Hochschule Konstanz
Department of Computer Science

Eingereicht von
David Wolpers
Matrikel-Nummer 295416

david.wolpers@htwg-konstanz.de

Bachelor These

Entwicklung von Entscheidungskriterien für die Verwendung von GraphQL gegenüber REST in modernen Webanwendungen

Konstanz, 30. Juni 2020

Bachelor These

Entwicklung von Entscheidungskriterien für die Verwendung von GraphQL gegenüber REST in modernen Webanwendungen

von

David Wolpers

zur Erlangung des Grades

Bachelor of Science
in Angewandte Informatik

an der Hochschule Konstanz Technik, Wissenschaft und Gestaltung,

Matrikel-Nummer:	295416
Abgabedatum:	30. Juni 2020
1. Betreuer:	**Prof. Dr. Marko Boger**
2. Betreuer:	**Michael Wagner**

Eine elektronische Version dieser Thesis ist Verfügbar auf https://github.com/Seitenbau/graphql-api-signalk.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, David Wolpers, geboren am 21.03.1997 in Nürnberg,

(1) dass ich meine Bachelorarbeit mit dem Titel:

Entwicklung von Entscheidungskriterien für die Verwendung von GraphQL gegenüber REST in modernen Webanwendungen

in der Fakultät Informatik unter Anleitung von Professor Dr. Marko Boger und Betreuer Michael Wagner selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angeführten Hilfen benutzt habe;

(2) dass ich die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

(3) dass die eingereichten Abgabe-Exemplare in Papierform und im PDF-Format vollständig übereinstimmen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 14.06.20 _____

Abstract

Beim Aufbau einer Web-API fällt die Entscheidung für die Technologie in den meisten Fällen auf REST. In den letzten Jahren wurden aber immer mehr Alternativen entwickelt. Dazu gehört auch GraphQL. Als Unternehmen stellt sich jetzt die Frage, wann sich der Einsatz von GraphQL lohnt. Ziel dieser Arbeit ist es, dafür Entscheidungskriterien aufzustellen und zu untersuchen.
Um dieses Ziel zu erreichen, werden erst einige Grundlagen erforscht. Dabei sollen zuerst die wichtigsten Technologien dieser Arbeit erläutert werden. Anschließend wird festgelegt, welche Technologien genutzt werden sollen, um Aspekte der APIs zu untersuchen. Dabei wird auch erläutert, welche Aspekte analysiert werden sollen. Daraufhin wird in einer Kombination aus Codebeispielen und Literaturinformationen die Analyse durchgeführt. Zuletzt werden Use-Cases aufgestellt und im Kontext der Ergebnisse der Analyse evaluiert.
Die Arbeit zeigt, dass die flexiblere Datenübertagungen bei GraphQL viele Vorteile ermöglicht. Es wurde aber auch ersichtlich, dass für sehr einfache APIs der Einsatz von REST zu empfehlen ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcode

Abkürzungsverzeichnis

API Application Programming Interface

HTTP Hypertext Transfer Protocol

REST Representational State Transfer

HATEOAS Hypermedia As The Engine Of Application State

URI Uniform Resource Identifier

SOAP Simple Object Access Protocol

URL Uniform Resource Locator

UI User Interface

BSI Bundesamt für Sicherheit in der Informationstechnik

SDL Schema Definition Language

1

Einleitung

Parallel zur Verbreitung des Internets ist die versendete Datenmenge stark angestiegen. Immer mehr Nutzer wollen auf immer größer werdende Datenmengen möglichst schnell zugreifen. Gleichzeitig ist der Anteil der Nutzer, welche über mobile Endgeräte auf die Daten zugreifen, weiter angestiegen [Enge 2019]. Diese Entwicklungen resultieren in neuen Anforderungen für die Datenübertragung. Zum einen brauchen vor allem mobile Endnutzer eine schnelle und datensparende Verbindung. Zum anderen führt die erhöhte Vielfalt der Geräte auch dazu, dass die benötigten Daten sich von Nutzer zu Nutzer stark unterscheiden können.
Diese Anforderungen müssen von der Schnittstelle zwischen Server und Client erfüllt werden. Mit der aktuell am weitesten verbreiteten Technologie REST wird dies schnell sehr aufwändig. Denn hier treten häufig Probleme wie Over- und Underfetching auf, d. h. bei einer Datenübertagungen werden zu viele oder zu wenige Daten übertragen. Auch Facebook hatte damit zu kämpfen und entschied sich daraufhin eine neue Technologie zu entwickeln - GraphQL. Diese sollten einen flexiblen, datensparenden Zugriff auf die entsprechenden APIs ermöglichen.

1.1. Ziel der Arbeit

Wann der Einsatz von GraphQL sinnvoll ist, lässt sich nicht pauschal sagen. Die Sprache ermöglicht zwar neue Herangehensweisen an eine API, doch könnten andere bekannte Funktionalitäten darunter leiden. Ziel der nachfolgenden Arbeit ist es, für Unternehmen und Entwickler eine Entscheidungshilfe zu formulieren, wann sich der Einsatz

von GraphQL gegenüber REST lohnen könnte. Konkret sollen dabei folgende Fragen beantwortet werden:

- Welche funktionellen Aspekte sind für den Aufbau der meisten APIs erforderlich?
- Wie können diese Aspekte in GraphQL und REST umgesetzt werden?
- In welchen Fällen ist der Einsatz welcher Technologie zu empfehlen?

1.2. Vorgehensweise

Um diese Fragen zu beantworten, wird erst aus aktueller Literatur entnommen, welche Aspekte entscheidend für den Aufbau einer guten API sind. Anschließend wird über Codebeispiele, welche sich auf eine bereits existierende Segel-API beziehen, untersucht, ob und wie diese in GraphQL und REST implementiert werden können. Diese Implementierungen werden daraufhin gegenübergestellt und analysiert. Zuletzt werden praxisnahe Use-Cases formuliert und anhand der Ergebnisse der Analyse eine Empfehlung gegeben, welche Technologie man für welchen Zweck auswählen sollte.

2

Technischer Hintergrund

Um die Arbeit verstehen zu können, müssen erst einige wichtige Technologien, welche die Arbeit behandelt, beschrieben werden. Zuerst wird erklärt, was eine API eigentlich ist. Anschließend werden mit REST und GraphQL die verschiedenen Herangehensweisen an eine API behandelt. Zuletzt wird noch gezeigt, inwiefern das Grundprinzip von GraphQL sich von REST unterscheidet, aber auch welche Gemeinsamkeiten sie haben.

2.1. Application Programming Interface (API)

Unter einem Application Programming Interface (API) versteht man die Kommunikationsstelle zwischen zwei Softwarekomponenten. Die API wird dabei von einer der Komponenten festgelegt und gibt an, wie die andere Komponente mit ihr kommunizieren kann. Konkret werden von der API alle Daten und Funktionalitäten zur Verfügung gestellt und dokumentiert, welche von außen zugänglich sein sollen. Eine API kann dabei nach [Spichale 2019] auf zwei Arten heruntergebrochen werden :

- **Programmiersprachen-APIs:** Programmiersprachen-APIs werden beispielsweise von Bibliotheken angeboten und sind sprach- und plattformabhängig.
- **Remote-APIs:** Die Datenübertragung bei Remote-APIs wird über Protokolle wie das Hypertext Transfer Protocol (HTTP) abgewickelt. Da HTTP hauptsächlich im Internet verwendet wird, heißen solche HTTP-APIs häufig Web-APIs. Ihr Vorteil ist, dass sie sprach- und plattformunabhängig sind.

APIs, welche REST und GraphQL nutzen, sind dabei den Remote-APIs zuzuordnen. Bei beiden werden zusätzlich noch die meisten APIs als Web-APIs aufgebaut, d. h. sie nutzten HTTP.

2.2. Representational State Transfer (REST)

Der Representational State Transfer (REST) Architekturstil wurde im Jahre 2000 von Roy Fielding im Rahmen seiner Dissertation veröffentlicht. REST ist ein Architekturstil zum Aufbau einer API für verteilter Systeme. Nach Roy Fielding handelt es sich dabei um eine hybride Mischung verschiedener, netzwerkbasierter Architekturstile, welche durch weitere Einschränkungen eine einheitliche Anschlussschnittstelle beschreiben. Damit sich eine API RESTful nennen darf, müssen die folgenden Prinzipien erfüllt werden [Fielding 2000]:

- **Client-Server:** Trennung der User-Oberfläche von der Datenspeicherung, wodurch die Portabilität und die Skalierbarkeit ansteigt.
- **Zustandslos:** Jede Anfrage des Clients zum Server muss alle benötigten Daten enthalten, um die Anfrage zu verstehen.
- **Cachebar:** Die Daten des Servers müssen so gekennzeichnet werden, das deutlich ist, ob sie cachebar oder nicht-cachebar sind. Cachebare Daten dürfen dann vom Client Cache verwendet werden.
- **Einheitliches Interface:** Durch den Einsatz des Allgemeingültigkeitsprinzips wird die Architektur vereinfacht und die die Sichtbarkeit der Interaktionen verbessert.
- **Schichtenaufbau:** Aufbau in Schichten, so dass die einzelnen Schichten nur ihre direkten Interaktionspartner sehen.
- **Code auf Abfrage (Optional):** Client Funktionalität soll durch downloadbare Skripte und Applets erweitert werden .

Das einheitliche Interface ist dabei nach Fielding das zentrale Feature, welches REST von anderen Architekturstilen unterscheidet. Um dieses Ziel zu erreichen, muss das Interface weitere Einschränkungen erfüllen: Identifikation von Ressourcen, Manipulation von Ressourcen über Repräsentationen, Selbsterklärende Nachrichten und das Hypermedia As The Engine Of Application State (HATEOAS) Prinzip [Fielding 2000].
Hierbei werden die Begriffe Ressourcen und Repräsentationen verwendet. Eine Ressource ist jede Form einer benennbaren Information. Dabei kann es sich sowohl um

klassische Dateien handeln als auch um Services oder reale, nicht virtuelle Personen. Jeder dieser Ressourcen muss über einen Uniform Resource Identifier (URI) identifizierbar sein [Spichale 2019]. Der Client greift dann über diese URI auf die Ressource zu und erhält als Antwort eine Repräsentation der Ressource. Diese Repräsentation kann verschiedene Formate annehmen, wie z. B. JSON, HTML oder XML. Jedoch sollten diese Repräsentationen zur Erfüllung des HATEOAS-Prinzip wie Hypertext aussehen [Fielding 2008]. Unter Hypertext versteht man, dass der Text bzw. die Repräsentationen auf weitere Ressourcen verlinken sollen [Fielding 2007]. Diese Definitionen geben die Einschränkungen eigentlich schon vor [Fielding 2000]:

- **Identifikation von Ressourcen:** Jede Ressource ist über einen eindeutigen Schlüssel, der URI, identifizierbar.
- **Manipulation von Ressourcen über Repräsentationen:** Jede Anfrage oder Änderung der Ressourcen wird über eine Repräsentation durchgeführt.
- **Selbsterklärende Nachrichten:** Alle Informationen, welche nötig sind, um eine Anfrage zu verstehen, müssen mitgeschickt werden.
- **HATEOAS:** Informationen über weitere Möglichkeiten sollen über Links mitgeschickt werden.

2.3. GraphQL

GraphQL wurde ab 2012 von Facebook intern entwickelt, um ihre Probleme mit REST zu umgehen. Im Laufe der Zeit ist dem Konzern dabei bewusst geworden, dass der Nutzen von GraphQL über Facebook hinausgeht. Darum wurde die GraphQL-Spezifikation 2015 öffentlich zugängig gemacht [Facebook 2015].

Bei GraphQL handelt es sich um Kombination aus eine Abfragesprache für APIs und einer Laufzeitumgebung um diese Anfragen serverseitig mit Daten zu füllen [GraphQL 2018]. Im Kontext von GraphQL werden dabei drei grundlegende Komponenten verwendet [1und1 2019]:

- **Abfragesprache:** Die Abfragesprache ermöglicht den Zugriff auf die GraphQL-API. Dabei können die Abfragen durch den Clienten flexibel formuliert werden, so kann bspw. genau angegeben werden, welche Datenfelder benötigt oder verändert werden.

- **Typsystem:** Das Typsystem wird genutzt, um für jede Sever-Implementierung eine individuelles Schema zu definieren. Dieses Schema gibt genau an, welche Daten der Client anfragen kann. Gleichzeitig dient es aber auch als Basis, um die Anfragen zur Laufzeit zu validieren. 2.1 zeigt einen Beispielimplementierung eines Schemas.
- **Laufzeitumgebung:** Diese Validierung wird von der Laufzeitumgebung übernommen. Zusätzlich werden hier die Anfragen geparst, so dass anschließend die Daten für die Antworten durch den Servercode ermittelt werden können. Anschließend serialisiert die Laufzeitumgebung die Antwort noch. Implementierungen dieser Laufzeitumgebung gibt es bereits in vielen Programmiersprachen.

```
1  type Query {
2      route(name: String): Route
3  }
4
5  type Route {
6      name: String
7      distance: Float
8      start: String
9      end: String
10     description: String
11 }
```

Quellcode 2.1: Beispielstruktur eines Schemas

Zusammengenommen entsteht daraus ”ein hochgradig wandelbares API-Gerüst” [1und1 2019], wodurch verschiedenste Plattformen effizient angebunden werden können. Dieser Effekt wird noch verstärkt, da das ganze Design von GraphQL auf Frontendentwickler zugeschnitten ist. Dabei lässt sich das Design nach [GraphQL 2018] auf folgende Prinzipen herunterbrechen:

- **Hierarchisch:** Die Datenstruktur des Schemas sollte hierarchisch aufgebaut sein. Eine GraphQL Abfrage folgt dieser Struktur und ist somit genauso aufgebaut, wie die Daten, welche sie zurückerhält. Dadurch werden auch kompliziertere, mehrschichtige Anfragen möglich.
- **Produktorientiert:** GraphQLs Sprache und Laufzeitumgebung soll die Anforderungen von Frontendentwicklungen erfüllen und ermöglichen.
- **Starke Typisierung:** Jede GraphQL Server-Implementierung definiert ein eigenes Typesystem. Dieses Typesystem kann abgefragt werden und die Abfragen können validiert werden. Dadurch kann der Server Struktur und Art der Antwort garantieren.

- **Clientspezifische Abfragen:** Clients können genau angeben, welche Daten sie benötigen. Dabei wird auf jedes einzelne Feld spezifisch eingegangen.
- **Introspektiv:** GraphQL ist introspektiv. Das heißt, die Struktur des definierten Schemas muss über eine GraphQL Abfrage selbst abgefragt werden können.

2.4. GraphQL und REST - Implementierungsunterschiede

Durch die strikten Vorgaben von GraphQL haben verschiedenen Implementierungen hier sehr viele Gemeinsamkeiten. Bei REST kann die Implementierung jedoch wesentlich flexibler interpretiert werden. Das Problem hierbei ist aber, dass mehr Aufwand vom Entwickler betrieben werden muss, um eine wirklich RESTful-API zu erzielen. Die meisten REST-APIs erfüllen nicht alle Voraussetzungen, vor allem das einheitliche Interface, der eigentlich ausschlaggebende Punkt, wird häufig nicht vollständig implementiert [Dupree 2018]. Das liegt aber nicht nur an der Ignoranz der Entwickler, sondern auch daran, dass eine vollständige Implementierung des einheitlichen Interfaces auch Nachteile mit sich zieht. Fielding selber sagt, dass durch die Standardisierung der Datenstruktur die Effizienz teilweise auf der Strecke bleibt [Fielding 2000]. Für den Kontext dieser Arbeit werden vollständige RESTful-APIs betrachtet. Um eine Übersichtlichkeit zu gewähren, werden aber HATEOAS-Elemente meistens weggelassen.
Für beide Technologien sind verschiedene Faktoren unabhängig von der Implementierung. Diese werden nun verglichen:

- **Gemeinsamkeiten:**
  - Zustandslos: Beide Implementierungen sind zustandslos, d. h. alle benötigten Information zur vollständigen Bearbeitung der Anfrage müssen vom Client mitgeschickt werden.
  - HTTP: Beide Implementierungen können theoretisch auch über andere Verfahren als HTTP versendet werden, jedoch hat sich HTTP als der Standard verbreitet. Vor allem REST funktioniert durch die CRUD-Methoden sehr gut über HTTP.
- **Unterschiede:**
  - Endpunkte: Während bei REST für jede Ressource ein separater Endpunkt bestehen muss, hat GraphQL nur genau einen Endpunkt.

- CRUD: Jede Anfrage bei REST über HTTP kann eine der HTTP-Methoden verwenden und damit auch eine andere Funktion der REST-API ansprechen. Bei GraphQL hingegen sind alle Anfragen entweder *HTTP-POST* oder *HTTP-GET*.
- Datenabfrage: Bei REST werden immer alle Daten einer Datenstruktur zurückgeben, falls der Server Daten nicht über Parameter filtert. Bei GraphQL muss bei jeder Anfrage genau angeben werden, welche Daten benötigt werden.

3

Methodik der Analyse

Ziel der Arbeit ist die Analyse der verschiedenen Aspekte von Web-APIs bei der Verwendung von REST oder GraphQL. Dafür muss jedoch der Kontext für die Analyse festgelegt werden. Dieses Kapitel beschäftigt sich zuerst im Abschnitt Technologien mit den dafür genutzten Technologien. Anschließend wird in der Sektion 3.2 auf die Aspekte eingegangen, welche im nachfolgenden Kapitel genauer analysiert wird. Dazu wird erklärt, wieso diese Aspekte wichtig sind und welche Faktoren man untersuchen muss. Dabei wird immer darauf eingegangen, wie die Analyse diese Faktoren untersuchen soll.

3.1. Technologien

Zur Analyse der Aspekte wird eine API sowohl in GraphQL als auch in REST umgesetzt. Diese API basiert auf einem Plugin der SignalK-API, eine API, welche das SignalK Datenformat versendet [SignalK 2020]. Eingesetzt wird das Datenformat im Marinebereich. Aufgabe des Plugins ist es, die Verwaltung von Segelrouten zu regeln. Für die gesamte API gab es mehrere Implementierungen in verschiedenen Programmiersprachen zur Auswahl. Die Entscheidung fiel auf *JavaScript* in der *nodejs* Laufzeitumgebung.Javascript kann sehr einfach und übersichtlich geschrieben werden, wodurch der Fokus auf den jeweiligen API-Technologien liegt. Ein weiterer Vorteil ist, dass damit auch ein Testclient aufgesetzt werden kann, wodurch die Anzahl an verwendeten Technologien sinkt.

API	Server	Client
REST	express	node-fetch
GraphQL	express-graphql graphql	node-fetch

Tabelle 3.1: Verwendete Bibliotheken

Nach der Wahl der Programmiersprache müssen entsprechende Bibliotheken ausgesucht werden. Die Bibliotheken gehören dabei zwei Lagern an. Auf der eine Seite hat man Bibliotheken, welche die Implementierung der jeweiligen Technologie in verschiedenen Sprachen anbieten. Auf der anderen Seite hat man Bibliotheken, welche hilfreiche Features liefern, um die Technologie besser nutzen zu können. Diese Bibliotheken sind aber nur unterstützend und können somit keine eigene API aufbauen. In vielen Fällen wie *Apollo* liefern die Implementierungsbibliotheken einige dieser Features schon mit. Ziel der Arbeit ist es jedoch nicht, die Qualität und Features der einzelnen Bibliotheken zu analysieren. Stattdessen sollen hauptsächlich die grundlegenden Designvorschriften und die daraus resultierenden Vor- und Nachteile betrachtet werden. Zu diesem Zweck werden als Bibliotheken möglichst einfache Implementierungen von REST und GraphQL für Server und den Client genommen. In einigen Sonderfällen werden jedoch bestimmte Funktionen erwähnt, welche durch Bibliotheken möglich sind. Tabelle 3.1 zeigt, welche Bibliotheken ausgewählt wurden. *Express* ist eigentlich ein ganzes Framework, welches jedoch einen sehr einfachen Aufbau von Servern ermöglicht. Durch Erweiterungen kann auch GraphQL problemlos umgesetzt werden. Für den Client muss in beiden Fällen nur eine Möglichkeit existieren, Anfragen über HTTP zu versenden. *Node-fetch* ermöglicht dies in einer einfachen, übersichtlichen Art und Weise.

3.2. Aspekte

Jetzt stellt sich noch die Frage, welche Aspekte genauer behandelt werden sollen. Dazu werden untere anderem verschiedene Prinzipien des API-Designs von REST-APIs betrachtet. Diese beziehen sich häufig auf universal nützliche Funktionalitäten. Die Aspekte sollten möglichst so ausgewählt werden, dass ihre Anwendung zu einer sicheren und nutzerfreundlichen API führt. Dafür werden auch Aspekte betrachtet, welche REST schlecht behandelt oder welche den Kern von GraphQL ausmachen. Für jeden dieser

Aspekte stellt sich noch die Frage, wie er untersucht werden kann. In vielen Fällen wird es sich dabei um eine Codegegenüberstellung handeln, es kann aber auch auf verschiedene Literaturquellen zurückgegriffen werden. Eine genauere Beschreibung der Analyse wird für jeden der Aspekte individuell aufgeführt.

3.2.1. Verbreitung und Ausgereiftheit

Ziel der vorliegenden Arbeit ist es eine Entscheidungshilfe zu liefern, ob sich der Einsatz von GraphQL gegenüber REST für das eigene Projekt lohnt. Wenn man sich mit den Technologien auseinandersetzt, stellt man schnell fest, dass man eigentlich die gleichen APIs aufsetzten kann. Die Implementierung wird sich zwar unterscheiden, aber die reine Funktionalität ist gleich. Es lohnt sich deswegen, erst auf Faktoren zu schauen, welche unabhängig vom Aufbau einer API sind. Dieser Aspekt will sich deswegen damit beschäftigen, wie weit verbreitet die jeweiligen Technologien sind. Vor allem bei neueren Technologien wie GraphQL kann das einen guten Hinweis darüber geben, wie gut und nützlich sie ist. Da GraphQL inzwischen eine Open Source Software ist, wird der Faktor noch verstärkt. Eine Open Source Software ist eine Software, bei der jeder weiterentwickeln kann. Es ist so bspw. erlaubt sie zu nutzen, um weitere Implementierungen in anderen Sprachen zu entwickeln. Warum diese Faktoren so wichtig sind, zeigt sich vor allem an drei Eigenschaften:

- **Qualität:** Eine neue Technologie, welche sich schnell weitverbreitet, deutet meist daraufhin, dass die Qualität bzw. Nützlichkeit weit über der Konkurrenz liegt.

- **Ausgereiftheit:** Eine neue Open Source Technologie lebt davon, dass die Nutzer sie verwenden und weiterentwickeln. Je mehr Nutzer es gibt, desto mehr Potential gibt es für die Weiterentwicklung der Technologie oder auch Bibliotheken zur Nutzung der Technologie. Und das wiederum führt dazu, dass mögliche Schwächen der Technologie wahrscheinlicher durch verschiedene Bibliotheken ausgemerzt werden.

- **Langlebigkeit:** Neue Technologien sind in der Softwarebranche an der Tagesordnung. Aufgrund dieses Überangebots kommt es häufig vor, dass auch hochwertige Technologien wieder aussterben. Als Unternehmen will man es vermeiden, seine Produkte auf Technologien aufzubauen, welche in drei Jahren niemand mehr nutzt. Für APIs ist das besonders kritisch, da hier im Laufe der Zeit möglicherweise neue Frontends angebunden werden sollen. Ist eine Technologie aber schon weit verbreitet, sinkt dieses Risiko natürlich stark.

Für die Analyse sollen nun zwei Faktoren betrachtet werden. Zum einen, welche Unternehmen die jeweilige API-Technologie verwenden und warum sie es tun. Daraus kann man erste Schlüsse ziehen, wann sich der Einsatz von GraphQL lohnen würde. Außerdem sollte auch betrachtet werden, welche Bibliotheken oder Frameworks zur Verfügung stehen. Zwar sollen sie im Rahmen der Arbeit nicht analysiert werden, aber ausführliche Bibliotheken für verschiedene Sprachen sind ein gutes Zeichen für die Verbreitung einer Technologie. Zusätzlich ist eine hohe Anzahl an Bibliotheken auch ein gutes Zeichen, dass es für mögliche Probleme der Technologie bereits eine Lösung in Form einer Bibliothek gibt.

3.2.2. Implementierung

Wenn man dann anfängt, eine API zu bauen, stellt sich zuerst immer die Frage, wie die Implementierung aussieht und vor allem, wie aufwändig diese ist. Die Analyse soll sich hierfür nur mit einer sehr einfachen Implementierung auseinandersetzen. Die Komplexität von aufwändigeren Implementierungen ist häufig das Resultat anderer Aspekte. Solche Aspekte werden in den jeweiligen Abschnitten individuell behandelt.
Ziel ist es, den Aufwand widerzuspiegeln, welcher ungefähr der Implementierung eines REST-Endpunktes entspricht. Dabei muss sowohl der Aufwand im Backend, also dem Server, betrachtet werden als auch der Aufwand im Frontend, dem Client. Die gleiche Funktionalität soll anschließend in GraphQL umgesetzt werden und die Ergebnisse verglichen und analysiert werden.

3.2.3. Dokumentation

Eine gute API besteht aber nicht nur aus der Implementierung. Wie in [Sturgeon 2015] erklärt wird, sollte stattdessen schon früh im Entwicklungsprozess damit begonnen werden, eine übersichtliche Dokumentation anzufertigen. Denn Entwickler, welche die API nutzen, brauchen Informationen über zwei Aspekte: Welche Daten stehen zur Verfügung und welche Möglichkeiten hat man, um auf diese Daten zuzugreifen? Für eine Übersicht der Möglichkeiten versuchen REST mit HATEOAS und GraphQL mit seinem Introspektionssystem eine Lösung zu finden. Doch ob damit alles abgedeckt wird und ob es anwenderfreundlich ist, muss untersucht und analysiert werden. Auf welche Daten über die Schnittstellen zugegriffen werden kann, muss ebenfalls ausführlich dokumentiert werden. GraphQL hat mit dem Typesystem schon eine strike Typisierung zur Definition dieser Datenstrukturen. Wie nützlich das für die Dokumentation ist und wie dieser Faktor bei REST umgesetzt werden kann, wird in der Analyse gezeigt. Außerdem

soll auch auf mögliche automatisierte Dokumentationsverfahren eingegangen werden.

3.2.4. Validierung

Ein weiterer wichtiger Faktor zum Aufbau einer guten und sicheren API ist die Validierung. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) schreibt hierzu:

> „Bei der Datenvalidierung geht es darum sicherzustellen, dass durch Ein- und Ausgaben keine ungewollten Aktionen ausgelöst bzw. Manipulationen durchgeführt werden können. “[BSI 2013]

Sie weisen auch darauf hin, dass man für Eingabedaten ein positives Sicherheitsmodell verwenden sollte, d. h. dass die Daten explizit angenommen und nicht abgelehnt werden sollen. [BSI 2013] GraphQL nutzt das Typesystem, um genau dieses Ziel zu erreichen. Wie gut das funktioniert, wird in der Analyse beantwortet werden. Gleichzeitig soll gezeigt werden, wie die Validierung in REST umgesetzt wird.

3.2.5. Antwortzeit

Beim Aufbau einer API ist Benutzerfreundlichkeit einer der auschlaggebenden Punkte. Diese ist von vielen Faktoren abhängig. Doch als erstes fällt jedem Nutzer die Ladegeschwindigkeit der Seite auf. Wenn jede Aktion mit einer Sekunde warten verbunden ist, sind die Nutzer schnell unzufrieden. Diese Ladegeschwindigkeit ist ein Resultat der Antwortzeit der API. Wie lange dauert es also, um eine Anfrage zur API zu schicken, die Daten für die Antwort zu bekommen und anschließend die Antwort zurück zum Client zu schicken. Folgende Faktoren sind dabei ausschlaggebend (teilweise [Rajak 2017]):

- **Latenz:** Die Latenz gibt an, wie lange einen Datenpaket, unabhängig von seiner Größe, braucht, um von der Quelle zum Ziel zu kommen. Dieser Faktor kann auf API-Ebene nicht verändert werden. Stattdessen hängt er von der Lokalisierung von Client und Server ab. Eine geringere Latenz kann also bspw. durch gut verteilte Server erreicht werden.
- **Datenübertragung:** Die Datenübertragung hängt von zwei Aspekten ab: der Bandbreite der Leitung und der Größe der Daten. Die Bandbreite gibt an, wie viele Daten wie schnell übertragen werden können. Es kann jedoch auf die Bandbreite der Nutzer kein Einfluss genommen werden. Als API-Entwickler muss man sogar davon ausgehen, dass diese sehr klein ist. Entsprechend sollte die Datenmenge

möglichst gering gehalten werden. Häufig kommt es vor, dass die Antwort mehr Daten enthält, als der Client eigentlich benötigt. Dieses sogenannte **Overfetching** soll bei einer gut designten API nach Möglichkeit vermieden werden.

- **Underfetching:** Underfetching bezeichnet den Vorgang, dass mehrere Anfragen geschickt werden müssen, um ein Problem zu lösen. Im schlimmsten Fall muss auf die Antwort einer Anfrage gewartet werden, bevor dann eine weitere Anfrage geschickt werden kann, um alle Daten zu erhalten. Deswegen sollte die API möglichst alle benötigten Daten in einer Anfrage beantworten können.
- **Caching:** Caching bezeichnet den Vorgang, Anfragen und ihre Antwort in einem Cache zu speichern. Dadurch müssen manche Anfragen nicht wiederholt vom Server beantwortet werden, sondern können auf dem Weg schon von einem Cache bedient werden. Mit gutem Caching können für manche Anfragen alle vorherigen Faktoren vermieden werden. Eine API sollte entsprechend einen Cachingsupport anbieten.
- **Anfragen-Batching:** Anfragen-Batching beschreibt den Vorgang, mehrere Anfragen im Client zu kombinieren und gemeinsam abzuschicken. Dadurch können unnötige *Round-Trip-Times* vermieden werden. Anfragen-Batching wird jedoch meistens über Bibliotheken umgesetzt. Eine eigene manuelle Lösung ist sowohl für REST als auch GraphQL unnötig komplex. Deswegen wird in dieser Arbeit nicht genauer darauf eingegangen.

Over- und Underfetching sind Aspekte, auf die genau eingegangen werden muss. Denn sie sind einer der ausschlaggebenden Punkte, welche für GraphQL sprechen. Bei REST-APIs sind diese Probleme hingegen häufig weit verbreitet.
Die Analyse dieses Punktes wird sich damit beschäftigen, wieso die Probleme in REST-APIs häufig auftreten, aber auch, wie sie umgangen werden können. Dem gegenüber wird betrachtet, wie GraphQL versucht das Problem schon im grundlegenden Konzept zu umgehen.

Der nächste Punkt, der von der API beeinflusst werden kann, ist das Caching. Dafür ist ein genaueres Verständnis von Caching nötig. Ein Cache ist ein Zwischenspeicher. Abgefragte Daten können dort gespeichert werden, um gleiche Anfragen nicht wiederholt ganz bis zum Server durchgeben zu müssen. Häufig liegen auf dem Pfad mehrere Caches. Diese können auf drei verschiedene Varianten heruntergebrochen werden [McKinnon 2019]:

- **Seitencache:** Der Seitencache speichert den Seiteninhalt ab. Entscheidend ist dabei, dass dieser Cache vom Client kontrolliert werden kann, d. h. er kann theo-

retisch beeinflussen, was und wie lange gespeichert werden soll. Häufig wird dies aber automatisch über HTTP-Header festgelegt.

- **Browsercache:** Der Browsercache funktioniert theoretisch genau so, jedoch speichert er Daten für alle Seiten ein, welche über den Browser aufgerufen werden. Außerdem wird dieser vom Browser kontrolliert. Der Endnutzer kann jedoch jederzeit den Cache leeren.
- **Servercache:** Der Servercache liegt, wie der Name schon sagt, auf der Serverseite. Dabei werden, falls das möglich ist, auch gleiche Anfragen verschiedener Nutzer und ihre Antwort im Cache abgespeichert. Damit sollen unnötige Abfragen an die Datenbank vermieden werden.

Alle Varianten haben gemeinsam, dass festgelegt werden muss, welche Daten wie lange gecacht werden können. Um die Daten zu cachen ist ein eindeutiger Schlüssel vonnöten. Dieser wird aus der Anfrage erstellt. Wenn eine weitere Anfrage den gleichen Schlüssel erzeugt, kann sie die Daten im Cache auslesen und muss nicht vom Server beantwortet werden. Im besten Fall sollten Anfragen, welche genau die gleichen Daten haben wollen, auch den gleichen Schlüssel bekommen. Dabei sollen bspw. vertauschte Parameter keinen Einfluss haben. Die Analyse wird sich damit beschäftigen, wie die verschiedenen Cachingvarianten bei REST und GraphQL umgesetzt werden können. Dazu muss der Aufwand untersucht werden, welcher nötig ist, um überhaupt ein Caching implementieren zu können.

4

Prototypische Analyse

Das vorherige Kapitel hat gezeigt, welche Aspekte wie analysiert werden sollen. Diese Analyse wird jetzt entsprechend anhand von Codebeispielen oder Recherchen durchgeführt.

4.1. Verbreitung und Ausgereiftheit

Zuerst soll dabei betrachtet werden, wie weit die jeweiligen Technologien verbreitet sind. Da REST als wesentlich ältere und nahezu konkurrenzlose Technologie im Raum stand, wird sich über das Warum hier kaum etwas sagen lassen. Interessanter sind dagegen Unternehmen, welche sich für einen Wechsel von REST zu GraphQL entschieden haben. Ein nützlicher Aspekt ist es auch zu wissen, welche Bibliotheken existieren.

4.1.1. Unternehmen

Twitter: Twitter ist das wohl bekannteste Beispiel einer REST-API. Sie erlaubt Entwicklern, nahezu alle Daten von Twitter auszulesen, ohne dabei auf der Twitter Website zu sein. Das Prinzip ist aber genau das gleiche: Eine Schnittstelle stellt bspw. das Login zur Verfügung. Der Token, der dabei als Antwort erzeugt wird, kann genutzt werden, um im eigenen Namen zu Tweets zu posten. Es können aber natürliche auch aktuelle Tweets der eigenen Twitter Timeline ausgelesen werden [Twitter 2020]. Dabei ist die Twitter-API eigentlich keine vollständige Implementierung der Designprinzipien [Ullen-

boom 2017]. Stattdessen nutzt die Implementierung nur die Aspekte, welche sie wirklich braucht. REST wird hier wahrscheinlich nur deswegen verwendet, weil es früher keine sinnvolle Alternative gab. Simple Object Access Protocol (SOAP) welches von REST vom Markt verdrängt wurde, hatte dem kaum was gegenüber zu setzen. Andere Unternehmen wie Github nutzen aber die neuen Alternativen, um die Technologie zu wechseln.

Github: Github ist ein Unternehmen, welches eine Plattform zur Verfügung stellt, um die Entwicklung von Software zu organisieren. Beim Wechsel der API von Version 3 auf Version 4 in 2016 wechselten sie auch von REST zu GraphQL. Sie geben dafür mehrere Gründe an.
Der erste ist der sehr weitgefächerte Punkt der Skalierbarkeit. Skalierbarkeit hängt von vielen Faktoren ab, in ihrem konkreten Beispiel verweisen sie aber vor allem auf die Menge der gesendeten Daten und Anfragen. Das Kapitel Antwortzeit hat schon beschrieben, dass es sich dabei um die Probleme des Over- und Underfetching handelt. Interessant ist aber, dass sie als Ursache für das Overfetching vor allem HATEOAS kritisieren. Im Abschnitt Over- und Underfetching werden diese Probleme genauer beleuchtet. Aus Githubs Perspektive löst GraphQL diese Probleme vollständig.
Das andere Problem, auf das Github genauer eingeht, ist komplizierter. Ihr Problem mit REST ist, dass Informationen über Endpunkte nicht nur schwer zu erhalten, sondern auch schwer zu bestimmen sind. Als Beispiel nennen sie dafür die Bestimmung der Zugriffsrechte, welche für jeden Endpunkt nötig sind. Aber auch weitere Features, welche den Nutzen der API vereinfachen, wie Typesicherheit von Userparametern oder automatische Dokumentation, hat ihnen bei REST gefehlt. Das Kapitel Dokumentation wird sich damit beschäftigen, wie diese Faktoren in GraphQL umgesetzt wurden und welche Vorteile sich daraus ergeben. [Github 2016]

Weitere bekannte Unternehmen haben aus ähnlichen Gründen gewechselt. Dazu gehören Paypal, Airbnb und Shopify.

4.1.2. Bibliotheken und Frameworks

Viele Aspekte von REST können durch Bibliotheken kaum unterstützt werden. Prinzipien wie HATEOAS oder die Identifizierung der Ressourcen müssen beim Design der API vom Entwickler selber bedacht werden. Trotzdem versuchten verschiedene Frameworks Entwickler beim Aufbau einer REST-API zu unterstützen. Namentlich kann bspw. Spring Boot genannt werden. Dieses Framework ist für Java verfügbar. Spring Boot bietet viele Funktionalitäten an. Darunter gehört auch die Unterstützung für den Aufbau

von REST-APIs. Neben der grundlegenden Implementierung vereinfacht Spring Boot bspw. auch die Validierung von User-Input [Spring 2020a].
Insgesamt gesehen gibt es für jede anständige Programmiersprache eine REST-Implementierung mit mehr oder weniger Funktionalitäten. So hat Javascript das bereits erwähnte Express, während Python mit Flask aufwartet. [Slant 2020]
GraphQL steht trotz der relativ kurzen Geschichte von fünf Jahren REST kaum nach. Viele der etablierten Frameworks von REST bieten inzwischen auch Lösungen für GraphQL an. So haben sowohl Spring Boot als auch Express Möglichkeiten, um eine GraphQL-API aufzubauen [Spring 2020b] [GraphQL 2020b]. Insgesamt werden mehr als 20 Sprachen unterstützt [GraphQL 2020a]. Zusätzlich gibt es aber noch Bibliotheken, welche auf der Clientseite angewandt werden. Diese dienen einerseits dazu, die komplizierteren Anfragen übersichtlicher zu formulieren. Andererseits versuchen sie aber auch Probleme von GraphQL zu lösen. FlacheQL oder das bekanntere Appollo übernehmen bspw. das kompliziertere Caching von GraphQL(s. Kapitel Caching für mehr Details) für den Nutzer [FlacheQL 2019] [Apollo 2020].

4.1.3. Fazit

Immer mehr Unternehmen wechseln bereits zu GraphQL. Und sie haben dafür gute Gründe. Dabei werden sie durch eine Vielzahl an Frameworks und Bibliotheken unterstützt. Bei nahezu jedem REST-Projekt kann theoretisch auf GraphQL gewechselt werden. Doch ob sich das für alle Unternehmen auch lohnt, werden die nächste Kapitel beleuchten.

4.2. Implementierung

Wie in Kapitel Implementierung angesprochen, wird sich diese Analyse nur mit einer einfachen Implementierung beschäftigen. Dafür soll sowohl hier als auch in den weiteren Abschnitten die Routen-API genutzt werden.

4.2.1. Routen-API

Die Routen-API ist eine API zur Verwaltung von Routen. Sie kann in der SignalK-API als Plugin hinzugefügt werden. Der Nutzer kann in der REST-Implementierung auf verschiedene Endpunkte zugreifen. Die Tabelle 4.1 zeigt, welche Möglichkeiten dafür zur

Verfügung stehen.

Endpunkt	Methode	Funktionalität
`/route`	POST	Fügt eine neue Route hinzu
`/route/{uuid}`	PATCH	Aktualisiert eine vorhandene Route
	GET	Gibt die Daten einer Route aus
	DELETE	Löscht eine Route
`/routes`	GET	Gibt alle abgespeicherten Routen zurück

Tabelle 4.1: Routen-API: Endpunkte

Wie die Datenstruktur einer solchen Route aufgebaut ist, kann im Code 4.1 gesehen werden.

```
1  route {
2    uuid: 'aeedd10s11223as',
3    geometry: {
4      coordinates: [[221.365, 200.233] [101.244, 135.313]]
5      distance: 1048.231,
6    },
7    start: 'Kiel',
8    end: 'Stockholm',
9    description: 'Ostsee',
10   name: 'Ostsee Toern',
11   timestamp: '1593076241483.0'
12 }
```

Quellcode 4.1: Beispiel Datenstruktur für eine Route

4.2.2. Implementierung

Für die Implementierungsanalyse soll nun der */routes*-Endpunkt genauer betrachtet werden. Ohne HATEOAS würde die Datenstruktur der Antwort ungefähr wie der Code 4.2 aussehen. Weitere Felder wurden aus Platzgründen weggelassen. Der Code 4.3 zeigt dabei, wie dieser Fall in einer REST-API umgesetzt werden kann. Mit Express besteht ein solcher Endpunkt nur aus zwei Teilen. Zum Einen muss der Uniform Resource Locator (URL) des Endpunktes angeben werden, welcher angibt, wohin der Client seine Anfrage schicken muss, um diesen Endpunkt anzusprechen. Außerdem noch die Funktion angegeben werden, welche aufgerufen wird, sobald eine Anfrage

am Endpunkt ankommt. Mögliche Parameter der Anfrage werden durch die Übergabeparameter der Funktion übergeben. Da die endgültige Datenabfrage an die Datenbank für REST und GraphQL identisch ist, kann die genaue Syntax ignoriert werden. Deswegen wird sie durch den Aufruf *db.getRoutesData()* abgehandelt.

```
routes: [
  route {
    name: 'Ostsee',
    start: 'Kiel',
    end: 'Stockholm',
    ...
  },
  route {
    name: 'Nordsee',
    start: 'Dangast',
    end: 'Langeoog',
    ...
  }
]
```

Quellcode 4.2: Beispiel Datenstruktur für eine Liste von Routen

```
app.get('/routes', async function (req, res, next) {
  var result = await db.getRoutesData()
  return res.json(result)
})
```

Quellcode 4.3: REST Server Implementierung

Diese sehr einfache Variante funktioniert nur in Sprachen, wo der Rückgabetyp nicht explizit angeben werden muss. Andere Programmiersprachen wie Java schreiben vor, dass die genaue Datenstruktur der Antwort vorher definiert werden muss. Das ist aber eigentlich kein zusätzlicher Aufwand. Denn wie bereits in Kapitel Dokumentation erklärt wurde, muss diese Datenstruktur in der Dokumentation sowieso aufgeführt werden. Nur dann können Anwender die API wirklich effizient nutzen.
Bei GraphQL ist die Datenstruktur immer Teil der Implementierung. Dafür muss ein Schema definiert werden. Doch das Schema gibt nicht nur die Datenstruktur vor, es zeigt auch an, wie Anwender auf die API zugreifen können. Dafür gibt es theoretisch zwei Möglichkeiten:

- *Query:* Eine Query entspricht dem Äquivalent einer *HTTP-GET*-Abfrage, d. h. dass hier nur Daten zurückgegeben werden. Falls also niemand anders die Daten der Datenbank verändert, müssen die gleichen Queries auch die gleichen Daten zurückliefern. Will man die Daten verändern, benötigt man eine Mutation.

- *Mutation:* Eine Mutation erfüllt die Aufgabe mehrere HTTP-Methoden. Denn eine Mutation ist jede Anfrage, welche die Daten im Backend verändert. Das können Updates verschiedener Parameter sein, aber auch das Löschen eines Objekts.

Um in GraphQL die gleiche Funktionalität wie der REST-Server zur Verfügung zu stellen, muss ein entsprechender Server aufgesetzt werden. Der Code 4.4 zeigt, wie das Ergebniss aussehen könnte. Der Endpunkt hat dabei mehrere Teile, und zwar einerseits wieder die URL, welche jedoch auf den immer gleich bleibenden GraphQL-Endpunkt verweist. Außerdem wird der Endpunkt mit dem Schema und einem Resolver verknüpft. Der Resolver parst die Anfrage, gibt sie an die entsprechenden Funktionen weiter und baut am Ende die richtige, benötigte Datenstruktur für die Antwort zusammen. Als letzter Punkt wird noch *graphiql* erwähnt. Dieses übersichtliche User Interface (UI) hilft bei der korrekten Formulierung der Anfragen. Genauer wird sie im Kapitel Dokumentation behandelt.

```
1  var schema = buildSchema(`
2    type Query {
3      routes: [Route],
4    },
5
6    type Route {
7      uuid: String,
8      geometry: GeometryObject,
9      start: String,
10     end: String,
11     description: String,
12     name: String,
13     timestamp: String
14   },
15
16   type GeometryObject {
17     coordinates: [[Float]],
18     distance: Float
19   }
20 `);
21
22 var resolver = {
23   async routes(args) {
24     var result = await db.getRoutesData()
25   }
26 }
27
28 app.use('/graphql', graphqlHTTP({
29   schema: schema,
30   rootValue: resolver,
31   graphiql: true,
```

```
32  }));
```

Quellcode 4.4: GraphQL Server Implementierung

Auf den ersten Blick ist diese Implementierung des GraphQL-Servers wesentlich aufwändiger. Wenn man jedoch bedenkt, dass die Datenstruktur des Schemas so oder zumindest so ähnlich auch für die Dokumentation der ersten Variante geschrieben werden muss, ist der Aufwand kaum größer. Die Schema Definition Language (SDL), welche genutzt wird, um die Schemata aufzubauen, erlaubt aber noch kompliziertere Strukturen. So können bspw. Interfaces genutzt werden, um den grundlegenden Aufbau mehrerer Typen festzulegen. Die meisten Strukturen dürften einem Entwickler bekannt vorkommen. Da sie aber mit SDL in einer neuen Programmiersprache formuliert werden müssen, muss diese am Anfang auch erst gelernt werden.
Wenn man vom Schema absieht, ist der größte Unterschied, dass bei der ersten Variante der spezifische Endpunkt */routes* angefragt wird, wie es im Code 4.5 zu sehen ist. Bei weiteren Funktionalitäten, welche sich mit Routen beschäftigen, müsste ein anderer Endpunkt angefragt werden. Bei GraphQL hingegen wird immer der *graphql* Endpunkt angefragt. Im Nachrichtenrumpf der Anfrage wird dann genauer spezifiziert wird, was gesucht wird. Ein Beispiel hierfür zeigt der Code 4.6.

```
1  fetch('http://localhost:3000/routes')
2    .then(r => r.json())
3    .then(data => {
4      console.log(data);
5    });
```

Quellcode 4.5: REST Client Implementierung

```
1  fetch('http://localhost:3000/graphql', {
2      method: 'POST',
3        headers: {
4            'Content-Type': 'application/json',
5        },
6        body: JSON.stringify({
7            query: `
8              query {
9          routes {
10           uuid,
11           name,
12           ...
13         }
14       }
15     `
16   })
17 })
18   .then(r => r.json())
```

```
19      .then(data => {
20        console.log(data)
21      });
```

Quellcode 4.6: GraphQL Client Implementierung

Bei den Clientanfragen sind die Unterschiede schon wesentlich deutlicher. Der REST-Endpunkt gibt unabhängig von der Anfrage immer die gleichen Daten zurück. Entsprechend können die Anfragen auch sehr klein gehalten werden. Die einzigen beiden Änderungen, welche realistisch wären, wären einerseits eine Angabe, welches Dateiformat als Antwort erwartet wird. Das könnte z. B. JSON oder XML sein. Andererseits könnten noch mögliche HTTP-Header gesetzt werden, um Caching zu ermöglichen. Das Kapitel Caching beschäftigt sich genauer damit. Bei GraphQL muss genau angeben werden, was benötigt wird. Dafür wird die namensgebende *Graph Query Language* verwendet. Um den Code kleiner zu halten, wurde hierbei auf eine vollständige Aufzählung aller Felder verzichtet. Falls man aber wirklich die gleichen Daten benötigt, welche auch die REST-Schnittstelle ausgibt, müsste man alle Felder angeben. Die resultierenden Anfragen sind dadurch aber auch wesentlich größer. Für einen unerfahrenen Entwickler kommt hier das gleiche Problem auf, welches schon beim Server aufgetreten ist. Auch hier muss für eine effiziente Nutzung von GraphQL erst eine eigene Sprache gelernt werden. In vielen Faktoren ist diese zwar selbsterklärend, an REST kommt sie damit aber nicht heran.
Zusammengefasst lässt sich sagen, dass als Entwickler ohne Erfahrung der Einstieg in REST wesentlich einfacher ist. Ein Blick in die Dokumentation und es können Abfragen abgeschickt werden. Bei GraphQL ist der Einstieg ein wenig höher. Da aber weder SDL noch die Graph Query Language wirklich kompliziert ist, hält sich der Aufwand auch hier in Grenzen. Doch auch bei erfahrenen Entwicklern sind die GraphQL-Anfragen immer noch aufwändiger als die REST-Anfragen.

4.3. Dokumentation

Das vorherige Kapitel hat die Notwendigkeit einer Dokumentation schon angeschnitten. HATEOAS und das GraphQL-Introspektionssystem sollen hier unterstützend sein. Doch wie funktionieren sie eigentlich genau und wie hilft das bei der Dokumentation?:

HATEOAS: HATEOAS gibt an, dass die Antworten von REST-Servern Links enthalten sollen. Diese Links sollen entweder auf verwandte Ressourcen oder auf mögliche andere Funktionalitäten verweisen. Dabei sollen alle Möglichkeiten abgedeckt werden,

welche der Anwender im Frontend zum aktuellen Zeitpunkt realistischerweise auswählen könnte. Dieses Verfahren hat theoretisch zwei Vorteile. Man könnte bspw. ein Frontend aufbauen, welcher nur die Links verwendet, um dem Nutzer zu zeigen, was er tun kann. Eine wirkliche UI ist damit aber nicht möglich. Denn man weiß ja im vornerein gar nicht, welche Daten ankommen. Entsprechend kann man sie auch nicht übersichtlich darstellen.
Die andere Möglichkeit ist es, dass ein Entwickler einen Einstiegspunkt wählt und von diesem aus Schritt für Schritt alle Funktionalitäten der API kennenlernt. Doch wer würde lieber mehrere Anfragen verschicken, als einmal in eine übersichtliche Dokumentation zu schauen, um eine Übersicht über die eigenen Möglichkeiten zu bekommen?
Zusammengefasst lässt sich sagen, dass HATEOAS als Dokumentationsersatz unbrauchbar ist.

Introspektionssystem: Das Introspektionssystem eröffnet die Möglichkeit, das komplette Schema der vorliegenden GraphQL-API auszulesen. Da im Schema auch die Zugriffspunkte für die API definiert werden, deckt diesem System eigentlich alle grundlegenden Funktionen ab, welche für eine Dokumentation benötigt werden. Sendet man aber eine Anfrage, welche wirklich alle Daten erhält, ab, ist die Antwort ziemlich unübersichtlich. GraphiQL ist eine UI, welche Abhilfe verschaffen soll. Sie ist in allen Implementierungen standartmäßig dabei und muss nur aktiviert werden. Das UI gibt einem dabei die Möglichkeit, im Browser Anfragen zu formulieren und abzuschicken. Gleichzeitig stellt es eine interaktive Form des Schemas zur Verfügung. Auch Kommentare im Schema werden hier mit ausgegeben. Nutzt man also ein gut auskommentiertes Schema, wird die Dokumentation komplett automatisiert. Ein Schemausschnitt wie der Code 4.7 würde eine GraphiQL Dokumentation, wie in Abbildung 4.1 zu sehen ist, erzeugen:

```
1  """Eine Route"""
2    type Route {
3      """Identifikator"""
4      uuid: String,
5
6      """Koordinatendaten"""
7      geometry: GeometryObject,
8
9      """Startort"""
10     start: String,
11     ...
```

Quellcode 4.7: Schema mit Kommentaren

Beim grundlegenden Design ist GraphQL also wesentlich hilfreicher, was den Aufbau einer übersichtlichen Dokumentation angeht. Bibliotheken können aber beiden Technologien weiterhelfen. Bei GraphQL dienen Bibliotheken wie *graphql-docs* hauptsächlich

Abbildung 4.1: GraphiQL Dokumentation

dazu, die GraphiQL-UI in eine übersichtliche Dokumentarform zu bringen [graphql-docs 2016]. Diese kann dann auch unabhängig vom Server genutzt werden. Bei REST bieten Tools wie *Swagger* eine automatisierte Dokumentation der eigenen API an [Swagger 2020]. Damit wird eigentlich die gesamte Dokumentationsfunktionalität von GraphiQL abgedeckt.
Zusammengefasst lässt sich sagen, dass das Introspektionssystem und die darauf aufgebaute GraphiQL-UI bereits eine ausführliche automatisierte Dokumentation ermöglicht. Um mit REST auf das gleiche Niveau zu kommen, müssen externe Bibliotheken genutzt werden. Diese bieten dann aber auch äquivalente Funktionalitäten oder sogar mehr an.

4.4. Validierung

Das Typesystem wird von GraphQL nicht nur für die Introspektion verwendet. GraphQL nutzt es auch, um Anfragen automatisch zu validieren. Damit kann die Struktur der Query, welche durch die *Graph Query Language* festgelegt wird, validiert werden. Ein Entwickler muss bei Anfragen also nicht überprüfen, ob sie vom Schema erfüllt werden können.
Zusätzlich garantiert GraphQL mit dem Typesystem auch, dass alle eingegebenen Pa-

rameter und Daten den richtigen Datentyp haben. Bei REST muss diese Überprüfung, manuell durchgeführt werden. Meistens ist das jedoch der einfache Teil der Validierung. Denn in vielen Fällen unterliegen die Daten noch Einschränkungen, welche sie erfüllen müssen. Ein E-Mail String sollte bspw. nicht jeder beliebige String sein, sondern nur ein String, welcher folgenden Regexausdruck erfüllt:

```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b
```

Diese Validierung lässt sich sowohl bei REST als auch bei GraphQL ohne Bibliotheken nur sehr aufwändig umsetzten. Denn alle Parameter müssen mühsam mit *if-Anweisungen* validiert werden. Das ist nicht nur unübersichtlich, sondern es führt auch sehr wahrscheinlich zu sich wiederholenden Code. Auf die Art und Weise würden somit nur sehr einfache und wenige Validierung umgesetzt. Meisten werden deswegen Bibliotheken genutzt, welche sich in vielen Faktoren ähnlich sind. Spring Boot bietet bspw. eine Validierungslösung für beide Technologien an. Werden also Validierung benötigt, die über eine Datentyp-Validierung hinausgehen, hat keine der Technologien einen entscheidenden Vorteil.

4.5. Antwortzeit

Auch eine ausführliche Dokumentation und einfache Implementierung helfen nicht, wenn die die Antwortzeit zu hoch und der Client entsprechend unzufrieden ist.

4.5.1. Over- und Underfetching

Die komplizierteren Anfragen der GraphQL-Implementierung können benutzt werden, um sowohl Over- als auch Underfetching zu umgehen. Bei REST muss hingegen die Implementierung so designt werden, dass diese Probleme möglichst klein gehalten werden.

Overfetching

Für einen Overfetching-Fall könnte man einfach eine Erweiterung der Funktionalität des Kapitels Implementierung betrachten. Zusätzlich zu allen Daten der Routen will jetzt ein Frontend diese Routen auflisten. Dafür werden aber nur der Name und der Start- und Endpunkt benötigt. Für GraphQL würde sich dabei nichts ändern. Eine Anfrage wie im Code 4.8 zu sehen ist, würde diese Anforderungen exakt erfüllen.

```
query {
  routes {
    name,
    start,
    end
  }
}
```

Quellcode 4.8: Client Anfrage welche nur drei Felder pro Route zurückgibt

Bei REST ist die Lösung nicht so einfach. Würde man hier keine Änderungen durchführen, hätte man nur die Anfrage des Codes 4.5 zur Verfügung. Diese würde aber natürlich viel zu viele Daten für jede Route ausgeben. Es gibt für REST aber einige Lösungsansätze, um das Problem zu umgehen:

Zusätzliche Endpunkte: Diese Variante ist wohl die am weitesten verbreitete und auf den ersten Blick logischste. Jeder neue Fall wird einfach mit einem weiteren Endpunkt abgedeckt, welcher nur die benötigten Daten liefert. Dabei würde sich in vielen Fällen nur der Datenbankzugriff ändern. Der Code 4.9 zeigt dabei, dass die Implementierung nahezu identisch zum 4.3 Code aussehen würde. Eine Anfrage würde entsprechend nur an diesen neuen Endpunkt weitergleitet.

```
app.get('/routesList', async function (req, res, next) {
  var result = await db.getRoutesListData()
  return res.json(result)
})
```

Quellcode 4.9: REST Server Implementierung

Allerdings hat diese Variante zwei entscheidende Nachteile. Zum einen würde die Anzahl der Endpunkte im Laufe der Entwicklung drastisch zunehmen. Dadurch würde einerseits der Entwicklungsaufwand des Servers steigen, andererseits würde die Nutzbarkeit der API eingeschränkt werden. Denn ein Frontendentwickler müsste sich jetzt durch eine immer komplizierter werdende Dokumentation durchschlagen, um den einen perfekt passenden Endpunkt auszuwählen. Das andere Problem liegt vor, wenn man die API auch öffentlich zur Verfügung stellt. Hier kann man gar nicht für jeden Fall einen Endpunkt zur Verfügung stellen, da man gar nicht weiß, was alles benötigt wird. Dadurch wäre die Entwicklung eines Projektes mit diesem Verfahren sehr aufwändig und eingeschränkt.

Feldparamter: Eine weitere Möglichkeit wäre die Nutzung eines zusätzlichen Parameters. Dieser würde dabei die Funktionalität von GraphQL imitieren. Der Client könnte diesen Parameter nutzen, um anzugeben, welche Felder er erhalten will. Der Code

4.10 zeigt wie eine Anfrage aussehen könnte, welche an einen Server wie 4.11 gesendet werden würde, um unseren Anwendungsfall abzudecken.

```
fetch('http://localhost:3000/routes/name,start,end')
```

Quellcode 4.10: Client Anfrage mit Feldparametern

```
app.get('/routes/:fields', async function (req, res, next) {
            var results = await db.getRoutesData()
            var answers = [];
            var fields = req.params.fields;
            if (fields) {
                fields = req.params.fields.split(',')
                for (const result of results) {
                    var answer = {};
                    const keys = Object.keys(result)
                    for (const key of keys) {
                        if (fields.includes(key)) {
                            answer[key] = result[key];
                        }
                    }
                    answers.push(answer);
                }
            }
            return res.json(answers)
        })
```

Quellcode 4.11: Server Implementierung für Feldparameter

Doch auch hier zeigen sich schnell die Nachteile dieser Variante. Zum einen ist diese Implementierung wesentlich aufwändiger und unübersichtlicher. Für jeden Endpunkt, welcher diese Funktionalität zur Verfügung stellt, müsste entsprechend auch die Implementierung angepasst werden. Dabei deckt diese Variante nicht mal den gleichen Bereich ab, wie es GraphQL tun würde. Denn es können hier nur Felder ausgewählt werden, welche den primären Feldern der Datenstruktur entsprechen. Will man jetzt bei den Routendaten vom Code 4.1 nur die Koordinaten der Route haben, müsste das ganze Feld *Geometry* auswählt werden und somit auch alle weiteren Daten, die darin enthalten sind. Zusätzlich ist diese Variante auch wesentlich anfälliger für Fehler. Der Entwickler muss dabei genau mit der Dokumentation arbeiten, um die exakten Feldernamen zu nutzen und das Backend sollte wahrscheinlich eine Form der Validierung für diese Parameter aufbauen.

HATEOAS: Als Ursache mancher dieser Probleme können die REST-Prinzipien hergenommen werden. Der Abschnitt Unternehmen zeigt bereits, dass Github vor allem

HATEOAS als Ursache sieht. Für eine vollständige RESTful-API wäre die Liste, welche */routes* ausgibt, keine Liste aus vollständigen Routenobjekten. Stattdessen wären nur ein Link und einige wenige identifizierende Daten für jede Route vorhanden. Zusätzlich wären noch weitere Links auf verwandte Ressourcen oder Funktionalitäten Teil der Antwort, wie z. B. das Hinzufügen einer neuen Route mit */route*. Dadurch würde eine mögliche Antwort wie der Code 4.12 aussehen. Das Problem des Underfetching, welches dabei auftritt, wird in Sektion Underfetching behandelt. Das entscheidende Problem beim Overfetching sind die verwandten Ressourcen und Funktionalitäten. Bei einer stark vernetzten API kann es schnell vorkommen, dass hierbei eine unglaubliche Menge an Daten in Form von Links mitgeschickt wird. Doch damit sich das wirklich lohnt, müsste das Frontend diese Links auch aktiv nutzen. Doch wie schon das Kapitel Dokumentation gezeigt hat, hat dies kaum einen Mehrwert. Viele Entwickler nutzten die Dokumentation, um die Links direkt in den Code zu schreiben. Trotzdem müssen sie sich noch mit der ganzen Liste an Links rumschlagen, die mitgeschickt wird. Würde man HATEOAS hier entfernen, wäre eine ausführliche Dokumentation vonnöten. Da dies aber eigentlich Grundvoraussetzung für jede API ist, wäre das kein Hindernis. Gleichzeitig ist HATEOAS nach [Fielding 2008] eines der Kernprinzipien von REST und somit wäre eine API ohne HATEOAS eigentlich auch keine REST-API mehr. Falls HATEOAS also wirklich ein Hindernis darstellt, sollte man vielleicht Github folgen und auf GraphQL wechseln.

```
routes : {
  self: {
    href:       '/routes'
    addRoute:   '/routes'
  }
  data: [
    {
      uuid:     '3f441c99-67c1-48cd-8e97-9db483621eff'
      name:     'Ostsee'
      get:      '/routes/3f441c99-67c1-48cd-8e97-9db483621eff'
      delete:   '/routes/3f441c99-67c1-48cd-8e97-9db483621eff'
      patch:    '/routes/3f441c99-67c1-48cd-8e97-9db483621eff'
    },
    {
      uuid:     'abcde-67c1-48cd-8e97-9db483621eff'
      name:     'Nordsee'
      get:      '/routes/abcde-67c1-48cd-8e97-9db483621eff'
      delete:   '/routes/abcde-67c1-48cd-8e97-9db483621eff'
      patch:    '/routes/abcde-67c1-48cd-8e97-9db483621eff'
    }
  ]
```

```
22    }
```

Quellcode 4.12: Beispielliste der Routen mit HATEOAS

Alle Varianten haben somit ihre Nachteile und sind kein äquivalenter Ersatz zu GraphQL. Eine beliebte Lösung ist eine Mischung aus verschiedenen Varianten. So können bspw. einige sicher benötigte Fälle mit individuellen Endpunkten abgehandelt werden. Zusätzlich werden noch weitere Endpunkte zur Verfügung gestellt, welche größere Mengen an Daten zurückgeben. Hierbei wird, wenn es möglich, nützlich und nicht zu aufwändig ist, mit Feldparametern gearbeitet.

Underfetching

Underfetching baut theoretisch auf den gleichen Problemen auf wie das Overfetching. Wie bereits erwähnt, könnte ein Underfetching-Fall bspw. auftreten, wenn die Liste des */routes*-Endpunktes das HATEOAS-Prinzip erfüllt. Der Code 4.12 macht dabei nochmal deutlich, dass die Daten, welche für jede Route geschickt werden, nur sehr begrenzt sind. Wenn der Anwender jetzt alle Daten für alle Routen haben will, muss er erst eine Anfrage an den */routes*-Endpunkt abschicken, um alle Routen zu erhalten. Anschließend muss für jede Route eine individuelle Anfrage an den `route/{id}` geschickt werden, um die Daten der Routen zu erhalten. Dieses Problem ist als N+1-Problematik bekannt, d. h. um die Daten von N Routen zu erhalten, muss ich N+1 Anfragen abschicken [Kheyrollahi 2014].

Wendet man HATEOAS vollständig an, lässt sich das Problem kaum umgehen. Die einzige Lösung wäre es, die Menge an Daten, welche in der Routenliste zurückgegeben werden, zu erhöhen. Damit könnten wieder häufige Anwendungsfälle abgedeckt werden. Wenn also bei der Diskussion mit den Anwendern herauskommt, dass sie eigentlich immer den Start- und Endpunkt der Route benötigen, können diese einfach direkt beim */routes*-Aufruf zurückgegeben werden. Dabei treten jedoch wieder zwei Probleme auf. Einerseits führen mehr Daten unweigerlich auch zu mehr Overfetching. Das sollte jedoch nach Möglichkeit vermieden werden. Und andererseits haben wir wieder das Problem, dass diese Informationen häufig gar nicht vollständig vorliegen. Entwickler, welche nicht an der API beteiligt sind, sie aber für ihr Frontend nutzen, könnten ihre Datenwünsche nur beschränkt mitteilen. Und wenn man dann noch versucht, alle Wünsche zu erfüllen, könnte man einerseits auf HATEOAS ganz verzichten und andererseits hätte man wieder ein Overfetchingproblem.

Bei GraphQL ist Underfetching kein Problem. Das hat zwei Ursachen:

- **Datenstruktur als Graph:** Zu einem gewissen Grad versucht GraphQL das gleiche zu erreichen wie HATEOAS. Doch statt mit Links sind Datenstrukturen, welche nahe beiander liegen, direkt verwandt. Mittels einer Query können dann alle

diese verwandten Daten ausgelesen werden. Der Code 4.4 zeigt, wie eine einfache Implementierung dafür aussehen könnte.

- **Mehrere Queries:** Doch auch wenn die Datenstrukturen mal nicht verwandt sind, kann GraphQL Underfetching vermeiden. Jede Anfrage kann aus einer beliebigen Kombination von Mutationen oder Queries bestehen. Nicht möglich ist jedoch eine Kombination der beiden. Der Code 4.13 zeigt, wie komplett unabhängige Datenstrukturen mit nur einer Anfrage ausgelesen werden können. Mit verschiedenen Bibliotheken ist es sogar möglich, das Resultat der ersten Query direkt an die nächste Query zu übergeben.

```
1 query {
2   health,
3   routes {
4     uuid,
5     name
6   }
7 }
```

Quellcode 4.13: Beispiel Anfrage mit mehreren Queries

Fazit

Dieser Abschnitt hat gezeigt, warum der Hauptgrund für einen Wechsel von REST zu GraphQL die Vermeidung von Over- und Underfetching ist. Zwar gibt es verschiedene Lösungsansätze, um das Problem mit REST zu umgehen, doch keiner davon ist so effizient wie GraphQL.

4.6. Caching

Eine weitere Methode, die Datenmenge zu reduzieren, ist das Caching. In Kapitel 4.6 wurden bereits die verschiedenen Formen des Caching behandelt. Für diese Cachingvarianten gibt es zwei Möglichkeiten, sie umzusetzen. Einerseits kann auf HTTP-Caching zurückgegriffen werden. HTTP stellt dafür Header zur Verfügung, die genutzt werden können, um zu konfigurieren, wie und ob die Daten gecached werden können. Andererseits gibt es Bibliotheken, welche Lösungen für das Caching anbieten. Diese Lösungen bauen häufig teilweise auf dem HTTP-Caching auf, erweitern es aber noch, um bspw. flexibler zu werden. Sowohl für das Clientcaching als auch das Servercaching sollte für REST und GraphQL auf Bibliotheken oder Frameworks zurückgegriffen werden. Doch das ist für REST häufig gar nicht nötig. Denn Browsercaching kann

Freshness	Validierung
Expires: So, 14 June 2020 18:00:00 GMT	
Cache-Control: max-age=100	
ETag: `"abcdefghijkl1234567"`	If-None-Match: `"abcdefghijkl8901234"`
Last-Modified: So, 14 June 2020 18:00:00 GMT	If-Modified-Since: Mo, 15 June 2020 18:00:00 GMT

Tabelle 4.2: Übersicht über mögliche Header für HTTP Caching

mit geringem Aufwand ähnliche Funktionalitäten erfüllen. Dafür wird aber nur HTTP-Caching verwendet. Der Browser cacht defaultmäßig schon alle *HTTP-GET*-Anfragen. Für einen guten Cache sollten aber die Antworten so konfiguriert werden, dass der Browser weiß, was und wie lange gecacht werden kann. Das ist theoretisch nicht nur für *GET*, sondern auch für *HTTP-POST*-Anfragen möglich. Verschieden Browser ignorieren das aber und cachen *POST* Anfragen nicht. Auch alle anderen HTTP-Methoden können nicht gecacht werden. Das ist beides aber meistens kein wirklicher Nachteil, da diese Anfragen Daten auf dem Server ändern und somit so oder so durchgeschickt werden sollten. Um den Aufwand dieses Aspekts im Rahmen zu halten, werden für REST nur die *GET*-Anfragen betrachtet.
Um die Anfragen richtig zu konfigurieren, müssen nur die entsprechenden Header gesetzt sein. Diese haben zwei verschiedene Aufgaben: Die einen geben Ausschluss darüber, wie lange eine Antwort gecacht werden kann, die sogenannte *Freshness*. Das kann sowohl ein abgegrenztes Zeitfenster sein als auch eine Angabe über den Zustand der Ressource, welche zurückgeben wird. Der Browser nutzt diese Header automatisch, um den Browsercache zu validieren. Der Client kann jedoch auch andere Header nutzen, um explizite Angaben darüber machen zu wollen, ob der gecachte Wert noch valide ist. Das nennt sich *Validierung*. Die Möglichkeiten dafür werden in der Tabelle 4.2 aufgeführt:

Mit dem *Expires*-Header kann genau angegeben werden, bis zu welchem Datum der Wert gecacht werden kann. Mit *Cache-Control* ist die Flexibilität jedoch höher. Es kann unter anderem angegeben werden, dass die Antwort gar nicht cachbar ist oder aber auch, dass bei jeder Abfrage der Server angefagt werden soll, ob der Wert noch gültig ist. Um das zu überprüfen, kann ein *ETag*-Header verwendet werden. Der *ETag* identifiziert den Zustand einer Ressource. Verändert sich die Ressource muss sich auch der

ETag ändern, wodurch der gecachte Wert invalide wird. Ein Client kann bei einer Anfrage auch selber den *ETag* im *If-None-Match*-Header angeben. Falls er übereinstimmt, wird nur der gecachte Wert zurückgegeben. Falls kein *ETag* angeben ist, wird der ungenauere Last-Modified Header genutzt. Dieser gibt an, wann die Ressource das letzte Mal verändert worden ist. Der *If-Modified-Since*-Header kann genutzt werden, um diesen Wert direkt zu überprüfen.
Auf den ersten Blick sollten diese Möglichkeiten auch für GraphQL zur Verfügung stehen. Doch neben kleineren Problemen ist das wichtigste Problem das Speichern der Antworten im Cache. Wie bereits in Kapitel 4.6 angeschnitten wurde, muss dafür ein eindeutiger Schlüssel aus der Anfrage generiert werden. Im Falle von REST ist das relativ einfach. Bei *GET*-Anfragen kann alleine über die URL bestimmt werden, welche Daten benötigt werden. Damit kann die URL als Schlüssel für den Cache verwendet werden. Bei GraphQL ist das auf den ersten Blick nicht möglich. Denn hier wird immer der gleiche Endpunkt angefragt, die URL ist also für jede Anfrage identisch. Die Informationen der benötigten Daten sind alle im Körper der Anfrage enthalten. Für Clientcaches kann dieser dann auch genutzt werden, um einen Schlüssel zu generieren. Bibliotheken wie *akamai* unterstützen Frontend-Entwickler dabei [Akamai 2020]. Das automatisierte Browsercaching damit jedoch nicht möglich.
Eine andere Möglichkeit ist es, statt *POST*-Anfragen *GET*-Anfragen zu verwenden. Denn auch wenn in den meisten Fällen *POST* verwendet wird, ist es möglich, einen Server aufzusetzen, der auf *GET*-Anfragen hört. Die angefragte Datenstruktur wird damit nicht im Körper der Anfrage, sondern am Ende der URL als Parameter übergeben. Im Best-Case Scenario würde die Anfrage vom Code 4.14 dann sogar wesentlich kleiner werden, wie der Code 4.15 zeigt.

```
1  fetch('http://localhost:3000/graphql', {
2      method: 'POST',
3        headers: {
4            'Content-Type': 'application/json',
5      'Cache-Control': 'max-age=100'
6        },
7        body: JSON.stringify({
8            query: `
9              query {
10               routes {
11                     uuid
12                 }
13             }
14             `
15    })
16    })
```

Quellcode 4.14: GraphQL POST-Anfrage

```
1  fetch('http://localhost:3000/graphql/query='{routes {uuid}}', {
2        headers: {
3      'Cache-Control': 'max-age=100'
4        }
5    })
```

Quellcode 4.15: GraphQL GET-Anfrage

Problematisch sind dabei mehrere Faktoren. So können riesige und unübersichtliche URLs entstehen. Die Entwicklung mit der Routen-API hat gezeigt, dass die URLs sich auch nicht mit der Unterstützung von *GraphiQL* schreiben. Denn *GraphiQL* funktioniert nur mit *POST* zusammen. Zusätzlich kann es vorkommen, dass die URL zu lang für Browser werden. Diese Maximale Länge ist je nach Browser mit ca. 2000 Zeichen jedoch sehr lang [Microsoft 2019] und sollte somit eigentlich nie erreicht werden. Ein weiteres Problem ist jedoch, dass auch *Mutationen* in der *GET*-Anfrage möglich sind. Diese würden im HTTP-Kontext eigentlich einer *POST*-Anfrage oder ähnlichem entsprechen. Der Cache wird diese Anfragen jetzt aber genau wie *GET*-Anfragen behandeln. Dadurch können möglicherweise Anfragen nicht weitergegeben werden, welche eigentlich durchgereicht werden sollten. Doch auch dafür gibt es eine Lösung. Der GraphQL-Server könnte auf zwei Endpunkte aufgeteilt werden. Der eine funktioniert mit *GET* und nimmt nur *Queries* an, während der andere nur *Mutations* über *POST* entgegen nimmt. Das einzige Problem, das dabei auftritt, ist, dass Entwickler jetzt ihre Anfragestruktur abändern müssen, je nachdem, ob sie eine *Query* oder eine *Mutation* abschicken. Da das Gleiche bei REST aber eigentlich auch nötig ist, ist das kein wirklicher Nachteil gegenüber REST.
In beiden Fällen kann das automatisierte Browsercaching somit genutzen werden. Für GraphQL muss dafür jedoch teilweise auf die nützlichen Funtionen von *GraphiQL* verzichtet werden. Und ob das Caching dann wirklich effizient ist, ist eine andere Frage. Hier ist die erhöhte Flexibilität von GraphQL ein klarer Nachteil. Sobald sich ein Parameter ändert, verändert sich auch die URL. Folglich würden die Daten nicht aus dem Cache genommen werden und die Anfrage müsste somit vom Server beantwortet werden. Das ist aber kein Problem, was nur bei GraphQL auftritt. Wenn man bspw. die Feldparameterlösung von Kapitel 4.5.1 mit ihrer Anfrage 4.10 betrachtet, dann tritt hier das gleiche Problem auf. Für den Client- und Servercache versuchen Lösungen wie FlacheQL, das Problem zu umgehen. Dafür werden die Antworten auf ihre einzelnen Typen und Felder heruntergebrochen. Wenn eine Anfrage jetzt durch Teilmengen anderer Abfragen abgedeckt ist, kann sie direkt vom Cache beantwortet werden [FlacheQL 2019].
Wer sich über Nachteile von GraphQL informiert, stößt schnell darauf, dass Caching damit nicht möglich ist. Dieser Abschnitt hat gezeigt, dass das Caching im Allgemeinen, aber speziell auch das automatsierte Browsercaching, nicht nur möglich ist, sondern

sich der Aufwand kaum von REST unterscheidet. Die einzigen Probleme, welche dabei entstehen, sind ein Verlust von *GraphiQL* und ein möglicherweise ineffizienter Cache aufgrund der hohen Flexibilität von GraphQL.

5

Evaluation

Das vorherige Kapitel ist auf einige wichtige Aspekte beim Aufbau einer API eingegangen. Doch welche Erkenntnisse sich daraus ziehen lassen, kann ohne Kontext nicht beantwortet werden. Dieses Kapitel wird einige Uses-Cases aufstellen und anschließend untersuchen, im welchem dieser Fälle welche Technologie zu empfehlen ist. Zur Definition der Use-Cases werden die folgenden Parameter genutzt 5.1:

Schnittstellen	Wie viele Funktionen gibt es? Wie kompliziert sind sie?
Daten	Wie viele Daten werden realistischerweise ausgegeben? Welche Struktur haben diese?
Eingaben	Können Parameter übergeben werden? Können die gespeicherten Daten verändert werden?
Entwicklung	Wie aufwändig ist die Entwicklung? Wie häufig gibt es Änderungen? Ist ein Frontend vorgesehen?
Zielgruppe	Ist die API für die Öffentlichkeit gedacht oder nur für einen bestimmten Personenkreis? Soll es gut möglich sein eigene Frontends mit dieser API aufzubauen?

Tabelle 5.1: Parameter eines Use-Cases

Diese Parameter sollen zwei Eigenschaften erfüllen. Einerseits sollen sie einen direkten

Einfluss darauf haben, wie eine API aufgebaut sein muss. Andererseits soll sie dabei auch Auswirkungen auf einen der in Kapitel 4 untersuchten Aspekte haben. *Schnittstellen* und *Daten* geben dabei Überblick über die Komplexität der API. Durch die *Eingaben* soll gezeigt werden, wie ausführlich und kompliziert die Validierung sein muss, während *Entwicklung* und *Zielgruppe* die benötigte Flexibilität der API-Technologie widerspiegeln sollen.

5.1. Use-Cases

Web-APIs werden aus ganz unterschiedlichen Gründen gebaut. Ein Use-Case entspricht im Kontext dieser Arbeit einem dieser Gründe.

5.1.1. Einfache API

Use Case	**Einfache API**
Schnittstellen	Nur wenige Funktionen mit einer geringen Komplexität
Daten	Wenige Daten in kleinen Datenstrukturen
Eingaben	Keine oder nur wenige Übergabeparameter und keine Veränderung der Daten
Entwicklung	Schnelle Entwicklung des Servers; kein Frontend; keine Änderungen vorgesehen
Zielgruppe	API ist für die Öffentlichkeit entwickelt; Ein eigenes Frontend ist zwar theoretisch möglich, aber aufgrund der geringen Datenmenge nicht sinnvoll

Tabelle 5.2: Use-Case: Einfache API

Eine einfache API stellt nur wenige Informationen bereit. In vielen Fällen wird dafür sogar nur eine einzige Funktionalität genutzt. APIs dieser Art geben nur eine sehr geringe Datenmenge aus und sollen vor allem leicht zu nutzen sein. Ihre Entwicklung ist meis-

tens nicht besonders aufwändig, was auch daran liegt, dass es keinen Frontend-Client gibt. Solche APIs sollen von anderen Anwendungen aufgerufen werden, wenn sie genau die Information benötigen. Deswegen sind sie eigentlich immer für die Öffentlichkeit konzipiert. Als Beispiel kann eine API betrachtet werden, welche die aktuelle Zeit einer Zeitzone ausgibt.

5.1.2. Einfache API mit großer Datenstruktur

Dieser Use-Case beschreibt eine Weiterführung des vorherigen Falles. Auch hier gibt es eine entscheidende Funktionalität, welche aber eine wesentlich größere Datenstruktur ausgibt, als beim vorherigen Fall. Unter eine größeren Datenstruktur versteht man im Kontext dieser Evaluation eine Struktur mit mehr als 15 unterschiedlichen Feldern. Damit einher geht häufig eine etwas höhere Komplexität der Schnittstelle. Als Beispiel für eine solche API kann die native SignalK-API genommen werden. Zwar sind hier mehrere Funktionalitäten möglich, doch die wichtigste Aufgabe ist die Rückgabe der Daten des eigenen Bootes. Diese sind Resultat von vielen verschiedenen Sensoren und bilden eine entsprechend große Datenstruktur.

Use Case	**Einfache API mit großer Datenstruktur**
Schnittstellen	Nur wenige Funktionen, durchschnittliche Komplexität
Daten	Wenige Daten in großen Datenstrukturen
Eingaben	Parameter können übergeben werden, aber die gespeicherten Daten können nicht verändert werden
Entwicklung	Schnelle Entwicklung des Servers; kein Frontend; Änderungen möglich
Zielgruppe	API ist für die Öffentlichkeit entwickelt; Ein eigenes Frontend ist zwar theoretisch möglich, aber aufgrund der geringen Datenmenge nicht sinnvoll

Tabelle 5.3: Use-Case: Einfache API mit großer Datenstruktur

5.1.3. Webseiten-API

Solche APIs werden für eine bestimmte Webseite oder App entwickelt. In vielen Fällen hat man aber festgestellt, dass es für die Nutzer von Vorteil ist, wenn sie direkten Zugriff und eine übersichtliche Dokumentation für die API erhalten. Die Nutzerzahlen einer solchen API sind meistens wesentlich höher. Als konkretes Beispiel kann hier die Twitter-API genannt werden.

Use Case	**Webseiten-API**
Schnittstellen	Viele Funktionalitäten mit durchschnittlicher Komplexität
Daten	Viele Daten in durchschnittlich großen Datenstrukturen
Eingaben	Es können Parameter übergeben werden und gespeicherte Daten können verändert werden
Entwicklung	Aufwändige Entwicklung; Frontend existiert meistens; Änderungen sehr wahrscheinlich
Zielgruppe	API ist für die Nutzung der Öffentlichkeit gedacht; Ein eigenes Frontend ist möglich und sinnvoll

Tabelle 5.4: Use-Case: Webseiten-API

5.2. Use-Case basierte Evaluation

Für jeden dieser Use-Cases muss jetzt bestimmt werden, ob sich der Einsatz von GraphQL lohnen würde. Dafür wird betrachtet, welche Auswirkungen die einzelnen Aspekte auf den jeweiligen Use-Case haben.

5.2.1. Einfache API

Die *Einfache API* braucht eigentlich nicht viel. Sie sollte vorzugsweise sehr schnell zu implementieren und einfach zu verwenden sein. Das Kapitel Implementierung hat gezeigt, dass dies mit REST einfacher umzusetzen ist. Die erhöhte Komplexität von Gra-

phQL, welche einen genaueren Datenzugriff ermöglicht kann hier keine Vorteile bringen. Wie im Kapitel Antwortzeit erläutert, führt die geringe Datenmenge dazu, dass die Antwortzeit des Servers eigentlich nur noch von der Latenz abhängt. In diesem Fall ist also der Einsatz von REST zu empfehlen.

Implementierung	Sehr einfache Implementierung sinnvoll
Dokumentation	Dokumentation ist sehr kurz, muss nicht automatisiert sein
Validierung	Sehr einfach, da kaum Übergabeparameter
Over- und Underfetching	Unwichtig, da Daten- und Funktionsmenge zu gering
Caching	Wenig nützlich, da Daten sich häufig ändern(Bsp.: Zeit ist immer anders)

Tabelle 5.5: Evaluation: Einfache API

5.2.2. Einfache API mit großen Datenstruktur

Auch bei diesem Use-Case sollte die Implementierung noch einfach umzusetzen sein. Nach Kapitel Implementierung wäre hierfür also wieder REST zu empfehlen. Je nach Größe und Komplexität der Datenstruktur kann es aber vorkommen, dass sehr viel Overfetching auftritt. Und das ist nicht nur für die Antwortzeit schlecht, es ist für einen Frontendentwickler auch unübersichtlicher zu nutzen. Der Abschnitt Over- und Underfetching hat hierfür einige Lösungsvorschläge genannt. Wenn eine große Flexibilität erwünscht ist, muss man GraphQL wählen. Meistens ist das aber nicht der Fall. Stattdessen will man bei einer solchen API entweder alles haben oder nur bestimmte Abschnitte, wie bspw. die Daten eines Sensors. Will man jedoch alles in einer großen Datenstruktur haben, kann die Anfrage bei GraphQL schnell riesig werden. Und um nur Abschnitte zu erhalten, reicht die Feldparameterlösung meistens aus. Einer API, wie der SignalK-API, ist also zu empfehlen, REST zusammen mit einer Feldparameterlösung zu wählen.

Implementierung	Einfache Implementierung sinnvoll
Dokumentation	Dokumentation ist sehr kurz, muss nicht automatisiert sein
Validierung	Einfach, da keine Datenveränderung möglich
Over- und Underfetching	Zu geringe Datenmenge für Underfetching, Overfetching kann vorkommen
Caching	Wenig nützlich, da Daten sich häufig ändern (Bsp.: Sensordaten des Bootes)

Tabelle 5.6: Evaluation: Einfache API mit großen Datenstruktur

5.2.3. Webseiten-API

Bei einer solchen API ist die Implementierung schon aufgrund der vielen Funktionalitäten komplizierter. Um die API noch gut nutzen zu können, ist eine ausführliche Dokumentation nötig, aber auch aufwändig. Wie in Kapitel Dokumentation gezeigt, eignet sich GraphQL dafür sehr gut. Die automatisierte Erstellung einer interaktiven Dokumentation spricht in diesem Fall klar für GraphQL. Da die Validierung hier genauerer Abfragen bedarf, hat GraphQL zwar einen leichten, aber keinen entscheidenden Vorteil. Denn das Kapitel Validierung hat schon gezeigt, dass für solche Lösungen beide Technologien auf Bibliotheken zurückgreifen müssen.
Um bei einer solchen API eine gute Antwortzeit zu erreichen, müssen viele Probleme gelöst werden. So liegt hier ein klarer Fall von Over- und Underfetching vor. Die hohen Datenmengen führen schnell zu einem Fall von Overfetching. Und die vielen Funktionalitäten haben gleich mehrere Nachteile. Einerseits würde der Einsatz von HATEOAS wohl dazu führen, dass sehr viele Links mitgeschickt werden müssten. Dadurch würde sich das Overfetching noch verschärfen. Andererseits ist es dadurch auch wahrscheinlicher, dass ein Underfetching-Fall auftritt. Wie der Abschnitt Over- und Underfetching gezeigt hat, könnte GraphQL eine Lösung für diese Probleme anbieten. Um eine gute Antwortzeit zu erreichen, muss aber vor allem ein gutes Caching vorliegen. Viele der Daten werden sich in kurzer Zeit kaum bis gar nicht verändern und können somit sehr gut gecacht werden. Falls der Cache nicht absolut effizient sein muss, kann auch hier noch GraphQL mit einer GET-Anfrage genutzt werden. Für bessere Lösungen sollten dann Bibliotheken genutzt werden.

Doch noch ein weiterer Aspekt spricht für GraphQL. *Webseiten-APIs* werden meistens dauerhaft weiterentwickelt. So können einerseits im Backend weitere Parameter hinzukommen, andererseits können neue Seiten im Frontend aufgebaut werden. GraphQL ermöglicht durch die erhöhte Flexibilität bei der Datenabfrage, dass solche Änderungen in vielen Fällen unabhängig zwischen Back- und Frontend durchgeführt werden können.
Zusammengefasst lässt sich sagen, dass es *Twitter* daher zu empfehlen ist, auf GraphQL umzusteigen.

Implementierung	Implementierung ist aufgrund der hohen Funktionalitätsanzahl komplizierter
Dokumentation	Dokumentation ist sehr aufwändig, Automatisierung ist zu empfehlen
Validierung	Kompliziert, viele Eingaben möglich, gespeicherte Daten können verändert werden
Over- und Underfetching	Hohe Datenmenge und Funktionalitätsanzahl führt zu Over- und Underfetchingproblemen
Caching	Sehr nützlich, da sich die meisten Daten kaum ändern

Tabelle 5.7: Evaluation: Webseiten-API

5.3. Zusammenfassung und Einstufung der Ergebnisse

Die Use-Cases haben gezeigt, dass sich der Einsatz von GraphQL eigentlich erst bei komplizierteren APIs lohnt. Die etwas kompliziere Implementierung, vor allem im Frontend ist dabei als Einstiegshürde für GraphQL zu nennen. Gleichzeitig kommen die Vorteile von GraphQL aber auch erst bei solchen APIs zum Tragen. Diese Vorteile liegen vor allem in der Datenabfrage. Ein Frontendentwickler kann immer genau das anfordern, was er benötigt und weiß immer genau, wie die Antwort aufgebaut ist. Das ist ein Resultat aus dem Typesystem und der *Graph Query Language*. Die erhöhte Flexibilität, die sich daraus ergibt, kann eine unabhängige Entwicklung von Back- und Frontend unterstützen.

Wie aussagekräftig diese Ergebnisse sind, hängt noch von vielen weiteren Faktoren ab. Dafür müssen einerseits noch weitere wichtige Aspekte untersucht werden, welche in dieser Arbeit keinen Platz gefunden haben. Dazu gehören Authentifikation und Autorisierung, d. h. wer ist der Nutzer und was darf er tun. Interessant ist dabei auch, wie gut man die Zugriffsrechte auf auf bestimmte Felder und Endpunkte konfigurieren kann. Außerdem muss gezeigt werden, wie gut sich große binäre Dateien wie Bilder mit der jeweiligen Technologie übertragen lassen. Auch Errorhandling wurde nicht behandelt, sollte aber beim Aufbau einer guten API immer eine wichtige Rolle spielen. Des Weiteren müssten die verschiedenen Bibliotheken und Frameworks genauer betrachtet werden, um zu schauen, welche Features sie bieten. Zuletzt muss noch der Architekturstil der Software betrachtet werden. Dabei müssen vor allem verteilte Architekturen wie Microservices betrachtet werden.

Use-Case	Empfohlene Technologie
Einfache API	REST
Einfache API mit großen Datenstruktur	REST
Webseiten-API	GraphQL

Tabelle 5.8: Evaluation: Zusammenfassung

6

Fazit

Ziel dieser Arbeit war es festzustellen, wann sich der Einsatz von GraphQL gegenüber REST für eine Web-API lohnt. Zu diesem Ziel wurden die Aspekte Verbreitung, Implementierung, Dokumentation, Validierung und Antwortzeit untersucht.
Bei der Verbreitung konnten kaum Unterschiede zwischen GraphQL und REST nachgewiesen werden. Beide können in eigentlich allen sinnvollen Programmiersprachen umgesetzt werden. Dasselbe gilt auch für die Validierung. Zwar hat hier GraphQL mit dem Typesystem einen leichten Vorteil, wenn man aber kompliziertere Validierungen durchführen muss, müssen beide Technologien auf Bibliotheken zurückgreifen. Diese Aspekte haben folglich kaum bis gar keinen Einfluss auf die Entscheidung.
Betrachtet man die grundlegende Implementierung, hat REST einen leichten Vorteil, welcher vor allem bei einfachen APIs zum Tragen kommt. Das liegt daran, dass bei GraphQL sowohl die Anfragen als auch das Schema in einer eigenen Sprache formuliert werden müssen. Die Anfragen sind dabei auch bei erfahrenen Entwicklern wesentlich komplizierter als die meisten REST-Anfragen.
Sobald aber die Komplexität ansteigt, kann GraphQL durch eine einfachere Dokumentation und eine verbesserte Antwortzeit punkten. Die Dokumentation wird dabei von GraphQL automatisch übernommen. Bei REST muss für die gleiche Funktionalität auf Bibliotheken zurückgegriffen werden. Die gute Antwortzeit ist ein Resultat des flexiblen Datenzugriffs von GraphQL. Dadurch können die bei REST verbreiteten Probleme des Over- und Underfetching vermieden werden. Dieser flexible Datenzugriff ermöglicht auch eine unabhängigere Entwicklung von Front- und Backend, was ein weiterer Vorteil für GraphQL ist. Der Aspekt des Caching, welcher bei GraphQL häufig kritisch gesehen wird, wurde in dieser Arbeit als relativ äquivalent zwischen beiden Technologien erkannt. Zwar funktioniert das automatische Browsercaching bei REST sofort, das

Gleiche kann bei GraphQL aber auch mit einem GET-Endpunkt erreicht werden. Nur in Ausnahmefällen versagt diese Lösung.
Zusammengefasst lässt sich sagen, dass für die untersuchten Aspekte einfache APIs eher REST wählen sollten, während kompliziertere APIs mehr auf GraphQL setzten sollten.
Wie aussagekräftig diese Empfehlungen sind, hängt aber noch von vielen weiteren Faktoren ab. Einige wichtige Aspekte konnten bspw. im Rahmen dieser Arbeit nicht mehr behandelt werden. Dazu gehören unter anderem, wie gut sich größere binäre Dateien, wie Bilder mit REST und GraphQL übertragen lassen. Außerdem müssen vorhandene Bibliotheken noch genauer betrachtet werden, da ihre Features möglicherweise eine Lösung für entscheidende Probleme bieten. Zuletzt sind noch einige Use-Cases nicht betrachtet worden, da hier noch weitere Arbeit vonnöten ist. Dazu gehören vor allem Softwarelösungen mit einer verteilten Architektur wie Microservices.

Literatur

Fielding, Roy (2000). *Representational State Transfer (REST)*. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (besucht am 03. 06. 2020).

— (Sep. 2007). *The Rest of REST*. URL: https://roy.gbiv.com/talks/200709_fielding_rest.pdf (besucht am 05. 06. 2020).

— (20. Okt. 2008). *REST APIs must be hypertext-driven*. URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (besucht am 05. 06. 2020).

BSI, Hrsg. (3. Sep. 2013). *Leitfaden zur Entwicklung sicherer Webanwendungen. Empfehlungen und Anforderungen an die Auftragnehmer*, S. 42. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Webanwendungen/Webanw_Auftragnehmer.pdf?__blob=publicationFile&v=2 (besucht am 19. 06. 2020).

Kheyrollahi, Ali (2. Juni 2014). *Web APIs and the n+1 Problem*. URL: https://www.infoq.com/articles/N-Plus-1/ (besucht am 17. 06. 2020).

Facebook (14. Sep. 2015). *GraphQL. A data query language*. URL: https://engineering.fb.com/core-data/graphql-a-data-query-language/ (besucht am 21. 06. 2020).

Sturgeon, Phil (11. Aug. 2015). *Build APIs you won´t hate*, S. 127. ISBN: 978-0692232699.

Github (2016). *The GitHub GraphQL API*. URL: https://github.blog/2016-09-14-the-github-graphql-api/ (besucht am 17. 06. 2020).

graphql-docs (2016). *GraphQL Documentation Explorer*. URL: https://www.npmjs.com/package/graphql-docs (besucht am 19. 06. 2020).

Rajak, Sanjay (1. Juli 2017). *API Latency vs Response time*. URL: https://medium.com/@sanjay.rajak/api-latency-vs-response-time-fe87ef71b2f2 (besucht am 16. 06. 2020).

Ullenboom, Christian (2017). *Java SE 9 Standard-Bibliothek*. Hrsg. von Rheinwerk Verlag. 3. Aufl., S. 867. ISBN: 978-3-8362-5874-6.

Dupree, Matt (24. Apr. 2018). *The so-called 'RESTful' web in 2018 and beyond*. URL: https://www.philosophicalhacker.com/post/rest-in-2018/ (besucht am 08. 06. 2020).

GraphQL (10. Juni 2018). *GraphQL Specification 2018*. URL: http://spec.graphql.org/June2018/ (besucht am 08. 06. 2020).

1und1 (3. Sep. 2019). *GraphQL: Flexible Abfragesprache und Laufzeitumgebung für Ihr Web-API*. URL: https://www.ionos.de/digitalguide/websites/web-entwicklung/graphql/ (besucht am 08. 06. 2020).

Enge, Eric (11. Apr. 2019). *Mobile vs. Desktop Usage in 2019*. Hrsg. von Perficient. URL: https://www.perficient.com/insights/research-hub/mobile-vs-desktop-usage-study (besucht am 27. 05. 2020).

FlacheQL (2019). *FlacheQL*. URL: https://github.com/FlacheQL/FlacheQL (besucht am 18. 06. 2020).

McKinnon, Jenni (29. Okt. 2019). *Site Cache vs Browser Cache vs Server Cache: What's the Difference?* URL: https://wp-rocket.me/blog/different-types-of-caching/ (besucht am 16. 06. 2020).

Microsoft (2019). *Maximum URL length is 2,083 characters in Internet Explorer*. URL: https://support.microsoft.com/en-us/help/208427/maximum-url-length-is-2-083-characters-in-internet-explorer (besucht am 24. 06. 2020).

Spichale, Kai (2. Mai 2019). *API Design. Praxishandbuch für Java- und Webservice-Entwickler*. Hrsg. von dpunkt.verlag. 2. Aufl., S. 7–9, 150–151. ISBN: 9783864906114.

Akamai (2020). *GraphQL caching*. URL: https://learn.akamai.com/en-us/webhelp/api-gateway/api-gateway-user-guide/GUID-7019E774-7A4D-44F9-A731-330F9780C34B.html (besucht am 20. 06. 2020).

Apollo (2020). *Introduction. What is Apollo Client?* URL: https://www.apollographql.com/docs/react/ (besucht am 18. 06. 2020).

GraphQL (2020a). *Code*. URL: https://graphql.org/code/ (besucht am 18. 06. 2020).

— (2020b). *Running an Express GraphQL Server*. URL: https://graphql.org/graphql-js/running-an-express-graphql-server/ (besucht am 24. 06. 2020).

SignalK (2020). *SignalK*. URL: https://github.com/SignalK/signalk-server-node (besucht am 12. 06. 2020).

Slant (2020). *What are the best web frameworks to create a web REST API?* URL: https://www.slant.co/topics/1397/~best-web-frameworks-to-create-a-web-rest-api (besucht am 18. 06. 2020).

Spring (2020a). *Building a RESTful Web Service*. URL: https://spring.io/guides/gs/rest-service/ (besucht am 18. 06. 2020).

— (2020b). *Getting Started with GraphQL and Spring Boot*. URL: https://www.baeldung.com/spring-graphql (besucht am 18. 06. 2020).

Swagger (2020). *API Development for Everyone*. URL: https://swagger.io/ (besucht am 19. 06. 2020).

Twitter (2020). *API reference index*. URL: https://developer.twitter.com/en/docs/api-reference-index (besucht am 17. 06. 2020).