. Gerade in Streamingdiensten, die für die Nutzer auf sehr vielen verschiedenen Endgeräten zur Verfügung stehen sollen, können sich die einzelnen Backend-for-Frontends schnell vervielfachen. So beschreibt auch RTL, dass der Aufwand für Entwicklung und Wartung der BFFs des zugehörigen Services RTL+ immer größer wurde und für neue Features immer wieder Code-Duplikate entstehen. [30] Auch die GraphQL-API von Netflix geriet mit zunehmender Skalierung an ihre Grenzen, da es für das Schnittstellen-Team mit zunehmender Größe und Komplexität des Graphens immer schwerer wurde, diesen allein zu verwalten, sodass an dieser Stelle immer wieder Development- Bottlenecks entstanden. [31]

Mittlerweile setzen beide Unternehmen auf eine neue Lösung als Schnittstelle zwischen den verschiedenen Frontends und der Microservice-Architektur im Backend - eine Federated GraphQL-API. In diesem Architekturmuster wird das gesamte GraphQL-Schema in unabhängige Teilgraphen (*Subgraphs*) aufgeteilt, die dann als eigene Services von verschiedenen Teams entwickelt und gewartet werden können. Die einzelnen Subgraphen werden schließlich wieder über ein Gateway zu einem einzigen Endpunkt zusammengefasst. Netflix zerlegt seinen Gesamtgraphen dabei

nach übergeordneten Themenbereichen (*Domains*), die über einen zugehörigen Domain Graph Service (DGS), jeweils von einem separaten Team verwaltet, einen Teil der gesamten API abdecken. Als eine zentrale Komponente ist dann das *Schema Registry* dafür verantwortlich, die jeweiligen Subgraphen der DGS zusammenzuführen, zu validieren und daraus ein Gesamtschema, den sogenannten *Supergraph*, zu erstellen. Dieser wird dann für ein zentrales GraphQL-Gateway bereitgestellt, das weiterhin als einheitlicher Einstiegspunkt für alle Clients dient. [31]

Das Gateway empfängt die eingehenden Client-Requests, plant die Ausführung der Teilabfragen und leitet diese dann an die entsprechenden DGS weiter. Dieses *Query Planning* sorgt dafür, dass die entsprechenden Teilabfragen dann je nach Abhängigkeiten der Teilergebnisse parallel oder sequentiell an die relevanten

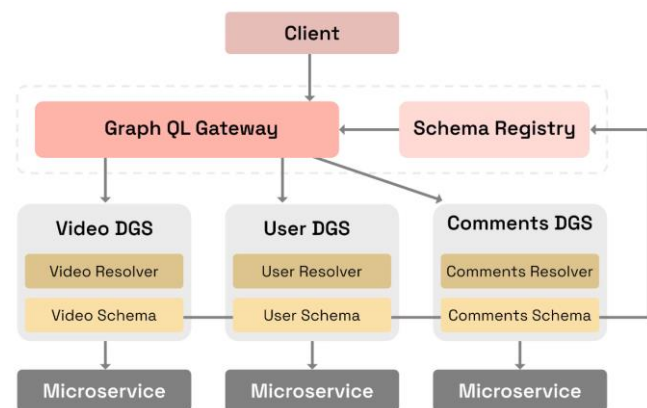


Abbildung 5: Aufbau einer prototypischen Federated GraphQL-API

DGS gesendet werden. Jeder Service führt nun seinen Teil des Querys aus und gibt die passenden Daten an das zentrale Gateway zurück, wo schließlich die Ergebnisse aggregiert und zurück zum Client gesendet werden. Die Aufteilung und Verteilung der Abfragen ermöglicht eine effiziente Nutzung der Backend-Ressourcen und verbessert die Performance und Skalierbarkeit der gesamten Schnittstelle. Da auch nicht mehr nur ein einziges Team den sehr komplexen Gesamtgraphen verwalten muss, kann durch die Aufteilung nach Domänen die Entwicklung der API dezentralisiert werden und die jeweiligen Teams ihre Dienste in schnelleren Zyklen unabhängig voneinander entwickeln und bereitstellen. [31] [32]

4. Fazit und Ausblick

Die Betrachtung von Frontends und Clients im Kontext großer, verteilter Systeme am Use Case eines Videostreamingdienstes zeigt die Komplexität und Vielfalt der Herausforderungen in diesem Bereich. So müssen bei der Planung und Umsetzung der Frontends eine Vielzahl an Aspekten berücksichtigt werden, um zum einen den Usern eine bestmögliche Applikation bereitzustellen, auf der anderen Seite jedoch die

Anforderungen von anderen Teilen und Diensten des Gesamtsystems wie der Videocodierung oder dem Backend zu erfüllen.

Für einen Streamingdienst spielt auch die Wahl geeigneter Videocodecs eine entscheidende Rolle, um eine optimale Balance zwischen Qualität, Kompression und Kompatibilität zu erreichen. YouTubes Strategie, mehrere Codecs wie H.264, VP9 und AV1 einzusetzen, ermöglicht es, ein breites Spektrum von Geräten zu bedienen und gleichzeitig die neuesten technologischen Fortschritte zu nutzen. Im Bereich der User Experience wird deutlich, dass nicht nur ein modernes und visuell ansprechendes Design für die Nutzer entscheidend ist, sondern diese durch eine gute Personalisierung und intelligente Darstellung der Inhalte unter Berücksichtigung psychologischer Prinzipien die Anwendung noch häufiger nutzen und die Bindung zum Service so verbessert werden kann.

Hinsichtlich der Verknüpfung verschiedener Clients mit einer Microservice- Architektur bestehen verschiedene API-Pattern, deren sinnvoller Einsatz jedoch auch immer vom einzelnen Anwendungsfall abhängt. Der vorgestellte Ansatz der Federated-GraphQL-Schnittstelle ist sicherlich keine Lösung, die sich für alle Arten von Anwendungen eignet, sondern hängt dabei auch immer davon ab, wie die Entwicklungsprozesse im Unternehmen aussehen und wie die einzelnen Teams organisiert werden sollen. Für eine erfolgreiche Umsetzung und Umstellung dieses API-Prinzips wie bei Netflix, RTL oder PayPal sind dabei auch Kompetenzen aus Bereichen des Testings, Loggings oder Monitorings von großer Bedeutung.

Die Entwicklungen im Bereich der Frontends und Clients sind weiterhin von hoher Dynamik geprägt und nie ganz abgeschlossen. Unternehmen hinter den Ultra Large Scale Systems stehen vor der Herausforderung, ihre Systeme kontinuierlich zu verbessern, anzupassen und so den wachsenden und sich wandelnden Anforderungen der Nutzer gerecht zu werden. Um trotz der großen Konkurrenz wettbewerbsfähig zu bleiben, wird es hier auch entscheidend sein neue, aufkommende Endgeräte sinnvoll zu unterstützen und technologische Innovationen wie effizientere Videocodes für die eigene Applikation zu nutzen.

Literaturverzeichnis

- [1] Can I Use... „MPEG-4/H.264 video format“, Can I Use, 2024. [Online]. Verfügbar: <https://caniuse.com/?search=h.264>. [Zugriff am: 26. Juni 2024].
- [2] MovPilot, "H.264 vs. H.265: Was ist der Unterschied und was ist besser?", MovPilot, 2024. [Online]. Verfügbar: <https://movpilot.com/de/blog/h264-vs-h265/>. [Zugriff am: 29. Juli 2024].
- [3] Can I Use... „HEVC/H.265 video format“, Can I Use, 2024. [Online]. Verfügbar: <https://caniuse.com/?search=h.265>. [Zugriff am: 26. Juni 2024].
- [4] Ashik, T.S. "AV1 vs HEVC: Which is the better codec?", Gumlet, 2024. [Online]. Verfügbar: <https://www.gumlet.com/learn/av1-vs-hevc/>. [Zugriff am: 27. Juni 2024].
- [5] Ozer, J. "Which codecs does YouTube use?", Streaming Learning Center, 2021. [Online]. Verfügbar: <https://streaminglearningcenter.com/codecs/which-codecs-does-youtube-use.html>. [Zugriff am: 27. Juni 2024].
- [6] Mohsin, M. "YouTube-Statistiken: 10 spannende Fakten für das Jahr 2023", Oberlo, 2023. [Online]. Verfügbar: <https://www.oberlo.com/de/blog/youtube-statistiken>. [Zugriff am: 27. Juni 2024].
- [7] Google. "YouTube", Google Play, 2024. [Online]. Verfügbar: <https://play.google.com/store/apps/details?id=com.google.android.youtube&hl=de>. [Zugriff am: 27. Juni 2024].
- [8] Apple. "YouTube", App Store, 2024. [Online]. Verfügbar: <https://apps.apple.com/de/app/youtube/id544007664>. [Zugriff am: 27. Juni 2024].
- [9] Francis, A. "The State of AV1 Playback Support: 2024", Bitmovin, 2024. [Online]. Verfügbar: <https://bitmovin.com/av1-playback-support>. [Zugriff am: 29. Juni 2024].
- [10] Francis, A. "Everything you need to know about Apple AV1 Support", Bitmovin, 2024. [Online]. Verfügbar: <https://bitmovin.com/av1-playback-support>. [Zugriff am: 29. Juni 2024].
- [11] Wittowski, P. "Fernseher-Betriebssysteme: Tizen, Android TV und Co. - das sind die Unterschiede", Netzwelt, 2023. [Online]. Verfügbar: <https://www.netzwelt.de/fernseher-smart-tvs/kaufberatung-fernseher-betriebssysteme-tizen-android-tv-co-unterschiede.html>. [Zugriff am: 29. Juni 2024].
- [12] Apple, "Swift", Apple, 2024. [Online]. Verfügbar: <https://www.apple.com/lae/swift/>. [Zugriff am: 04. Juli 2024].
- [13] Schmitz, J. "UX bei Netflix: Erforschung nutzergenerierter Inhalte auf Netflix", Justin Schmitz Blog, 2023. [Online]. Verfügbar: <https://justinschmitz.de/blog/ux-netflix/>. [Zugriff am: 05. Juli 2024].

- [14] Everything UX, "The Power of Personalization in UX: Unveiling Netflix's Success Story", Bootcamp UX Design, 2023. [Online]. Verfügbar: <https://bootcamp.uxdesign.cc/the-power-of-personalization-in-ux-unveiling-netflixs-success-story-55a693dacbf6>. [Zugriff am: 05. Juli 2024].
- [15] Amina, S., "Netflix und Personalisierung: Wie Personalisierung die Streaming-Industrie revolutioniert", Trbo, 2024. [Online]. Verfügbar: <https://www.trbo.com/de/blog/netflix-personalization-how-customization-revolutionized-the-streaming-industry/>. [Zugriff am: 07. Juli 2024].
- [16] Bennett, J. "Infinite Scrolling vs. Pagination In Web Design: Which to Use for Improving UX," Medium, 2023. [Online]. Verfügbar: <https://medium.com/@jessicajournal/infinite-scrolling-vs-pagination-in-web-design-which-to-use-for-improving-ux-bd3700334af5>. [Zugriff am: 09. Juli 2024].
- [17] Babich, N. "UX: Infinite Scrolling vs. Pagination," UX Planet, 2016. [Online]. Verfügbar: <https://uxplanet.org/ux-infinite-scrolling-vs-pagination-1030d29376f1>. [Zugriff am: 09. Juli 2024].
- [18] Hilotin, J. "Deadly scroll without end: How infinite scroll hacks your brain and why it is bad for you," Gulf News, 2023. [Online]. Verfügbar: <https://gulfnews.com/special-reports/deadly-scroll-without-end-how-infinite-scroll-hacks-your-brain-and-why-it-is-bad-for-you-1.1676965239566>. [Zugriff am: 09. Juli 2024].
- [19] Vos, K. "Is the infinite scroll a dark pattern and how does it impact our media consumption?," Medium, 2022. [Online]. Verfügbar: <https://medium.com/@koencreations/is-the-infinite-scroll-tis-a-dark-pattern-and-how-does-it-impact-our-media-consumption-2e3b1002a1a0>. [Zugriff am: 09. Juli 2024].
- [20] Chen, E. "The Psychology of Waiting: Skeletons," Medium, 2023. [Online]. Verfügbar: <https://medium.com/@elenech/the-psychology-of-waiting-skeletons-ca3b309e12a2>. [Zugriff am: 09. Juli 2024].
- [21] Jaye, H. "What's a skeleton screen?," UX Design Institute, 2022. [Online]. Verfügbar: <https://www.uxdesigninstitute.com/blog/whats-a-skeleton-screen/>. [Zugriff am: 10. Juli 2024].
- [22] Khan, B. A. "Laws of UX that Netflix follows," Bootcamp UX Design, 2023. [Online]. Verfügbar: <https://bootcamp.uxdesign.cc/laws-of-ux-that-netflix-follows-6f4819cf9a58>. [Zugriff am: 10. Juli 2024].
- [23] Montagno, J., Buck, A. "Direct client-to-microservice communication versus the API Gateway pattern," Microsoft Learn, 2023. [Online]. Verfügbar: <https://learn.microsoft.com/de-de/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern>. [Zugriff am: 08. Juli 2024].

- [24] Okpala, N. "API underfetching, and overfetching mitigation" Medium, 2024. [Online]. Verfügbar: <https://medium.com/@koodu-platform/introduction-d75372443b50>. [Zugriff am: 08. Juli 2024].
- [25] Siddiqui, M. "Backend for Frontend Pattern: Why You Need to Know It," MobilePeople, 2021. [Online]. Verfügbar: <https://medium.com/mobilepeople/backend-for-frontend-pattern-why-you-need-to-know-it-46f94ce420b0>. [Zugriff am: 11. Juli 2024].
- [26] Fränkel, N. "Discussing Backend For Front-end - Nicolas Fränkel - NDC London 2023" YouTube, 2023. [Online]. Verfügbar: https://www.youtube.com/watch?v=oKLVzK_pqgg. [Zugriff am: 14. Juli 2024].
- [27] Andani, P. "Understanding Backend for Frontend (BFF) Architecture: A Comprehensive Guide with Examples," Medium, 2024. [Online]. Verfügbar: <https://medium.com/@prashant.andani/unlocking-frontend-magic-with-bff-enhancing-user-experiences-92898451ac56>. [Zugriff am: 14. Juli 2024].
- [28] the native web. " GraphQL versus REST" YouTube, 2020. [Online]. Verfügbar: https://www.youtube.com/watch?v=oKLVzK_pqgg. [Zugriff am: 14. Juli 2024].
- [29] Nsong, F. "Complete Introduction to GraphQL," Medium, 2023. [Online]. Verfügbar: <https://medium.com/@fitzgeraldnsong/complete-introduction-to-graphql-d38a24210756>. [Zugriff am: 14. Juli 2024].
- [30] Jumpertz, O., Steinert, P. "Federated GraphQL in Microservice-Umgebungen," Informatik Aktuell, 2022. [Online]. Verfügbar: <https://www.informatik-aktuell.de/entwicklung/methoden/federated-graphql-in-microservice-umgebungen.html>. [Zugriff am: 18. Juli 2024].
- [31] Netflix Technology Blog, "How Netflix Scales its API with GraphQL Federation (Part 1)," Netflix Technology Blog, 2020. [Online]. Verfügbar: <https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2>. [Zugriff am: 21. Juli 2024].
- [32] Bohutskyi, S. "GraphQL Federation," Medium, 2024. [Online]. Verfügbar: <https://bohutskyi.com/graphql-federation-9a6183aa1956>. [Zugriff am: 23. Juli 2024].