Hochschule für Technik und Wirtschaft Dresden

Fachbereich Informatik/Mathematik

Bachelorarbeit

im Studiengang Wirtschaftsinformatik

Thema: Vergleich der Web API Ansätze REST und GraphQL

eingereicht von: Fabian Meyertöns

eingereicht am: 9. Dezember 2019

Betreuer: Prof. Dr. Thomas Wiedemann

2. Gutachter: Prof. Dr. Jürgen Anke

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation und Zielstellung

Das Internet und Web-Anwendungen unterliegen einem starken Wandel. Die Kommerzialisierung, zunehmende Verbreitung und Erweiterung der Möglichkeiten haben zu einer veränderten Nutzung und Entwicklung von Internetanwendungen beigetragen. Vom Netzwerk zum Teilen von Hypertextdokumenten ist es zur Plattform/Medium (TODO) für alle Arten interaktiver betrieblicher und sozialer Software geworden.

Um der zunehmenden Größe, Komplexität und mangelnden Wartbarkeit der Anwendungen Herr zu werden, hat sich auch in der Webentwicklung der anerkannte Grundsatz „Teile und Herrsche“ durchgesetzt. Namentlich ist der Trend von sogenannter monolithischer zu service-orientierter Software zu beobachten. Verbesserungen in der Skriptausführung von Webbrowsern und die Weiterentwicklung von JavaScript selbst hatte außerdem zur Folge, dass immer mehr Teile der Programme auf dem Client ausgeführt werden. Zeuge dieser Entwicklung sind die vielen JavaScript Frameworks (z.B. Angular, React, Vue), welche besonders innerhalb der letzten 10 Jahre populär geworden sind und mittlerweile zum Standard der Webentwicklung gehören, sowie Projekte wie Electron und NativeScript, welche Webtechnologien nutzen, um native Anwendungen zu entwickeln [vgl. 25]. Mit diesen Frameworks entwickelte Single Page Applications (SPA) generieren Ansichten und Dialoge auf Basis von Daten, welche über Web Application Programming Interfaces (API) mit dem Server ausgetauscht werden. Oft kommunizieren Anwendungen mit einer Vielzahl von Webservices, z. B. um unternehmensinterne oder externe Datenquellen oder Dienste zu integrieren.

Um eine einheitliche Kommunikationsschnittstelle für die Interaktion von Clientanwendungen und Webservices zu schaffen, werden APIs häufig nach dem Representational State Transfer Architekturkonzept (REST) entwickelt. Da REST aber nur Rahmenbedingungen und die Grundgedanken für Web-APIs liefert, finden sich in der Praxis viele Formate und Modelle, welche versuchen Implementierungsdetails und Best-Practices auszuführen. Die Bezeichnung REST-API ist somit aufgeweicht worden und Entwickler sind, aufgrund mangelnder Standards, zu API spezifischen Anpassungen gezwungen.

GraphQL wurde 2015 von Facebook veröffentlicht und bringt eine Spezifikation und Abfragesprache für die Client-Server Kommunikation. Die Implementierung einer GraphQL-API unterscheidet sich gravierend von bestehenden REST-APIs. Mittlerweile existieren aber wichtige Entwicklerwerkzeuge und die ersten GraphQL-APIs haben sich in der Praxis bewährt [vgl. 27].

Die vorliegende Arbeit soll mehrere Fragen bezüglich der beiden Ansätze für bestehende Web-APIs und Neuentwicklungen klären:

1. Welche Vorteile bietet GraphQL?
2. Welche Probleme entstehen erst durch GraphQL?

3. Für welche Anwendungszwecke kann eine GraphQL-API eine REST-API ersetzen?

4. Ist ein gemeinsamer Einsatz der beiden Ansätze möglich und sinnvoll?

1.2 Aufbau der Arbeit

Um die API Ansätze REST und GraphQL zu vergleichen soll zunächst die zugrundeliegende Client-Server Architektur betrachtet werden. Weiterhin muss der Begriff des Application Programming Interface (API) allgemein und speziell im Zusammenhang mit dem Web definiert werden. Die REST Architektur stellt seit ihrer Erklärung im Jahr 2000 die Grundlage für viele Webanwendungen und APIs dar, wurde aber über die Jahre verschieden interpretiert und standardisiert. Erst 2015 wurde GraphQL als Alternative veröffentlicht. Die Grundgedanken beider Ansätze und ihre Implementierung werden daher nach ihrer chronologischen Reihenfolge untersucht. Anschließend soll im Vergleich festgestellt werden, wie die Technologien die klassischen Probleme bei der API Entwicklung lösen und welche Vorteile und Probleme erkennbar sind. Zum praktischen Vergleich wurde eine Single-Page Anwendung und jeweils eine REST- und GraphQL-Serveranwendung entwickelt, mit denen verschiedene API Szenarien dargestellt und die Performance gemessen werden kann. Aus diesen Ergebnissen sollen die zuvor genannten Forschungsfragen beantwortet werden, bevor abschließen auch auf den kombinierten Einsatz der beiden Technologien eingegangen wird.

2 Vorbetrachtungen

2.1 Client-Server Architektur

Die Basis für die meisten Anwendungen im Web bildet die Client-Server Architektur. Der Nachrichtenaustausch in einem verteilten System wird dabei auf zwei Akteure und eine Interaktion fokussiert. Der Client initiiert die Interaktion, indem er eine Nachricht an den Server sendet. Der Server reagiert, indem er mit dem Resultat der Auswertung der Nachricht antwortet. Ein Client kann mit mehreren Servern in Interaktion treten und mehrere Clients können mit dem gleichen Server kommunizieren, wie in Abbildung 2.1 dargestellt. Durch die Anfrage werden die Rollen von Client und Server festgelegt, welche jedoch nur für die Dauer der Interaktion gelten. So kann ein Akteur, der bei einem Nachrichtenaustausch als Server angefragt wird, während eines anderen selbst als Client die Kommunikation beginnen. Bedingungen für die Client-Server Architektur sind, dass dem Client zu Beginn der Kommunikation der Server bekannt ist und dass beide das gleiche Kommunikationsprotokoll unterstützen.

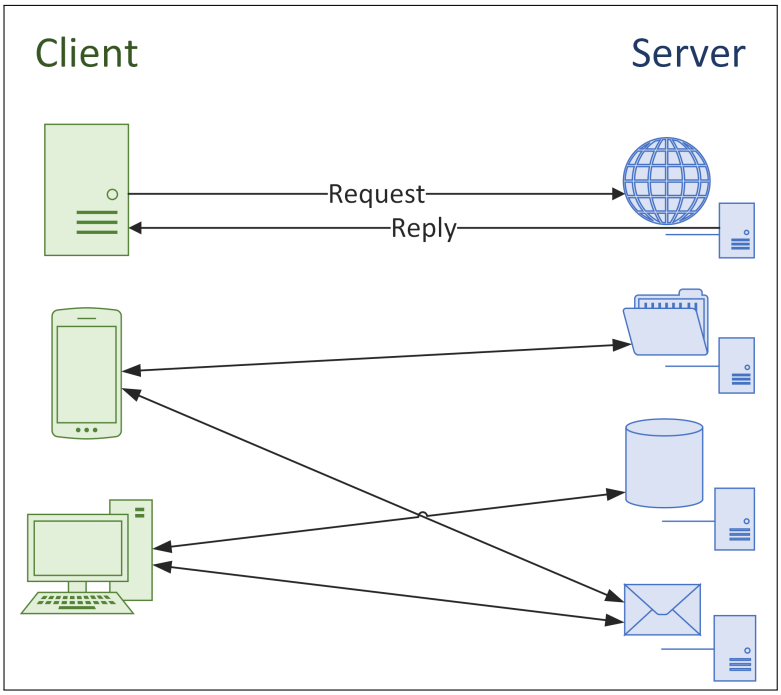

Abbildung 2.1: Mehrere Clients treten mit verschiedenen Servertypen in Interaktion, in Anlehnung an [3, S. 23]

Die Entwicklung einer Anwendung nach dieser Architektur hat ein erklärtes Ziel und Hauptvorteil: Die Trennung der Verantwortlichkeiten.[vgl. 9, S. 78]

Client und Server sind logisch und oft auch physikalisch getrennte Teile einer Anwendung und können unabhängig voneinander entwickelt werden, solange sie ihre

Kommunikationsschnittstelle wahren. Die damit verbundene Trennung der Verantwortlichkeiten führt zu einer Reduzierung der Komplexität der einzelnen Anwendungsteile. Da Client und Server unabhängig voneinander ausgeführt werden, beeinflusst der Ausfall des einen nicht die Stabilität des anderen. Wird der Client als Benutzeroberfläche und der Server als Datenspeicher genutzt, so können beispielsweise mehrere Clients für verschiedene Geräte oder Anwendungsfälle entwickelt und optimiert werden, wodurch die Portabilität der Oberfläche verbessert wird. Die Verteilung auf mehrere Geräte erhöht außerdem die Skalierbarkeit des Gesamtsystems.

2.2 Web APIs

Eine Software kann einen Teil seiner Funktionalität über Schnittstellen externen Programmen zur Verfügung stellen. Solche Application Programming Interfaces (API) ermöglichen die Kommunikation zwischen Programmen durch das parametrisierte Aufrufen der öffentlich gemachten Funktionen einerseits und das Antworten mit dem Ergebnis der Funktion andererseits. Ein großer Vorteil der Verwendung von APIs ist das Verstecken der zugrundeliegenden Komplexität und führt zu Unabhängigkeit von Implementierungsdetails. So ist es z.B. die Aufgabe eines Betriebssystems, Programmen den Zugriff auf Hardwarekomponenten, wie Dateisystem, Netzwerk, Sensoren und Peripheriegeräte, zu abstrahieren und bereitzustellen. Externe Programme werden durch die Benutzung jedoch abhängig von der Schnittstelle, welche daher als Kommunikationsobjekt Vertragscharakter bekommt. Für die meisten Programmiersprachen existieren Programmbibliotheken und Frameworks, die zur Entwicklungszeit grundlegende und immer wieder benötigte Funktionalität bereitstellen, aber besonders Cloudservices und öffentliche APIs haben in den letzten Jahren an Popularität erlangt, da sie mit HTTP eine sprachunabhängige Schnittstelle bieten und die Integration von Anwendungs- und Nutzerdaten ermöglichen. Die vorliegende Arbeit beschränkt sich auf API von verteilten Systemen im Web. Webanwendungen auf Basis von JavaScript-Frameworks werden häufig auch als Fat-Client Anwendungen bezeichnet, da ein beträchtlicher Teil der Programmlogik, sowie die Benutzeroberfläche auf Clientseite ausgeführt wird. Da der Server in solchen Anwendungen als Zugangspunkt und Speicher für Daten dient, sind die APIs geprägt von der Übertragung der Daten zur Anzeige im Client und der Modifikation der Daten auf dem Server. Gemäß den Hauptinteraktionen werden diese APIs auch als CRUD (Create, Read, Update, Delete) APIs bezeichnet.

Abgrenzung zu anderen Web API Ansätzen

Neben REST und GraphQL existieren weitere API-Technologien auf Basis der Client-Server-Architektur wie das Simple Object Access Protocol (SOAP) und Remote Procedure Call (RPC). SOAP verwendet XML zur Kommunikation, welches an vielen Stellen durch das leichtgewichtigere JSON abgelöst wurde. Ziel einer RPC-Kommunikation ist das Aufrufen einer Aktion auf dem Server, wobei die Orientierung an Funktionsnamen zu einer engen Kopplung von Client und Server führt. Die Namensgebung der Funktionen erfordert außerdem Dokumentation und Richtlinien, um einen Überblick über bestehende Funktionen zu erhalten und die Zahl dersel-

ben nicht unnötig zu vergrößern und damit die Komplexität der API zu erhöhen. RPC ist aufgrund seiner leichtgewichtigen Nachrichten für Anwendungen mit hohen Performanceansprüchen oder befehlorientierte APIs geeignet [2, vgl.]. Beide Technologien finden jedoch in der Entwicklung von Single-Page-Apps kaum Verwendung und sollen daher im Folgenden nicht weiter betrachtet werden.

3 Das REST Architekturkonzept

3.1 Entstehung

Das Architekturkonzept des Representational State Transfer wurde erstmals im Jahr 2000 von Roy T. Fielding erwähnt, definiert und mit dem Akronym REST bezeichnet.[9, S. 76] Es enthält eine Reihe von Prinzipien für die Entwicklung von verteilten Systemen im Web mit besonderem Fokus auf den Schnittstellen zwischen den einzelnen Komponenten. Dabei baut REST auf der Client-Server-Architektur auf und hatte durch die Arbeit vib Fielding in den HTTP- und URI-Arbeitsgruppen direkten Einfluss auf die Entwicklung von der beiden Standards, wodurch jene Schnittstellen im Web definiert werden.[vgl. 9, 4, 105f.] REST beschreibt größtenteils die Architektur des Web selbst und ist daher nicht auf APIs ausgelegt oder beschränkt[vgl. 26, S. 1].

3.2 Grundlagen

Ziele von REST

Die Ziele und Vorteile der REST Architektur lassen sich folgendermaßen zusammenfassen:

> REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. [9, S. 105]

Diese Ziele werden erreicht durch eine Reihe von Anforderungen.

Komponenten und Prinzipien

Die REST Architektur baut auf der weit verbreiteten Client-Server Architektur auf und definiert den Begriff der Ressource als Abstraktion für jede Art von Information, die Ziel einer Client-Anfrage ist.[vgl. 9, 88f.] Als Beispiele für Ressourcen nennt Fielding selbst: „a document or image, a temporal service [. . .], a collection of other resources, a non-virtual object (e.g. a person).“ [9, S. 88] Eine Ressource wird durch einen Bezeichner bekannt gemacht und ist damit für einen Client abrufbar. Der Bezeichner bleibt gleich, selbst wenn sich die damit identifizierte Ressource ändert. Die Kommunikation einer Ressource vom Server zum Client erfolgt über eine Repräsentation der Ressource, d.h. ein Datenformat, welches vom Server festgelegt oder durch Content-Negotiation zwischen Client und Server ausgehandelt wird. Somit bleibt der Ursprung der Daten hinter der Serverschnittstelle versteckt. Die Antwort des Servers enthält die Daten der Repräsentation und deren Metadaten. Anfrage und

Tabelle 3.1: REST Data Elements, in Anlehnung an [9, S. 88]

Data Element	Web Equivalent	Example
resource	the intended conceptual target of a hypertext reference	a list of events
resource identifier	URI	http://example.com/events
representation data	HTML, JSON Document, PNG image	{events:[{id:4,title:...},{...}]}
representation metadata	media type, last-modified time, ETag	Content-Type: application/json, Content-Encoding: gzip
control data	if-modified-since, cache-control, request method	GET, Cache-Control: no-cache

Antwort enthalten außerdem Kontrolldaten, z. B. HTTP Request Methods, Status Codes und Header, um die Bedeutung der Nachricht zu vermitteln.[vgl. 9, 90f.]

REST ist zustandslos, sodass jede Anfrage vom Client zum Server alle für deren Verarbeitung notwendigen Informationen beinhalten muss. Ein Client speichert gewöhnlich seinen Zustand, während auf Serverseite keine Zustandsinformationen über mehrere Anfragen hinweg gespeichert werden, wie es zum Beispiel bei Sessions der Fall ist. Aufgrund dieser Eigenschaft wird im Zusammenhang mit REST auch von einer "Client-Cache-Stateless-Server[-Architektur]" [9, S. 83] gesprochen. Hieraus ergeben sich drei Vorteile:[vgl. 9, S. 79]

- Sichtbarkeit: Jede Anfrage kann separat untersucht und gecacht werden, ohne weitere Informationen zu benötigen.
- Zuverlässigkeit: Anfragen können atomar betrachtet und bei Teilfehlern kann ein stabiler Systemzustand leicht wiederhergestellt werden.
- Skalierbarkeit: Da ein Server Zustandsinformationen nur für die Dauer der Verarbeitung einer Anfrage speichert, können Ressourcen schnell wieder freigegeben werden.

Nachteil dieser Anforderung ist eine verringerte Netzwerkperformance, da bei sequenziellen Anfragen an einen Server, Daten zur Clientidentifikation, Authentifizierung oder aus vorangegangene Antworten erneut gesendet werden müssen. Die verringerte Kopplung führt ebenfalls dazu, dass die Kontrolle des Servers über das Verhalten der Clientanwendung reduziert wird.

Clientseitiges Speichern von erhaltenen Daten (Caching) erlaubt die Wiederverwendung früherer Serverantworten für zukünftige, gleiche Anfragen und ist sinnvoll, da die gefühlte Performance durch Verringerung der durchschnittlichen Latenz erhöht wird. Dadurch wird die Effizienz und Skalierbarkeit der Anwendung erhöht, obwohl die Latenz jeder einzelnen Anfrage durch den Cache-Lookup erhöht wird. Je mehr die gespeicherten Daten von den tatsächlichen Daten abweichen, desto mehr

verringert sich die Verlässlichkeit derselben.[vgl. 9, S. 80] GET-Requests werden von Webbrowsern implizit gecacht, wobei Server das gewünschte Caching-Verhalten explizit erlauben und verbieten können.

REST beschreibt drei Typen von Komponenten:[vgl. 9, S. 96]

- User Agent: Dies kann ein Webbrowser bzw. die Benutzeranwendung sein, in jedem Fall jedoch der letztendliche Empfänger der Antwort.

- Origin Server: Die endgültige Quelle der Repräsentation einer Ressource und der letztendliche Empfänger von Request, die zu Modifikationen der Daten führen.

- Intermediary (Zwischenkomponenten): Diese befinden sich zwischen User Agent und Origin Server und können sowohl als Client, als auch als Server agieren. Sie leiten Anfragen und Antworten weiter, können diese aber auch modifizieren. Je nach Anwendungsfall spricht man hier von einem Gateway oder Proxy.

Für die Schnittstellen der Komponenten gelten folgende Grundsätze:[vgl. 9, S. 82]

- Ressourcen müssen durch einen Bezeichner eindeutig identifizierbar sein. Die URI für eine Ressource wird auch als Endpunkt bezeichnet.

- Die Interaktion mit einer Ressource, d.h. das Abfragen und Manipulieren derselben erfolgt über eine Repräsentation der Ressource.

- Nachrichten müssen so selbsterklärend sein, dass sie von der Schnittstelle verstanden und verarbeitet werden können.

- "hypermedia as the engine of application state" [9, S. 82] (HATEOAS): Der Hypertext beschreibt den Zustand der Anwendung und die Möglichkeiten der Veränderung desselben. Folgt der Client einem Link im Hypertext und fordert eine neue Repräsentation an, so ändert er seinen Zustand. Dieses Prinzip ist daher auch namensgebend für die Architektur des Representational State Transfer [vgl. 18, 103f.].

Durch die generischen Client- und Serverschnittstellen, die selbstbeschreibenden Nachrichten und die zustandslose Kommunikation kann eine REST Anwendung leicht von Schichten profitieren. Keine einzelne Komponente benötigt Kenntnis des gesamten Informationsweges, sondern sieht nur die Komponenten, mit denen sie direkt interagiert, sodass zwischen User-Agent und Origin-Server Zwischenkomponenten eingeführt werden können, um spezifische Aufgaben, wie Caching, Kapselung, Load Balancing und Datentransformation, zu übernehmen.[vgl. 9, S. 99] Die so verbesserte Skalierbarkeit des Servers bringt jedoch mehr Datenverarbeitung durch die Zwischenkomponenten und somit höhere Latenzzeiten mit sich.[vgl. 9, S. 83, 98] TODO: a, b, c erklären

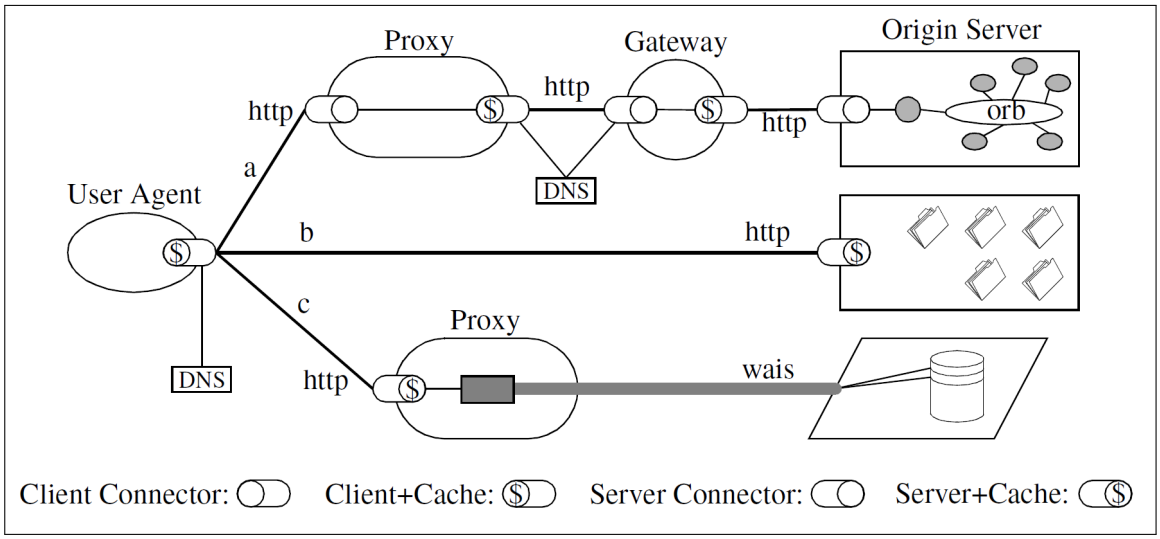

Abbildung 3.1: REST [9, S. 84]

3.3 Implementierung von RESTful APIs

Eine API, die dem REST Architekturstil folgt, wird RESTful genannt. Da die Dissertation von Fielding jedoch einige Implementierungsdetails schuldig bleibt bzw. bewusst „Details der Komponentenimplementierung und Protokollsyntax [ignoriert]“ [9, S. 86], wurde der Begriff RESTful API mit der Zeit aufgeweicht und für APIs verwendet, die grundlegende Anforderungen nicht erfüllen.[vgl. 10] Die REST Architektur ist generell protokollunabhängig, obwohl durch die Arbeit von Fielding an HTTP und URI diese für die Implementierung sehr geeignet sind und auch das erste Anwendungsfeld waren [vgl. 9, S. 109, 116]. Laut Richardson enthalten die Webstandards URI, HTTP und HTML die Randbedingungen der REST-Architektur, sodass eine API dieselben nur richtig gebrauchen muss, um als RESTful zu gelten [vgl. 20].

Richardson Maturity Model

Richardson beschreibt vier Level, um Webservices qualitativ zu klassifizieren und ihre Konformität zum REST-Architekturstil auszudrücken [vgl. 6]. Im Folgenden sind HTTP Anfragen bzw. Antworten exemplarisch und vereinfacht dargestellt.

Level 0 bezeichnet einen Webservice, bei der jede Interaktion mit der selben URI und HTTP Methode (HTTP Verb) erfolgt. Diese Methode ist zumeist *POST* damit Daten sowohl empfangen, als auch gesendet werden können, ohne den Restriktionen des URI Query Strings zu unterliegen. Die Semantik einer Abfrage ist allein durch den Request Body ersichtlich, wodurch die automatische Verarbeitung durch Zwischenkomponenten erschwert wird.

APIs in Level 1 teilen, im Vergleich zu Level 0, die Komplexität eines Endpunktes auf verschiedene Ressourcen auf und gebrauchen URIs um diese zu identifizieren. Ein Beispiel für eine öffentliche API dieser Art ist die Amazon Product Advertising API [1], welche man teilweise ausschließlich mit GET-Requests aufrufen kann. Das Strukturieren der API nach Ressourcen erlaubt eine getrennte Entwicklung der Endpunkte und Zwischenkomponenten, wie Load Balancer, können auf Anfragen

leichter Einfluss nehmen.

```
POST /customer/4/cart HTTP/1.1

{
  addProduct: {
    id: 3
  }
}
```

Abbildung 3.2: Level 1 API POST Request

Level 2 erweitert die Verwendung von Webstandards um die von HTTP definierten Methoden und Response Status Codes. Jede HTTP Methode drückt eine bestimmte Intention aus und ist an semantische Beschränkungen gebunden [vgl. 8, 21ff.].

Tabelle 3.2: Bedeutung oft genutzter HTTP Methoden, in Anlehnung an [8, S. 22]

HTTP Methode	Bedeutung
GET	Anfragen der Repräsentation einer Ressource
POST	Bedeutung kann variieren. Zum Erstellen oder Modifizieren genutzt
PUT	Erstellt eine Ressource bzw. ersetzt eine vorhandene Repräsentation.
PATCH	Modifiziert eine vorhandene Repräsentation.
DELETE	Löscht eine Ressource

Die häufigsten Interaktionen mit Ressourcen werden durch diese Methoden beschrieben. Die zusätzlichen Eigenschaften geben Zwischenkomponenten weitere Möglichkeiten zur Optimierung [vgl. 20]. Ein GET-Request ist als *safe* definiert, d.h. dass die Ressource nicht verändert wird, sodass die Antwort bis zum Empfang eines verändernden Request gecacht werden kann. Die Idempotenz von GET, PUT und DELETE bedeutet, das bei mehrmaliger Ausführung des gleichen Requests der Zustand der Ressource gleich bleibt. Schlägt eine solche Anfrage fehl, was am Response Status Code erkennbar ist, kann diese ohne Nebeneffekte wiederholt werden.

```
DELETE /events/1/booking/3 HTTP/1.1
```

Abbildung 3.3: Level 2 DELETE Request

Der Grundgedanke von Level 3 und dem Einsatz von Hypermedia ist durch das Human Web, die Interaktion von Nutzern mit HTML Dokumenten im Webbrow-

ser, inspiriert. Da der Browser Kenntnis von der Semantik von HTML hat, kann er im Dokument enthaltene Links als solche darstellen, sodass der Benutzer über die Links durch die Anwendung navigieren kann. Haben Clientanwendung und Server ein gemeinsames Hypermedia-Format, so können darin enthaltenen Links und Form-Elemente genutzt werden, um dem Client das Navigieren zu zugehörigen Daten und das Manipulieren der Repräsentation zu ermöglichen. Eine solche API ist damit selbstbeschreibend, benötigt keine externe Dokumentation und kann einige wenige Listenressourcen als Einstiegspunkte veröffentlichen, von denen aus alle weiteren Ressourcen über Links erreichbar sind [vgl. 26, S. 78]. Der Anwendungsfluss ist durch den Server steuerbar, z. B. durch das Anbieten von Links zur nächsten oder vorherigen Ressource, wobei Client und Server entkoppelt bleiben durch das gemeinsame Verständnis des Hypermedia-Formats und URI [vgl. 20]. So schreibt auch Fielding: „A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state.“ [10]

```
HTTP/1.1 200 OK

{
  book: {
    id: 1,
    title: "REST in practice",
    author: "http://api.com/author/10",
    coverImage: "http://api.com/books/1/cover.jpg",
    reviews: "http://api.com/books/1/reviews?first=10",
    relatedBooks: [
      "http://api.com/books/14"
    ],
    next: "http://api.com/books/2",
  }
}
```

Abbildung 3.4: Level 3 Hypermedia JSON Response

Das Richardson Maturity Model erleichtert das Einschätzen der Konformität einer API zum REST Architekturstil. Die Frage, wann eine API als RESTful bezeichnet werden kann, beantwortet es jedoch nicht. Mit Sicherheit kann man sagen, dass eine API auf Level 0 keine REST-API ist und Fowler schreibt sogar, dass Level 3 eine Voraussetzung für REST ist [vgl. 11]. Deutlich wird, dass jedes Level ein anderes Problem löst, doch nicht jede API vor jedem dieser Probleme steht. So könnte eine automatisch generierte API Dokumentation mit *Swagger*[22] eine Alternative zur Selbstdokumentation durch Hypermedia (Level 3) sein.

Repräsentationsformate und Query String Erweiterungen

REST gebietet die Übertragung von Daten mit Hypermedia-Formaten, deren Verständnis für die Kommunikation von Client und Server essenziell ist. Es gibt einige Formate, wie HTML und ATOM [15], welche weit verbreitet sind und Unterstützung für Hypermedia-Elemente wie Links mitbringen. Mit dem vermehrten Einsatz von JavaScript in Webanwendungen ist jedoch die JavaScript Object Notation (JSON) [4] immer beliebter geworden, sodass für die meisten Programmiersprachen mittlerweile Bibliotheken zum Serialisieren und Parsen des JSON-Formats existieren. Während HTML von Browsern verstanden und angezeigt werden kann, ist JSON sehr viel leichtgewichtiger und leichter lesbar. JSON definiert nur wenige Datentypen und enthält „keine standardisierte Darstellung von Links, was primär daran liegt, dass für Links kein spezieller Datentyp in JavaScript existiert.“ [26, S. 89] Um die Probleme bei der Entwicklung von APIs mit JSON zu bewältigen wurden über die Zeit verschiedene Spezialisierungen ausgearbeitet. So definieren die Hypertext Application Language (HAL) [17] und JSON-LD [23] eine Semantik für Verlinkungen und Einbeziehen zugehöriger Daten in JSON und JSON Schema [21] ermöglicht das Validieren von JSON-Dokumenten. Die OpenAPI [19] und JSON:API [16] Spezifikationen gehen noch weiter und beschreiben sowohl Abfragesyntax im Query String der URI, als auch die Struktur und Semantik der Antwort. Eine JSON:API-Abfrage und Antwort könnte folgendermaßen aussehen:

```
1 GET /customers/2?include=car&fields[people]=name,age&fields[car]=brand HTTP/1.1
2 Accept: application/vnd.api+json
```

Abbildung 3.5: Eine JSON:API Abfrage

An diesem Beispiel wird deutlich, welche Möglichkeiten solche Spezifikationen bieten. Die Vorteile sind außerdem, dass allgemeine Bibliotheken und Anwendungen für APIs nach einer Spezifikation entwickelt und in Entwicklerwerkzeuge integriert werden können [vgl. 24] Je mehr Abfragelogik jedoch in den URI Query String verlagert wird, desto mehr Rechenarbeit muss der Server leisten, um die Antwort zu erzeugen und HTTP Caching kann aufgrund der speziellen Antwort weniger genutzt werden.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "http://api.com/customers/2"
  },
  "data": [{
    "type": "people",
    "id": "2",
    "attributes": {
      "name": "Franz Meier",
      "age": 34
    }
  }],
  "included": [{
    "type": "car",
    "id": "48",
    "attributes": {
      "brand": "VW"
    }
  }]
}
```

Abbildung 3.6: Die JSON:API Antwort zu 3.5

4 GraphQL

4.1 Entwicklung

GraphQL ist eine Abfragesprache für APIs und Laufzeitumgebung, um auf Abfragen zu antworten [13]. Seit 2012 wurde GraphQL von Facebook entwickelt und eingesetzt. Im Jahr 2015 veröffentlichte Facebook GraphQL auf Github, wo die aktuelle Spezifikation von der GraphQL Foundation mittlerweile eigenständig weiterentwickelt wird und 2018 die letzte Version veröffentlicht wurde. Ziel der Entwicklung war es, das Datenmodell in einer Client-Server-Anwendung durch ein Schema zu beschreiben, vom Server bereitzustellen und darauf aufbauend Abfragen zu formulieren und auszuwerten [vgl. 7]. Ein Client sollte alle für eine Ansicht nötigen Daten mit einer einzigen Anfrage vom Server präzise anfordern können und daraufhin exakt diese Daten erhalten. Das Overfetching und Underfetching von REST-APIs, d.h. dass ein Endpunkt mehr oder weniger Daten sendet, als benötigt werden, kann so verhindert werden, was sich besonders auf den Datenverkehr in Mobilfunknetzwerken auswirken soll. Ein weiterer Vorteil ist die Entkopplung von den Datenbedürfnissen des Clients von den Endpunkten des Servers, wodurch getrennte Weiterentwicklung und die Wiederverwendbarkeit der API sichergestellt wird. Die Bestandteile von GraphQL sind: „a type system, query language and execution semantics, static validation, and type introspection.“ [14] Diese sollen im Folgenden erörtert werden.

4.2 Spezifikation und Funktionsweise

Typsystem

Das Schema beschreibt die API eines GraphQL-Servers durch die GraphQL Schema Definition Language (SDL). Teil der SDL ist das Typsystem, womit ausgedrückt werden kann, welche Datenobjekte die API zur Verfügung stellt [vgl. 14]. Einstiegspunkt in das Schema bilden die drei Root-Types, welche nach Konvention *query*, *mutation* und *subscription* genannt werden. Alle Felder dieser drei Objekte stellen die Operationen dar, welche die API anbietet. Unter *query* werden alle Abfragetypen gesammelt, welche in einer REST-API mit GET angefragt werden würden, während zu *mutation* alle Modifikationsoperationen gehören, entsprechend PUT, POST, PATCH, DELETE. *Subscription* ermöglicht es Datenströme zu senden. Jeder Typ im Schema kann eine beliebige Zahl an Feldern enthalten, welche selbst Objekte, skalare Datentypen oder Arrays sind. Die von GraphQL definierten skalaren Datentypen dieser Felder sind *string*, *int*, *float*, *boolean* und *id* [vgl. 7]. Diese können jedoch um neue Datentypen erweitert werden. Das Typsystem erlaubt es außerdem Typen zu verbinden (Union Type) und abstrakte Typen (Interface) zu definieren, welche von konkreten Typen implementiert werden. Jedem Feld können benannte Argumente übergeben werden, die Einfluss auf die Auswertung des Feldes haben. *Null* ist ein erlaubter Rückgabewert für alle Felder, solange diese nicht mit einem Ausrufezeichen als non-nullable gekennzeichnet werden [vgl. 7]. In Abbildung 4.1 ist

der Root-Query Typ mit dem Feld *events* dargestellt, dem ein optionales Argument *amount* mit dem Standardwert 100 übergeben wird. Der Rückgabewert des Feldes ist ein Array, welches Eventobjekte enthält.

```
type Query {
    events(amount: Int = 100): [Event!]!
    users: [User!]!
}

type Event {
  id: ID!
  title: String!
  description: String!
  price: Float!
  date: String
  creator: User!
}

type User {
  id: ID!
  email: String!
  password: String
  createdEvents: [Event!]
}
```

Abbildung 4.1: GraphQL Beispielschema

- Query Syntax
  - Abbildung Beispiel Query
  - GraphQL Abfrage beschreibt deklarativ welche Daten erwartet werden
  - Antwort ist üblicherweise JSON mit der gleichen Struktur
  - Abfragen können geschachtelt werden
  - bei Objekttyp muss mind 1 Feld angegeben werden
  - Fragmente
    - Abbildung Beispiel Fragment
    - sammlung von Feldern von einem Typ
    - verhindert Dopplung von mehreren Feldern in Abfrage
    - ermöglicht typbasierte Feldselektion
  - mehrere gleiche queries mit alias möglich
  - query können variablen in map übergeben werden, no manual string construction escaping
- Introspektion

  - Spezialfelder beginnend mit doppelt Unterstrich
  - __schema, __typename
  - Metadaten über GraphQL Schemas
  - Sinn ist Nutzung durch Entwicklungstools
  - ermöglicht statische Validierung: GraphQL Abfrage kann zu Entwicklungszeit geprüft werden

- Referenzen werden unsichtbar, da von GraphQL Server automatisch aufgelöst (Performance beachten!)
- ein Endpunkt (kein Nutzen von URIs)
- jede Abfrage mit POST (kein Nutzen von HTTP Methoden), query in query feld in json payload
- immer 200 OK status code
- keine URI nutzung -> level 0 mmr
- application/graphql media type
- nicht im Spec festgelegt (da netzwerkunabhängig), aber praxis
- Semantik der Abfrage von Server ausgewertet (welche der CRUD Operationen)

```
HTTP/1.1 200 OK

{
  book: {
    id: 1,
    title: "REST in practice",
    author: "http://api.com/author/10",
    coverImage: "http://api.com/books/1/cover.jpg",
    reviews: "http://api.com/books/1/reviews?first=10",
    relatedBooks: [
      "http://api.com/books/14"
    ],
    next: "http://api.com/books/2",
  }
}
```

Abbildung 4.2: Level 3 Hypermedia JSON Response

4.3 Server-Execution

- request empfangen
- lexing: zeichenstream in graphql tokens umgewandelt
- parsing: semantik korrekt, struktur
- lexer und parser algorithmus nicht im spec festgelegt
- ermöglicht spezifische optimierungen
- server validiert query mit schema, inputvalidierung
- validierungsregeln im spec oder selbst festgelegt:
- validierung erfolgt depth first
- query Feld für Feld parallel ausgeführt
- jedes Feld hat eigenen resolver
- resolver für jedes Feld aufgerufen
- resolver erfüllen wert nach schema
- breadth-first (Abbildung), 3 und 4 parallel
- parent result an child resolver weitergegeben
- Felder in manchen Implementierungen als trivial automatisch aufgelöst
- rückgabewert der Felder an Schema geprüft, evtl konvertiert
- JSON kennt nur number, nicht int und float
- graphql sendet nie etwas was nicht im schema ist

5 Vergleich

5.1 Testumgebung

Für den Vergleich der beiden API Ansätze wurde eine Single-Page Anwendung mit dem Vue.js Framework als Client und jeweils eine REST API und GraphQL API mit der Node.js Serverlaufzeitumgebung entwickelt. Als Datenquelle dient zum einen eine MongoDB Datenbank, sowie Dateien im JSON Format. Durch die Verteilung der jeweiligen Softwarekomponenten auf physikalisch unterschiedliche Rechner lassen sich mehrere Anwendungsfälle abbilden. So wurden die Clientanwendung, die Serveranwendungen und die Datenbank in verschiedenen Kombinationen lokal und von Cloudumgebungen (Azure, Heroku, MongoDB Atlas) gehostet und ausgeführt. Die JSON Dateien als Alternative zur Datenbank ermöglichen es, das Kontingent der Clouddatenbank nicht unnötig zu belasten und den durch Datenbankabfragen entstehenden Overhead, auf die deutlich schnellere Dateilesezeit zu minimieren. Bei den Messungen wurde auf das Angleichen der Testbedingungen geachtet, besonders die im Zusammenhang mit Clouddiensten gemessenen Werte sind jedoch stark von der je nach Tageszeit unterschiedlichen, unbekannten Belastung des Dienstes abhängig.

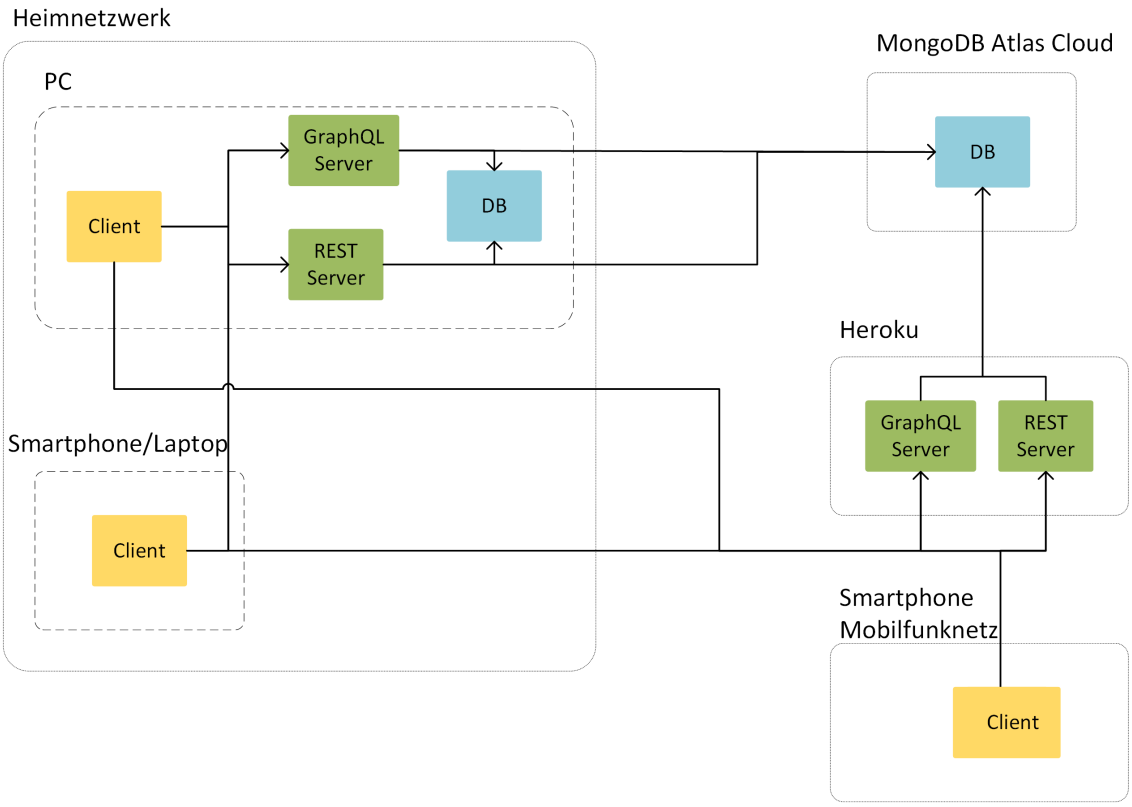

Abbildung 5.1: Architektur der Testumgebung

5.2 Abfrageflexibilität

- GraphQL
  - unterscheidet Feldselektion, Sortieren, Filtern, Optionen
  - keine Syntax für Sortieren/Filtern, aber Möglichkeit über Feldargumente
  - an Query sind Performanceprobleme evtl. schwer erkennbar (siehe Tools)
- REST
  - fields=“ ” für Feldselektion
  - field=“ ” zum Filtern
  - sort=[field],sort-dir=desc zum Sortieren
  - option=“ ” für Optionen
  - includes empfohlen bei JSON:API, aufbrechen von HATEOAS?
- Upload ist schwierig für GraphQL (serialization), REST kann multipart/form-data header nutzen

Bei der Entwicklung von Client-Anwendungen ist die Flexibilität der genutzten API ein entscheidender Einflussfaktor. Dabei steht meistens die Flexibilität für das Generieren der Abfragen zum Empfangen von Daten im Vordergrund, aber auch Operationen zum Modifizieren von Ressourcen müssen betrachtet werden.

GraphQL wurde als Abfragesprache entwickelt. Eines der Entwicklungsziele ist es, dass eine Clientanwendung mit einer einzigen Abfrage alle für eine Ansicht nötigen Daten erhalten kann. Der Server gewährt Zugriff auf das Datenschema, welches alle verfügbaren Daten und deren Relationen darstellt, sowie die Einstiegspunkte (Root Query), auf deren Basis ein Client seine Anfrage aufbauen kann. Das Schema stellt den Rahmen für alle Abfragen dar. Alle Daten, die der Server zur Verfügung stellt, werden durch Typen und deren Felder repräsentiert. Jedes Feld, welches eine Clientanwendung erhalten möchte, muss in der Query angegeben werden. Es existiert keine Möglichkeit, alle Felder eines Typs zu erhalten, ohne jedes davon anzugeben, z. B. durch Wildcard Zeichen. Der Client ist damit eng an das Schema gekoppelt, erhält aber auch nur die Daten, die tatsächlich angefragt wurden. Die Möglichkeit eines GraphQL-Servers, beliebige Abfragen innerhalb des Schemas auszuwerten, eignet sich besonders für öffentliche APIs, bei denen dem API-Entwickler die individuellen Anforderungen und damit die Abfragen verschiedenster Anwendungen im Voraus nicht bekannt sind. Die Relationen zwischen den Datentypen werden durch das Schema festgelegt. Eine Anfrage kann jedoch beliebig viele, unabhängige Queries beinhalten, z. B. um verschiedene Daten zu erhalten, die in keiner Beziehung zueinander stehen oder deren Beziehung durch das Schema nicht dargestellt wird. Der Server kann für jedes Feld Parameter anbieten, welche die Auswertung des Feldwertes verändern. Über solche Feldargumente können Optionen, die Geschäftslogik darstellen, aber auch allgemeine Abfragefunktionen, wie z. B. Sortieren [vgl. 12] und Filtern [vgl. 5] zur Verfügung gestellt werden. Einem Feld „Breite“ könnte beispielsweise die Maßeinheit als Option übergeben werden.

Während das Anfragen von Daten sehr viel Flexibilität für Clients bietet, ist das Modifizieren von Ressourcen nach Konvention nur über einzelne Mutations möglich. Üblicherweise wird für jede Operation (Erstellen, Verändern und Löschen) einer Ressource eine separate Mutation zur Verfügung gestellt, welche durch den Namen eindeutig identifiziert wird. Die Antwort auf eine Mutation kann wieder ein Objekt sein und genutzt werden, um das Resultat der Operation zu übermitteln oder den neuen Stand der Ressource abzufragen.

In einer REST API ist jede Ressource über einen separaten Endpunkt erreichbar. Das Veröffentlichen einer neuen Ressource geschieht durch das Einführen eines neuen Endpunktes. Ein Endpunkt kann alle Methoden zum Abfragen und Ändern einer Ressource bereitstellen und durch die HTTP Verben semantisch unterscheiden. Die Antwort auf einen GET Request an einen Endpunkt enthält alle Felder der Ressource bzw. die komplette Repräsentation. In Beziehung stehende Daten, die nicht Teil einer Ressource sind, müssen über mehrere Requests vom Server angefragt werden. Der Grundgedanke von HATEOAS ist, dass diese verbundenen Ressourcen Verweise aufeinander enthalten, wodurch ihre Beziehung dargestellt wird. Das Zusammenbauen von Requests anhand der Links in erhaltenen Ressourcen ermöglicht große Flexibilität, erfordert jedoch, dass Daten erst ausgewertet werden müssen und danach erst neue Anfragen abgesetzt werden können. Dadurch ist das Datenmodell nicht durch ein Schema vorgegeben, sondern kann zur Laufzeit geändert und explorativ genutzt werden, da es selbstdokumentierend ist. Oftmals ist bei der Entwicklung von Clientanwendungen die erforderliche Datenstruktur jedoch bekannt und ist es wünschenswert, möglichst alle benötigten Informationen von wenigen Endpunkten zu erhalten, um die Performance zu steigern. Dazu gibt es verschiedene Ansätze, wie der Query-String einer URI genutzt werden kann, um in Beziehung stehende Daten mit einem einzigen Request von einem Endpunkt zu erhalten. Weder REST noch HTTP spezifizieren Syntax und Semantik des Query Strings bzw. betrachten diese als Implementierungsdetail. Query-Parameter und Matrixparameter werden häufig gebraucht, um die Möglichkeiten von Abfragesprachen für REST zu implementieren. Sie ermöglichen z. B. die gewünschten Felder des angefragten Objektes, ein Feld nach dem sortiert oder ein Feld und Wert nach dem gefiltert werden soll, anzugeben. Somit können Clientanwendungen die Antwort je nach ihren Anforderungen besser beeinflussen. Abhängig davon wie unterschiedlich diese sind, muss die Entscheidung über die Erweiterung der Parameteroptionen oder das Einführen eines separaten Endpunktes getroffen werden. Über sogenannte „includes“ lassen sich bei JSON:API verbundene Objekte in die Antwort einbeziehen. Includes sind außerdem eine effektive Lösung für das N+1 Problem. Diese speziellen Auswertungen des Query-Strings müssen durch den REST Server angeboten werden und sind aus Entwurfssicht ein Tausch von erhöhter Abfragekomplexität und Clientflexibilität gegen Serverperformance.

Da GraphQL für die Übertragung von Daten in Textform konzipiert wurde, müssen Informationen als Text vorliegen oder serialisiert werden. Das kann besonders beim Hochladen von Binärdaten, wie Audio- oder Videodateien, zu sehr großen Datenmengen führen. Eine REST API kann Binärdaten mit dem MediaType „multipart/form-data“ empfangen und verarbeiten. Für eine GraphQL API muss entweder ein Dateiupload-Service vorgeschaltet sein und nur der Link übertragen oder eine Bibliothek, wie

„graphql-upload“, verwendet werden.

5.3 Weiterentwicklung und Versionierung

- GraphQL
  - neue Felder hinzufügen ohne existierende Queries zu beeinflussen
  - neue Felder nicht automatisch gesendet
  - Felder als deprecated markieren → Tool kann Entwickler warnen
  - Monitoring auf Feldlevel möglich
- REST
  - Monitoring: werden sparse fieldsets oder includes verwendet können genutzte Felder aufgezeichnet werden
  - neue Version, neuer Endpunkt example.com/v2/contacts/...
  - für kleine Veränderungen ungeeignet
  - Hinzufügen von Ressourcen → keine Versionierung notwendig
  - Semantische Versionierung
  - Hinzufügen/Umordnen von einzelnen Elementen sollte Client akzeptieren
  - Versionierung über Accept-Header, URI oder query string

Langlebige Software befindet sich in ständiger Entwicklung. Ein Server muss auf Veränderungen des der API zugrunde liegenden Datenmodells reagieren können und diese gegebenenfalls Clients zugänglich machen. Neue Anforderungen an die API erfordern das Erweitern der Abfragemöglichkeiten oder das Verringern derselben, wenn alte Funktionalität nicht mehr unterstützt oder durch neue ersetzt werden soll. Da jede API einen Vertrag zwischen Server und Client darstellt, sollten Veränderungen einem festgelegten Ablauf und Kommunikationskanal folgen, um sukzessive Migration zu gewährleisten und Abfragefehler zu vermeiden. Zu diesem Zweck wird in der Softwareentwicklung häufig Versionierung verwendet, bei der durch eine Zahl oder Zahlenkombination ein bestimmter Stand der Software bereitgestellt oder abgefragt wird.

Um eine REST API um zusätzliche Ressourcen zu erweitern, ist keine Versionierung notwendig. Stattdessen wird ein neuer Endpunkt hinzugefügt und über die Verlinkung von anderen Ressourcen oder als Einstiegspunkt in der Dokumentation bekanntgemacht. Eine API Clientanwendung sollte so entwickelt werden, dass eine Antwort ausgewertet werden kann, solange sie semantisch eindeutig verständlich ist (Postel'sches Gesetz). Das Umordnen und Hinzufügen von Elementen innerhalb einer Antwort sollte somit nicht zu Fehlern führen und die Validierung demgegenüber tolerant sein. Normalerweise muss erst versioniert werden, wenn keine Rückwärtskompatibilität mehr gewährleistet werden kann. Eine neue Version einer Ressource kann über mehrere Wege bereitgestellt und angefragt werden.

1. Jeder Request kann einen Accept-Header enthalten, der zur Identifikation der zu sendenden Repräsentation genutzt werden kann. Sendet ein Client z. B. einen Accept-Header mit dem Wert application/vnd.format.v2+json, so kann der Server durch Content-Negotiation anhand des „v2“ die zu sendende Version ermitteln.

2. Die URI als Identifikationsmittel kann ebenfalls genutzt werden, um verschiedene Versionen einer Ressource kenntlich zu machen. Die Anfrage an example.com/v2/events entspricht dem Erstellen eines neuen Endpunktes. Der Server muss nun auf einem weiteren Endpunkt antworten können, die Komplexität jedes einzelnen Endpunktes wird jedoch nicht erhöht.

3. Für kleine Änderungen, wie das Ändern oder Entfernen eines einzelnen Feldes, ist normalerweise kein neuer Endpunkt nötig. Würde für jede dieser verhältnismäßig kleinen Änderungen ein neuer Endpunkt erstellt werden, ginge schnell die Übersichtlichkeit über die Menge der Endpunkte und deren Unterschiede verloren. Die URI der Ressource kann für verschiedene Versionen genutzt werden und stattdessen die Versionsinformationen im Query String transportiert werden (z. B. example.com/events?version=2). Diese Variante teilt den gleichen Nachteil, den alle Query String Optionen haben, dass der Serverhandler für diesen Endpunkt komplexer und die Performance verringert wird. Es ist denkbar, dass nicht nur jede einzelne Option geprüft werden muss, sondern dass die Option für die Version der Ressource auch Auswirkung auf andere Optionen hat, sodass der Server einen Entscheidungsbaum auswerten muss, bevor eine Antwort gesendet werden kann.

Wurde eine neue Version einer Ressource zur Verfügung gestellt, muss diese auch erreichbar sein. Da die Änderungen von Variante 2 und 3 sich allein auf die URI auswirken, müssen in verknüpften Ressourcen die Links zur neuen Version aktualisiert werden. Da Header in einem Link nicht repräsentiert werden können, ist es bei der ersten Variante notwendig, dass der Client über die neue Version informiert wird und allen Anfragen an die Ressource den Accept-Header hinzufügt.

Beim Entfernen von Funktionalität ist es sinnvoll, die aktuelle Nutzung derselben durch Monitoring festzustellen und notwendig, die geplanten Änderungen zu kommunizieren. Da normalerweise die gesamte Repräsentation einer Ressource als Antwort gesendet wird, kann nur die Nutzung des Endpunktes und nicht die tatsächliche Notwendigkeit einzelner Teile der Ressource bzw. Felder ermittelt werden. Das ist jedoch möglich, wenn bei jeder Anfrage mittels Includes die gewünschten Felder angegeben werden müssen. Sollen einzelne Elemente einer Ressource nicht mehr gesendet werden, so können diese zeitweise den für den Datentyp üblichen Nullwert enthalten, bevor sie endgültig entfernt werden. Wird ein Endpunkt der API entfernt, so sollte der Server mit dem entsprechenden HTTP Status Code (410 Gone) antworten und alle Links in verknüpften Ressourcen müssen gelöscht werden. Ist eine Ressource lediglich unter einem anderen Endpunkt erreichbar, so kann der Code 310 „Moved Permanently“ den Client automatisch zur neuen URI weiterleiten.

Erklärtes Ziel von GraphQL ist es, Versionierung zu verhindern und stattdessen die API kontinuierlich, abwärtskompatibel weiterzuentwickeln. Da jedes Feld, welches ein Client erhalten möchte, in der Query explizit angegeben werden muss, ist

detailreiches Monitoring bis auf Feldebene möglich. Im Datenmodell neu eingeführte Felder beeinflussen bestehende Abfragen nicht, können aber jederzeit in diese aufgenommen werden. Felder, die zukünftig aus der API entfernt werden sollen, können mit isDeprecated als veraltet gekennzeichnet und ein Grund als Metainformation angegeben werden. Diese Kennzeichnung beeinflusst die Abfrage des Feldes nicht, kann aber per Introspektion zur Entwicklungszeit von Tools abgefragt und dem Entwickler als Warnung angezeigt werden. Enthält eine Abfrage Felder, die im Schema nicht mehr enthalten sind, schlägt die Abfrage fehl. Änderungen in der Auswertungslogik einzelner Felder können über optionale Feldargumente bereitgestellt werden. Auch diese beeinflussen bestehende Abfragen ohne diese Argumente nicht. Für Feldargumente und Input Typen existiert zur Zeit noch keine Möglichkeit, diese als veraltet zu kennzeichnen, ist aber für den nächsten Release der GraphQL Spezifikation geplant. Ein GraphQL Server stellt ein Schema und damit alle darin enthaltenen Abfragen unter einer URI bereit. Ergibt sich die Notwendigkeit, eine neue Version unter einer anderen URI zu veröffentlichen, so können Abfragen die Schemata nicht verbinden und werden vom Server nur im Kontext eines Schemas ausgewertet.

5.4 Performance

Zum Vergleich der Abfrageperformance von GraphQL und REST wurden jeweils nacheinander eine Reihe von Abfragen durchgeführt und gemessen, um die Charakteristik der Daten zu erforschen. Die kostenfreie Laufzeitumgebung von Heroku (Dyno) unterliegt einem monatlichen Stundenlimit und wird nach 30 Minuten, in denen keine Anfragen gestellt werden, in den Schlafzustand versetzt. Erhält eine schlafende Dyno eine Anfrage, so wird sie nach einer kurzen Verzögerung wieder aktiv. TODO Zitat Heroku Diese anfängliche Verzögerungszeit ist nach der angegebenen 30 minütigen Pause, in geringerem Maße aber auch nach nur wenigen Sekunden Inaktivität zu beobachten.5.2 Diese anfänglichen Ausreißer sind unabhängig von der verwendeten API, wurden aber aufgrund der hohen Schwankungen in den weiteren Untersuchungen nicht einbezogen. Je nach für die Abfrage verwendetem Gerät unterscheiden sich die Abfragezeiten deutlich.

5.5 Datenaufkommen/Netzwerklast

- Transfer, Verarbeiten und Speichern unnötiger Daten (Felder) sollte vermieden werden
- GraphQL
  - automatisch kleinstmöglicher Request
  - Query muss an Server gesendet werden
- REST
  - standardmäßig gesamte Repräsentation
  - viele APIs bieten Feldselektion an (Partials)

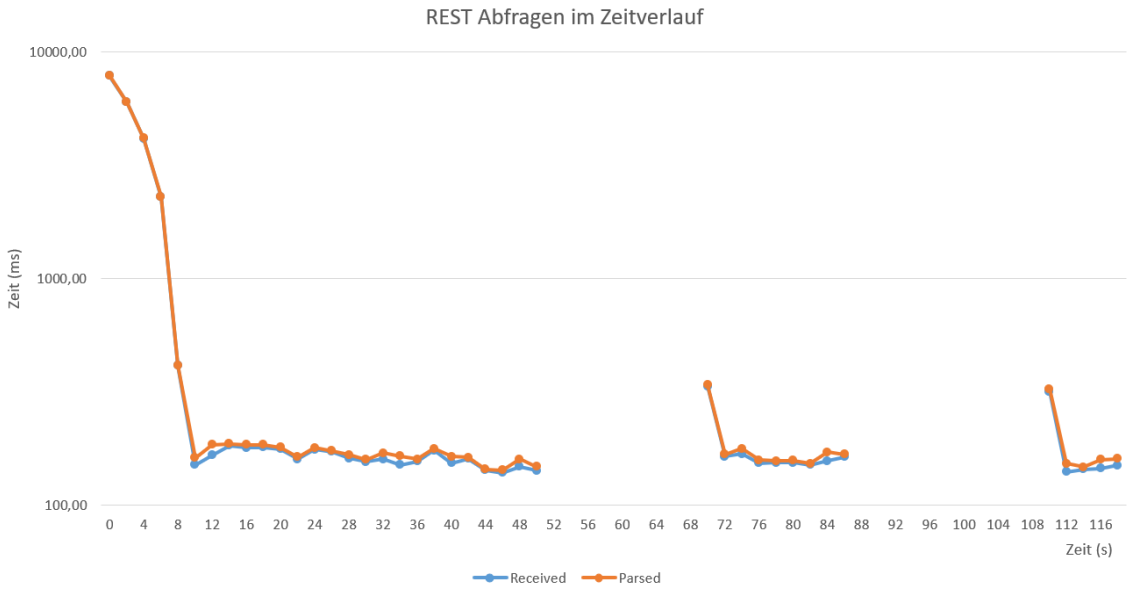

Abbildung 5.2: Abfragen im Zeitverlauf

```
1 const startTime = performance.now();
2 const response = await fetch('http://localhost:5000/events');
3 const received = performance.now() - startTime;
4 const data = await response.json();
5 const parsed = performance.now() - startTime;
```

Abbildung 5.3: Messung von Received- und Parsed-Zeit, Vereinfachte Darstellung aus *EventFetch.vue* Z. 92f.

  - query string enthält Feldselektion nach bestimmter Syntax
  - je komplexer, desto mehr Daten
  - jede Ressource ist extra Endpunkt
  - für mehrere Ressourcen müssen mehrere Requests gemacht werden, n+1 Problem
  - includes beziehen verbundene Daten in Response ein → ein Request für mehrere Ressourcen

- GraphQL und REST Partials unterscheiden Objekt und Array nicht → Wissen über API notwendig um Performance einzuschätzen

- mehr Daten um Request genauer zu machen, sinnvoll um deutlich weniger Daten als Response zu erhalten

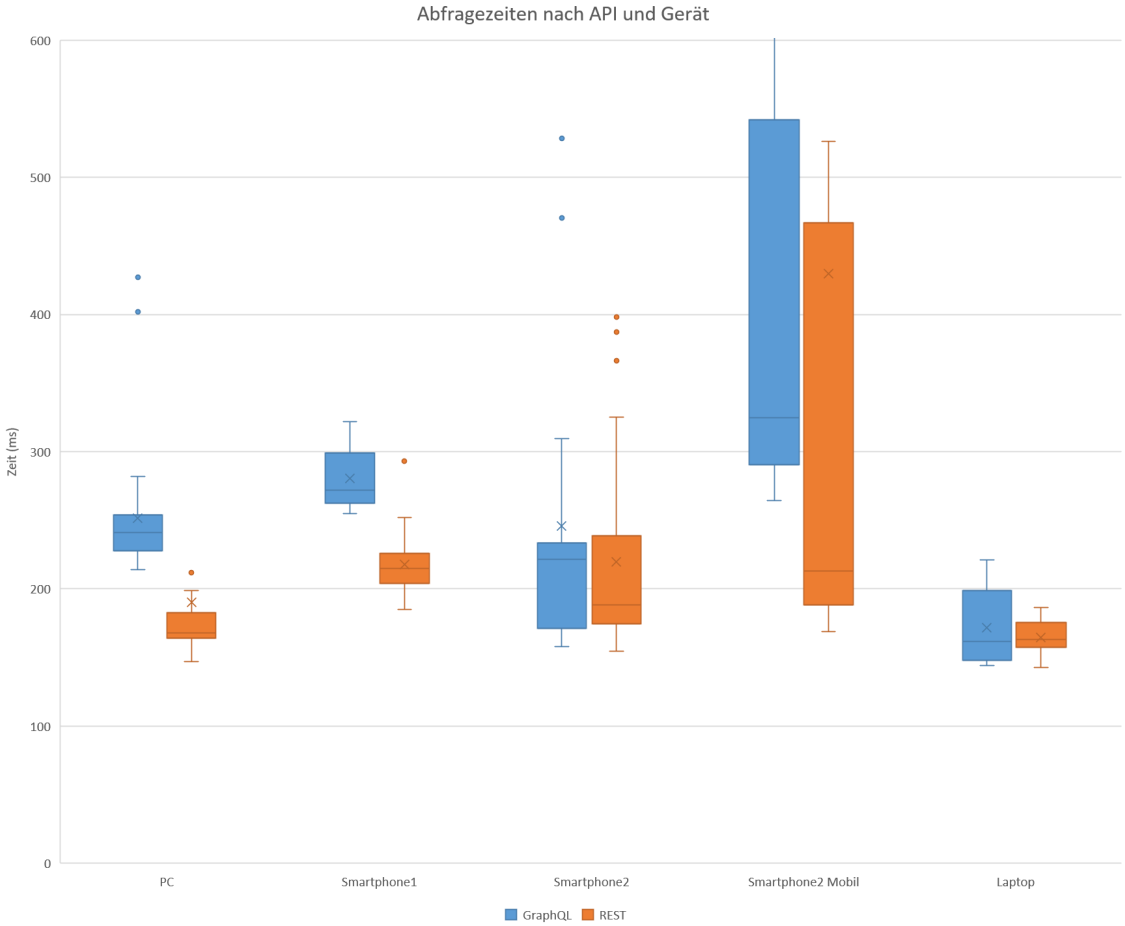

Abbildung 5.4: Abfragen nach Gerät

5.6 Caching

- Flexibilität gegen Caching: je spezieller die Abfrage, desto schwieriger (weniger sinnvoll) caching
- nicht Antwort direkt cachen (HTTP), sondern manuell Objekte anhand ID cachen (JavaScript)
- REST
  - Browser HTTP caching automatisch genutzt
  - Caching basierend auf Endpunkt
  - URL ist cache ID für die Ressource
  - ermöglicht HTTP cache proxies
  - je spezieller der query string, desto weniger cache Treffer
- GraphQL
  - POST response wird normalerweise nicht gecacht

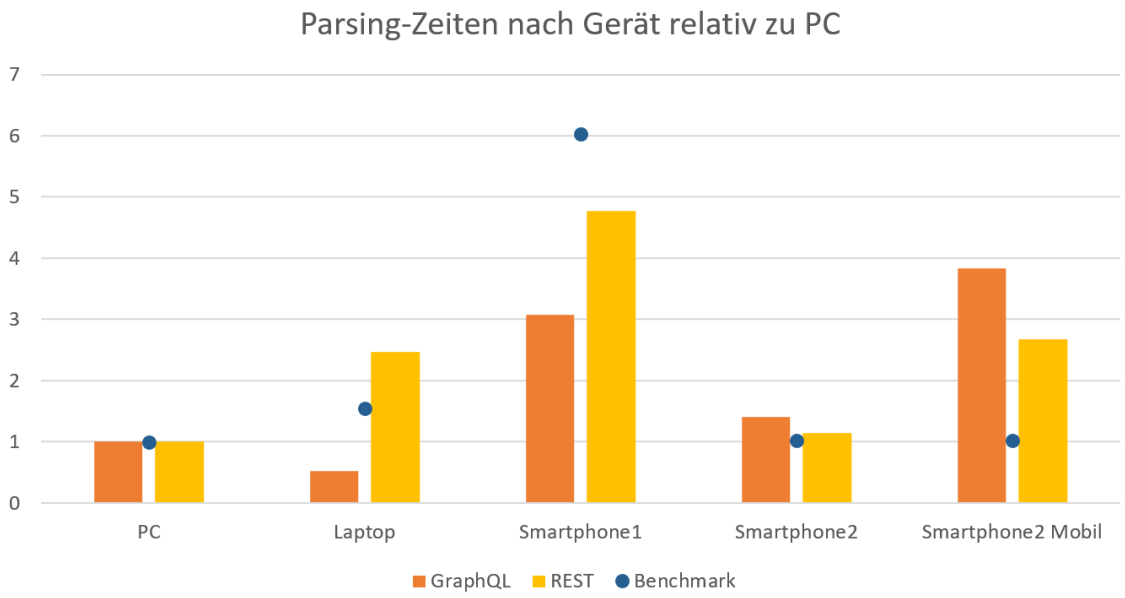

Abbildung 5.5: Parsingzeiten nach Gerät

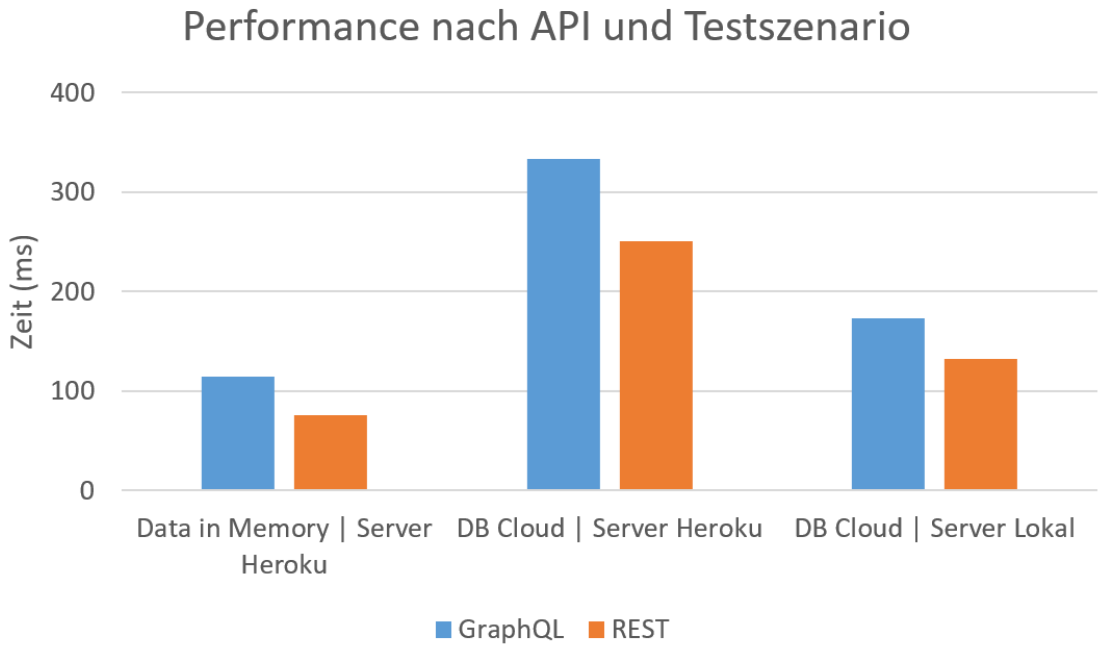

Abbildung 5.6: Abfrageperformance nach Testszenario

- standardmäßig keine ID für caching vorhanden. Empfehlung: API sollte ID pro Objekt bereitstellen
- globale ID über alle Tabellen/Backendservices nötig, mit schema typ verbinden
- einzelne queries cachen aufgrund verschachtelung nicht sinnvoll, objekte aus query extrahieren

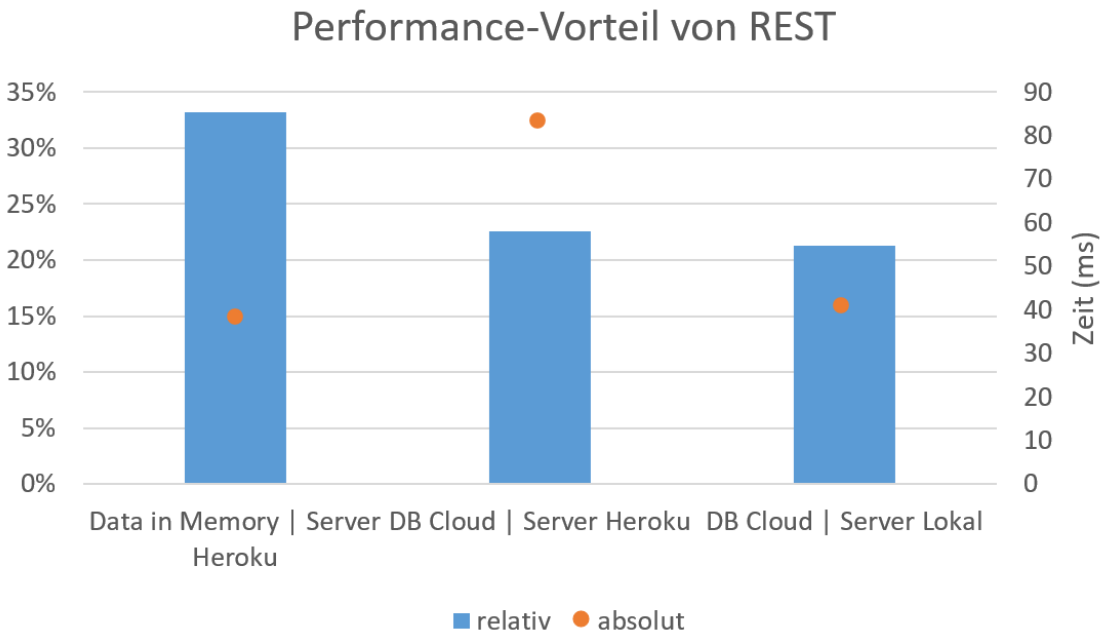

Abbildung 5.7: Performancedifferenz GraphQL-REST

5.7 Batching/Deduping

- GraphQL queries können parallel ausgewertet werden, Mutations nicht
- Dataloader verhindert gleiche Daten mehrmals Anforderungen
- wartet auf alle Resolver, minimale Anzahl requests ausgeführt

5.8 Fehlerbehandlung

- GraphQL gibt error Feld zurück
- 200 OK auch bei Fehler

5.9 Sicherheit

- queries können komplex werden und server stark belasten
- apollo safe listing: tatsächlich genutzte Queries zur Entwicklungszeit extrahieren und whitelisten
- nur query id schicken, ast kann vorher berechnet werden -> schnellere ausführung, nicht möglich bei public api
- angriffe und missbrauch vermeiden

5.10 Kosten

5.11 Lernkurve, Fehlersuche, Community

5.12 Bibliotheken und Tools

- GraphQL
  - offizielle Spezifikation vorhanden
  - Referenzimplementierung in JavaScript
  - Zusatztools von Facebook (Dataloader, Relay)
  - Tool kann Abfragekomplexität zu Entwicklungszeit ermitteln (mit vergangenen Messwerten)
  - Introspektion hilft, statische Validierung (type checking)
  - Apollo Client/server: Laufzeitanalyse, graphnutzung und graphänderungen tracken
- REST: fehlende Standards (in Bezug auf linkelemente) macht die Entwicklung von Tools schwierig

6 Fazit

6.1 Zusammenfassung

6.2 Kombinierte Verwendung von GraphQL und REST

6.3 Ausblick

Literaturverzeichnis

[1] Amazon Web Services, Inc., *Requests - Product Advertising API*, 2019. Adresse: https://docs.aws.amazon.com/en_pv/AWSECommerceService/latest/DG/CHAP_MakingRequestsandUnderstandingResponses.html (Abruf am 15. 11. 2019).

[2] N. Barbettini. (März 2018). Nate Barbettini – API Throwdown: RPC vs REST vs GraphQL, Iterate 2018. Veröffentlicht von OktaDev, Adresse: https://www.youtube.com/watch?v=IvsANO0qZEg (Abruf am 17. 11. 2019).

[3] G. Bengel, *Grundkurs Verteilte Systeme*, fourth. Springer Vieweg, 2014, S. 355, ISBN: 978-3-8348-1670-2. DOI: 10.1007/978-3-8348-2150-8.

[4] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, 2017.

[5] L. Byron, *Proposal: Basic expression language for filter inside GraphQL*, 2018-10-02. Adresse: https://github.com/graphql/graphql-spec/issues/271#issuecomment-426167175 (Abruf am 06. 12. 2019).

[6] J. Desrosiers, *The Hypermedia Maturity Model | 8th Light*, 2018-05-30. Adresse: https://8thlight.com/blog/jason-desrosiers/2018/05/30/the-hypermedia-maturity-model.html (Abruf am 11. 11. 2019).

[7] Facebook, Inc., *GraphQL*, 2018. Adresse: https://graphql.github.io/graphql-spec/June2018/# (Abruf am 10. 08. 2019).

[8] R. T. Fielding und J. F. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, 2014.

[9] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures“, Doctoral dissertation, University of California, Irvine, 2000. Adresse: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (Abruf am 25. 10. 2019).

[10] ——, *REST APIs must be hypertext-driven*, 2008. Adresse: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (Abruf am 25. 11. 2019).

[11] M. Fowler, *Richardson Maturity Model*, 2010-03-18. Adresse: https://www.martinfowler.com/articles/richardsonMaturityModel.html (Abruf am 10. 11. 2019).

[12] I. Goncharov, *Trouble with graphql orderBy*, 2018-03-23. Adresse: https://github.com/graphql/graphql-spec/issues/427#issuecomment-375764325 (Abruf am 06. 12. 2019).

[13] GraphQL Foundation, *GraphQL | A query language for your API*, 2019. Adresse: https://graphql.org/ (Abruf am 09. 08. 2019).

[14] ——, *graphql/graphql-spec*, 2019. Adresse: https://github.com/graphql/graphql-spec (Abruf am 10. 08. 2019).

[15] J. Gregorio und B. de hOra, *The Atom Publishing Protocol*, 2007.

[16] Y. Katz, D. Gebhardt und G. Sullice, *JSON:API - Latest Specification (v1.0)*, 2015-05-29. Adresse: `https://jsonapi.org/format/` (Abruf am 15. 11. 2019).

[17] M. Kelly, *JSON Hypertext Application Language*, 2013-10-03. Adresse: `https://tools.ietf.org/html/draft-kelly-json-hal-06` (Abruf am 10. 11. 2019).

[18] I. Melzer, *Service-orientierte Architekturen mit Web Services Konzepte - Standards - Praxis*, 3. Aufl. Heidelberg: Spektrum Akad. Verl., 2008, ISBN: 382741993X. Adresse: `http://slubdd.de/katalog?TN_libero_mab21658382`.

[19] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky und U. Sarid, *OpenAPI Specification*, 2018-10-08. Adresse: `http://spec.openapis.org/oas/v3.0.2` (Abruf am 15. 11. 2019).

[20] L. Richardson, *JWTUMOIM: Act 3*, 2009-01-16. Adresse: `https://www.crummy.com/writing/speaking/2008-QCon/act3.html` (Abruf am 10. 11. 2019).

[21] J. Schema, *JSON Hypertext Application Language*, 2019-09. Adresse: `https://json-schema.org/` (Abruf am 15. 11. 2019).

[22] SmartBear Software, *Create Great API Documentation | Swagger*, 2019. Adresse: `https://swagger.io/solutions/api-documentation/` (Abruf am 06. 12. 2019).

[23] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler und N. Lindström, *JSON-LD 1.0 - A JSON-based Serialization for Linked Data*, 2014-01-16. Adresse: `https://www.w3.org/TR/json-ld/` (Abruf am 10. 11. 2019).

[24] Stacee, *Postman Supports OpenAPI 3.0 - Postman Blog*, 2018-12-11. Adresse: `https://blog.getpostman.com/2018/12/11/postman-supports-openapi-3-0/` (Abruf am 20. 10. 2019).

[25] Stack Exchange Inc., *Stack Overflow Developer Survey*, 2019. Adresse: `https://insights.stackoverflow.com/survey/2019#technology-_-web-frameworks` (Abruf am 06. 12. 2019).

[26] S. Tilkov, M. Eigenbrodt, S. Schreier und O. Wolf, *REST und HTTP Entwicklung und Integration nach dem Architekturstil des Web*, 3., aktualisierte und erw. Aufl. Heidelberg: dpunkt-Verl., 2015, ISBN: 3864916445. Adresse: `http://slubdd.de/katalog?TN_libero_mab22)500195210`.

[27] G. Torikian, K. Daigle, D. Celis, C. Somerville, B. Swinnerton und B. Black, *The GitHub GraphQL API - The GitHub Blog*, 2016-09-14. Adresse: `https://github.blog/2016-09-14-the-github-graphql-api/` (Abruf am 06. 12. 2019).

Anlagen

Selbständigkeitserklärung

Ich versichere, dass ich die Bachelorarbeit mit dem Titel **Vergleich der Web API Ansätze REST und GraphQL** selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

.....	.....
Ort, Datum	**Student**