Die Potenziale von GraphQL als Alternative zu REST

Isabell Waas
Fakultät Informatik
Hochschule für angewandte Wissenschaften Hof
Hof, Deutschland
isabell.waas@hof-university.de

***Zusammenfassung*—In der vorliegenden Arbeit soll untersucht werden, ob das noch sehr neue GraphQL eine gleichwertige Alternative zu dem etablierten REST Architekturstil darstellt. Hierfür werden die beiden Technologien zunächst separat vorgestellt. Im anschließenden Vergleich wird auf verschiedene Kriterien eingegangen, die bei der Implementierung von APIs eine Rolle spielen. Darunter sind die Schnittstellendefinition, Client-Server-Kommunikation, Abfrage- und Antwortmöglichkeiten, Skalierbarkeit, Versionierung, Performance und Caching, der Entwicklungsaufwand auf Client- und Serverseite sowie das Ökosystem. Letztendlich kann festgestellt werden, dass es für beide Technologien geeignete Anwendungsfälle gibt, da beide Stärken und Schwächen aufweisen. GraphQL kann somit durchaus als Alternative zu REST gesehen werden.**

***Index Terms*—REST, RESTful Webservice, GraphQL, Schema, Webservice, API, Schnittstelle, Interface, Client-Server-Anwendung**

I. Ziel der Arbeit

Mit der Digitalisierung haben sich auch Webanwendungen in den Alltag zahlreicher Menschen integriert, darunter beispielsweise Online-Shops, Wetter-Apps oder Soziale Medien. Die meisten dieser Applikationen kommunizieren im Hintergrund mit weiteren Diensten, die als Webservices bezeichnet werden und entweder von denselben Entwicklern oder von Drittanbietern über das Internet bereitgestellt werden. Ein Webservice kapselt bestehende Funktionalität oder Daten und ermöglicht es, auf diese über ein Application Programming Interface (API) zuzugreifen. Hierzu kann eine Webanwendung, der sogenannte Client, eine Anfrage senden, woraufhin der Webservice, der auch als Server bezeichnet wird, diese verarbeitet und beantwortet. [1]

Die Verwendung von Webservices bringt mehrere Vorteile mit sich. So können die über die Schnittstelle zur Verfügung gestellten Daten und Funktionen von mehreren Clients zugleich genutzt und somit wiederverwendet werden. Ebenso erfolgt die Kommunikation plattformunabhängig, sodass die Technologien auf Client- und Server-Seite unterschiedlich sein können. [1]

Über die Jahre haben sich einige Technologien für die Implementierung solcher Schnittstellen im Web etabliert. Das Unternehmen Postman, dessen gleichnamige Software ein bekanntes Tool zum Erstellen und Testen von APIs ist, veröffentlicht jährlich seinen „State of the API“ Bericht. In diesem wurden im letzten Jahr mehr als 40.000 Entwickler und API-Fachleute aus unterschiedlich großen Unternehmen und verschiedenen Branchen zu Themen rund um APIs befragt. Die folgende Grafik wurde der „State of the API“ Studie von 2023 entnommen und zeigt die beliebtesten Technologien bei der Schnittstellen-Entwicklung. [2]

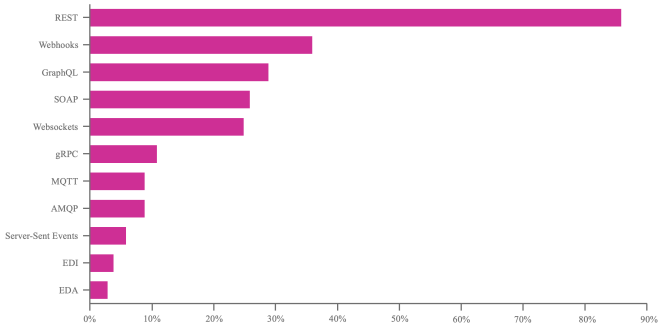

Abbildung 1. Die beliebtesten API Architekturen im Jahr 2023 [3]

Postmans „State of the API“ Berichte zeigen bereits seit 2020, dass Representational State Transfer (REST) am populärsten ist. Auch wenn der Anteil an REST-Nutzern jährlich ein wenig gesunken ist, liegt der Architekturstil noch immer mit großem Abstand vorne, wenn es um die Erstellung von Webservices geht. Das etwas ältere Protokoll SOAP ist eine der verbreitetsten Alternativen zu REST. Obwohl beide ihre Vor- und Nachteile haben, wurde SOAP durch seinen leichtgewichtigeren Konkurrenten mehr und mehr verdrängt und ist inzwischen primär in Altsystemen zu finden. In den Umfragen von Postman hielt sich SOAP dennoch bis einschließlich 2022 auf dem dritten Platz, bis es 2023 von einer nach und nach aufstrebenden, innovativen Technologie überholt wurde: GraphQL. Es existiert seit ungefähr 10 Jahren und ist damit weniger als halb so alt wie REST und SOAP. Dass GraphQL mittlerweile dennoch beliebter als SOAP ist, beweist, dass es sich zu einem ernstzunehmenden Wettbewerber entwickelt hat. Mit einem Blick auf Abbildung 1 fällt auf, dass die zweitplatzierten Webhooks bisher unerwähnt blieben. Das ist dadurch begründet, dass die HTTP-Nachrichten zur ereignisgesteuerten Kommunikation zwar oft in Verbindung mit Webservices genutzt werden, allerdings nicht der Erstellung solcher Schnittstellen dienen. [3]–[9]

In der vorliegenden Arbeit sollen nun die beiden aktuell populärsten der vorgestellten Technologien zur Implementierung von APIs betrachtet werden: REST und GraphQL. Während sie zunächst einzeln vorgestellt werden, sollen sie danach hinsichtlich verschiedener Kriterien miteinander verglichen werden. Im vierten Kapitel werden für jede der Technologien jeweils drei Anwendungsfälle erläutert. Abschließend soll ein Fazit darüber gezogen werden, ob GraphQL eine gleichwertige Alternative zu REST darstellt.

II. Übersicht über die betrachteten API-Architekturen

A. REST

1) Geschichte: Bei REST handelt es sich um ein Paradigma, das sich im Allgemeinen auf die Architektur verteilter Systeme bezieht, jedoch insbesondere bei der Implementierung von Webservices Anwendung findet. Bevor REST im Jahr 2000 im Kontext der Dissertation des Informatikers Roy Fielding veröffentlicht wurde, kam bei der Erstellung von Schnittstellen im Web größtenteils das schwergewichtigere SOAP zum Einsatz. Dagegen ermöglicht REST eine wesentlich einfachere Kommunikation zwischen Webanwendungen basierend auf mehreren Prinzipien, die im nachfolgenden Abschnitt erklärt werden. Heutzutage gilt REST für die meisten Entwickler als Standard, wenn es um die API-Implementierung geht, und es gibt unzählige Frameworks für sämtliche bekannte Programmiersprachen. Über das Internet stellen zahlreiche namhafte Unternehmen Daten und Funktionalität über sogenannte RESTful Webservices zur Verfügung, darunter Google, PayPal und X. [11]–[15]

2) REST Prinzipien und Charakteristiken von RESTful APIs: Im Mittelpunkt eines RESTful Webservices stehen Ressourcen. Das sind gemäß Fielding Informationen, die benannt werden können und auf die ein Client über eine API im Web zugreifen kann. Stellt ein Client eine Anfrage, so übermittelt (englisch: transfer) der Webservice den derzeitigen Zustand (englisch: state) der jeweiligen Ressource in der gewünschten Darstellung (englisch: representation) an den Client. Damit erklärt sich auch der Name Representational State Transfer, der mit REST abgekürzt wird. [11], [19]
Der REST-Architekturstil schreibt grundsätzlich kein bestimmtes Protokoll vor, jedoch wird er primär mit dem Hypertext Transfer Protocol (HTTP) beziehungsweise dem Hypertext Transfer Protocol Secure (HTTPS) umgesetzt. Zudem basiert REST auf verschiedenen Prinzipien, wobei die folgenden fünf besonders ausschlaggebend sind.

- Identifizierbarkeit: Zunächst muss jede Ressource mithilfe eines eindeutigen Identifiers adressiert werden können. Hierfür wählt der Entwickler einen geeigneten Uniform Resource Identifier (URI). Bei der Ressource Author könnte dieser zum Beispiel https://api/v1/authors/ sein. [11], [19]
- Repräsentationen: Dass Ressourcen in verschiedenen Darstellungen vorliegen können, steckt bereits in dem Wort Representational, für das die ersten beiden Buchstaben von REST stehen. Bei diesen Repräsentationen handelt es sich um Formate wie Extensible Markup Language (XML) oder JavaScript Object Notation (JSON). Bei der Kommunikation zwischen einem Client und einem Webservice gibt bei jeder Anfrage Ersterer an, welche Repräsentationen einer Information er bevorzugt. Dies geschieht über Multipurpose Internet Mail Extensions (MIME) Typen, z.B. application/json, die in dem Accept-Header des HTTP-Requests angegeben werden. Die anschließende Antwort des Webservices enthält die Ressource in einem Format, das er selbst unterstützt und die Präferenzen des Clients bestmöglich berücksichtigt. Der Client kann dem Header Content-Type der HTTP-Response entnehmen, welcher MIME-Typ für die angefragte Information genutzt wird. Der beschriebene Vorgang der Einigung auf ein Format der übermittelten Ressource heißt Content Negotiation und ist Teil des HTTP Standards. [11], [22]
- Selbstbeschreibung: Des Weiteren soll eine REST-API selbsterklärend sein. Dies wird vor allem durch die standardisierten HTTP-Methoden erreicht, die im Kontext von REST jeweils für bestimmte, auf Ressourcen ausführbare Aktionen stehen. Bedeutend sind hauptsächlich POST, GET, PUT, PATCH und DELETE, da sich mit ihnen die sogenannten CRUD (Create, Read, Update, Delete) Operationen umsetzen lassen. Dazu werden Endpunkte bereitgestellt, die Aktionen auf eine Ressource oder eine Liste an Ressourcen beinhalten und über eine definierte URI in Verbindung mit dem passenden HTTP-Verb aufrufbar sind. Bei den Antworten des Webservices wird sich zudem noch eines anderen standardisierten Elements von HTTP bedient: der Statuscodes. Diese in fünf Gruppen eingeteilten Nummern zeigen den Erfolgs- oder Fehlerstatus der HTTP-Response an und helfen, diesen schneller zu erfassen. [11], [19], [22]
- Hypermedia: Eine API-Antwort enthält oft nicht nur die eigentlich verlangte Information, sondern darüber hinaus Verlinkungen zu weiteren Ressourcen, die mit der angefragten Ressource in Verbindung stehen. Je nach Repräsentation stehen folglich URIs in bestimmten Attributen, bei JSON können sie beispielsweise in „links“ und Unterattributen wie „self“ angegeben werden. Auf diese Weise wird dem Client eine Art Navigation durch die verschiedenen Ressourcen ermöglicht, sodass er die API anhand eines Ausgangspunkts weiter erkunden kann. Ein häufiger Anwendungsfall des von Fielding als Hypermedia as the Engine of Application State (HATEOAS) bezeichneten Prinzips ist es, beim Abruf einer Ressourcenliste mit Pagination Verlinkungen auf beispielsweise die nächste und vorherige Seite mitzuliefern. [11], [19], [20]
- Zustandslosigkeit: Als Letztes muss genannt werden, dass die Kommunikation bei REST zustandslos ist, nicht zuletzt, weil HTTP beziehungsweise HTTPS genutzt wird. Das bedeutet, dass jeder HTTP-Request separat

betrachtet wird, unabhängig davon, ob es sich um die erste Anfrage eines Client handelt oder dieser bereits Nachrichten geschickt hat. Der Server speichert keine Informationen über den Client, folglich muss Letzterer bei jedem Request alle relevanten Daten übermitteln, zum Beispiel einen Authentifizierungstoken. [11], [19]

Abschließend soll ein Blick auf das von Leonard Richardson entwickelte Richardson Maturity Model geworfen werden, das in Abbildung 2 zu sehen ist. Mit diesem kann bewertet werden, wie REST-konform ein Webservice ist, denn in der Praxis werden nicht selten APIs als RESTful bezeichnet, die die Prinzipien des Architekturstils nur teilweise umsetzen. Richardson stellt vier Ebenen vor. Bei der untersten Stufe handelt sich um einen ganz grundlegenden Service mit einem einzigen Endpunkt, der Daten lediglich als Plain Old XML (POX) übermittelt. Dagegen kommen auf der darüber liegenden Ebene Ressourcen hinzu, welche über URIs adressierbar sind. Im nächsten Schritt werden nun die verschiedenen standardisierten HTTP-Methoden genutzt. Die letzte Stufe führt schließlich HATEOAS ein. Erst wenn eine API die oberste Ebene erreicht hat, ist sie tatsächlich RESTful. [21]

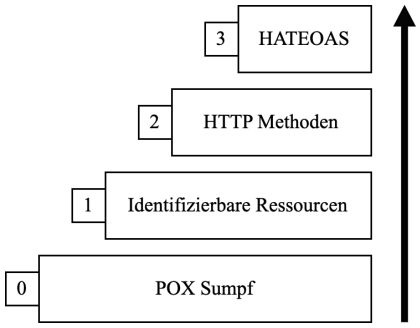

Abbildung 2. Das Richardson Maturity Model

B. GraphQL

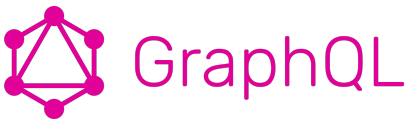

Abbildung 3. Das Logo von GraphQL [10]

1) Geschichte: GraphQL fand seinen Ursprung bei dem bekannten amerikanischen Unternehmen Meta, welches damals noch den Namen Facebook trug. Mit der zunehmenden Beliebtheit von Mobilgeräten setzte Facebook sich 2012 das Ziel, für diese eine eigene native Applikation zu entwickeln, die performanter und stabiler ist als die bisherige. Als bei der Technologieauswahl unter anderem das Problem auftrat, dass die Möglichkeiten bei der Datenabfrage mit REST zu unflexibel sind und daher sehr viele Abfragen stattfinden müssen, schlossen sich die Softwareentwickler Lee Byron, Dan Schafer und Nick Schrock zusammen und begannen mit der Entwicklung von GraphQL. Ursprünglich sollte GraphQL nur für den News Feed genutzt werden, wurde jedoch nach und nach intern in sämtlichen Client-Server-Anwendungen von Facebook eingesetzt. Schließlich entschied Facebook im Jahr 2015, die GraphQL Spezifikation als Open Source zu veröffentlichen. Seitdem hat sich eine relativ große Community gebildet, die Versionen der GraphQL Runtime in verschiedensten Programmiersprachen implementiert hat. Neben Meta bieten auch einige andere Unternehmen wie zum Beispiel GitHub bereits GraphQL APIs an. [15]–[18]

2) Grundlegende Konzepte: Auch wenn GraphQL häufig als Datenabfragesprache oder Laufzeitumgebung bezeichnet wird, handelt es sich dabei eigentlich um eine Spezifikation, die die Syntax und Semantik einer Abfragesprache sowie noch weitere Anforderungen festlegt. Aufbauend auf dieser kann GraphQL in verschiedenen Programmiersprachen implementiert werden. [26]

Eine wichtige Eigenschaft der GraphQL Spezifikation ist ihre starke Typisierung. Hierfür gibt es vordefinierte Typen, darunter insbesondere Enums und Skalare, das heißt einfache Typen wie String, Int oder ID. Ebenso können sogenannte Type Modifier genutzt werden, um Listen zu erzeugen oder Typen als nicht nullable zu kennzeichnen. Daneben gibt es Interfaces, Union Types und Input Types, die in dieser Arbeit aber nicht näher erläutert werden. Um die Typisierung zu gewährleisten, muss für jede Anwendung ein GraphQL Schema definiert werden, das in der Schema Definition Language (SDL) verfasst ist. Es repräsentiert die Typen von Daten, die ein Client abfragen kann. Das sind Objekte, zum Beispiel eine Person oder ein Produkt, mit ihren Attributen, die Felder genannt werden und wiederum selbst Typangaben besitzen. Über Felder werden mit dem Schema auch die Beziehungen zwischen den verschiedenen Typen erstellt. Letztendlich ergibt sich ein Graph aus Knoten, das heißt, den Objekten, und Kanten, also den Verbindungen zwischen den Objekten. Diese hierarchische Struktur bildet das Herzstück von GraphQL. [25]–[27]

Als nächstes soll näher auf die Kommunikation zwischen Client und Server eingegangen werden. GraphQL zeichnet sich insbesondere dadurch aus, dass der Client spezifische Daten anfordern kann und somit sehr viel Kontrolle über die Struktur der vom Server gesendeten Informationen hat. Dazu wird zunächst eine Operation in Form einer Zeichenkette konstruiert, die der GraphQL Spezifikation folgt und den durch den Client gewünschten Aufbau und Inhalt der Antwort des Servers definiert. Die Bausteine dieser Zeichenkette stellen Felder dar, womit sowohl Attribute von Objekten als auch Objekte selbst gemeint sein können. Für diese können wiederum verschachtelte Felder, sogenannte nested fields, angegeben werden. Auf diese Weise lassen sich auch Beziehungen in die Zeichenkette und damit in eine einzige Anfrage des Client integrieren. Die fertige Operation wird schließlich über ein nicht vorgegebenes Protokoll, jedoch in der Regel über HTTP beziehungsweise HTTPS, an den meist genau einen Endpunkt des Servers gesendet. Dort wird sie mithilfe des GraphQL Schemas interpretiert und es werden Resolver aufgerufen. Das sind Funktionen, die die entsprechende Logik beinhalten, um

die Daten für jedes Feld aus der Datenquelle abzurufen und schließlich an den Client zu senden. Entsprechend kann es auch sein, dass Resolver Daten verändern und eine Bestätigung darüber an den Client geschickt wird. [11], [23]–[25]
Es soll noch angemerkt werden, dass in den GraphQL Operationen auch Argumente verwendet werden können, um Feldern bestimmte Werte zu geben. Das kann zum Beispiel eine ID sein, mit der ein ganz spezifisches Objekt abgerufen werden kann. Ebenso können Variablen zur Definition von Platzhaltern genutzt werden, die dynamisch bei der Ausführung der Operation gefüllt werden können. Operationen lassen sich dadurch flexibel gestalten. Des Weiteren besteht durch Fragments die Möglichkeit, Teile von Operationen auszulagern und wieder zu verwenden, wodurch diese lesbarer und wartbarer werden. Erwähnenswert sind zudem Directives. Diese können die Struktur des Ergebnisses einer Operation beeinflussen, indem sie an ein Feld oder Fragment angehängt werden und dieses beispielsweise nur in das Ergebnis aufgenommen wird, wenn ein bestimmtes Argument den Wert true liefert. Abgesehen davon gibt es noch ein paar weitere Besonderheiten wie zum Beispiel Aliases, auf die im Rahmen dieser Arbeit aber nicht genauer eingegangen wird. [24], [25]
Anschließend werden nun die konkreten Aktionen vorgestellt, die ein Client ausführen kann. Hierfür existieren in GraphQL die im Folgenden aufgelisteten drei Operationstypen, von denen jeder als Typ im GraphQL Schema definiert werden muss. Dabei sind die Felder die verschiedenen Arten von Daten, auf die der Client zugreifen kann. Alle drei Typen stellen Einstiegspunkte in das Schema dar, von denen ausgehend Operationen stattfinden können. [24], [27], [28]

- Query: Wie der Name vermuten lässt, dient eine Query dazu, Daten von dem Server abzurufen, das heißt, lesende Aktionen auszuführen. [26]
- Mutation: Mit Mutations können Daten erstellt, verändert oder gelöscht werden. Obwohl dies theoretisch auch mit einer Query implementiert werden könnte, ist es Konvention für jegliche schreibende oder lesende und schreibende Aktion eine Mutation zu verwenden. Ein wichtiger Unterschied zwischen beiden Operationstypen ist zudem, dass Queries parallel ausgeführt werden und Mutations sequentiell. [24], [25]
- Subscription: Zuletzt besteht mit dem Operationstyp Subscription die Möglichkeit, Echtzeitdaten des Servers zu abonnieren und auf Ereignisse zu reagieren. [28]

Bei jeder Operation muss ihr entsprechender Typ genannt werden, außer es wird eine bestimmte Kurzschreibweise verwendet. Diese gibt es nur für Query, weshalb in diesem Fall klar ist, um welchen Operationstyp es sich handelt. Neben ihrem Typen sollte zudem für jede Operation stets ein sinnvoller Name angegeben werden, der ähnlich wie ein Funktionsname beispielsweise das Finden von Fehlern und das Lesen von Log-Dateien vereinfacht. [24]
Zuletzt soll noch ein Konzept vorgestellt werden, das dabei hilft, das GraphQL Schema zur Laufzeit zu erkunden und zu analysieren: Introspektion. Dabei werden spezielle Abfragen genutzt, um zum Beispiel herauszufinden, welche Typen und Felder verfügbar sind. [28]

III. Vergleich von REST und GraphQL

Nachdem die Entstehung sowie die Konzepte der beiden Technologien nun einzeln erläutert wurden, sollen sie im Folgenden miteinander verglichen werden. Dabei wird auf verschiedene Aspekte eingegangen, die für die Wahl der richtigen API-Architektur eine Rolle spielen.

A. Schnittstellendefinition

Eine Schnittstellendefinition beschreibt den Umfang und die Struktur der über die API erreichbaren Informationen. Zudem bestimmt sie, wie Daten abgefragt werden können. Eine präzise Definition kann sowohl die Programmierung als auch das Verständnis der Schnittstelle für die Entwickler erleichtern.
Was REST betrifft, so steht den Programmierern einer API vollkommen frei, inwieweit diese spezifiziert werden soll. Aufgrund dessen sind die Dokumentationen von RESTful Webservices oft unvollständig oder veraltet. Wird sich für eine Schnittstellenbeschreibung entschieden, so wird häufig auf die standardisierte OpenAPI Spezifikation zurückgegriffen, die besser unter ihrem früheren Namen Swagger bekannt ist. API-Beschreibungen können mit bestimmten Tools und Frameworks auch aus dem Code generiert werden. [29]
GraphQL wird hingegen als selbstdokumentierend bezeichnet, da es für jede API die Definition eines präzisen Schemas vorschreibt. Dieses wird entweder händisch erstellt oder automatisch aus den Resolvern erzeugt. Es stellt eine Beschreibung der verfügbaren Abfragen und Datentypen, inklusive deren Felder und Beziehungen dar. Mithilfe von Introspektion können Entwickler die API sehr leicht erkunden. Hierzu werden manuelle Abfragen oder interaktive Tools wie die in den Browser eingebettete Entwicklungsumgebung (IDE) GraphiQL genutzt. [29], [30]

B. Client-Server-Kommunikation

Des Weiteren muss die Kommunikation zwischen Client und Server betrachtet werden. Diese besteht im Austausch von Daten über ein Netzwerk. Hierbei sendet der Client Anfragen und der Server Antworten.
Zunächst können an dieser Stelle ein paar Gemeinsamkeiten zwischen REST und GraphQL genannt werden. So unterstützen beide das Client-Server-Modell und sind hierbei weitgehend zustandslos. Auch wenn weder das Protokoll noch das Format beim Datentransfer vorgeschrieben sind, werden beide Technologien in der Regel mit HTTP beziehungsweise HTTPS und JSON verwendet. [29], [32]

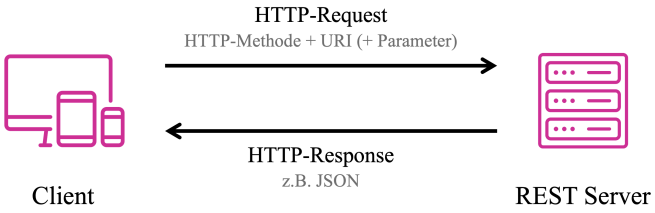

Abbildung 4. Client-Server-Kommunikation bei REST [40], [41]

Das REST-Paradigma zeichnet sich durch eine sehr einheitliche und vorhersehbare Schnittstelle aus. Dabei gibt es Ressourcen, die jeweils über eine eindeutige URI identifiziert werden. Der Zugriff auf Ressourcen erfolgt über vordefinierte Endpunkte, die die standardisierten HTTP-Methoden verwenden, insbesondere POST, GET, PUT, PATCH und DELETE. Typischerweise gibt es pro Ressource mindestens eine Route, welche jeweils mit verschiedenen HTTP-Verben genutzt werden kann. Auf diese Weise wird eine klare Trennung zwischen den Ressourcen ermöglicht. [29], [31]

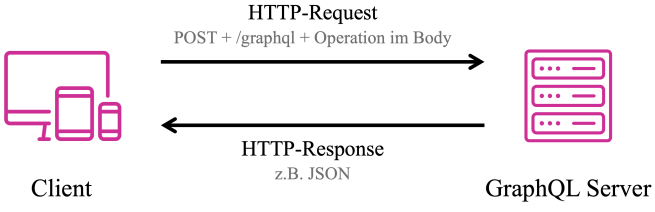

Abbildung 5. Client-Server-Kommunikation bei GraphQL [40], [41]

Während es bei RESTful Webservices in der Regel sehr viele Routen gibt, so bietet eine GraphQL Schnittstelle nur einen einzigen Endpunkt für sämtliche Operationen an. Bei den verschiedenen Anfragen kommt nur das HTTP-Verb POST zum Einsatz. Dadurch wird die API-Struktur wesentlich vereinfacht. [29], [31]

C. Flexibilität

1) Abfrage- und Antwortmöglichkeiten: Ein sehr relevantes Kriterium bei der Wahl einer Schnittstellen-Architektur sind die Abfrage- und Antwortmöglichkeiten. Sie entscheiden darüber, wie flexibel und effizient die Kommunikation zwischen Client und Server erfolgt. Durch die Unterstützung individueller Requests, bei denen die gewünschten Daten genau spezifiziert werden, kann beispielsweise überflüssiger Datentransfer umgangen werden. Auf diese Weise lässt sich sowohl die Performance der API als auch das Erlebnis für den Benutzer optimieren. [30]
Durch die starke Ressourcenorientierung mit serverseitig festgelegten Endpunkten sind die Möglichkeiten von Anfragen und Antworten bei REST sehr eingeschränkt. Clients können hier nicht etwa angeben, welche Attribute einer Ressource sie anfragen möchten, sondern erhalten eine starr festgelegte Response. Aus diesem Grund werden sie oft mit Over- und Underfetching konfrontiert. Bei Ersterem lädt der Client wesentlich mehr Informationen herunter, als in der Anwendung tatsächlich gebraucht werden. Underfetching meint hingegen, dass die clientseitig benötigten Daten nicht von einem einzigen Endpunkt bereitgestellt werden, sodass mehrere Requests gestellt werden müssen. In diesem Zusammenhang tritt oft das besonders ineffiziente N+1 Problem auf. Hierbei wird zum Beispiel eine Liste an Autoren abgerufen und für jeden Autor erfolgt dann eine zusätzliche Abfrage seiner Bücher. [25], [36]
GraphQL erlaubt stattdessen wesentlich flexiblere Anfragen und Antworten. Indem der Client nur genau die Daten erhält, die er in seinem Request verlangt hat, werden Over- und Underfetching erfolgreich verhindert. Das bedeutet auch, dass stets nur eine einzige Anfrage erforderlich ist, wodurch die Anzahl der notwendigen API-Aufrufe und die Menge der übertragenen Daten erheblich reduziert werden. [23], [25]

2) Skalierbarkeit: Unter Skalierbarkeit wird die Fähigkeit eines Servers verstanden, mit einer steigenden Anzahl an Clients zuverlässig und effizient umzugehen. Beeinflusst wird sie vor allem durch die verfügbaren Ressourcen wie zum Beispiel Serverleistung und Netzwerkbandbreite, durch Load-Balancing und durch Caching. [31]
REST APIs sind im Allgemeinen sehr gut skalierbar. Dies wird unter anderem durch eine klare Trennung von Client und Server erreicht. Die Tatsache, dass RESTful Webservices zustandslos sind, sodass alle Informationen in jeder Anfrage mitgeschickt werden, ermöglicht ebenfalls eine Lastverteilung auf mehrere Server. Darüber hinaus kann Caching unkompliziert realisiert werden (siehe Abschnitt III-D). [29]
Im Gegensatz dazu ist das Caching bei GraphQL wesentlich schwieriger (siehe Abschnitt III-D). Des Weiteren sind Subscriptions nicht zustandslos und erfordern eine dauerhafte Verbindung zwischen Client und Server, die Ressourcen beansprucht. Dennoch muss auch festgestellt werden, dass die flexibleren Abfragemöglichkeiten ohne Over- und Underfetching die Serverlast reduzieren. Insgesamt ist die Skalierbarkeit bei GraphQL daher auch gut. [31]

3) Versionierung: Wie bei jeder Software muss es auch bei APIs möglich sein, neue Funktionalität hinzuzufügen sowie vorhandene Funktionalität zu ändern und zu verbessern. Die Versionierung dient hierbei dazu, ältere Varianten der Schnittstelle weiterhin anzubieten, sodass bestehende Clients nicht beeinträchtigt werden.
Bei Fieldings Architekturstil wird die Versionierung normalerweise innerhalb der URI umgesetzt. Das bedeuted, dass bei dem „/v1/“ innerhalb des Pfads „https://api/v1/authors/“ das „v“ für Version steht und „1“ durch die jeweilige Versionsnummer ersetzt wird. Es muss erwähnt werden, dass REST zu keiner Versionierung verpflichtet, allerdings ist sie sehr üblich. Häufig laufen mehrere Varianten der API parallel und Nutzer entscheiden über die URI, welche sie nutzen möchten. [32]
Da bei GraphQL der Client und nicht der Server genau spezifiziert, welche Daten und Felder er abfragen will, wird keine Versionierung benötigt. Stattdessen sollen APIs abwärtskompatibel sein. Neue Funktionen können problemlos ergänzt werden, ohne bestehenden Queries zu schaden. Sollen Funktionen, zum Beispiel Felder, entfernt werden, so werden sie zunächst als veraltet markiert. Der folgende Ausschnitt eines Schemas zeigt dies beispielhaft. [23], [32], [42]

```
type Author {
  id: ID
  firstName: String
  lastName: String
  biography: String @deprecated
   (reason: "Field is deprecated.")
  books: [Book]
}
```

Den Clients wird über eine Warnung mitgeteilt, wenn Felder veraltet sind. Wurden die API-Abfragen nach und nach umgeschrieben, um nicht mehr auf die betroffenen Felder zuzugreifen, können die Felder entfernt werden. Statt einer Warnung verursacht die Abfrage der Felder dann eine Fehlermeldung. [23], [32], [42]

D. Performance und Caching

Die Performance von Schnittstellen ist ebenfalls ein zu betrachtendes Vergleichskriterium, insbesondere, wie schnell und effizient Daten bereitgestellt werden. In Verbindung damit steht Caching. Hierbei werden oft angeforderte Informationen zwischengespeichert, sodass sie bei zukünftigen Anfragen nicht neu abgerufen werden müssen. Dies verkürzt die Antwortdauer und entlastet den Server.
Eine REST API stellt konkrete Endpunkte zur Verfügung, wobei die Struktur der Antworten serverseitig vorgegeben ist. Sind mehrere Ressourcen vonnöten, so müssen in der Regel auch mehrere Anfragen erfolgen. Es kommt dabei oft zu Over- und Underfetching, worunter die Performance leidet. Durch die eindeutige Adressierbarkeit von Ressourcen über URIs, ist jedoch das Caching in der Regel sehr einfach möglich. Es kann auf Ressourcenebene stattfinden, wobei der Cache an den Identifier gebunden werden kann. Dies ist zusammen mit den Caching-Mechanismen von HTTP mit wenig Aufwand umsetzbar. [23], [29], [31], [33]
Grundsätzlich performen GraphQL APIs besser als RESTful Webservices, da der Client mit einem einzigen Request gezielt nur die benötigten Informationen anfordern kann. Jedoch erhöhen stark verschachtelte Abfragen die Serverlast. Caching ist in GraphQL wesentlich komplexer als bei REST, da jede Query unterschiedliche Daten zurückgeben kann, selbst wenn sie sich auf dieselbe Entität bezieht. Folglich muss das Caching hier auf Ebene der Felder stattfinden. Mittlerweile gibt es aber schon einige leistungfähige Lösungen, vor allem für clientseitiges Caching. Hier ist beispielsweise Apollo Client zu nennen. [23], [29]–[31]

E. Entwicklungsaufwand

Selbstverständlich wird die Entscheidung über die API-Architektur auch von dem Entwicklungsaufwand auf Client- und Serverseite beeinflusst. Das bedeutet hauptsächlich, wie komplex die Programmierung ist und wie viel Zeit dafür aufgebracht werden muss. Nicht nur lassen sich durch eine effiziente Entwicklung Kosten sparen, es ergeben sich auch langfristige Vorteile wie eine geringere Fehleranfälligkeit und vereinfachte Wartung.

1) Clientseitige Implementierung: Auf der Seite des Client bietet REST besonders in Bezug auf die Requests wenig Flexibilität, wie in Abschnitt III-C1 bereits angesprochen wurde. So müssen sich Anfragen an die API stets aus den folgenden, starr definierten Bestandteilen zusammensetzen: Die HTTP-Methode, die die jeweilige Aktion spezifiziert, eine URI als Identifier für eine Ressource und optional Parameter und Werte, die entweder in der URI oder im Body mitgeliefert werden. Eine beispielhafte Abfrage der Autor-Ressource mit der ID 2 wäre GET https://api/v1/authors/2. Werden zu diesem Autor auch die Titel seiner Bücher benötigt, so wird je nach Definition des Endpunkts sehr wahrscheinlich noch ein oder sogar mehrere weitere Requests erforderlich sein. Zu erwähnen ist außerdem, dass REST schwach typisiert ist. Das bedeutet, dass der Client das Format der von der Schnittstelle abgerufenen Informationen selbst interpretieren und die Daten auf Gültigkeit prüfen muss. [32]
Die ebenfalls in Abschnitt III-C1 erläuterten flexiblen Abfragemöglichkeiten zeichnen dagegen GraphQL aus. Der Client kann hier zwischen einer Query, Mutation oder Subscription wählen und ansonsten seinen Request sehr individuell gestalten. So kann bei dem Abruf des Autor mit der ID 2 anders als mit REST angepasst werden, welche Felder mitgeliefert werden sollen. Das folgende Beispiel zeigt, wie diese Query aussehen könnte, wenn zu dem gewünschten Autor die ID und der Nachname sowie all seine Bücher mit jeweils ihrem Titel erhalten werden sollen. Sie steht im Body des HTTP Request. Als HTTP-Verb wird POST genutzt und die URI könnte https://graphql sein. [32], [37]

```
query {
  authorWithBooks(id: 2) {
      id
      lastName
      books {
          title
      }
  }
}
```

Die Daten sind des Weiteren durch das von der GraphQL Spezifikation geforderte Schema stark typisiert. Dies ist sowohl für den Server als auch den Client von Vorteil, da beiden das Format der untereinander übertragenen Informationen bekannt ist und die API mithilfe von Introspektion erkundet werden kann. Zudem werden Requests, die nicht gültig sind, durch das Schema abgewiesen, sodass eine Validierung der Daten aus der Response durch den Client überflüssig wird. Die Antworten des Servers sind für den Client insgesamt vorhersehbar. [29], [31], [32]

2) Serverseitige Implementierung: Was die serverseitige Implementierung betrifft, so müssen bei einem RESTful Webservice in erster Linie Ressourcen und Endpunkte definiert werden. Dies gestaltet sich durch die umfangreichen Bibliotheken (siehe Abschnitt III-F1), die es mittlerweile gibt, in der Regel recht einfach und effizient. Da die Struktur der Requests serverseitig fest vorgegeben ist, ist die Logik

innerhalb der Endpunkte in der Regel wenig komplex. Der in dem vorherigen Abschnitt III-E1 genannte Aufruf GET https://api/v1/authors/2 könnte beispielsweise folgende Daten als Antwort erhalten. [31]

```
{
  "id": 2,
  "firstName": "John Ronald Reuel",
  "lastName": "Tolkien"
}
```

Die Kommunikation mit Datenquellen wie einer relationalen Datenbank wird ebenfalls mit etablierten Frameworks wie Spring Boot und hilfreichen Tools, die beispielsweise Object-Relational Mapping (ORM) ermöglichen, vereinfacht. Was Fehler betrifft, so werden diese mit den klar definierten HTTP-Statuscodes angezeigt. Auch hierbei ist es von Vorteil, dass die möglichen Abfragen, die ein Client tätigen kann, festgelegt sind, da das Finden und Beheben von Fehlern so einfacher ist. [29]
Das bereits mehrfach angesprochene Schema bringt bei der Entwicklung einer GraphQL API viele Vorteile. Allerdings muss es zunächst serverseitig implementiert werden. Ein beispielhaftes Schema ist in unten stehendem Listing dargestellt. Darin werden die Typen Query, Author und Book mit ihren jeweiligen Feldern definiert. [38]

```
type Query {
  authorWithBooks(Id: ID): Author
}

type Author {
  id: ID
  firstName: String
  lastName: String
  books: [Book]
}

type Book {
  id: ID
  title: String
  pageCount: Int
}
```

Abgesehen von dem Schema muss auf Serverseite eine Datenquelle angebunden werden, was wie bei REST durch das recht große Ökosystem an Frameworks (siehe Abschnitt III-F1) nicht allzu schwierig ist. Um die dynamischen, oft verschachtelten Abfragen zu ermöglichen sind zudem Resolver zu entwickeln. Die teilweise komplexe Logik dieser Funktionen beschreibt, wie die in dem Request gewünschten Felder aus der Datenquelle abgerufen werden können. Abfragen wie die in Abschnitt III-E1 genannte authorWithBooks Query erhalten somit beispielsweise folgende Antwort von der API. [11], [31], [38]

```
{
  "data": {
    "id": 2,
    "lastName": "Tolkien",
    "books": [
      {
        "title": "The Hobbit"
      },
      {
        "title": "The Lord of the Rings"
      }
    ]
  }
}
```

Auch wenn eine GraphQL API zunächst anspruchsvoll zu entwickeln sein kann, ist sie langfristig gesehen vor allem durch ihre flexiblen Abfragemöglichkeiten besser anpassbar und erweiterbar als ein RESTful Webservice. Zuletzt soll die Fehlerbehandlung angesprochen werden. Hierbei hilft das GraphQL Schema, da es ungültige Requests ablehnt und somit eine automatische Validierung erfolgt. Auch werden dabei meist detaillierte Fehlermeldungen erzeugt. Jedoch kann es für manche Entwickler irritierend sein, dass GraphQL APIs stets auch im Fehlerfall den Statuscode 200 angeben, der eigentlich eine erfolgreiche Anfrage anzeigt. [11], [29], [32]

F. Ökosystem

1) Technologien: Die Entwicklung von APIs kann vereinfacht und beschleunigt werden, wenn auf etablierte Tools, Bibliotheken und Best Practices zurückgegriffen werden kann. Ein umfassendes Ökosystem hilft zudem bei der späteren Wartung der Schnittstelle.
Dass es REST nun schon seit über 20 Jahren gibt, zeigt sich auch in den umfassenden Tools, Bibliotheken und Frameworks, die den Alltag zahlreicher Entwickler vereinfachen. Hier kann zum Beispiel Postman genannt werden, das in Kapitel I erwähnt wurde. Über die Jahre hat sich ein so großes Ökosystem aufgebaut, dass RESTful Webservices mit sämtlichen bekannten Programmiersprachen und vielen verschiedenen Datenquellen implementiert werden können. Beliebte Frameworks sind beispielsweise Spring Boot für Java oder ASP.NET Core für C#. Wurden die Konzepte des Architekturstils einmal verstanden, kann der Entwickler die Technologien nahezu komplett frei wählen. [23], [29], [34]
Auch wenn GraphQL vergleichsweise jung ist, wächst das Ökosystem an Technologien sowohl auf Server- als auch auf Clientseite stetig. So gibt es die zuvor angesprochene interaktive Browser-IDE GraphiQL sowie GraphQL Playground gewissermaßen als Entsprechung zu Postman bei REST. Ebenfalls wird die Entwicklung durch Bibliotheken wie Gatsby.js und Frameworks für nahezu jede Programmiersprache und Datenquelle unterstützt, darunter Apollo für JavaScript sowie auch Spring Boot für Java. [23], [34], [35]

2) Entwickler-Community: Das Ökosystem einer API besteht nicht nur aus Technologien, sondern auch aus den Entwicklern, die als Community bezeichnet werden. Durch den Erfahrungsaustausch und die gegenseitige Hilfe bei Problemen wird der Code einer Schnittstelle in vielen Fällen schneller geschrieben und ist qualitativ hochwertiger.
Wirft man einen Blick zurück auf Abbildung 1 in Kapitel I, so wird klar, wie beliebt Fieldings Architekturstil unter Entwicklern ist. Dementsprechend umfangreich und erfahren ist die Community, die sich über viele Jahre aufgebaut hat. Sie investiert wiederum auch Zeit und Ressourcen, um hilfreiche Tools und Frameworks für REST zu implementieren und das Ökosystem zu erweitern. [29]
Ebenso wie die Zahl der Tools und Frameworks, wächst auch die Entwicklergemeinschaft bei GraphQL stetig und schnell. So entstehen zum Beispiel auch Konferenzen oder Podcasts, die GraphQL thematisieren. Dennoch ist REST aktuell noch weitaus populärer, wie auch die angesprochene Grafik 1 in Kapitel I zeigt. [23]

IV. Anwendungsfälle in der Praxis

Im direkten Vergleich zeigt sich somit, dass sowohl REST als auch GraphQL Stärken und Schwächen haben und insgesamt nicht gesagt werden kann, dass eine der beiden Technologien in jedem Fall überlegen ist. Für beide gibt es in der Praxis Anwendungsfälle, von denen jeweils drei in diesem Kapitel betrachtet werden sollen. [34]

A. REST-basierte Anwendungsfälle

1) Kleine Anwendungen mit einfachen Daten: Im Allgemeinen ist ein RESTful Webservice etwas leichter zu implementieren als eine GraphQL API. Außerdem gibt es den Architekturstil bereits deutlich länger, weshalb die Zahl der Entwickler, die schon mit REST gearbeitet hat, größer ist. Daher ist REST für kleine Anwendungen mit einfachen Daten empfehlenswerter. [32], [34]

2) Wenig komplexe Datenabfragen: Ein weiterer Anwendungsfall für REST sind APIs, die keine komplexen Datenabfragen ermöglichen sollen. Das bedeutet beispielsweise, dass sie keine verschachtelten Felder beinhalten, sondern genau die Ressourcen angefordert werden, für die Endpunkte existieren. [32], [34]

3) Einheitliche Datenabfragen: Zuletzt sollte REST gewählt werden, wenn die große Mehrheit an Requests einheitlich ist und die Flexibilität von GraphQL nicht benötigt wird. Klar definierte Endpunkte haben dann mehr Vorteile, beispielsweise vereinfachtes Caching. [32], [34]

B. GraphQL-basierte Anwendungsfälle

1) Begrenzte Bandbreite: Mit GraphQL können mit einer einzigen API-Anfrage genau die durch den Client benötigten Daten erhalten werden. Auf diese Weise kann Over- und Underfetching vermieden, Datentransfer reduziert und somit Bandbreite gespart werden. [32], [34]

2) Kombination mehrerer Datenquellen: Des Weiteren können mit GraphQL mehrere Datenquellen in einem einzigen Endpunkt verknüpft werden. Dazu sammelt der zugehörige Resolver die Informationen zunächst von den unterschiedlichen Quellen, wandelt sie dann in das gewünschte kombinierte Format um und sendet sie an den Client. [32], [34], [39]

3) Uneinheitliche Datenabfragen: Wie besonders in Kapitel III immer wieder betont wurde, zeichnet sich GraphQL hauptsächlich durch die sehr flexiblen Abfragemöglichkeiten aus. Wird daher angenommen, dass sehr viele uneinheitliche Requests an eine Schnittstelle gestellt werden, so sollte sie mit GraphQL implementiert werden. [32], [34]

V. Fazit

Insgesamt kann gesagt werden, dass GraphQL seit seiner Veröffentlichung bereits stark gewachsen ist. Es ist davon auszugehen, dass sich diese Entwicklung in den nächsten Jahren fortsetzen wird. Auch wenn REST in Punkten wie Skalierbarkeit, Caching oder dem Ökosystem noch vorne liegt, beweist GraphQL seine Stärken in den flexiblen Abfrage- und Antwortmöglichkeiten und der starken Typisierung. Daher kann man zu dem Schluss kommen, dass GraphQL durchaus eine gleichwertige Alternative zu REST darstellt und beide Technologien ihre Daseinsberechtigung haben.

Literatur

[1] "Webservices: Dienste von Maschine zu Maschine". IONOS SE. https://www.ionos.de/digitalguide/websites/web-entwicklung/webservice/ (abgerufen 14.04.2024).
[2] "2023 State of the API Report". Postman Inc. https://www.postman.com/state-of-api/ (abgerufen 14.04.2024).
[3] "2023 State of the API Report. API technologies". Postman Inc. https://www.postman.com/state-of-api/api-technologies/#api-technologies (abgerufen 07.04.2024).
[4] "2022 State of the API Report. API technologies". Postman Inc. https://www.postman.com/state-of-api/2022/api-technologies/#api-technologies (abgerufen 14.04.2024).
[5] "2021 State of the API Report. API technologies". Postman Inc. https://www.postman.com/state-of-api/2021/api-technologies/#api-technologies (abgerufen 14.04.2024).
[6] "2020 State of the API Report. API technologies". Postman Inc. https://www.postman.com/state-of-api/2020/api-technologies/#api-technologies (abgerufen 14.04.2024).
[7] "Was ist ein Webhook?". Red Hat Inc. https://www.redhat.com/de/topics/automation/what-is-a-webhook (abgerufen 21.04.2024).
[8] "A query language for your API". The GraphQL Foundation. https://graphql.org/ (abgerufen 21.04.2024).
[9] "SOAP: Das Netzwerkprotokoll erklärt". IONOS SE. https://www.ionos.de/digitalguide/websites/web-entwicklung/soap-simple-object-access-protocol/ (abgerufen 21.04.2024).
[10] "GraphQL Logo & Brand Guidelines". The GraphQL Foundation. https://graphql.org/brand/ (abgerufen 16.04.2024).
[11] P. Ackermann, "Webentwicklung. Das Handbuch für Fullstack-Entwickler", Bonn: Rheinwerk Verlag, 2021.
[12] "Google APIs Explorer". Google. https://developers.google.com/apis-explorer?hl=de#p/ (abgerufen 21.04.2024).
[13] "Get started with PayPal REST APIs". PayPal. https://developer.paypal.com/api/rest/ (abgerufen 21.04.2024).
[14] "X API". X Corp. https://developer.twitter.com/en/products/twitter-api (abgerufen 21.04.2024).
[15] S. Sharma, "Modern API Development with Spring and Spring Boot", Birmingham: Packt Publishing, 2021.
[16] L. Byron, "GraphQL: A data query language". Engineering at Meta. https://engineering.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/ (abgerufen 22.04.2024).

[17] “Graph API”. Meta. https://developers.facebook.com/docs/graph-api (abgerufen 22.04.2024).
[18] “Dokumentation zu GitHub GraphQL-API”. GitHub, Inc. https://docs.github.com/de/graphql (abgerufen 22.04.2024).
[19] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”, Dissertation, University of California, Irvine, 2000.
[20] “HATEOAS: Das steckt hinter dem Akronym”. IONOS SE. https://www.ionos.de/digitalguide/websites/web-entwicklung/hateoas-alle-informationen-zu-der-rest-eigenschaft/ (abgerufen 25.04.2024).
[21] M. Fowler, “Richardson Maturity Model”. Martin Fowler. https://martinfowler.com/articles/richardsonMaturityModel.html (abgerufen 25.04.2024).
[22] F. Doglio, “REST API Development with Node.js. Manage and Understand the Full Capabilities of Successful REST Development”, Berkeley, CA: Apress, 2018.
[23] R. Wieruch, “The Road to GraphQ. Your journey to master pragmatic GraphQL in JavaScript with React.js and Node.js.”, Berlin: Robin Wieruch, 2018.
[24] “Queries and Mutations”. The GraphQL Foundation. https://graphql.org/learn/queries/ (abgerufen 02.05.2024).
[25] K. Goetsch, “GraphQL for Modern Commerce. Complement Your REST APIs with the Power of Graphs”, O’Reilly Media, Inc., 2020.
[26] N. Aleks, D. Farhi, “Black Hat GraphQL. Attacking Next Generation APIs”, San Francisco, CA: No Starch Press, 2023.
[27] “Schemas and Types”. The GraphQL Foundation. https://graphql.org/learn/schema/ (abgerufen 05.05.2024).
[28] E. Porcello, A. Banks, “Learning GraphQL. Declarative data fetching for modern web apps”, Beijing: O’Reilly Media, Inc., 2018.
[29] G. Bello, “GraphQL vs. REST”. Postman, Inc. https://blog.postman.com/graphql-vs-rest/ (abgerufen 18.05.2024).
[30] “Decision Guide to GraphQL Implementation”. Amazon Web Services, Inc. https://aws.amazon.com/de/graphql/guide/ (abgerufen 18.05.2024).
[31] R. Ala-Laurinaho, J. Mattila, J. Autiosalo, J. Hietala, H. Laaki, K. Tammi, “Comparison of REST and GraphQL Interfaces for OPC UA”, Computers, vol. 11, no. 5, pp. 6-9, May 2022, doi: 10.3390/computers11050065.
[32] “Was ist der Unterschied zwischen GraphQL und REST?”. Amazon Web Services, Inc. https://aws.amazon.com/de/compare/the-difference-between-graphql-and-rest/ (abgerufen 18.05.2024).
[33] “GraphQL: Flexible Abfragesprache und Laufzeitumgebung für Ihr Web-API”. IONOS SE. https://www.ionos.de/digitalguide/websites/web-entwicklung/graphql/ (abgerufen 19.05.2024).
[34] C. R. China, “GraphQL vs. REST API: What’s the difference?”. IBM. https://www.ibm.com/blog/graphql-vs-rest-api/ (abgerufen 20.05.2024).
[35] “Code Using GraphQL”. The GraphQL Foundation. https://graphql.org/community/tools-and-libraries/ (abgerufen 20.05.2024).
[36] “GraphQL is the better REST”. howtographql. https://www.howtographql.com/basics/1-graphql-is-the-better-rest/ (abgerufen 22.05.2024).
[37] “Make an HTTP call with GraphQL”. Postman, Inc. https://learning.postman.com/docs/sending-requests/graphql/graphql-http/ (abgerufen 25.05.2024).
[38] “Building a GraphQL service”. Broadcom. https://spring.io/guides/gs/graphql-server (abgerufen 25.05.2024).
[39] E. Herrera, “GraphQL vs. REST APIs: Why you shouldn’t use GraphQL”. LogRocket. https://blog.logrocket.com/graphql-vs-rest-api-why-you-shouldnt-use-graphql/(abgerufen 26.05.2024).
[40] “Flaticon”. Freepik Company S.L. https://www.flaticon.com/free-icon/data-center_8911431 (abgerufen 23.06.2024).
[41] “Flaticon”. Freepik Company S.L. https://www.flaticon.com/free-icon/devices_408533 (abgerufen 23.06.2024).
[42] “GraphQL”. Facebook, Inc. https://spec.graphql.org/June2018/ (abgerufen 01.07.2024).