Vergleich zwischen den API-Architekturstilen Representational State Transfer und GraphQL

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science
des Fachbereichs Informatik und Medien der
Technischen Hochschule Brandenburg

vorgelegt von:
Maximilian Milz

Betreuer: Prof. Dr.-Ing. Thomas Preuß
Zweitgutachter: Dipl.-Inform. (FH) Stefan Pratsch, M. Sc.

Brandenburg an der Havel, 11. September 2021

Kurzfassung

Representational State Transfer (REST) ist ein ressourcenorientierter Ansatz für die Realisierung von Web-APIs, welcher sich mit Anfang der 2000er etablierte und bis heute als de-facto Standard angesehen wird. In dem Jahr 2015 veröffentlichte *Facebook* seine quelloffene Datenabfrage- und Manipulationssprache *GraphQL*. Das Ziel dieses Projektes ist die Entwicklung einer Alternative zu REST und einer SQL-ähnlichen Abfragesprache für die Entwicklung von Webanwendungen. Um GraphQL und REST bewerten zu können, wird im Zusammenhang dieser Arbeit ein Leistungsvergleich zwischen den beiden API-Architekturstilen durchgeführt.

Schlüsselwörter

GraphQL, REST, Anwendungs-Programmierschnittstelle

Abstract

Representational State Transfer (REST) is a resource-oriented approach for the realization of web APIs, which became established in the early 2000s and is still considered the de-facto standard today. In the year 2015 *Facebook* released its open-source data query and manipulation language *GraphQL*. The goal of this project is to develop an alternative to REST and an SQL-like query language for web application development. To evaluate GraphQL and REST, a performance comparison between the two API architectural styles is performed in the context of this work.

Keywords

GraphQL, REST, Application Programming Interface

Inhaltsverzeichnis

1 Einleitung

Studiert man öffentliche API-Verzeichnisse fällt auf, dass von Jahr zu Jahr die Tendenz Webanwendungen zu modularisieren stetig steigt. Das Onlineportal ProgrammableWeb veröffentlichte im Jahr 2019 einen Artikel über die Wachstumsrate ihres Verzeichnisses. Santos beschreibt, dass sich die Entwicklung von APIs von Anfang der 2000er von einer Seltenheit bis in das Jahr 2019 zu einem regelrechten Trend entwickelte. Auffällig ist dabei der hohe Anteil von REST-APIs. (Vgl. Santos, 2019) Dabei wurden in den letzten Jahren neben dem REST Ansatz zahlreiche Alternativen zur Entwicklung von Webanwendungen vorgestellt. Unter anderem gehört dazu die GraphQL-Spezifikation, welche 2015 von dem Tech-Giganten Facebook veröffentlicht wurde.

In der einschlägigen Literatur sind bereits zahlreiche Vergleiche zwischen REST und GraphQL vorhanden. In diesen vergleichenden Studien wird GraphQL tendenziell eine bessere Leistung bestätigt, wobei zu bemängeln ist, dass Entwicklungsszenarien, in denen die Verwendung von REST sinnvoll wäre von den Autoren außer Acht gelassen werden.

1.1 Ziel

Im Vordergrund dieser Arbeit steht die Untersuchung und Vergleich der Leistung von sowohl eines RESTful-API als auch eines GraphQL-API. Dadurch soll anhand von realen Messwerten ein Leistungsvergleich zwischen den beiden API-Architekturstilen erarbeitet werden, um Anwendungsszenarien herauszufiltern, in denen die Verwendung von REST sinnvoller ist als die Verwendung von GraphQL. Die vorliegende Arbeit setzt sich die Beantwortung zweier Kernfragen zum Ziel:

1. Sind GraphQL-APIs performanter als RESTful-APIs?
2. In welchen Entwicklungsszenarien wäre die Anwendung von REST sinnvoller?

1.2 Herangehensweise

Zuerst erscheint es notwendig, die Grundlagen zu den Begriffen API, REST und GraphQL zu klären (Kapitel 2). Aufbauend auf das Grundlagenwissen wird im Folgenden eine Beispielanwendung mithilfe des quelloffenen Java-Frameworks Spring Boot entworfen und umgesetzt, welche sowohl ein RESTful-API als auch ein GraphQL-API anbieten soll (Kapitel 3). Aufbauend darauf wird ein Leistungsvergleich entworfen, welcher sowohl die Definition der Leistung als auch die Formulierung einer Methodik zur Erfassung dieser Leistung miteinbezieht (Kapitel 4). Die Methodik dient dazu Metriken zu definieren, welche für den Vergleich der Leistungen zwischen den APIs verwendet werden. Anhand der Methodik wird ein Experiment entworfen, womit die Leistung der entworfenen APIs verglichen werden soll. Durch das Experiment werden Aufgaben definiert, welche beide APIs erfüllen müssen. Während der Abarbeitung der Aufgaben wird mithilfe der Methodik die Leistung der APIs erfasst. Im nachfolgenden Kapitel 5 wird dieses Experiment durchgeführt. Anhand der erlangten Daten können die APIs anschließend miteinander verglichen werden. Die Beantwortung der Kernfragen wird durch den Leistungsvergleich erreicht.

2 Application Programming Interfaces

Application Programming Interfaces (dt. Anwendungs-Programmierschnittstelle, API) spielen eine bedeutende Rolle in der heutigen Softwareentwicklung. Im Kontext dieser Arbeit steht ein Leistungsvergleich zwischen dem Architekturstil REST und der GraphQL-Spezifikation. Dafür wird in dem nachfolgenden Kapitel eine Beispielanwendung entworfen und umgesetzt, welche sowohl ein RESTful-API als auch ein GraphQL-API anbieten soll. Um diese Anwendung umsetzen zu können wird im Vorfeld eine Einführung in APIs gegeben, der Architekturstil REST vorgestellt und die GraphQL-Spezifikation beschrieben.

2.1 Definitionsansätze eines Application Programming Interface

Der Journalist David Berlind des Onlineportals ProgrammableWeb definiert APIs als Nutzerschnittstellen, die im Fokus jeweils andere Nutzer haben: “Application Programming Interfaces are user interfaces – just with different users in mind.” (Berlind, 2015) Diese prägnante Formulierung macht auf einen elementaren Punkt aufmerksam: Während eine Nutzerschnittstelle die Kommunikation zwischen einem menschlichen Akteur und einer Anwendung realisiert, liegt der Fokus bei einem API auf der Interaktion zwischen verschiedenen Anwendungen. Anders als bei dem Datenaustausch über eine Nutzerschnittstelle, so hebt es Spichale hervor, benötigt eine Maschine keine stilvolle graphische Oberfläche oder für den Menschen lesbare Zahlensysteme. Vielmehr wird eine Syntaxdefinition benötigt, um den Datenaustausch und die damit eingehende Kommunikation zwischen den Anwendungen zu standardisieren. Die Kommunikation zwischen Anwendungen begrenzt sich dabei auf die vorher definierten Schnittstellen eines APIs. Dabei beschreiben die Schnittstellen in welcher Form Informationen eingegeben und die daraus resultierenden Daten ausgegeben werden. (Vgl. Spichale, 2019 S. 7)

Joshua Bloch beschreibt ein API als eine Softwarekomponente, die näher spezifiziert wird durch die Methoden, die sie anwendet, und die Ein- und Ausgaben, die sie benutzt. Ihr Hauptzweck besteht aus der Beschreibung einer Anzahl an Methoden, unabhängig von ihrer Implementierung, denn die Implementierung kann variieren, ohne dass der Nutzer der jeweiligen Softwarekomponente beeinträchtigt wird.

> An application programming interface (API) specifies a component in terms of its operations, their inputs, and outputs. Its main purpose is to define a set of functionalities that are independent of their implementation, allowing the implementation to vary without compromising the users of the component. (Bloch, 2014 S. 34)

Die Definition von Bloch legt nahe, dass API, betrachtet man nur den Begriff, eine Bezeichnung für viele unterschiedliche Ausprägungen ist.

Auf der höchsten Abstraktionsebene wird zwischen Programmiersprachen- und Remote-APIs unterschieden. Unter Programmiersprachen-APIs werden beispielsweise Bibliotheken kategorisiert, welche typischerweise sprach- und plattformabhängig sind. Diesen APIs stehen die Remote-APIs gegenüber. Typische Remote-APIs wären RESTful- und SOAP-Webservices, sowie Messaging-APIs. Diese APIs verwenden meist Protokolle wie beispielsweise HTTP, welche sie sprach- und plattformunabhängig machen. (Vgl. Spichale, 2019 S. 7-8)

Des Weiteren kann zwischen öffentlichen und privaten APIs unterschieden werden. Öffentliche APIs besitzen keine Einschränkungen bei dem Zugriff auf das API, da sie für die Allgemeinheit öffentlich nutzbar sind. Allgemein verfolgt diese Typisierung eine hohe Verbreitung und Verwendung des API. Private APIs hingegen stehen nur Internen und eventuellen Partnern des API-Anbieters zur Verfügung. (Vgl. Kress, 2020 S. 7-8)

Darüber hinaus werden APIs auch als Web-APIs bezeichnet. Ein Web-API ist das Muster von HTTP-Anfragen und -Antworten, das für den Zugriff auf eine Webseite oder Plattform, die im Web zu erreichen ist verwendet wird, die für den Zugriff durch beliebige Computerprogramme und nicht durch von Menschen verwendete Webbrowser spezialisiert ist (vgl. Apigee S. 8). Da in den nachfolgenden Kapiteln zwei Web-APIs entworfen und umgesetzt werden, wird die Bezeichnung API als Ersatz für den Begriff Web-API verwendet. Diese APIs basieren auf den API-Architekturstilen Representational State Transfer (REST) und GraphQL, welche in den nächsten beiden Abschnitten beschrieben werden.

2.2 Architekturstil Representational State Transfer

Representational State Transfer (dt. repräsentative Zustandsübertragung, REST) ist ein Paradigma für die Realisierung von verteilten Systemen, welches auf dem Hypertext Transfer Protokoll aufsetzt und erstmalig von Roy Thomas Fielding in seiner Dissertation aus dem Jahr 2000 beschrieben wurde. Laut Fielding ist REST selbst kein Protokoll oder Standardisierung, vielmehr beschreibt er REST als Leitfaden, an dem sich ein Entwickler oder Entwicklerin bei der Implementierung von Webservices richten soll. (Vgl. Fielding 2000) Dieser Leitfaden stütz sich auf drei Kategorien – Ressourcen, Prinzipien und Methoden – welche im Folgenden ausführlich dargestellt werden.

2.2.1 REST-Ressourcen

Fielding beschreibt Ressourcen als jede beliebige Information, welchem einem Namen zugeordnet werden kann (vgl. ebd. S. 88). Beispielsweise könnte eine Ressource ein*e Benutzer*in einer Webseite oder der aktuelle Wetterstatus einer Wetterapplikation sein. Ressourcen werden über Uniform Resource Identifier (dt. einheitliche Ressourcenbezeichner, URI) identifiziert. Die Informationen, welche beim Aufruf dieser Ressource zurückgegeben werden, sind allgemein als Repräsentation bekannt.

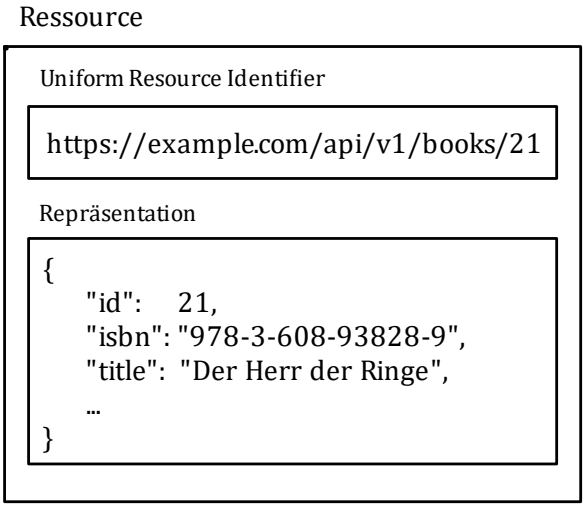

Abbildung 1 REST Ressource

Mit der Abbildung 1 wird exemplarisch eine Buch Ressource veranschaulicht. Das Buch wird als JSON repräsentiert und kann mithilfe der *id* des Buches über eine eindeutig identifizierbare URL abgerufen werden. Anzumerken ist, dass das Buch auch durch andere Datenformate wie XML oder beispielsweise als PDF repräsentiert werden kann.

2.2.2 REST-Designprinzipien

Fielding formuliert mit REST sechs Prinzipien für die Architektur von verteilten Systemen. Mit diesen Prinzipien werden Eigenschaften umgesetzt, welche laut Fielding für eine Webarchitektur empfehlenswert sind (vgl. ebd. S. 66-71). Im Folgenden werden diese Eigenschaften dargelegt.

- *Geringe Einstiegsbarriere*: Zum einen sollen REST-Dienste eine geringe Einstiegsbarriere anbieten, sodass die Benutzer*innen des Dienstes ohne Vorkenntnisse leicht die Funktionalität des Dienstes verstehen und ihn verwenden können. Durch eine geringe Einstiegsbarriere wird gewährleistet, dass viele Nutzer*innen erreicht werden können. (Vgl. ebd. S. 67)
- *Veränderbarkeit*: Neben der geringen Einstiegsbarriere soll ein REST-Dienst veränderbar sein (vgl. ebd. S. 68). Unter der Veränderbarkeit wird die Evolvierbarkeit, Erweiterbarkeit, Konfigurierbarkeit und Wiederverwendbarkeit verstanden (vgl. ebd. S. 33-35). Selbst wenn die Entwicklung einer Anwendung gelingen sollte, welche genau den Anforderungen eines Benutzers oder einer Benutzerin entspricht, werden sich diese Anforderungen nach der Inbetriebnahme der Anwendung wahrscheinlich verändern - deshalb muss eine Architektur veränderbar sein.
- *Verteiltes Hypermedia*: Darüber hinaus stellt Fielding fest, dass ein REST-Dienst verteilte Hypermedien anbieten soll. Unter verteilten Hypermedien wird die Angabe von Steuerinformationen verstanden, durch denen Benutzer*innen beispielsweise Zustandsübergänge zu anderen Ressourcen ermöglicht wird.
- *Skalierbarkeit*: Als letzte Eigenschaft erwähnt Fielding die Skalierbarkeit eines Dienstes. (Vgl. ebd. S. 68-69)

Die Realisierung dieser Eigenschaften wird durch folgende REST-Prinzipien erreicht.

Client-Server-Modell

Das Client-Server-Modell beschreibt die Kommunikation zwischen einem Server, welcher beliebig viele Dienste zur Verfügung stellt und einem Client, welcher diese Dienste nutzt. Der Client schickt dabei eine Anfrage an den Server und bekommt daraufhin eine Antwort zurückgeliefert. (Vgl. ebd. S. 45-46)

Zustandslosigkeit

Unter der Zustandslosigkeit wird die zustandlose Kommunikation zwischen dem Client und dem Server verstanden. Das heißt, dass der Server keine Informationen bezüglich der Anfrage des Clients besitzt. Alle relevanten Informationen, welche für die Erfüllung einer Anfrage benötigt werden, müssen von dem Client mitgeschickt werden. (Vgl. ebd. S. 47)

Mehrschichtige Systeme

Laut Fieldings Darstellung sollen REST-Dienste mehrschichtig aufgebaut sein. Das bedeutet, dass zwischen Client und Server noch weitere Komponenten eingebaut werden können, welche als Vermittler zwischen Client und Server dienen. Komponenten einer Schicht können dabei nur mit Komponenten benachbarter Schichten kommunizieren, wodurch die dahinterliegenden Schichten für die Komponente nicht sichtbar sind. Eine Aufgabe eines Vermittlers könnte beispielsweise die Umformung einer Nachricht sein. (Vgl. ebd. S. 46)

Einheitliche Schnittstelle

Unter dem Prinzip der einheitlichen Schnittstelle versteht Fielding, dass Server und Client auf Basis eines Protokolls und eines Datenformates kommunizieren und dabei die Semantik dieser einhalten. Beispielsweise verwenden die sogenannte charakterisierte RESTful-Implementierung das Hypertext Transfer Protokoll und zum Beispiel das Datenformat JavaScript Object Notation (JSON) für die Kommunikation und dem Datenaustausch zwischen Client und Server. Das Prinzip der einheitlichen Schnittstellen gliedert sich in weitere Unterprinzipien:

- *Adressierbarkeit*: Jede Ressource besitzt einen URI über den diese identifizierbar ist.
- *Selbstbeschreibende Nachrichten:* Nachrichten besitzen sämtliche Informationen, welche für die Realisierung einer Anfrage benötigt werden.
- *Hypermedia für den Anwendungsstatus:* Der Grundgedanke hinter diesem Prinzip ist, dass Benutzer ohne Vorwissen über das API außer der Einstiegs-URL in der Lage sind über Hypermedia die Zustandsübergänge des API zu verwenden.
- *Manipulation von Ressourcen durch Repräsentation:* Für die Manipulation einer Ressource muss eine veränderte Repräsentation der Ressource an den Server geschickt werden. (Vgl. ebd. S. 81-82)

Caching

Ein Cache beschreibt in der Informatik einen schnellen Speicher, in welchem aufwendig berechnete Daten zwischengespeichert und dadurch wiederverwendbar gemacht werden. Laut Fielding soll ein solcher Cache zwischen Client und Server als Abfangmechanismus eingebaut werden. Durch die zustandlose Kommunikation und den selbstbeschreibenden Nachrichten können dadurch Anfragen, welche bereits von einem oder mehreren Clients gestellt wurden, vom Cache abgerufen werden, ohne dass der Server diese Daten neu berechnen muss. (Vgl. ebd. S. 44, 48)

Code-On-Demand

Laut Fielding ist dieses Prinzip optional. Aus Fieldings Beschreibung geht hervor, dass es möglich sein sollte mit REST nachladbare Programmteile als Erweiterung des Dienstes zu übertragen. Dabei könnte es sich beispielsweise um ein JavaScript handeln. (Vgl. ebd. S. 53)

2.2.3 REST-Methoden

Mit REST wird ein ressourcenorientiertes Paradigma für die Entwicklung von Webdiensten formuliert. Nach Fieldings Vorstellungen soll REST das Hypertext Transfer Protokoll verwenden (vgl. ebd. Abschnitt 6.3). Für den Zugriff auf Ressourcen werden primär die HTTP-Methoden GET, POST, PUT und DELETE verwendet, wodurch

die typischen CRUD-Operationen[1] abgebildet werden. Durch diese Methoden wird der schreibende und manipulierende Zugriff auf eine spezifische Ressource oder komplette Menge aus Ressourcen verwirklicht. (Vgl. Apigee S. 6)
Mit der Tabelle 1 wird exemplarisch die Verwendungsweise der HTTP-Methoden im Zusammenhang mit dieser Arbeit dargestellt.

Ressource	GET read	POST create	PUT update	DELETE delete
/books	Alle Bücher auflisten	Neues Buch erstellen	funktioniert nicht	funktioniert nicht
/books/23	Buch darstellen	funktioniert nicht	Aktualisieren eines Buches, wenn vorhanden	Löschen eines Buches

Tabelle 1 REST Methoden

Durch die GET-Methode kann eine Menge von Ressourcen oder eine spezifische Ressource dargestellt werden. Mithilfe der POST-Methode kann die Repräsentation einer neuen Ressource an den Server geschickt werden, welcher diese anschließend erstellt. Für eine nachträgliche Bearbeitung einer Ressource wird die PUT-Methode verwendet. Mit dieser Methode muss die überarbeitete Repräsentation über die HTTP-Anfrage mitgeschickt werden. Für das Löschen einer Ressource wird die DELETE-Methode verwendet.
Nachdem ausführlich der Architekturstil REST dargestellt wurde, ist es folgerichtig auf die GraphQL-Spezifikation einzugehen.

2.3 Alternative zu REST: GraphQL-Spezifikation

Die GraphQL-Spezifikation wurde 2015 von dem Tech-Giganten Facebook veröffentlicht. Die Spezifikation beschreibt GraphQL zum einen als Sprache zur Abfrage und Manipulation von Daten, sowie als serverseitige Laufzeitumgebung. Das Ziel des Projektes ist die Entwicklung einer Alternative zu REST-APIs, sowie eine SQL-ähnliche Abfragesprache. (Vgl. Byron, 2018)

[1] Create, Read, Update, Delete

REST verfolgt einen ressourcenorientierten Ansatz, bei dem jede Ressource über einen gekapselten Endpunkt aufgerufen werden kann. Byron betrachtet diesen Ansatz als problematisch, da durch ihn oft nicht die benötigten Daten abgerufen werden. Deswegen werden mit GraphQL die Anwendungsdaten nicht als Ressourcen betrachtet. Vielmehr werden mit GraphQL diese als Objekte definiert, welche in einem Graphen angeordnet werden.

> We were frustrated with the differences between the data we wanted to use in our apps and the server queries they required. We don't think of data in terms of resource URLs, secondary keys, or join tables; we think about it in terms of a graph of objects and the models we ultimately use in our apps like NSObjects or JSON. (Byron, 2015)

Mit der Abbildung 2 wird ein Graph dargestellt. Die Knoten des Graphen repräsentieren die Objekte *Book*, *Author* und *Category*. Die Kanten, welche die Knoten miteinander verbinden, setzen die Relationen zwischen den Objekten um.

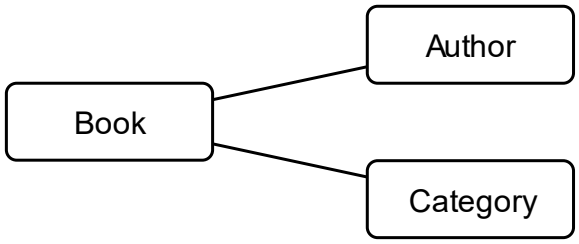

Abbildung 2 Beispielgraph

Ruft ein Client beispielsweise das Book-Objekt auf, kann er zusätzlich mit dem gleichen Aufruf das Author-Objekt und/oder Category-Objekt aufrufen.
Mit dem nachfolgenden Kapitel werden die GraphQL-Komponenten näher dargelegt.

2.3.1 GraphQL-Komponenten

GraphQL setzt sich aus drei elementaren Komponenten zusammen. Der Abfragesprache, das Typsystem und die Laufzeitumgebung. Für ein gutes Verständnis über die Funktionsweise von GraphQL werden folglich diese Komponenten vorgestellt und definiert.

Abfragesprache

Um als Client eine Anfrage an einen GraphQL-Dienst zu stellen, wird die GraphQL Query Language (dt. GraphQL Abfragesprache) verwendet. Die Anfragequellen werden allgemein als GraphQL Dokumente bezeichnet. Ein solches Dokument kann sowohl die drei Operationen Query, Mutation und Subscription als auch Fragmente enthalten. (Vgl. Byron, 2018 Kapitel 2)

- *Operationen von GraphQL*

  Query: stellt den Lesezugriff auf Daten zur Verfügung.

  Mutation: ermöglicht das Schreiben von Daten sowie das anschließende Lesen der geschriebenen Daten.

  Subscription: setzt eine langlebige Anfrage um, welche Daten als Reaktion auf Ereignisse aufruft. (Vgl. ebd. Abschnitt 2.3)

- *Auswahlmenge*

  Bei der Angabe einer GraphQL-Operation muss ein Client eine Menge an Informationen angeben, welche der GraphQL-Dienst zurückliefern soll. Diese Menge an Informationen wird Selection Set (dt. Auswahlmenge) genannt. Der Client erhält ausschließlich die Informationen, welche er in dieser Auswahlmenge angegeben hat. (Vgl. ebd. Abschnitt 2.4)

  Genauer beschrieben ist eine Auswahlmenge eine Komposition aus Feldern. Ein Feld beschreibt eine einzelne Information, welche innerhalb der Auswahlmenge ausgewählt werden kann. Ein Feld kann zum einen primitive Daten, wie zum Beispiel eine Zeichenkette oder Nummern, als auch komplexere Daten enthalten, wie beispielweise eine Relation auf andere Daten. Komplexere Daten können wiederum eine eigene Auswahlmenge besitzen, aus welcher der Client weitere Informationen auswählen kann. (Vgl. ebd. Abschnitt 2.5) Abbildung 3 veranschaulicht eine GraphQL Anfrage.

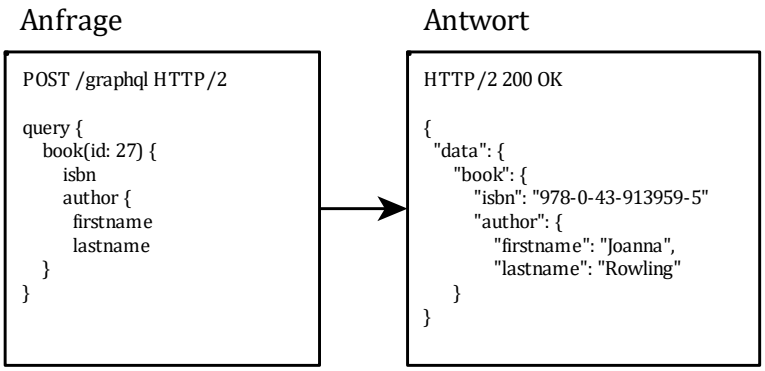

Abbildung 3 Beispiel einer GraphQL Abfrage

Bei der Anfrage werden Informationen eines Buches abgerufen. Aus der Auswahlmenge des Buches werden die Felder *isbn* und *author* ausgewählt. Das ISBN Feld setzt sich aus dem primitiven Datentyp einer Zeichenkette zusammen, wohingegen das Feld *author* eine Relation auf den Autor des Buches ist. Der Autor besitzt eine eigene Auswahlmenge, wodurch bei der Anfrage auf die zwei weiteren Felder *firstname* und *lastname* zugegriffen werden kann.

- *Fragmente*

Fragmente realisieren Kompositionen von Feldern innerhalb von GraphQL Anfragen. Sie definieren wiederverwendbare Auswahlmengen, welche Dopplungen in den Anfragen reduzieren.

```
fragment countryFields on Country {
  name
  native
}

query withFragments {
  country1: country(code: "DE") {
    ...countryFields
  }
  country2: country(code: "NL") {
    ...countryFields
  }
}
```

Listing 1 GraphQL Beispiel für die Verwendung von Fragmenten

Mit dem Listing 1 wird exemplarisch die Definition und Verwendung von einem Fragment veranschaulicht. Durch das Fragment *countryFields* wird eine wiederverwendbare Auswahlmenge für das Objekt Country umgesetzt. Mit den Anfragen country1 und country2 werden zwei Länder angefragt. Durch die Verwen-

dung von *...countryFields* wird die wiederverwendbare Auswahlmenge des Fragments für diese Anfragen festgelegt. Fragmente wirken sich positiv auf die Übersicht von umfangreichen GraphQL Dokumenten aus, indem die wiederverwendbaren Auswahlmengen zentral festgelegt und in den Operationen verwendet werden.

Typsystem

Das Typsystem ist die zweite elementare Komponente von GraphQL und beschreibt die Fähigkeiten eines GraphQL-Servers. Eine Hauptaufgabe dieser Komponente ist die Überprüfung von Anfragen auf ihre Gültigkeit. Des Weiteren beschreibt das Typsystem die Eingabetypen des GraphQL-Servers. Dadurch kann zur Laufzeit festgestellt werden, ob die eingegebene Anfrage valide ist. Zusammengesetzt wird das System aus den Schema-, Typ- und Direktivdefinitionen, welche durch den API-Entwickler beschrieben werden. (Vgl. ebd. Kapitel 3)

- *Schemadefinition*

  Die kollektiven Typsystemfähigkeiten eines GraphQL-Dienstes werden durch das Schema definiert. Ein Schema wird durch den Typen und Direktiven beschrieben, welche es unterstützt sowie durch den Root-Operation-Type (dt. Wurzel-Operationstyp) für jegliche Art von Operationen: Query, Mutation und Subscription. Das Schema ist der Ort des Typsystems, an dem die Operationen des Systems definiert werden. (Vgl. ebd. Abschnitt 3.2)

```
type City {
    id: ID!
    name: String!
    country: Country!
    district: String!
    localName: String!
    surfaceArea: Int
}

type QueryRootType {
    city(id: ID!): City
}


schema {
    query: QueryRootType
}
```

Listing 2 GraphQL Schema Definition

Durch das Listing 2 wird das Objekt City beschrieben. Mithilfe des *QueryRootType* Objektes werden die Query Operationen des Schemas deklariert. Durch dieses

Objekt wird beispielsweise die Query city definiert, welche eine *id* als Eingabeargument erwartet und ein City Objekt zurückliefert, falls ein solches Objekt zu der angegebenen *id* existiert. Das *QueryRootType* Objekt wird in dem Schema Objekt anschließend als Root-Operation-Type für den Query Operationstypen festgelegt. Des Weiteren wird durch ein Ausrufezeichen gekennzeichnet, dass der Rückgabewert oder Eingabewert eines Typs nicht leer sein darf.

- *Typdefinition*

  Die fundamentale Einheit des Typsystems ergibt sich aus der Definition eines *Typs*. Laut der GraphQL-Spezifikation existieren sechs Arten von Typen (vgl. ebd. Abschnitt 3.4). Die unterste Ebene der Typen ergibt sich aus dem *Scalar* und *Enum*. Werden alle Typen des Typsystems in einem Graphen betrachtet, repräsentieren diese Typen die Blätter des Graphen. Ein Scalar ergibt sich aus einem primitiven Wert, wohingegen ein Enum die Werte, welche dieses annehmen kann, fest vorschreibt.

```
firstname: String!
```

Listing 3 GraphQL Scalar Type

Durch das Listing 3 wird exemplarisch die Definition des Scalars firstname veranschaulicht. Als erstes wird der Name eines Scalars angegeben, gefolgt von einem Doppelpunkt und anschließend dem Datentyp, den dieser Scalar Typ annehmen darf.

```
enum Weather {
  sunny
  rainy
}
```

Listing 4 GraphQL Enum Type

In dem Listing 4 wird die Definition eines Enum aufgeführt. Das Enum erhält den Namen Weather, kann die Werte sunny und rainy annehmen und dient als neuer Datentyp des Typsystems.

Die nächsthöhere Ebene der Typdefinitionen erschließt sich aus dem *Objekt*. Ein Objekt definiert eine Menge von Feldern, wobei sich jedes Feld aus einem anderen Typen des Systems ergibt. Durch Objekte wird eine hierarchische Struktur des Systems geschaffen.

```
type User {
  id: ID!
  firstname: String!
  lastname: String!
  comments: [Comment]
}

type Comment {
  id: ID!
  content: String!
}
```

Listing 5 GraphQL Object Type

In dem Listing 5 wird die Definition der Objekte User und Comment veranschaulicht. Das User Objekt beschreibt eine Menge aus den Feldern id, firstname, lastname und comments. Während id, firstname und lastname primitive Datentypen annehmen, verweist comments auf eine Liste des Objektes Comment. Durch das Verweisen auf andere Objekte wird eine hierarchische Struktur des Typsystems geschaffen.

Des Weiteren können abstrakte Typen definiert werden. Das *Interface* definiert ebenfalls eine Menge von Feldern. Jedes Objekt, welches dieses Interface implementiert, muss die durch das Interface definierten Felder ebenfalls definieren.

```
interface Human {
  firstname: String!
  lastname: String!
}

type User implements Human{
  id: ID!
  firstname: String!
  lastname: String!
}
```

Listing 6 GraphQL Interface Type

Durch das Listing 6 wird das Interface *Human* beschrieben. Das Interface wird von dem User Objekt implementiert, wodurch die Scalar Typen firstname und lastname durch das Objekt deklariert werden müssen.

Eine *Union* (dt. Vereinigung) dahingegen definiert eine Liste von möglichen Typen. Bei dem Aufruf einer solchen Vereinigung, wird einer der angegebenen Typen zurückgeliefert.

```
union searchResult = Video | Photo

type searchQuery {
  firstSearchResult: searchResult!
}
```

Listing 7 GraphQL Union Type

Mit dem Listing 7 wird die Vereinigung *searchResult* definiert. Als Rückgabetypen werden die Objekte Video und Photo festgelegt. Durch das Aufrufen eines Feldes, welches diese Vereinigung verwendet, muss eines der beiden Objekte zurückgeliefert werden.

Als letzte Typdefinition wird der *Input*-Typ (dt. Eingabe) vorgestellt. Mit diesem Typen kann die Komplexität der Schemadefinition minimiert werden. Input-Typen definieren eine Menge von Feldern, welche die erwarteten Felder einer Clienteingabe fest vorschreiben können.

```
input UserInput {
  firstname: String!
  lastname: String!
}

type MutationResolver{
  createUser(user: UserInput!): User!
  updateUser(id: ID! user: UserInput!): User!
}
```

Listing 8 GraphQL Input Type

Listing 8 implementiert den Input *UserInput*. Durch diesen Eingabetypen wird festgelegt, welche Felder bei der Eingabe eines Users eingegeben werden müssen. Der UserInput kann beispielsweise als Eingabeargument für eine Mutation verwendet werden, um die Schemadefinition übersichtlicher zu gestalten.

- *Direktivdefinition*

  Direktiven eines Typsystems setzten dekorative Funktionen im Schema um. Durch Dekorierer wird eine Typdefinition durch zusätzliche Funktionalität erweitert. Dadurch können beispielsweise Konfigurationsparameter festgelegt oder Informationen über veraltete Felder des Schemas an den Client ausgegeben werden.

```
type User {
  name: String! @deprecated(reason: "Outdated. Use firstname/lastname.")
  firstname: String!
  lastname: String!
}
```

Listing 9 GraphQL Direktive

Das Listing 9 veranschaulicht exemplarisch die Verwendung der *@deprecated Direktive*. Versucht ein Client das Feld *name* aufzurufen, wird diese Anfrage durch die Direktive mit der Fehlermeldung: „Outdated. Use fistname/lastname“ abgelehnt.

Laufzeitumgebung

Die dritte und letzte Komponente von GraphQL ist die Laufzeitumgebung eines GraphQL Servers. Sie ist für die Übersetzung, Validierung und Bearbeitung von GraphQL-Anfragen gedacht.

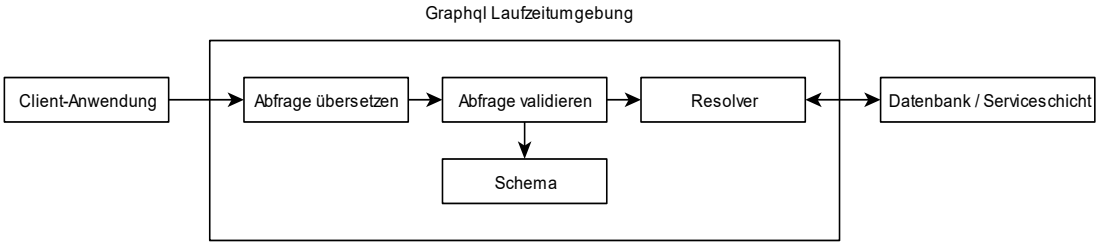

Abbildung 4 GraphQL Laufzeitumgebung

Stellt eine Client-Anwendung eine Anfrage, wird diese zuerst von der Laufzeitumgebung übersetzt, um diese anschließend mithilfe des Typsystems und das daraus resultierende Schema zu validieren. Nach einer erfolgreichen Validierung der Anfrage wird diese an einen entsprechenden Resolver weitergereicht, welcher die für die Bearbeitung erforderliche Geschäftsebene des Dienstes anstößt. Sollte bei der Validierung jedoch ein Fehler aufgetreten sein, dann wird das dem Client über die eingebaute Fehlerbehebung mitgeteilt. Der Fehler wird dabei von der Laufzeitumgebung abgefangen und als *error* Objekt dem Client als JSON repräsentiert zurückgeliefert.

Nachfolgend werden die Designprinzipien formuliert, welche durch die Spezifikation verfolgt werden und bei der Umsetzung der GraphQL Komponenten durchgesetzt wurden.

2.3.2 GraphQL-Designprinzipien

Die GraphQL-Spezifikation setzt fünf Designprinzipien um, welche zu einer leistungsstarken und produktiven Umgebung für die Entwicklung von Client-Anwendungen beitragen. Nachfolgend werden diese Prinzipien vorgestellt.

Hierarchische Struktur

GraphQL-Anfragen sind hierarchisch strukturiert. Felder können ineinander verschachtelt aufgerufen werden und die Daten, welche als Antwort zurückgeliefert werden, sind identisch zu der in der Auswahlmenge festgelegten Struktur aufgebaut.

Produktorientiert

GraphQL ist eindeutig von den Anforderungen und Ansichten der Front-End-Entwicklern, die sie schreiben, geprägt. GraphQL beginnt mit ihrer Denkweise und ihren Anforderungen und entwickelt die Sprache und die Laufzeit, die notwendig sind, um diese umzusetzen.

Starke Typisierung

GraphQL-Server definieren ein anwendungsspezifisches Typsystem. Abfragen an den Server werden im Kontext dieses Typsystems ausgeführt. Bevor eine Abfrage ausgeführt wird, wird mit dem Typsystem sowohl festgestellt, ob die Abfrage syntaktisch korrekt ist als auch, ob sie innerhalb des GraphQL-Typsystems gültig ist. Sollte dies nicht der Fall sein, reagiert der GraphQL-Server mit einem Hinweis an den Client.

Client-spezifische Abfragen

GraphQL-Server definieren das Maß der Fähigkeiten, welches durch einen Client benutzt werden darf über das Typsystem. Die Aufgabe des Clients ist zu bestimmen, wie diese Fähigkeiten des GraphQL-Servers genutzt werden sollen. Anders als bei anderen Client-Server-Anwendungen, bei denen der Server bestimmt, welche Daten zurückgeliefert werden, kann mit einer GraphQL-Abfrage genau bestimmt werden, welche Daten von dem GraphQL-Server zurückgeliefert werden.

Introspektiv

Ein GraphQL-Server unterstützt die Introspektion des eignen Schemas. Das Schema kann durch die eigene GraphQL-Abfragesprache abgerufen werden und enthält alle Informationen über dieses. Durch Introspektion wird Client-Entwicklern das Erforschen des Schemas ermöglicht, wodurch mit GraphQL selbst leistungsfähige Client-Tools entwickelt werden können. (Vgl. Byron, 2018 Kapitel 1)

3 Entwurf und Umsetzung einer Beispielanwendung

Nachdem in dem vorherigen Kapitel der Begriff API definiert und kategorisiert, die Designprinzipien und der Zweck von REST beschrieben und danach die GraphQL-Spezifikation als Alternative zu REST vorgestellt wurde, wird in diesem Kapitel auf der Grundlage von dem quelloffenen Java-Framework *Spring Boot* eine Beispielanwendung für das Erfassen und Verändern von Daten entworfen und umgesetzt. Für diese Anwendung werden öffentlich zugängliche Datensätze benutzt. Diese Datensätze bilden Kontinente, Länder, Städte und Sprachen ab, welche zueinander in Beziehung stehen. Der Zugriff und das Verändern von den Daten soll über ein API verwirklicht werden. Eine wichtige Anforderung an diese Anwendung ist die Bereitstellung von sowohl eines RESTful- als auch eines GraphQL-API, da diese für einen Leistungsvergleich in den nächsten beiden Kapiteln benötigt werden.
Dafür wird im Folgenden das Framework *Spring Boot* vorgestellt (Abschnitt 3.1) und die Anwendungsarchitektur entworfen (Abschnitt 3.2). Anschließend werden die Architekturschichten nacheinander umgesetzt (Abschnitte 3.2.1ff). Das Kapitel schließt mit einem ersten Vergleich von REST und GraphQL, es werden ihre Eigenschaften in Bezug auf Entwurf und Umsetzung unter die Lupe genommen (Abschnitt 3.2). Der Entwicklungsprozess wird durch ein Git Repository[2] veranschaulicht.

3.1 Anwendungsentwicklung nach dem Prinzip „Konvention über Konfiguration“: *Spring Boot*

Spring ist ein quelloffenes Framework aus der Java-Welt, welches von Rod Johnson entwickelt und erstmals 2002 in seinem Buch *Expert One-On-One Design and Development* erwähnt wurde. Im Juni 2003 erscheint die erste *Spring* Version unter der Apache 2.0 Lizenz, womit die bis heute anhaltende Erfolgsgeschichte des Frameworks beginnt.

[2] Git Repository: https://github.com/maximilianmilz/sample-service

Das fundamentale Ziel von *Spring* ist es die Entwicklung von Java-Anwendungen zu vereinfachen. Dafür bietet *Spring*, im Gegensatz zur Java Standard Edition, vorgefertigte Lösungsansätze zur Entwicklung solch einer Software an und verfolgt dabei vier Kernstrategien[3] (vgl. Walls, 2012 S. 4-15):

- Leichtgewichtige und minimale Entwicklung mit Plain Old Java Objects (dt. einfaches altes Java-Objekt, POJO)
- Lockere Kopplung durch Injizieren von Abhängigkeiten und Interface Orientierung (Dependency Injection, DI)
- Deklarative Programmierung durch Aspekte und übliche Konventionen (aspektorientierte Programmierung, AOP)
- Reduzierung von Boilerplate-Code durch Aspekte und Vorlagen

Durch den modularen Aufbau von *Spring* ist das Entwickeln von adaptiven Anwendungen möglich. Allgemein bietet das Framework durch die Bereitstellung von insgesamt 20 Standardmodulen eine Fülle an Eigenschaften. Diese Module werden auf Basis ihrer Funktionalität in den Gruppen Web, Datenzugriff/Integration, Instrumentierung, AOP, Kern-Container und Test unterteilt (vgl. Golubski, 2019 S. 11-12).
Spring Boot ist eines der vielen Unterprojekte von *Spring.* Es wird für die Umsetzung dieses Projektes verwendet, denn es bietet mit dem Richtsatz „convention over configuration“ (dt. Konvention über Konfiguration) einen Ansatz, welcher ermöglicht, dass aufwendige Konfigurationen im Projekt entfallen können.
Darüber hinaus verwendet das Framework einen HTTP-Server, welcher es ermöglicht, dass die entwickelte Anwendung zu jedem Zeitpunkt der Entwicklung gestartet und getestet werden kann.
Die Konfiguration eines Projektes erfolgt allgemein dabei über das Build-Managment-Tool *Maven*[4]. Dabei werden die Abhängigkeiten, die eine Software zu anderen Softwareprojekten besitzt, in der sogenannten Projektdefinitionsdatei *pom.xml* angegeben.

[3] Für eine ausführliche Erklärung wird auf das Buch *Spring im Einsatz* von Craig Walls verwiesen.
[4] Apache Maven: https://maven.apache.org/

3.2 Anwendungsarchitektur

Die Beispielanwendung für das Erfassen und Bearbeiten von Anwendungsdaten wird mithilfe einer Schichtenarchitektur entworfen. Der Vorteil einer solchen Architektur ist es, dass der Austausch der Funktionalität und Implementierungsweise in einer Schicht erfolgt, ohne dass andere Schichten dadurch beeinträchtigt werden. Schichten können dabei mit anderen benachbarten Schichten kommunizieren.
Mit der nachfolgenden Abbildung 5 wird die Anwendungsarchitektur veranschaulicht.

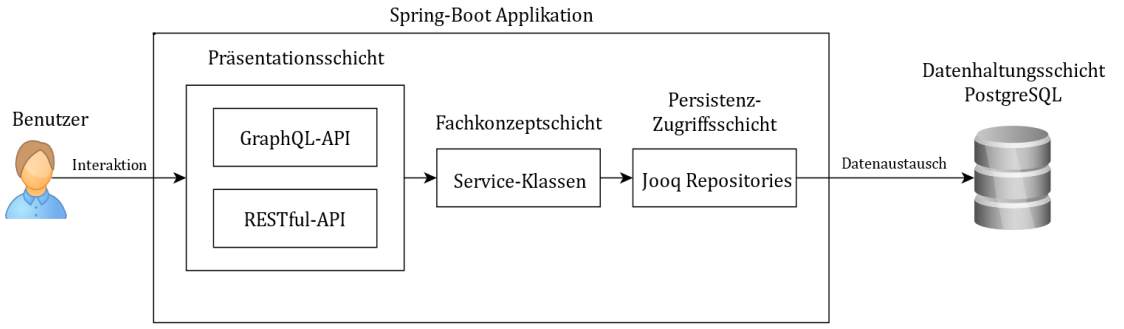

Abbildung 5 Systemarchitektur der Beispielanwendung

Die Anwendung setzt sich aus der Datenhaltungs-, Persistenz-Zugriffs-, Fachkonzept- und der Präsentationsschicht zusammen. Die *Datenhaltungsschicht* ist für die Haltung und Verwaltung der Anwendungsdaten zuständig, wohingegen die *Persistenz-Zugriffsschicht* den einheitlichen Zugriff auf diese Daten ermöglicht, unabhängig von der Art der Speicherung. Die *Fachkonzeptschicht* stellt die eigentliche Geschäftslogik der Applikation dar. Sie umfasst die Bearbeitung sowie Aufbereitung der Anwendungsdaten. Schlussendlich realisiert die *Präsentationsschicht* die Darstellung der Anwendungsdaten sowie die Interaktionsmöglichkeiten mit der Anwendung. Letztere Schicht soll sowohl ein RESTful-API als auch GraphQL-API umsetzen. Im Zusammenhang mit dem Leistungsvergleich, welcher in den nachfolgenden Kapiteln entworfen und umgesetzt wird, ist es wichtig, dass die Leistung nur durch den API-Architekturstil beeinflusst wird. Erreicht wird dieser Punkt dadurch, dass beide APIs durch dieselbe Anwendung realisiert werden und dieselbe Serviceschicht für das Erfüllen von Anfragen anstoßen.

3.2.1 Datenhaltungsschicht

Mit der in dieser Arbeit konzipierten Anwendung soll die Verwaltung und Aufbereitung von Anwendungsdaten umgesetzt werden. Die Anwendungsdaten sollen durch eine Datenbank bereitgestellt werden. Diese Datenbank stellt die Entitäten *Stadt*, *Land*, *Kontinent*, *Landessprache* und *Sprache* bereit. Dafür wird der Datenbank-Dump WorldDB[5] aus dem Jahr 2019 verwendet.

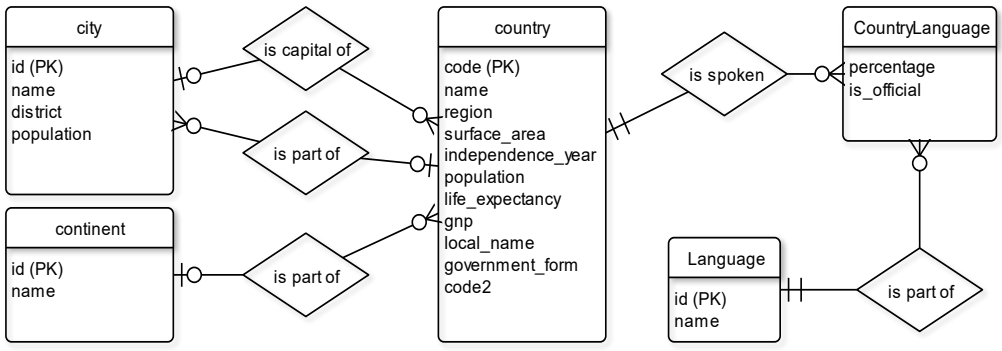

Abbildung 6 Entity-Relationship-Modell

Im Mittelpunkt des in Abbildung 6 dargestellten Entity-Relationship-Modells steht das Land. Dieses ist über einen dreistelligen Ländercode eindeutig identifizierbar und enthält allgemeine Informationen wie beispielsweise dessen Größe, Rechtsform oder Einwohnerzahl. Jedes Land kann einem Kontinent zugeordnet werden. Dieser besitzt lediglich einen Namen und verweist auf beliebige viele Länder, welche zu diesem Kontinent gehören. Des Weiteren verweist ein Land auf beliebige viele Städte, sowie eine einzelne Stadt, woraus sich die Hauptstadt des Landes ergibt. Städte besitzen Informationen über Namen, Bezirk und Einwohnerzahl einer Stadt. Schlussendlich verweist ein Land auf beliebig viele Sprachen, die in diesem Land gesprochen werden. Eine solche Ländersprache verweist immer auf eine Sprache und gibt Informationen darüber, wie hoch der prozentuale Anteil dieser Sprache in diesem Land ist und ob sie die offizielle Landessprache ist.

Umgesetzt wird die Datenhaltungsschicht durch das objektrelationale Datenbankmanagementsystem *PostgreSQL*. Mit dem Listing 10 wird ein Auszug aus der SQL-Datei geboten. Durch diese Datei wird das Schema definiert und die Datenbank mit Daten gefüllt.

[5] World Database: https://github.com/worlddb/world.db

```
-- Tabellen Definitionen
create table city
(
    id           bigserial              not null primary key,
    name         varchar(35) default '' not null,
    country_code char(3)     default '' not null,
    district     varchar(20) default '' not null,
    population   integer     default 0  not null
);

-- Daten
INSERT INTO city (id, name, country_code, district, population)
VALUES (1, 'Kabul', 'AFG', 'Kabol', 1780000);
INSERT INTO city (id, name, country_code, district, population)
VALUES (2, 'Qandahar', 'AFG', 'Qandahar', 237500);
```

Listing 10 Auszug aus der Datenbankschema Definition

Als erstes wird in diesem Auszug die Datenbanktabelle city mit den nötigen Attributen definiert. Anschließend wird diese Tabelle mit Beispieldaten gefüllt, welche von der Anwendung als Anwendungsdaten verwendet werden. Im Folgenden wird der Zugriff von der Anwendung auf diese Daten durch die Persistenz-Zugriffsschicht erläutert.

3.2.2 Persistenz-Zugriffsschicht

Eine der Hauptaufgaben der Persistenz-Zugriffsschicht ist der einheitliche Zugriff auf die Anwendungsdaten, unabhängig von der Art der Speicherung. Diese Schicht wird in der *Spring Boot* Applikation durch Repository-Klassen repräsentiert. Um die objektorientierte Java-Welt mit der relationalen PostgreSQL-Welt miteinander zu verbinden, wird ein sogenannter Objekt-Relationaler-Mapper benötigt. Bei dem Objekt-Relationalen-Mapping werden Konstrukte aus der objektorientierten Welt auf die relationale abgebildet. Dieses Mapping soll mit der Datenbank-Mapping-Softwarebibliothek *Java Object Oriented Querying*[6] (dt. Java Objektorientiertes Abfragen, jOOQ) umgesetzt werden. Die Bibliothek ist sowohl relational als auch objektorientiert und stellt mit ihrer *Domain Specific Language* (dt. Domänenspezifische Sprache, DSL) Methoden zur Verfügung, um Datenbankabfragen aus generierten Klassen zu konstruieren. Für die Generierung der jOOQ-Klassen wird ein separates *Maven*-Pro-

[6] jOOQ, Datenbank-Mapping-Softwarebibliothek: https://www.jooq.org/

jekt implementiert, welches mehrere Java Packages zu den relevanten Datenbanktabellen generieren kann. In dem nachfolgenden Listing 11 wird das Mapping anhand der Datenbanktabelle *City* veranschaulicht.

```
<execution>
    <id>city-schema</id>
    <phase>generate-sources</phase>
    <goals>
        <goal>generate</goal>
    </goals>
    <configuration>
        <jdbc>
            <driver>${spring.datasource.driver-class-name}</driver>
            <url>${spring.datasource.url}</url>
            <user>${spring.datasource.username}</user>
            <password>${spring.datasource.password}</password>
        </jdbc>
        <generator>
            <database>
                <includes>city</includes>
                <inputSchema>public</inputSchema>
                <outputSchemaToDefault>true</outputSchemaToDefault>
            </database>
            <generate>
                <javaTimeTypes>true</javaTimeTypes>
            </generate>
            <target>
                <packageName>de.thb.world.service.city.jooq</packageName>
                <directory>../src/main/java</directory>
            </target>
        </generator>
    </configuration>
</execution>
```

Listing 11 jOOQ-Mapping City

Das Ergebnis ist das Java Package *de.thb.world.service.city.jooq*, welches alle relevanten Klassen zur Generierung einer SQL-Abfrage für die Datenbanktabelle *City* zur Verfügung stellt. Folglich wird die Umsetzung einer SQL-Abfrage für den Aufruf von allen Städten veranschaulicht. Die typische PostgreSQL-Abfrage sieht wie folgt aus.

```
select * from city order by id asc
```

Listing 12 PostgreSQL Beispielabfrage

Nachfolgend wird die jOOQ Abfrage der Repository-Klasse veranschaulicht, welche Inhaltlich identisch zu der PostgreSQL-Abfrage ist.

```
import de.thb.world.service.city.jooq.tables.City;
import de.thb.world.service.city.jooq.tables.records.CityRecord;

@Repository
@RequiredArgsConstructor
public class CityRepository {
    private final DSLContext context;

    public List<CityRecord> findAll() {
        return context.selectFrom(City.CITY)
                .orderBy(City.CITY.ID.asc())
                .fetch();
    }
}
```

Listing 13 jOOQ Beispielabfrage

Für die Datenbankabfrage wird der DSLContext verwendet. Laut der offiziellen jOOQ Dokumentation ist der DSLContext der Haupteinstiegspunkt für Client-Code, um auf jOOQ-Klassen und -Funktionen zuzugreifen, die mit der Ausführung von Abfragen zusammenhängen (vgl. JOOQ, 2021).

3.2.3 Fachkonzeptschicht

Die Fachkonzeptschicht realisiert die eigentlich Geschäftslogik der Anwendung. Allgemein wird diese Schicht in der *Spring Boot* Applikation durch sogenannte Service-Klassen realisiert. Service-Klassen werden von der Präsentationsschicht oder einer anderen Service-Klasse angestoßen. Diese Klasse stößt wiederum die Persistenz-Zugriffsschicht an, um Daten aus der Datenhaltungsschicht zu holen. Die erhaltenen Daten werden in der entsprechenden Service-Methode bearbeitet, gefiltert oder anderweitig aufbereitet, um anschließend an die Präsentationsschicht zurückgeliefert zu werden. In dem nachfolgenden Listing 14 wird beispielsweise die Servicemethode für die Abfrage aller Städte dargestellt. Diese Methode benutzt das CityRepository, um sich über die Persistenz-Zugriffsschicht die Daten zu allen Städten zu holen. In derselben Serviceklasse werden die durch das CityRepository bereitgestellten Datensätze (CityRecord) in POJOs (City) gemappt. Anschließend wird eine Liste dieser gemappten Objekte an die Präsentationsschicht zurückgegeben.

```
@Service
@RequiredArgsConstructor
public class CityService {
    private final CityRepository cityRepository;

    public List<City> findAll() {
        return cityRepository.findAll()
                .stream()
                .map(this::map)
                .collect(Collectors.toList());
    }

    public City map(CityRecord record) {
        return City.builder()
                .id(record.getId())
                .name(record.getName())
                .countryCode(record.getCountryCode())
                .district(record.getDistrict())
                .population(record.getPopulation())
                .build();
    }
}
```

Listing 14 Auszug aus dem CityService

3.2.4 Präsentationsschicht

Mithilfe der Präsentationsschicht werden die Möglichkeiten zur Interaktion mit der Anwendung geschaffen. Für die Interaktion mit der Anwendung wird sowohl ein RESTful- als auch eines GraphQL-API entworfen und entwickelt. Für eine gerechte Gegenüberstellung der beiden API-Architekturstilen müssen beide APIs den gleichen Funktionsumfang anbieten.
Nachfolgend werden der Entwurf und die Umsetzung der beiden APIs dargestellt.

3.3 Vergleich von Entwurf und Umsetzung in dem REST- bzw. GraphQL-API

3.3.1 Ressourcenorientierte Ansatz: RESTful-API

Das RESTful-API verfolgt einen ressourcenorientierten Ansatz. Die Ressourcen, welche durch das API angeboten werden, ergeben sich aus den Entitäten des relationalen Modells. Alle Endpunkte sind unter der Basis-URL */rest/v1/* erreichbar. Ressourcen werden in der *Spring Boot* Applikation in separaten Controller-Klassen repräsentiert. Aus den Methoden einer solchen Klasse ergeben sich die HTTP-Methoden,

welche auf diese Ressource angewendet werden können. Diese Methoden stoßen die jeweilige Servicemethode an, welche für die Erfüllung einer Anfrage benötigt wird. Eine GET-Anfrage, um alle Städte abzurufen, könnte wie folgt aussehen.

```
@RestController
@RequiredArgsConstructor
@RequestMapping("/rest/v1/cities")
public class CityController {
    private final CityService cityService;

    @GetMapping
    public List<City> getAll() {
        return cityService.findAll();
    }
}
```

Listing 15 Auszug aus dem CityController

Die *getAll* Methode des *CityController* verwendet die Serviceklasse *CityService*, welche die Methode zum Abrufen aller Städte bereitstellt. Durch die *@GetMapping* Annotation wird festgelegt, dass die Methode mit einem GET-Request über den /rest/v1/cities Endpunkt aufgerufen wird.

Im Rahmen der Entwicklung des RESTful-API werden weitere Controllermethoden umgesetzt, woraus sich die Menge aller Endpunkte ergibt. In der nachfolgenden Tabelle 2 werden diese Endpunkte veranschaulicht.

Verb	URL	Beschreibung
LanguageController URL: /languages		
GET	/	Aufrufen aller Sprachen
POST	/	Erstellen einer Sprache
PUT	/{id}	Aktualisieren einer Sprache
DELETE	/{id}	Entfernen einer Sprache
GET	/{id}	Aufrufen einer Sprache
GET	/{id}/countries	Aufrufen aller Länder, in denen die Sprache gesprochen wird
CountryLanguageController URL: /country-languages		
GET	/	Aufrufen aller Länder-Sprachen-Relationen
POST	/	Erstellen einer Ländersprache
PUT	/{code}-{id}	Aktualisieren einer Ländersprache
DELETE	/{code}-{id}	Entfernen einer Ländersprache
GET	/{code}-{id}	Aufrufen einer bestimmten Ländersprache
CountryController URL: /countries		
GET	/	Aufrufen aller Länder
POST	/	Erstellen eines Landes
PUT	/{code}	Aktualisieren eines Landes
DELETE	/{code}	Löschen eines Landes
GET	/{code}	Aufrufen eines Landes

GET	/{code}/languages	Aufrufen aller Sprachen eines Landes
GET	/{code}/cities	Aufrufen aller Städte eines Landes
GET	/{code}/capital	Aufrufen der Hauptstadt eines Landes

CityController URL: /cities

GET	/	Aufrufen aller Städte
POST	/	Erstellen einer Stadt
PUT	/{id}	Aktualisieren einer Stadt
DELETE	/{id}	Entfernen einer Stadt
GET	/{id}	Aufrufen einer Stadt
GET	/{id}/country	Aufrufen des Landes einer Stadt

ContinentController URL: /continents

GET	/	Aufrufen aller Kontinente
GET	/{id}	Aufrufen eines Kontinents
GET	/{id}/countries	Aufrufe aller Länder eines Kontinents

Tabelle 2 RESTful-API Endpunkte

Relevant ist die Darstellung der Endpunkte, da eine Teilmenge dieser Endpunkte in dem Leistungsverglich in den beiden nachfolgenden Kapiteln verwendet wird.

3.3.2 GraphQL-API

Allgemein sollte bei der Entwicklung eines GraphQL-APIs die Reihenfolge der Entwicklungsschritte eingehalten werden. Der Entwickler oder die Entwicklerin sollte zuerst das Schema definieren, um anschließend zu wissen, welche *Resolver* umgesetzt werden müssen. Das Schema setzt sich aus den Direktiven- sowie Typdefinitionen als auch aus den Root-Operation-Typen der jeweiligen Operationstypen zusammen (vgl. Abschnitt 2.3.1 Abfragesprache *Operationen von GraphQL*). Als erstes sollten bei der Definition des Schemas die Direktiven und Typen formuliert werden, um darauf aufbauend die Operationen des Schemas beschreiben zu können. Das Listing 16 zeigt exemplarisch die Definition des Objekts *City*. Für das Objekt wird in dem QueryRootType die Query *cities* definiert, welche eine Liste von Städten zurückliefert.

```
type QueryRootType {
    cities: [City]
}

type City {
    id: ID!
    name: String
    country: Country!
    district: String
    population: Int
}

schema {
    query: QueryRootType
}
```

Listing 16 Auszug des GraphQL Schemas

Mithilfe des Schemas werden die Operationen des Typsystems beschrieben. Für die Ausführung einer solchen Operation wird ein Vermittler zwischen dem GraphQL-Typsystem und dem Dienst, welcher diese Operation annehmen und weiterleiten soll, benötigt. Dieser Vermittler wird in der GraphQL-Termini *Resolver* genannt. Für die Realisierung eines *Resolvers* in der Präsentationsschicht der *Spring Boot* Anwendung wird das quelloffene *Domain Graph Service Framework* von Netflix[7] verwendet. Das Framework vereinfachtet die Implementierung von GraphQL Diensten. Eines der Hauptmerkmale, welche das Framework von dem Standard Spring Boot GraphQL Framework unterscheidet, ist die Bereitstellung der Annotationsbasierten Spring Boot Programmierung (vgl. Bakker, 2021). Nachfolgend wird in dem Listing 17 ein Auszug des *CityResolver* veranschaulicht. Über die *@DgsData* Annotation wird der Name des Typen sowie ein Feld angegeben, welches zu der Auswahlmenge des Typs gehört. Aus diesen beiden Attributen kann jedes Feld eines Typsystems identifiziert werden. Mit der zu der Annotation dazugehörigen Methode wird bestimmt, wie dieses Feld berechnet werden soll. Zusammengefasst bedeutet das, dass ein Resolver eine ähnliche Aufgabe wie ein REST-Controller besitzt. Durch ihn wird die benötigte Serviceklasse angestoßen, um eine Aufgabe zu bearbeiten. Der Unterschied ist, dass durch einen Resolver keine Endpunkte definiert werden, sondern die Berechnung der Felder.

[7] GraphQL Domain Graph Service Netflix: https://netflix.github.io/dgs/

```java
@DgsComponent
@RequiredArgsConstructor
public class CityResolver {
    private final CityService cityService;
    private final CountryService countryService;

    @DgsData(parentType = "QueryRootType", field = "cities")
    public List<City> getAll() {
        return cityService.findAll();
    }

    @DgsData(parentType = "City", field = "country")
    public Country getCountryOfCity(DgsDataFetchingEnvironment dfe) {
        City city = dfe.getSource();
        return countryService.findByCode(city.getCountryCode()).get();
    }
}
```

Listing 17 Auszug des CityResolver

Durch das Definieren des Schemas und der Umsetzung der Resolver entstehen weitere Operationen. Die Menge aller Operationen, welche für das Typsystem bereitstehen, wird in der Tabelle 3 veranschaulicht.

Name	Typ	Argumente	Beschreibung
language	Query	id: ID	Abrufen einer bestimmten Sprache
languages	Query		Abrufen aller Sprachen
addLanguage	Mutation	input: LanguageInput	Hinzufügen einer Sprache
updateLanguage	Mutation	id: ID	Aktualisieren einer Sprache
		input: LanguageInput	
removeLanguage	Mutation	id: ID	Entfernen einer Sprache
addCountryLanguage	Mutation	input: CountryLangInput	Hinzufügen einer Landessprache
updateCountryLanguage	Mutation	id: ID!	Aktualisieren einer Landessprache
		countryCode: String	
		input: CountryLangInput	
removeCountryLanguage	Mutation	id: ID	Entfernen einer Landessprache
country	Query	code: String	Abrufen eines bestimmten Landes
countries	Query	-	Abrufen aller Länder
addCountry	Mutation	input: CountryInput	Erstellen eines Landes
updateCountry	Mutation	code: String	Aktualisieren eines Landes
		input: CountryInput	
removeCountry	Mutation	code: String	Entfernen eines Landes
continent	Query	id: ID	Abrufen eines Kontinents
continents	Query	-	Abrufen aller Kontinente
city	Query	id: ID	Abrufen einer Stadt
cities	Query	-	Abrufen aller Städte
addCity	Mutation	input: CityInput	Hinzufügen einer Stadt
updateCity	Mutation	id: ID!	Aktualisieren einer Stadt
		input: CityInput	
removeCity	Mutation	id: ID	Entfernen einer Stadt

Tabelle 3 GraphQL-API Operationen

Einstimmig zu den REST-Endpunkten wird eine Teilmenge der GraphQL-Operationen für den nachfolgenden Leistungsvergleich zwischen REST und GraphQL benötigt, weshalb die Darstellung aller Operationen notwendig ist.
Auf Grundlage der entwickelten Beispielanwendung wird im Folgenden der Entwurfsprozess des Leistungsvergleiches beschrieben.

4 Entwurf eines Leistungsvergleichs zwischen REST und GraphQL

In diesem Kapitel wird ein Leistungsvergleich zwischen REST und GraphQL entworfen. In dem ersten Abschnitt dieses Kapitels wird dafür die Leistung definiert. Aufbauend darauf werden typische Leistungsprobleme von APIs beschrieben als auch eine Methodik zur Erfassung der Leistung eines API entworfen. Das schließt die Herangehensweise für die Erfassung sowie die Formulierung von Metriken ein. Im Anschluss wird auf Grundlage dieser Methodik ein Experiment beschrieben, womit die Leistung der beiden APIs aus dem Abschnitt 3.2.4 ermittelt und anhand der daraus gewonnenen Ergebnisse miteinander verglichen werden soll.

4.1 Leistung

Laut Fielding ist die Leistung eines der wichtigsten Aspekte bei der Entwicklung einer netzwerkbasierten Anwendung, da diese der bestimmende Faktor für die Benutzerwahrnehmung über die Qualität der Software ist. Die gewählte Architektur spielt eine entscheidende Rolle in Bezug auf die Leistung. Die Leistung einer netzwerkbasierten Anwendung wird zunächst durch die Anforderungen der Anwendung, dann durch den gewählten Interaktionsstil, danach durch die realisierte Architektur und schließlich durch die Implementierung der einzelnen Komponenten bestimmt. (Vgl. Fielding, 2000 S. 29)
Grundlegend wird die wahrgenommene Leistung eines API durch die Netzwerkleistung beeinflusst. Jede Interaktion mit einem API führt typischerweise zu einem HTTP-Request. Besitzt der Client oder Server eine schlechte Netzwerkanbindung, wird daraus eine schlechte Leistung geschlossen. Darüber hinaus existieren typische Probleme, welche mit der Wahl des API-Architekturstils zusammenhängen und ein API negativ in Bezug auf dessen Leistung beeinflussen können. In den nachfolgenden Abschnitten werden die typischen Leistungsprobleme eines API beschrieben sowie eine Methodik zur Erfassung der Leistung eines API formuliert.

4.1.1 Leistungsprobleme von APIs

Typische Probleme, welche in Bezug auf die Leistung eines API auftreten, sind das *Over-Fetching* und *Multiple-Round-Trips*. In beiden Fällen bekommt ein Client nicht die Daten zurückgeliefert, welche benötigt werden. Nachfolgend werden diese Probleme spezifiziert.

Over-Fetching

Unter *Over-Fetching* wird das Zurückliefern von zu vielen Informationen als Antwort auf eine Client-Anfrage verstanden. Unwesentliche Daten, die ein Client nicht benötigt, werden *Overhead* genannt. Grundsätzlich kann durch jede API-Anfrage ein *Overhead* entstehen, welcher sich gravierend auf die Leistung eines API auswirken kann. Beispielsweise tritt ein *Overhead* auf, wenn ein Client nur eine bestimmte Teilmenge von Attributen einer bestimmten Ressource benötigt, aber durch den API-Architekturstil nur das Abrufen der kompletten Ressource unterstützt wird. Der Client bekommt einen *Overhead* als Antwort mit übermittelt. Umso größer der *Overhead* ist, desto schlechter wird die Leistung des API.

Multiple-Round-Trips

Multiple-Round-Trips sind genau das gegenteilige Problem. Unter diesem Begriff wird verstanden, dass mit einer API-Anfrage nicht genügend Informationen abgefragt werden können, sodass ein Client eine weitere oder mehrere API-Anfragen stellen muss, damit er die für sich relevanten Informationen erhält. Durch dieses Problem ist ein Client oft dazu gezwungen mehrere API-Anfragen zustellen. Beispielsweise könnte ein Client eine Buch-Ressource einer Bibliotheks-API abfragen. Zusätzlich dazu benötigt der Client die Autor-Ressource, welche mit dem Buch in Relation steht. Können beide Ressourcen nicht über eine API-Anfrage abgefragt werden, dann spricht man von Multiple-Round-Trips.

4.1.2 Leistungsmetriken zur Leistungserfassung von APIs

In diesem Abschnitt werden Metriken zur Erfassung der Leistung eines API beschrieben. Gilling beschreibt in einem Bericht dreizehn API Metriken, welche jedes Entwicklerteam verfolgen sollte (vgl. Gilling, 2020). Auf Grundlage von diesen Metriken werden die nachfolgenden Leistungsmetriken definiert

API-Aufrufe pro Geschäftsvorgang

Unter API-Aufrufe pro Geschäftsvorgang wird die Anzahl der benötigten API-Anfragen zur Erfüllung einer Aufgabe verstanden. Laut Gilling hat die Niedrighaltung der API-Aufrufe eine hohe Priorität in Bezug auf die Leistung eines API. Im Folgenden wird die Formel für die Berechnung der API-Aufrufe pro Geschäftsvorgang veranschaulicht.

$$A_G = \sum_{i=0}^{n} A_i$$

Die *API-Aufrufe pro Geschäftsvorgang* (A_G) ergibt sich aus der Summe aller definierten API-Anfragen (A).

Gesamt- und Durchschnittslatenz

Einer der auschlaggebendsten Metriken für die Beurteilung der Benutzerwahrnehmung ist laut Gilling die Latenzzeit einer API-Anfrage. APIs mit einer hohen Latenzzeit, werden von Benutzern als schlecht empfunden und unter Umständen nicht weiter benutzt. Eine niedrige Latenzzeit wirkt sich also positiv auf die Benutzerwahrnehmung aus. In dieser Arbeit wird die Latenzzeit als genau diese Zeit definiert, welche von dem Senden des ersten Bytes der Client-Anfrage bis zu dem Erhalten des letzten Bytes der Server-Antwort vergeht. Die Latenzzeit wird mit L angegeben.

Wichtig bei der Messung der Latenzzeiten ist die Gesamtlatenz der API-Aufrufe pro Geschäftsvorgang. Durch diese wird die gesamte Zeit ermittelt, welche für die Erfüllung einer Aufgabe benötigt wird. Die Formel für die Berechnung der Gesamtlatenz sieht wie folgt aus.

$$L_G = \sum_{i=0}^{n} L_i$$

Die Gesamtlatenzzeit (L_G) berechnet sich aus der Summe aller gemessenen Latenzzeiten (L).

Darüber hinaus bietet sich das Ermitteln der Durchschnittslatenz an. Diese gibt die durchschnittliche Latenz aller API-Anfragen an, welche für einen Geschäftsvorgang bearbeitet werden müssen. Die Formel der Durchschnittslatenz schaut wie folgt aus:

$$L_D = \frac{L_G}{A_G}$$

Die durchschnittliche Latenzzeit (L_D) wird aus dem Quotienten der Gesamtlatenzzeit (L_G) und den API-Aufrufen pro Geschäftsvorgang (A_G) ermittelt.

Gesamte und durchschnittliche Bytegröße der Rückgabedaten

Unter Bytegröße der Rückgabedaten wird die Bytegröße der Repräsentation einer Ressource verstanden. Betrachtet wird dabei nur die Größe der tatsächlich repräsentierenden Daten. Das heißt, dass nur der Body eines HTTP-Response, ohne den dazugehörigen Headerfeldern, betrachtet wird.

Die gesamte Bytegröße der Rückgabedaten (B_G) gibt dabei die Summe aus den Bytegrößen der Rückgabedaten (B) aller API-Anfragen an, welche für die Erfüllung einer API-Anfrage benötigt werden:

$$B_G = \sum_{i=0}^{n} B_i$$

Die durchschnittliche Bytegröße der Rückgabedaten (B_D) wird aus der Differenz der gesamten Bytegröße der Rückgabedaten (B_G) und den API-Aufrufen pro Geschäftsvorgang (A_G) gebildet:

$$B_D = \frac{B_G}{A_G}$$

4.2 Methodische Vorgehensweise zur Erfassung der Leistung

Basierend auf den beschriebenen Leistungsmetriken wird im Folgenden die methodische Vorgehensweise für die Erfassung der Leistung eines API erläutert. Es wird davon ausgegangen, dass ein API auf dem zustandlosem Übertragungsprotokoll HTTP aufbaut. Auf Grundlage dieser Annahme dürfen HTTP-Requests nur sequenziell abgearbeitet werden. Das bedeutet, dass jeder Request über eine eigene TCP Verbindung bearbeitet wird und keine Konzepte wie HTTP/2 Multiplexing oder HTTP/1.1 Pipelining hinzugezogen werden.

Für die Erfassung der Leistung eines API müssen zuerst API-Anfragen beschrieben werden, anhand derer die Leistung gemessen werden kann. Nach der Definition dieser Anfragen werden diese nacheinander, ihrer Reihenfolge nach abgearbeitet. Währenddessen wird für jede Anfrage sowohl die *Latenzzeit* als auch die *Bytegröße*

der Rückgabedaten gemessen und auf die *gesamte Latenzzeit* und die *gesamte Bytegröße* addiert. Anschließend werden aus den gemessenen Ergebnissen die *durchschnittliche Latenzzeit* als auch die *durchschnittliche Bytegröße* berechnet. Dieser Prozess wird durch die Abbildung 7 graphisch veranschaulicht.

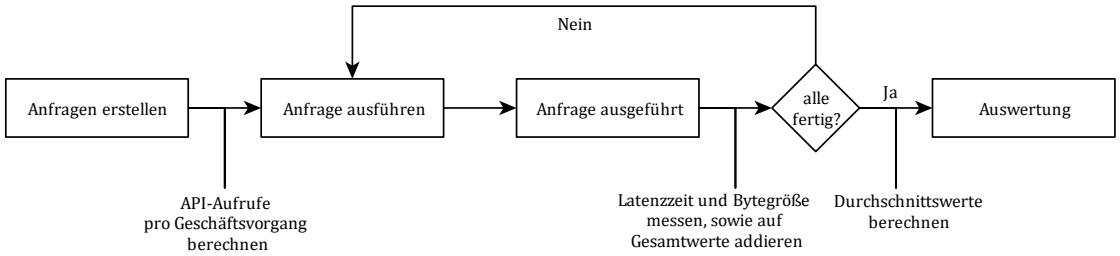

Abbildung 7 Methodik zur Erfassung der Leistungsmetriken

Umgesetzt wird die entworfene Methodik durch ein Python Skript, welches in dem Listing 18 veranschaulicht wird. Dieses verwendet das Python Paket *Requests,* um API-Anfragen zu verschicken. Standardmäßig wird durch das Paket die *ElapsedTime* und der *Content* einer Anfrage berechnet, woraus sich die *Latenzzeit* und die Rückgabedaten einer Anfrage ergeben. Aus den Rückgabedaten kann anschließend die *Bytegröße der Rückgabedaten* ermittelt werden. Durch die ermittelten Ergebnisse werden im Anschluss die Gesamt- und Durchschnittswerte kalkuliert. Die berechneten Ergebnisse werden abschließend auf der Kommandozeile ausgegeben.

```
import requests

REST_BASE_URL = "https://sample-world-service.herokuapp.com/rest/v1"
GRAPHQL_ENDPOINT = "https://sample-world-service.herokuapp.com/graphql"


def evaluate(responses):
    print("{0:>8} {1:>15} {2:>15} {3:>15}"
        .format("", "Status Code", "Latency", "Byte size"))
    data = [[r.status_code,
             r.elapsed.total_seconds(),
             len(r.content)] for r in responses]
    latencies = [r.elapsed.total_seconds() for r in responses]
    content_lengths = [len(r.content) for r in responses]

    total_latency = sum(latencies)
    average_latency = total_latency / len(latencies)
    total_byte_size = sum(content_lengths)
    average_byte_size = total_byte_size / len(content_lengths)

    for i, row in enumerate(data):
        print("{0:>8} {1:>15} {2:>15} {3:>15}"
              .format(i, row[0], row[1], row[2]))

    print("{0:>8} {1:>15} {2:>15} {3:>15}"
        .format("Average", "/", average_latency, int(average_byte_size)))
    print("{0:>8} {1:>15} {2:>15} {3:>15}"
        .format("Total", "/", total_latency, int(total_byte_size)))


if __name__ == '__main__':
    rest = [
        requests.get(url=f"{REST_BASE_URL}/cities")
    ]

    graphql = [
        requests.post(url=GRAPHQL_ENDPOINT,
                      json={"query": "{cities{name}}"})
    ]

    print("\n------------------- REST Evaluation -------------------")
    evaluate(rest)
    print("\n----------------- GraphQL Evaluation ------------------")
    evaluate(graphql)
```

Listing 18 Python Code zur Erfassung der Leistung

4.3 Entwurf eines Experiments für die Erfassung der Leistung

Auf Grundlage der APIs aus dem Abschnitt 3.3 wird ein Experiment zur Erfassung dessen Leistungen entworfen. Im Zusammenhang dieses Experiments müssen beide APIs die gleichen Aufgaben bewältigen. Eine Aufgabe setzt sich aus einer oder mehreren API-Anfragen in einem spezifischen Anwendungsszenario zusammen. Während des Experiments wird davon ausgegangen, dass eine Aufgabe erst abgearbeitet wird, wenn die vorherigen Aufgaben bereits abgearbeitet wurden, da diese aufeinander aufbauen können. Um mit dem Experiment repräsentative Ergebnisse zu ermitteln und um mehrere Clients zu simulieren, müssen alle Aufgaben zehnmal abgearbeitet werden.

Folgende Aufgaben müssen beide APIs während der Durchführung des Experimentes abarbeiten.

- *A1* Abfragen des Attributs *Name* aller Städte.
- *A2* Abfragen der Attribute *id*, *Name*, *District* und *Population* aller *Städte*.
- *A3* Abfragen des Attributs *Name* einer spezifischen *Stadt* sowie des Namens des Landes und Kontinents, zu dem diese Stadt in Relation steht.
- *A4* Hinzufügen des Landes *Rohhan*, sowie der anschließenden Rückgabe des *Ländercodes*.
- *A5* Das Land aus der Aufgabe *A4* soll in Rohan umbenannt werden.
- *A6* Hinzufügen der Städte *Edoras*, *Aldburg*, *Grimslade* und *Hochborn* zu dem Land *Rohan* aus Aufgabe *A4*.
- *A7* Löschen des *Landes* aus der Aufgabe *A4*.

Zusammengefasst wird durch ein Python Skript die entworfene Methodik umgesetzt. Mit diesem Python Skript können die genannten Aufgaben abgearbeitet werden. Auf Grundlage des beschriebenen Experiments wird im Folgenden der Leistungsvergleich zwischen den in dem Abschnitt 3.3 entwickelten APIs durchgeführt.

5 Durchführung des Leistungsvergleichs zwischen REST und GraphQL

Nachdem die methodische Herangehensweise für die Erfassung der Leistung eines APIs auf Grundlage von Leistungsmetriken geklärt wurde, wird in diesem Kapitel das beschriebene Experiment aus dem Abschnitt 4.3 mithilfe der entwickelten APIs aus dem Abschnitt 3.3 durchgeführt.
Als erstes werden die Spezifikationen des Testsystems beschrieben, auf dem die Beispielanwendung ausgeführt wird. Anschließend werden mithilfe dieses Testsystems die beschriebenen Aufgaben des Experiments ausgeführt und ausgewertet.

5.1 Spezifikationen des Testsystems

Die entwickelte Beispielanwendung aus dem Kapitel 3 wird bei der Cloud-Plattform Heroku[8] gehostet. Heroku Applikationen laufen auf leichtgewichtigen Linux Containern, welche von dem Heroku-Team als *Dyno* (dt. Prüfstand) beschrieben werden. Als Benutzer des Cloud-Dienstes kann ein solcher *Dyno* kostenlos für das Testen von Anwendungen verwendet werden. Bei der Verwendung eines kostenlosen *Dyno* muss der Benutzer gewisse Einschränkungen beachten.
Zu den Einschränkungen zählt unter anderem, dass sich die Anwendung nach dreißig Minuten in einen Stand-By-Modus versetzt, wenn diese in dieser Zeit keinen Web-Traffic erhalten hat. Das bedeutet, dass die Anwendung vor dem Vergleich als erstes durch einen HTTP-Request geweckt werden muss, damit der Anwendungsstatus nicht zu fehlerhaften Messerwerten führt.

[8] Anwendungs-URL: http://sample-world-service.herokuapp.com/

5.2 Durchführung des Experiments

In diesem Abschnitt wird das beschriebene Experiment aus dem Abschnitt 4.3 durchgeführt. Zu jeder Aufgabe werden zuerst die notwendigen API-Anfragen für das Erfüllen der Aufgaben formuliert. Anschließend werden diese Anfragen mithilfe des Python Skripts, welches in dem Abschnitt 4.2 eingeführt wurde, abgeschickt, woraus sich die Ergebnisse für dieses Experiment ergeben.

Als Hinweis wird erwähnt, dass sowohl die Spalten durchschnittliche Latenzzeit (L_D) und gesamte Latenzzeit (L_G) als auch durchschnittliche Bytegröße der Rückgabedaten (B_D) und gesamte Bytegröße der Rückgabedaten (B_G) zusammengefasst werden, wenn nur eine API-Anfrage für das Erfüllen einer Aufgabe benötigt wird.

Aufgabe A1: Abfragen des Attributs Name aller Städte.

Mit der Aufgabe A1 soll das Attribut *Name* aller *Städte* abgefragt werden. Während die Abfrage für das RESTful-API über den */cities*-Endpunkt erfolgt, kann mit dem GraphQL-API die *cities*-Operation verwendet werden. Aus der Auswahlmenge einer Stadt wird das Feld *name* gewählt. Die API-Anfragen werden in dem Listing 19 veranschaulicht. Beiden APIs benötigen lediglich eine API-Anfrage. Nachfolgend werden die Ergebnisse tabellarisch vorgestellt.

	RESTful-API					GraphQL-API				
Iter.	A_G	L_D *(ms)*	L_G *(ms)*	B_D *(Bytes)*	B_G *(Bytes)*	A_G	L_D *(ms)*	L_G *(ms)*	B_D *(Bytes)*	B_G *(Bytes)*
1	1	289		381.064		1	283		84.218	
2	1	286		381.064		1	296		84.218	
3	1	241		381.064		1	276		84.218	
4	1	278		381.064		1	254		84.218	
5	1	270		381.064		1	261		84.218	
6	1	287		381.064		1	246		84.218	
7	1	270		381.064		1	245		84.218	
8	1	254		381.064		1	312		84.218	
9	1	282		381.064		1	245		84.218	
10	1	269		381.064		1	317		84.218	
Ø	1	273		381.064		1	274		84.218	

Tabelle 4 Aufgabe A1 Ergebnisse

Das RESTful-API weist eine durchschnittliche Gesamtlatenzzeit von 273 ms auf und ist damit um 1 ms schneller als das GraphQL-API. Dagegen liefert das GraphQL-API nur 84218 Bytes zurück, 77,90 Prozent weniger als das RESTful-API, welches

381064 Bytes zurückgeliefert hat. Die Rückgabedaten der API-Anfragen werden in dem Listing 26 (im Anhang) veranschaulicht.

Aufgabe A2: Abfragen der Attribute id, Name, District und Population aller Städte.

Aufgabe A2 steht inhaltlich im Zusammenhang zu der Aufgabe A1. Sowohl der REST-Endpunkt als auch die GraphQL-Operation bleiben identisch. Lediglich die Auswahlmenge der GraphQL-Operation ändert sich, indem diese mit den Feldern *id*, *District* und *Population* ergänzt wird. Für die Abarbeitung dieser Aufgabe benötigen beide APIs, analog zur Aufgabe A1, nur eine API-Anfrage. Diese API-Anfragen werden in dem Listing 20 dargestellt. Nachfolgend werden die Messergebnisse präsentiert.

	RESTful-API					GraphQL-API				
Iter.	A_G	L_D *(ms)*	L_G *(ms)*	B_D *(Bytes)*	B_G *(Bytes)*	A_G	L_D *(s)*	L_G *(s)*	B_D *(Bytes)*	B_G *(Bytes)*
1	1	264		381.064		1	256		307.662	
2	1	261		381.064		1	301		307.662	
3	1	262		381.064		1	260		307.662	
4	1	264		381.064		1	267		307.662	
5	1	273		381.064		1	249		307.662	
6	1	261		381.064		1	256		307.662	
7	1	258		381.064		1	259		307.662	
8	1	265		381.064		1	283		307.662	
9	1	262		381.064		1	259		307.662	
10	1	251		381.064		1	246		307.662	
Ø	1	262		381.064		1	264		307.662	

Tabelle 5 Aufgabe A2 Ergebnisse

Ähnlich zu den Ergebnissen aus der Aufgabe A1 ist das RESTful-API wieder mit 262 ms minimal schneller als das GraphQL-API mit 264 ms. Bei der Bytegröße der Rückgabedaten sieht der Unterschied jedoch größer aus. Das GraphQL-API liefert 307662 Bytes zurück, während das RESTful-API, so wie in der Aufgabe A1, 381064 Bytes zurückliefert. Das Ergebnis des GraphQL-API ist nur noch 19,3 Prozent weniger als das von dem RESTful-API. Die Rückgabedaten der API-Anfragen werden in dem Listing 27 (im Anhang) dargelegt.

Aufgabe A3: Abfragen des Attributs Name einer spezifischen Stadt sowie des Namens des Landes und Kontinents, zu dem diese Stadt in Relation steht.

Mit der Aufgabe A3 wird auf eine spezifische Stadt zugegriffen. Im Kontext dieser Aufgabe wird die Stadt mit der *id 42* aufgerufen. Zusätzlich zu dieser Stadt müssen sowohl das Land als auch der Kontinent, zu dem diese Stadt gehört, abgerufen werden. Mit dem RESTful-API werden dafür drei Endpunkte aufgerufen. Zum einen der */cities/42*, dann */cities/42/country* und danach */countries/DZA/continent*. Bei der Durchführung mit dem GraphQL-API hingegen wird die *city*-Operation verwendet. Die Operation erwartet die *id* einer Stadt als Eingabeargument übergeben und liefert anschließend die dazugehörige Stadt zurück. Aus der Auswahlmenge der Stadt wird das *country*-Feld ausgewählt und aus der Auswahlmenge des *country*-Feldes schließlich das Feld *continent*. Die API-Anfragen werden in dem Listing 23 (im Anhang) dargestellt. Nachfolgend werden tabellarisch die Ergebnisse zu der Aufgabe A3 veranschaulicht.

	RESTful-API					GraphQL-API				
Iter.	A_G	L_D *(ms)*	L_G *(ms)*	B_D *(Bytes)*	B_G *(Bytes)*	A_G	L_D *(s)*	L_G *(s)*	B_D *(Bytes)*	B_G *(Bytes)*
1	3	270	810	165	495	1	224		94	
2	3	279	837	165	495	1	235		94	
3	3	289	868	165	495	1	216		94	
4	3	281	843	165	495	1	219		94	
5	3	278	833	165	495	1	221		94	
6	3	272	815	165	495	1	254		94	
7	3	298	894	165	495	1	222		94	
8	3	278	833	165	495	1	269		94	
9	3	281	841	165	495	1	218		94	
10	3	276	828	165	495	1	210		94	
Ø	3	280	840	165	495	1	229		94	

Tabelle 6 Aufgabe A3 Ergebnisse

Das RESTful-API benötigt für eine API-Anfrage im Durchschnitt 280 ms und für die Durchführung der kompletten Aufgabe im Schnitt 840 ms, während das GraphQL-API die komplette Aufgabe in 229 ms abarbeitet. Daraus geht hervor, dass das GraphQL-API 266,81 Prozent performanter ist. Auch wenn die Leichtgewichtigkeit betrachtet wird, ist das GraphQL-API im Vorteil. Insgesamt werden durch die GraphQL-API-Anfrage 94 Byte zurückgeliefert. Das RESTful-API liefert mehrere Antworten zurück. Im Durchschnitt werden 165 Bytes pro Antwort zurückgegeben. Die gesamte Bytegröße der Rückgabedaten beträgt somit 495 Bytes. Das GraphQL-API

ist sowohl gegenüber der durchschnittlichen als auch der gesamten Bytegröße der Rückgabedaten leichtgewichtiger als das RESTful-API. Wird der Gesamtwert für die Auswertung betrachtet, dann ist das GraphQL-API 81 Prozent leichtgewichtiger als das RESTful-API. Mit dem Listing 28 (im Anhang) werden die Daten aufgeführt.

Aufgabe A4: Hinzufügen des Landes Rohhan, sowie der anschließenden Rückgabe des Ländercodes.

In Aufgabe *A4* wird das fiktionale Land Rohhan hinzugefügt. Nach dem Hinzufügen soll anschließend der Ländercode des Landes zurückgegeben werden. Für das Hinzufügen eines Landes mit dem RESTful-API wird ein POST-Request an */countries* gestellt. Mit dieser Anfrage wird eine JSON als Repräsentation des Landes an den Server übergeben. Auf der Seite des GraphQL-API wird die Operation *addCountry* verwendet. Diese verlangt das Operationsargument *CountryInput*. Über dieses *CountryInput*-Objekt werden die Daten des Landes festgelegt. Anschließend kann über die Auswahlmenge dieser Operation der Ländercode gewählt werden. Die entsprechende API-Anfragen werden in dem Listing 21 (im Anhang) aufgelistet. Im Folgenden werden die Messergebnisse tabellarisch dargestellt.

	RESTful-API					GraphQL-API				
Iter.	A_G	L_D (ms)	L_G (ms)	B_D (Bytes)	B_G (Bytes)	A_G	L_D (s)	L_G (s)	B_D (Bytes)	B_G (Bytes)
1	1	266		241		1	209		38	
2	1	256		241		1	219		38	
3	1	269		241		1	226		38	
4	1	267		241		1	215		38	
5	1	257		241		1	213		38	
6	1	253		241		1	221		38	
7	1	260		241		1	216		38	
8	1	265		241		1	224		38	
9	1	257		241		1	214		38	
10	1	267		241		1	223		38	
Ø	1	262		241		1	218		38	

Tabelle 7 Aufgabe A4 Ergebnisse

Während das RESTful-API die Aufgabe in durchschnittlich 262 ms erfüllt, benötigte das GraphQL-API nur 218 ms. Daraus folgernd ist das GraphQL-API in Bezug auf diese Aufgabe 20,18 Prozent performanter. Im Kontext der gesamten Bytegröße der Rückgabedaten ist das GraphQL-API mit 38 Bytes 84 Prozent leichtgewichtiger als

das RESTful-API mit 241 Bytes. In dem Listing 29 (im Anhang) werden die Rückgabedaten aufgezeigt.

Aufgabe A5: Das Land aus der Aufgabe A4 soll in Rohan umbenannt werden.

Aufbauend auf Aufgabe A4 wird mit der Aufgabe A5 das hinzugefügte Land Rohhan überarbeitet. Das Land wird in Rohan umbenannt. Für das Aktualisieren einer Ressource mit dem RESTful-API wird ein PUT-Request formuliert, welcher sich auf */countries/ME1* bezieht. Mit dieser API-Anfrage wird die überarbeitete JSON Repräsentation an den Server geschickt. Mit der GraphQL-API hingegen wird die Operation *updateCountry* verwendet. Diese Operation erwartet sowohl den *Ländercode* als auch ein *CountryInput* als Argument übergeben. Das Listing 22 (im Anhang) veranschaulicht die API-Anfragen.

	RESTful-API					GraphQL-API				
Iter.	A_G	L_D *(ms)*	L_G *(ms)*	B_D *(Bytes)*	B_G *(Bytes)*	A_G	L_D *(s)*	L_G *(s)*	B_D *(Bytes)*	B_G *(Bytes)*
1	1	261		240		1	217		41	
2	1	259		240		1	226		41	
3	1	272		240		1	220		41	
4	1	254		240		1	228		41	
5	1	261		240		1	209		41	
6	1	254		240		1	215		41	
7	1	257		240		1	211		41	
8	1	274		240		1	261		41	
9	1	244		240		1	257		41	
10	1	262		240		1	216		41	
Ø	1	260		240		1	226		41	

Tabelle 8 Aufgabe A5 Ergebnisse

Beide APIs können die Aufgabe mit nur einer API-Anfrage erfüllen. Das RESTful-API benötigt dafür 260 ms, wohingegen das GraphQL-API die gleiche Operation in 226 ms ausführt. Das GraphQL-API ist also ungefähr 15,04 Prozent schneller als das RESTful-API. Auch wenn die Rückgabedaten der APIs miteinander verglichen werden, sprechen die Ergebnisse für das GraphQL-API. Mit nur 41 Bytes ist das API 83 Prozent leichtgewichtiger als das RESTful-API. Das Listing 30 (im Anhang) legt die Rückgabedaten der API-Anfragen dar.

Aufgabe A6: Hinzufügen der Städte Edoras, Aldburg, Grimslade und Hochborn zu dem Land Rohan aus Aufgabe A4.

Dem Land aus der Aufgabe A4 werden vier Städte hinzugefügt. Mit dem RESTful-API können mithilfe eines POST-Request über den */cities*-Endpunkt Städte hinzugefügt werden. Jede Stadt wird dabei über einen eigenen API-Aufruf hinzugefügt. Mit der GraphQL-API hingegen können mehrere Operationen gleichzeitig mit einer API-Anfrage ausgeführt werden. Das Hinzufügen einer Stadt erfolgt bei dem GraphQL-API über die *addCity*-Operation. Die API-Anfragen sind durch das Listing 24 (im Anhang) aufgeführt. Tabellarisch werden nachfolgend die Ergebnisse dargestellt.

	RESTful-API					GraphQL-API				
Iter.	A_G	L_D *(ms)*	L_G *(ms)*	B_D *(Bytes)*	B_G *(Bytes)*	A_G	L_D *(s)*	L_G *(s)*	B_D *(Bytes)*	B_G *(Bytes)*
1	4	224	894	92	370	1	235		86	
2	4	227	907	92	370	1	221		86	
3	4	229	916	92	370	1	219		86	
4	4	227	906	92	370	1	224		86	
5	4	231	923	92	370	1	219		86	
6	4	224	899	92	370	1	230		86	
7	4	222	888	92	370	1	216		86	
8	4	230	919	92	370	1	232		86	
9	4	227	909	92	370	1	223		86	
10	4	231	923	92	370	1	222		86	
Ø	4	227	908	92	370	1	224		86	

Tabelle 9 Aufgabe A6 Ergebnisse

Das GraphQL-API ist in der Lage diese Aufgabe mit einer durchschnittlichen Latenzzeit von 224 ms durchzuführen und liefert mit jeder Iteration 86 Bytes an Rückgabedaten zurück. Mit dem RESTful-API müssen vier voneinander getrennte API-Anfragen gestellt werden. Eine Anfrage wird im Durchschnitt in 227 ms abgearbeitet und liefert 92 Bytes an Rückgabedaten zurück. Hochgerechnet auf die Gesamtlatenzzeit ergibt das für alle vier Anfragen 908 ms und 370 Bytes an Rückgabedaten. Das GraphQL-API ist somit ungefähr 305,36 Prozent schneller und 76,5 Prozent leichtgewichtiger als das RESTful-API. Mit dem Listing 31 (im Anhang) werden die Rückgabedaten der API-Anfragen veranschaulicht.

Aufgabe A7: Löschen des Landes aus der Aufgabe A4.

Durch die Aufgabe A7 wird das Land und alle mit dem Land verbundenen Ressourcen gelöscht. Das Löschen erfolgt bei dem RESTful-API über einen DELETE-Request. Mit der DELETE-Methode und dem Ländercode des Landes wird dafür der */countries/ME1*-Endpunkt aufgerufen. Zurückgeliefert wird dadurch die Repräsentation der gelöschten Ressource. Alle anderen Ressourcen, die zu diesem Land in Beziehung stehen, werden gleichzeitig durch die Geschäftslogik der Anwendung entfernt. Um die Aufgabe mit dem GraphQL-API zu erfüllen, wird die Operation *deleteCountry* verwendet. Mit der Benutzung dieser Operation muss der Ländercode des Landes als Argument übergeben werden. Die zu der Aufgabe dazugehörigen API-Anfragen werden in dem Listing 25 (im Anhang) aufgeführt. In der Tabelle 10 werden die Messergebnisse zu dieser Aufgabe veranschaulicht und im Anschluss ausgewertet.

	RESTful-API					GraphQL-API				
Iter.	A_G	L_D *(ms)*	L_G *(ms)*	B_D *(Bytes)*	B_G *(Bytes)*	A_G	L_D *(s)*	L_G *(s)*	B_D *(Bytes)*	B_G *(Bytes)*
1	1	227		122		1	230		41	
2	1	199		122		1	225		41	
3	1	211		122		1	242		41	
4	1	213		122		1	205		41	
5	1	219		122		1	208		41	
6	1	204		122		1	220		41	
7	1	217		122		1	245		41	
8	1	221		122		1	248		41	
9	1	192		122		1	230		41	
10	1	223		122		1	207		41	
Ø	1	213		122		1	226		41	

Tabelle 10 Aufgabe A7 Ergebnisse

Beide APIs können die Aufgabe mit nur einer API-Anfrage abarbeiten. Während das GraphQL-API 226 ms benötigt, kann die Aufgabe durch das RESTful-API mit einer gesamten Latenzzeit von nur 213 ms abgearbeitet werden. Das bedeutet, dass das RESTful-API minimal performanter mit 6,10 Prozent Unterschied ist. Wird dahingegen die gesamte Bytegröße der Rückgabedaten betrachtet, dann liefert das GraphQL-API 41 Bytes zurück, während das RESTful-API 122 Bytes zurückgibt. Das GraphQL-API ist zusammengefasst 66,4 Prozent leichtgewichtiger. Durch das Listing 32 (im Anhang) werden die Rückgabedaten veranschaulicht.

5.3 Auswertung des Experiments

In diesem Abschnitt werden die präsentierten Messergebnisse analysiert und bewertet.

Auffällig ist, dass das RESTful-API für komplexere Aufgaben mehrere API-Anfragen pro Geschäftsvorgang benötigt, woraus sich das Problem der Multiple-Round-Trips ergibt. Dahingegen kann das GraphQL-API dieses Problem durch den funktionalen Vorteil verhindern, dass mehrere GraphQL-Operationen über eine API-Anfrage abgearbeitet werden können. Zusammengefasst kann das GraphQL-API jede Aufgabe mit nur einem HTTP-Request bewältigen.

Durch die API-Anfragen pro Geschäftsvorgang können auch die gemessenen Latenzzeiten begründet werden. Mit der Abbildung 8 werden die erhobenen Gesamtlatenzzeiten pro Aufgabe zusammengefasst.

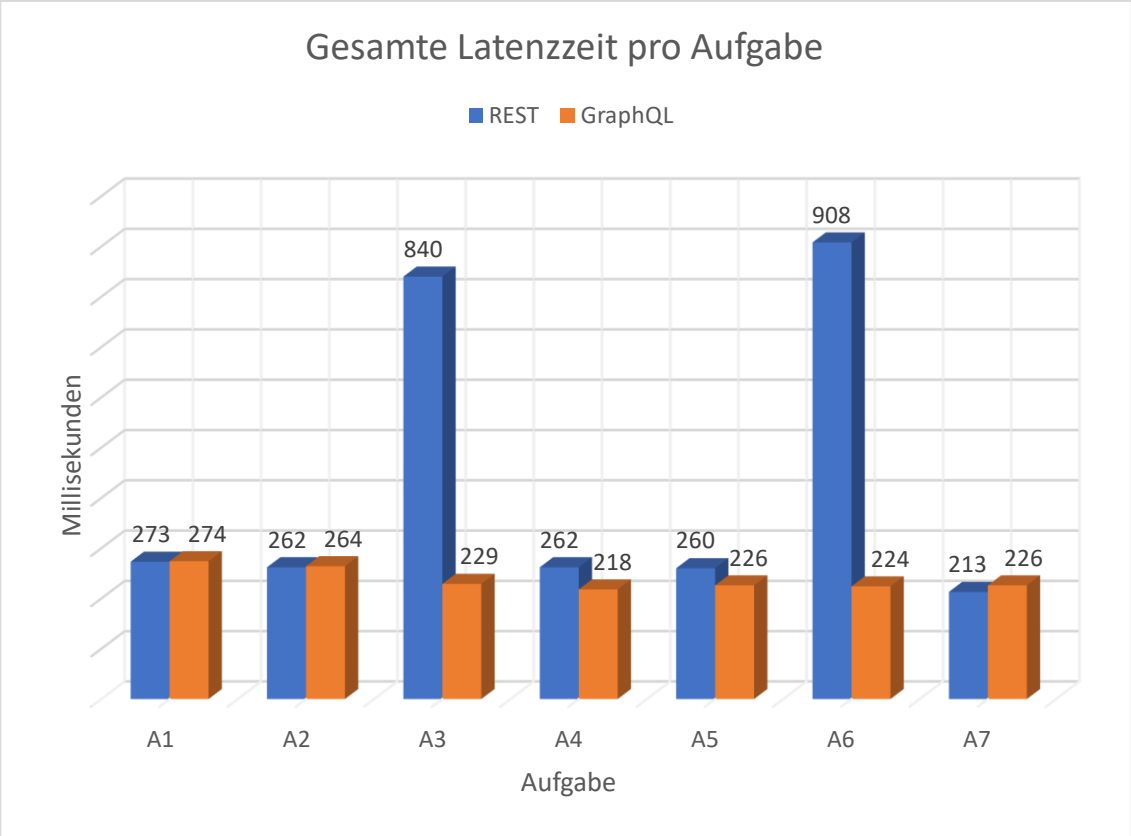

Abbildung 8 Messergebnisse zu der Gesamtlatenzzeit

Beide APIs können die Aufgaben annähernd gleich performant abarbeiten. Lediglich die Aufgaben *A3* und *A6* fallen auf. Aufgabe *A3* und *A6* sind die Aufgaben, bei denen auf der Seite des RESTful-API das Problem der Multiple-Round-Trips auftritt. Dar-

aus ist zuschließen, dass wenn das RESTful-API mehrere API-Anfragen pro Geschäftsvorgang benötigt, dann steigt die Gesamtlatenzzeit linear zu der Anzahl der API-Anfragen. Daraus folgernd ist festzustellen, dass das GraphQL-API durchschnittlich für komplexere Aufgaben deutlich performanter ist als das RESTful-API. Einfache Aufgabe hingegen können beide APIs annähernd gleich performant ausführen. Durch die Abbildung 9 werden die erhobenen Messergebnisse zu der gesamte Bytegröße pro Aufgabe veranschaulicht.

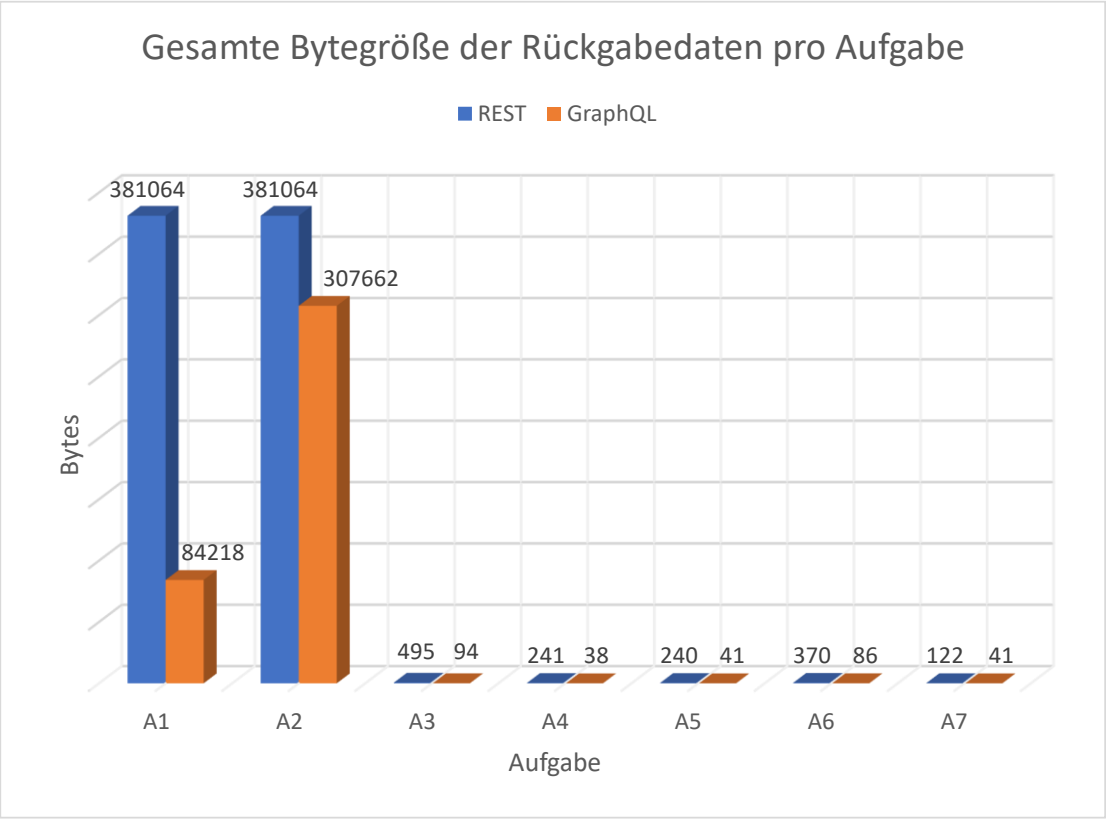

Abbildung 9 Messergebnisse zu der gesamten Bytegröße der Rückgabedaten

Im Zusammenhang mit der Bytegröße der Rückgabedaten fällt zusätzlich auf, dass durch GraphQL in jeder Aufgabe deutlich weniger Daten zurückgeliefert werden. RESTful-APIs sind anfällig für das Problem des Over-Fetching. Dieses wird von GraphQL dadurch verhindert, dass die Benutzer des API selbst entscheiden können, welche Daten zurückgeliefert werden sollen.

Schlussfolgernd lässt sich also feststellen, dass das GraphQL-API die Probleme der Multiple-Round-Trips und des Over-Fetching verhindert, wodurch es, in Anbetracht der hier herangezogenen Kriterien, performanter und leichtgewichtiger als das RESTful-API ist.

6 Zusammenfassung und Fazit

Auf den Grundlagen von *Application Programming Interfaces*, *Representational State Transfe*r und *GraphQL* (Kapitel 2) wurde eine Beispielanwendung entworfen und implementiert (Kapitel 3). In dem Abschnitt 3.3 wurden für diese Anwendung sowohl ein RESTful-API als auch ein GraphQL-API entworfen und umgesetzt. Anhand dieser Beispielanwendung wurde ein Vergleich in Bezug auf die Leistung der beiden entwickelten APIs entworfen, durchgeführt und bewertet (Kapitel 4 und 5).
Zusammengefasst hat sich ergeben, dass umso komplexer das Datenmodell einer Anwendung ist, desto mehr kann von einem GraphQL-API profitiert werden. Durch ein GraphQL-API können typische Leistungsprobleme wie das Over-Fetching und Multiple-Round-Trips verhindert werden, welche im Zusammenhang mit ressourcenorientierten Ansätzen auftreten. Diese Probleme werden von GraphQL dadurch verhindert, dass zum einen mehrere Operationen über nur einen POST-Request abgearbeitet werden können und zum anderen der Benutzer selbst bestimmen kann, welche Daten eine Anfrage zurückgeben soll. Dadurch können die Latenzzeit und die Größe der Rückgabedaten verringert werden, wodurch GraphQL-APIs allgemein performanter und leichtgewichtiger als RESTful-APIs sind. Nichtsdestotrotz bietet sich die Wahl eines RESTful-API in manchen Entwicklungsszenarien an. Soll beispielsweise eine Anwendung mit einem einfachen und flachen Datenmodell umgesetzt werden, dann kommt REST als Architekturstil durchaus in Frage, da durch die Implementierung eines RESTful-API die aufwendige Schemadefinition, die mit GraphQL anfallen würde, vermieden werden kann und beide Ansätze annähernd gleich performant wären.

7 Ausblick

In dieser Arbeit wurde die Leistung von GraphQL und REST miteinander verglichen. Dabei wurden die API-Anfragen während der Abarbeitung sequenziell abgearbeitet. Aufbauend auf diese Arbeit könnte ein ähnlicher Vergleich durchgeführt werden, welcher Konzepte wie das HTTP/1.1 Pipelining oder HTTP/2 Multiplexing berücksichtigt. Eine Vermutung wäre, dass das RESTful-API in den Aufgaben mit mehreren API-Anfragen dadurch eine bessere Leistung zugesprochen werden kann und annähernd gleich performant wie das GraphQL-API wäre.
Des Weiteren könnten REST und GraphQL noch durch andere Vergleichskriterien gegenübergestellt werden. Beispielsweise könnte die Änderbarkeit der beiden API-Architekturstilen untersucht werden, um zu ermitteln, welcher Ansatz eine bessere Abwärtskompatibilität anbietet. Wahrscheinlich hat GraphQL durch die Verwendung von Auswahlmengen einen Vorteil, da ein Feld nur zurückgeliefert wird, wenn dieses von einem Client angegeben wird. Wird beispielsweise ein neues Feld hinzugefügt, dann bleibt dieses dem Client verborgen. Ein REST-Client dahingegen bekommt eine aktualisierte Repräsentation der Ressource zurückgeliefert, weswegen der Client in der Lage sein muss die neuen Daten zu verarbeiten.
Des Weiteren erscheinen sicherheitsbezogene Aspekte interessant. Daher, dass GraphQL nur einen POST-Endpunkt verwendet, stellt sich die Frage wie einzelne GraphQL-Operationen durch zum Beispiel einen Token oder Key gesichert werden können. Außerdem könnte das Caching von GraphQL-APIs untersucht werden. Während durch die zustandlose Kommunikation und den selbstbeschreiben Nachrichten REST-Anfragen zwischengespeichert werden können, könnte GraphQL durch die Verwendung von nur einem POST-Endpunkt einen Nachteil haben.
Zusammengefasst bedeutet das, dass REST und GraphQL noch durch weitere Kriterien verglichen werden können, woraus sich weitere Entwicklungsszenarien herausstellen, in denen die Verwendung von REST oder GraphQL sinnvoller erscheint.

8 Literaturverzeichnis

Apigee (2012). *Web API Design: The Missing Link.*

Bakker, Paul. (2021). *Open Sourcing the Netflix Domain Graph Service Framework: GraphQL for Spring Boot*. *Zugriff am 24. August 2021*. https://netflixtechblog.com/open-sourcing-the-netflix-domain-graph-service-framework-graphql-for-spring-boot-92b9dcecda18

Berlind, David. (2015). *APIs are user interfaces – just different users in mind*. Zugriff am 23. Juni 2021. https://www.programmableweb.com/news/apis-are-user-interfaces-just-different-users-mind/analysis/2015/12/03

Bloch, Joshua. (2014). *A Brief Opinionated History of the API*. Zugriff am 25. Juni 2021. https://2014.splashcon.org/details/plateau2014/2/Invited-Speaker-A-Brief-Opinionated-History-of-the-API

Byron, Lee. (2015). *GraphQL: A data query language*. Zugriff am 14. August 2021. https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/

Byron, Lee. (2018). *GraphQL specification*. Zugriff am 29. Juni 2021. https://spec.graphql.org/June2018/

Fielding, T., Roy. (2000). Architectural Styles and the Design of Network-based Software Architectures. https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Gilling, Derric. (2020). 13 API Metrics That Every Platform Team Should be Tracking. Zugriff am 24. Juli 2021. https://www.moesif.com/blog/technical/api-metrics/API-Metrics-That-Every-Platform-Team-Should-be-Tracking/

Golubski, Wolfgang. (2019). *Entwicklung von verteilten Anwendungen*. Springer Verlag

JOOQ. (2021). *DSLContext*. Zugriff am 03. September 2021. https://www.jooq.org/javadoc/latest/org.jooq/org/jooq/DSLContext.html

Kress, Dominik. (2020). *GraphQL: Eine Einführung in APIs mit GraphQL*. dpunkt.Verlag

Santos, Wendell. (2019). *APIs show Faster Growth Rate in 2019 than Previous Years*. Zugriff am 25. Juni 2021l. https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17

Spichale, Kai. (2019). *API-Design: Praxishandbuch für Java- und Webservice-Entwickler*. dpunkt.Verlag

Walls, Craig. (2012). *Spring im Einsatz*. Hanser Verlag

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Anhang

```
rest = [
    requests.get(url=REST_BASE_URL + "/cities")
]

graphql = [
    requests.post(url=GRAPHQL_ENDPOINT,
                  json={"query": '{cities{name}}'})
]
```

Listing 19 Aufgabe A1 API-Anfragen

```
rest = [
    requests.get(url=REST_BASE_URL + "/cities")
]

graphql = [
    requests.post(url=GRAPHQL_ENDPOINT,
                  json={"query": '{cities{id name district population}}'})
]
```

Listing 20 Aufgabe A2 API-Anfragen

```
rest = [
    requests.post(url=REST_BASE_URL + "/countries",
                  json={
                      "code": "ME1",
                      "name": "Rohhan",
                      "continentId": 1,
                      "region": "Mittelerde",
                      "surfaceArea": 5824000,
                      "indpendenceYear": "213",
                      "governmentForm": "",
                      "code2": "M1",
                      "population": 2500000,
                      "lifeExpectancy": 23.57,
                      "gnp": 4567,
                      "localName": "Rohan",
                      "capitalId": 1
                  })
]

graphql = [
    requests.post(url=GRAPHQL_ENDPOINT,
                  json={"query": """mutation {addCountry(country: {
                                        code: "ME1",
                                        name: "Rohhan",
                                        continentId: 1,
                                        region: "Mittelerde",
                                        surfaceArea: 5824000,
                                        indpendenceYear: "213",
                                        governmentForm: "",
                                        code2: "M1",
                                        population: 2500000,
                                        lifeExpectancy: 23.57,
                                        gnp: 4567,
                                        localName: "Rohan",
                                        capitalId: 1}) {code}}"""})
]
```

Listing 21 Aufgabe A4 API-Anfragen

```
rest = [
    requests.put(url=REST_BASE_URL + "/countries/ME1",
                 json={
                     "code": "ME1",
                     "name": "Rohan",
                     "continentId": 1,
                     "region": "Mittelerde",
                     "surfaceArea": 5824000,
                     "indpendenceYear": "213",
                     "governmentForm": "",
                     "code2": "M1",
                     "population": 2500000,
                     "lifeExpectancy": 23.57,
                     "gnp": 4567,
                     "localName": "Rohan",
                     "capitalId": 1
                 })
]

graphql = [
    requests.post(url=GRAPHQL_ENDPOINT,
                  json={"query": """mutation {updateCountry(code: "ME1"
country: {
                                        code: "ME1",
                                        name: "Rohan",
                                        continentId: 1,
                                        region: "Mittelerde",
                                        surfaceArea: 5824000,
                                        indpendenceYear: "213",
                                        governmentForm: "",
                                        code2: "M1",
                                        population: 2500000,
                                        lifeExpectancy: 23.57,
                                        gnp: 4567,
                                        localName: "Rohan",
                                        capitalId: 1}) {code}}"""})
]
```

Listing 22 Aufgabe A5 API-Anfragen

```
rest = [
    requests.get(url=REST_BASE_URL + "/cities/42"),
    requests.get(url=REST_BASE_URL + "/cities/42/country"),
    requests.get(url=REST_BASE_URL + "/countries/DZA/continent")
]

graphql = [
    requests.post(url=GRAPHQL_ENDPOINT,
                  json={"query":
                        "{city(id:42){name country{name continent{name}}}}"
                        })
]
```

Listing 23 Aufgabe A3 API-Anfragen

```
rest = [
    requests.post(url=REST_BASE_URL + "/cities",
                  json={"name": "Edoras",
                        "countryCode": "ME1",
                        "district": "Mittelerde",
                        "population": 157000}),
    requests.post(url=REST_BASE_URL + "/cities",
                  json={"name": "Aldburg",
                        "countryCode": "ME1",
                        "district": "Mittelerde",
                        "population": 535000}),
    requests.post(url=REST_BASE_URL + "/cities",
                  json={"name": "Grimslade",
                        "countryCode": "ME1",
                        "district": "Mittelerde",
                        "population": 210000}),
    requests.post(url=REST_BASE_URL + "/cities",
                  json={"name": "Hochborn",
                        "countryCode": "ME1",
                        "district": "Mittelerde",
                        "population": 705000})
]

graphql = [
    requests.post(url=GRAPHQL_ENDPOINT,
                  json={"query": """
                  mutation {
                    c1: addCity(city: {name: "Edoras"
                                       district: "Mittelerde"
                                       population: 157000
                                       countryCode: "ME1"}) {id}
                    c2: addCity(city: {name: "Aldburg"
                                       district: "Mittelerde"
                                       population: 535000
                                       countryCode: "ME1"}) {id}
                    c3: addCity(city: {name: "Grimslade"
                                       district: "Mittelerde"
                                       population: 210000
                                       countryCode: "ME1"}) {id}
                    c4: addCity(city: {name: "Hochborn"
                                       district: "Mittelerde"
                                       population: 705000
                                       countryCode: "ME1"}) {id}
                  }
                  """})
]
```

Listing 24 Aufgabe A6 API-Anfragen

```
rest = [
    requests.delete(url=REST_BASE_URL + "/countries/ME1")
]

graphql = [
    requests.post(url=GRAPHQL_ENDPOINT,
                  json={"query":
                          'mutation {removeCountry(code: "ME1"){code}}'})
]
```

Listing 25 Aufgabe A7 API-Anfrage

```
REST Rückgabedaten
[
  {
    "id": 1,
    "name": "Kabul",
    "countryCode": "AFG",
    "district": "Kabol",
    "population": 1780000
  },
  {
    "id": 2,
    "name": "Qandahar",
    "countryCode": "AFG",
    "district": "Qandahar",
    "population": 237500
  },
  ...
  {
    "id": 4079,
    "name": "Rafah",
    "countryCode": "PSE",
    "district": "Rafah",
    "population": 92020
  }
]

GraphQL Rückgabedaten
{
  "data": {
    "cities": [
      {
        "name": "Kabul"
      },
      {
        "name": "Qandahar"
      },
      ...
      {
        "name": "Rafah"
      }
    ]
  }
}
```

Listing 26 Rückgabedaten Aufgabe A1

REST Rückgabedaten

```
[
  {
    "id": 1,
    "name": "Kabul",
    "countryCode": "AFG",
    "district": "Kabol",
    "population": 1780000
  },
  {
    "id": 2,
    "name": "Qandahar",
    "countryCode": "AFG",
    "district": "Qandahar",
    "population": 237500
  },
  …

  {
    "id": 4079,
    "name": "Rafah",
    "countryCode": "PSE",
    "district": "Rafah",
    "population": 92020
  }
]
```

GraphQL Rückgabedaten

```
{
  "data": {
    "cities": [
      {
        "id": "1",
        "name": "Kabul",
        "district": "Kabol",
        "population": 1780000
      },
      {
        "id": "2",
        "name": "Qandahar",
        "district": "Qandahar",
        "population": 237500
      },
      …
      {
        "id": "4079",
        "name": "Rafah",
        "district": "Rafah",
        "population": 92020
      }
    ]
  }
}
```

Listing 27 Rückgabedaten Aufgabe A2

REST Rückgabedaten

Anfrage 1


```
{
  "id": 42,
  "name": "Skikda",
  "countryCode": "DZA",
  "district": "Skikda",
  "population": 128747
}
```


Anfrage 2


```
{
  "code": "DZA",
  "name": "Algeria",
  "continentId": 3,
  "region": "Northern Africa",
  "surfaceArea": 2381741.00,
  "independenceYear": "1962",
  "population": 31471000,
  "lifeExpectancy": 69.7,
  "gnp": 49982.00,
  "localName": "Al-Jaza'ir/Algérie",
  "governmentForm": "Republic",
  "capitalId": 35,
  "code2": "DZ"
}
```


GraphQL Rückgabedaten


```
{
  "data": {
    "city": {
      "name": "Skikda",
      "country": {
        "name": "Algeria",
        "continent": {
          "name": "Africa"
        }
      }
    }
  }
}
```


Listing 28 Rückgabedaten Aufgabe A3

```
REST Rückgabedaten
{
  "code": "ME1",
  "name": "Rohhan",
  "continentId": 1,
  "region": "Mittelerde",
  "surfaceArea": 5824000.00,
  "independenceYear": null,
  "population": 2500000,
  "lifeExpectancy": 23.6,
  "gnp": 4567.00,
  "localName": "Rohan",
  "governmentForm": "",
  "capitalId": 1,
  "code2": "M1"
}

GraphQL Rückgabedaten
{
  "data": {
    "city": {
        "code": "ME1"
      }
   }
}
```

Listing 29 Rückgabedaten Aufgabe A4

```
REST Rückgabedaten
{
  "code": "ME1",
  "name": "Rohan",
  "continentId": 1,
  "region": "Mittelerde",
  "surfaceArea": 5824000.00,
  "independenceYear": null,
  "population": 2500000,
  "lifeExpectancy": 23.6,
  "gnp": 4567.00,
  "localName": "Rohan",
  "governmentForm": "",
  "capitalId": 1,
  "code2": "M1"
}


GraphQL Rückgabedaten

{
  "data": {
    "city": {
        "code": "ME1"
      }
   }
}
```

Listing 30 Rückgabedaten Aufgabe A5

```
REST Rückgabedaten
Anfrage 1
{
  "id": 4481,
  "name": "Edoras",
  "countryCode": "ME1",
  "district": "Mittelerde",
  "population": 157000
}
Anfrage 2
{
  "id": 4482,
  "name": "Aldburg",
  "countryCode": "ME1",
  "district": "Mittelerde",
  "population": 535000
}
Anfrage 3
{
  "id": 4483,
  "name": "Grimslade",
  "countryCode": "ME1",
  "district": "Mittelerde",
  "population": 210000
}
Anfrage 4
{
  "id": 4484,
  "name": "Hochborn",
  "countryCode": "ME1",
  "district": "Mittelerde",
  "population": 705000
}

GraphQL Rückgabedaten
{
  "data": {
    "c1": {
      "id": "4485"
    },
    "c2": {
      "id": "4486"
    },
    "c3": {
      "id": "4487"
    },
    "c4": {
      "id": "4488"
    }
  }
}
```

Listing 31 Rückgabedaten Aufgabe A6

```
REST Rückgabedaten
{
  "code": "ME1",
  "name": "Rohan",
  "continentId": 1,
  "region": "Mittelerde",
  "surfaceArea": 5824000.00,
  "independenceYear": null,
  "population": 2500000,
  "lifeExpectancy": 23.6,
  "gnp": 4567.00,
  "localName": "Rohan",
  "governmentForm": "",
  "capitalId": 1,
  "code2": "M1"
}

GraphQL Rückgabedaten
{
  "data": {
    "removeCountry": {
      "code": "ME1"
    }
  }
}
```

Listing 32 Rückgabedaten Aufgabe A7

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Ort, Datum

Unterschrift
(Maximilian Milz)