Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

placeholder

1.2 Zielsetzung

placeholder

Kapitel 2

Grundlagen

2.1 API

Der Begriff API steht für Application Programming Interface (auf Deutsch Anwendungs-Programmier-Schnittstelle). Die Grundlagen heutiger APIs wurden 1952 von David Wheeler, einem Informatikprofessor an der Universität Cambrigde, in einem Leitfaden verfasst. Dieser Leitfaden beschreibt das Extrahieren einer Sub-Routinen-Bibliothek mit einheitlichem dokumentierten Zugriff. [4] Ira Cotton und Frank Greatorex erwähnten den Begriff erstmalig auf der Fall Joint Computer Con.[2] Dabei erweiterten sie den Leitfaden von David Wheeler um einen wesentlichen Punkt: Der konzeptionellen Trennung der Schnittstelle und Implementierung der Sub-Routinen-Bibliothek. Somit kann die Implementierung auf die die Schnittstelle zugreift ohne Einfluss auf die Benutzer ausgetauscht werden.[3]

> APIs sind wie User Interfaces - nur mit anderen Nutzern im Fokus. David Berlind [1]

APIs sind also wie UIs für die Interaktion mit Benutzern gedacht. Der wesentliche Unterschied zwischen UI-Schnittstellen und einer API liegt aber an der Art der Nutzer die auf das Interface zugreifen. Bei UIs spricht man von einem *human-readable-interface*, das bedeutet das ein menschlicher User mit dem System interagiert. Bei einer API spricht man von einem *machine-readable-interface*, also von einer Schnittstelle die für die Kommunikation zwischen Maschinen gedacht ist.
Ein abstraktes Beispiel für eine API wäre beispielsweise die Post. Angenommen eine Person will einen Brief an eine andere Person senden, um diese Person zum Essen einzuladen. Was passiert ist dann folgendes:

- Person 1 verfasst eine Nachricht und packt diesen in einen Briefumschlag
- Der Briefumschlag wird mit einer Briefmarke und der Adresse des Empfängers und des Absenders versehen
- Der Brief wird nun an die Post übergeben
- Die Post kümmert sich um die Zustellung des Briefes
- Person 2 erhält den Brief

Damit dieser Zugriff der unterschiedlichen Systeme (hier Menschen) auf die Post funktioniert muss eine genaue Definition des Service vorliegen. Die genaue Spezifikation des Service ist dabei folgende:

- Das zu versendende Objekt (Brief, Paket, etc.) muss an einem Sammelposten der Post abgegeben werden
- Das Objekt ist dabei mit einer Empfängeradresse und einer Absenderadresse zu versehen, zusätzlich muss eine Briefmarke gekauft werden

Das stellt die Art des Services dar und legt somit fest was über die API versendet wird. Zudem wird die Repräsentation der API definiert, also in welcher Form der Service im System des Servicenutzers integiert wird. Der Briefkasten / Sammelposten ist dabei die eigentliche Schnittstelle - also die API. Die Zustellung ist dabei Implementierungsdetail, der Weg vom Sammelposten der Post über die Verteilerzentren und mit dem Briefträger zum Empfänger. Dieses Implementierungsdetail kann beliebig angepasst werden, die Zustellung kann beispielsweise über den Land- oder Luftweg erfolgen. Der Benutzer welcher den Brief versendet hat ist davon nicht betroffen.

2.2 REST-API

REST steht für Representational State Transfer [4]. REST ist dabei aber keine konkrete Technologie oder ein Standard. REST beschreibt einen Architekturstil welcher im Jahr 2000 von Roy Fielding konzipiert wurde. Bei REST werden Daten als Ressourcen gesehen und in einem spezifischen Format übertragen. Ursprünglich wurde von Fielding dabei XML verwendet. XML wird aber in den letzten Jahren verstärkt durch JSON abgelöst.

Deswegen ist JSON besser als XML für den Austausch einfacher Daten mittels einer API geeignet:

- JSON wurde speziell für den leichtgewichtigen Datenaustausch konzipiert.
- JSON ist schneller als XML
- JSON hat weniger Overhead als XML da XML deklarativ ist und somit wesentlich mehr Daten als JSON beinhaltet

Zum Vergleich hier ein Buch welches in JSON und XML repräsentiert wird:

```
1 {
2     "isbn": "978-3551555557",
3     "title": "Harry Potter und der Orden des Phönix",
4     "releaseDate": "2003-11-15T00:00:00.000Z"
5 }
```

```
1 <book>
2     <isbn name="isbn" type="string">978-3551555557</isbn>
3     <title name="title" type="string">Harry Potter und der Orden des Phönix</title>
4     <releaseDate name="isbn" type="dateTime">2003-11-15T00:00:00.000Z</releaseDate>
5 </book>
```

Wie in den obigen Code Beispielen ersichtlich produziert JSON (134 Bytes) wesentlich weniger Daten als XML(238 Bytes). Man kann zusammengefasst also sagen, wenn es

rein um die Datenübertragung geht, ist JSON deutlich leichtgewichtiger und damit auch performanter.

2.2.1 REST-Bedingungen

Die Begriffe Web und HTTP müssen von REST unterschieden werden, da REST lediglich einen Architekturstil beschreibt. Das Web hingegen besteht aus mehreren Architekturstilen und nutzt als Standargkommunikationsprotokoll HTTP. HTTP ist als REST-konforme Implementierung zu erachten.

REST hat folgende architektonische Bedingungen:

1. **Einheitliche Schnittstelle:** Jeder REST-Client der auf den Server zugreift muss auf dieselbe Art und Weise zugreifen. Dabei ist es egal ob der Client ein Browser, eine Desktop-Applikation oder eine mobile Anwendung ist. Dabei kommt hier HTTP mit der Verwendung einer URI - *Unique Resource Identifier*, welche dann eine URL ist wie zum Beispiel: *https://api.twitter.com*, zu tragen. Desweiteren müssen Anfragen autark sein - der Client überträgt alle Informationen, die der Server für die Abarbeitung der Anfrage benötigt, mit. Zudem wird die Implementierung von dem Service der sie zur Verfügung stellt entkoppelt, man kann also die Implementierung beliebig weiterentwickeln oder austauschen.
2. **Client-Server Architektur:** Durch das Client-Server Prinzip werden verteilte Systeme beschrieben, die mittels netzwerkbasierter Kommunikation miteinander kommunizieren. In der REST-Architektur sind der Client und der Server klar voneinander abgetrennt. Damit ist es möglich beide Komponenten unabhängig voneinander weiterzuentwickeln, ohne dabei Einfluss auf die jeweils andere Kompnente zu haben
3. **Zustandslosigkeit (Stateless):** Zustandlosigkeit bedeutet, dass der Client alle Informationen die der Server benötigt mitsendet. Das resultiert darin, dass der Server keine Overhad Daten des Clients speichern muss um zukünfige Anfragen abarbeiten zu können. Dadurch ist es auch möglich eine Lastverteilung (also mehrere Serverinstanzen nebeneinander laufen zu lassen) zu realisieren. Dies wiederum erleichtert die Skalierbarkeit des Servers.
4. **Schichtsystem:** Geschichtete Systeme sind Systeme die aus mehreren hierarchischen angeordneten Schichten bestehen. Ein geschichtetes System wäre in .NET beispielsweise so umzusetzen:

   - Controller (Leitet die Anfrage an die Logik weiter)
   - Business-Logik (Setzt die geforderte Aktion um)
   - Datenbankzugriff

   Durch dieses Schichtsystem werden die Abhängikeiten im System reduziert und die Austauschbarkeit der einzelnen Komponenten erleichtert.
5. **Zwischenspeicher (Cache):** Um wiederkehrende Anfragen performanter abwickeln zu können, kann dem Server mithilfe des *Cache-Control-Headers* mitgeteilt werden, ob die angefragten Daten zwischengespeichert werden sollen. Zusätzlich muss definiert werden wielange dieser Cache gültig ist. Nach Ablauf der Gültigkeit werden die Daten erneut vom Server geladen. Durch das Zwischenspeichern von

Daten wird die Performanz des Servers gesteigert. Ein Nachteil besteht dahingehend, dass dem Client womöglich nicht immer die aktuellsten Daten zur Verfügung stehen.

6. **Code auf Anfrage:** Hierbei handelt es sich um eine optionale Bedingung, die Code-Snippets für die lokale Ausführung an den Client senden kann. Beispielsweise als JavaScript-Code innerhalb einer HTML-Antwort.

2.2.2 Kommunikation mit dem Server

Die Kommunikation zwischen Server und Client wird bei einer RESTful-API mittels HTTP und dessen Methoden realisert. Um CRUD (Create, Read, Update, Delete) Operationen auf Ressourcen abzusetzen werden folgende Methoden verwendet:

1. **GET:** Wird als lesender Zugriff auf eine Ressource verwendet.
2. **PUT:** Wird verwendet um eine Ressource zu aktualisieren.
3. **POST:** Wird verwendet um eine neue Ressource zu erstellen.
4. **DELETE:** Wird für das Löschen einer Ressource verwendet.

Zusätzlich wird dem Client mittels HTTP-Statuscodes mitgeteilt ob die Anfrage erfolgreich war oder fehlgeschlagen ist. Beispielsweise werden hier Statuscodes angeführt die an den Client zurückgesendet werden:

- **Erfolgreich:** 200
- **Anfrage fehlerhaft:** 400
- **Server interner Fehler:** 500

2.3 GraphQL

Die Informationen für dieses und das folgende Kapitel wurden aus diesen Quellen bezogen [3, 5–7]

2015 veröffentlichte Facebook unter der MIT-Lizenz GraphQL. GraphQL ist die Spezifikation einer plattformunabhängigen Query-Sprache. GraphQL dient als Übersetzer der Kommunikation zwischen Client und Server. Die Kommunikation erfolgt wie bei REST über ein Request-Response-Schema. Der wesentliche Unterschied zu REST-APIs besteht darin, dass GraphQL genau einen Endpunkt, anstatt für jede Ressource einen eigenen Endpunkt, zur Verfügung zu stellt. Desweiteren werden nur POST-Anfragen von GraphQL akzeptiert, diese können lesend (Query) oder schreibend (Mutation) sein.

2.3.1 Entwurfsprinzipien

GraphQL definiert keine Implementierungsdetails sondern diese Entwurfsprinzipien:

1. **Produktzentriert:** Die Anforderungen der Clients (Darstellung der Daten) stehen im Mittelpunkt. GraphQL bietet mit der Abfragesprache dem Client die Möglichkeit genau die Daten abzufragen, die er tatsächlich benötigt. Die Hierarchie der Abfrage wird dabei durch eine Menge ineinander geschachtelter Felder abgebildet.
2. **Hierarchisch:** Jede Anfrage ist wie die Daten die er anfordert geformt. Das bedeutet, dass der Client die Daten genau in dem Format erhält wie in der Anfrage

spezifiziert. Das ist ein intuitiver Weg für den Client um seine Datenanforderungen zu definieren.

3. **Strenge Typisierung:** Jeder GraphQL-Service definiert ein für das System spezifisches Typ-System. Anfragen werden im Kontext dieses Typ-Systems ausgeführt. Mit Tools wie zum Beispiel GraphiQL kann man vor Ausführung der Anfrage sicherstellen, dass sie syntaktisch und semantisch korrekt sind.
4. **Benutzerdefinierte Antwort:** Durch das Typ-System veröffentlicht der GraphQLService die Möglichkeiten, die ein Client hat um auf die dahinterliegenden Daten zuzugreifen. Der Client ist dafür verantwortlich genau zu spezifizieren wie er diese Möglichkeiten nutzen will. Bei einer normalen REST-API liefert ein Endpunkt jene Daten einer Ressource zurück welche vom Server definiert wurden. Bei GraphQL werden hingegen genau jene Daten einer Ressource zurückgegeben, die vom Client in seiner Anfrage definiert wurden.
5. **Introspektion:** Das Typ-System eines GraphQL-Services kann direkt mit der GraphQL-Abfragesprache abgefragt werden. Diese Abfragen werden für die Erstellung von Tools für GraphQL benötigt.

2.4 GraphQL vs. REST

2.4.1 Endpunkte

Eine REST-API bietet für jede Ressource verschiedene Endpunkte an um CRUD Operationen für die jeweilige Ressource auszuführen. GraphQL hingegen bietet nur einen Endpunkt. An diesem kann der Client eine Abfrage mit einer Query oder einer Mutation senden um auf die jeweilige Ressource zuzugreifen.

2.4.2 Over-fetching und Under-fetching

Als Over-fetching wird das Laden von zuvielen oder nicht benötigten Daten bezeichnet. Under-fetching bedeutet, dass man mehr Daten benötigt als der Server dem Client zurückgibt. Dieses Problem tritt bei REST-APIs auf die viele verschiedene Clients mit Daten versorgen müssen (beispielsweise eine Desktop-Applikation, eine Mobile-Anwendung und ein Web-Client). Grundsätzlich wollen alle drei Clients diesselbe Ressourcen abfragen, mit dem Unterschied, dass die Mobile Anwendung beispielsweise nicht alle Daten benötigt. Die Desktop-Applikation möchte aber alle Daten einer Ressource bekommen um diese darzustellen. Eine Lösung dafür wäre beispielsweise einen Endpunkt für jeden Client zu definieren um die für ihn benötigen Daten zur Verfügung zu stellen. Dies resultiert aber in mehr Programmieraufwand und damit auch einer höheren Komplexität der Anwendung.

Dieses Problem kann bei GraphQL nicht auftreten da der Client in seiner Anfrage genau die Daten definiert die er braucht.

2.4.3 Was ist für welches Szenario besser geeignet?

Möchte man ein System entwickeln auf welches verschiedene Clients (die verschiedene Anforderungen an die Ressourcen haben) zugreifen, dann ist GraphQL die bessere Wahl. Würde man dieses System mit einer REST-API umsetzen, müsste man entweder Probleme mit Under-fetching und Over-fetching in Kauf nehmen, oder für jeden Client eine eigene Route definieren. GraphQL ist in diesem Szenario deshalb als besser zu erachten, weil man sich mit der einmaligen Implementierung des Schemas zusätzliche Routendefinitionen erspart. Dadurch hat man automatisch weniger Entwicklungsaufwand und generell weniger Komplexität.

Verfolgt man aber das Ziel ein System zu entwickeln, welches keine komplexen Daten enthält und beispielsweise nur mit einem Web-Client kommuniziert, ist eine REST-API die bessere Wahl.

Kapitel 3

GraphQL

3.1 Typ-System

Das GraphQL-Typ-System wird zur Definierung eines Schemas verwendet. Ein Schema beschreibt einen GraphQL-Service und besteht aus den abrufbaren Ressourcen, ihren Relationen zueinander und ihren Interaktionsmöglichkeiten.
Eine eingehende Anfrage wird durch die im Schema definierte Datenstruktur validiert. Wenn die in der Anfrage enthaltene Query durch das Typ-System erfolgreich validiert wurde, wird die beinhaltete Operation an die Implementierung weitergeleitet. Dafür zerlegt GraphQL die übergebene Query und gibt sie an den jeweiligen Resolver weiter. Diese Resolver interagieren mit der Geschäftslogik und füllen die angeforderten Felder mit Daten. Die kumulierten Ergebnisse werden als Antwort an den Client zurückgeschickt.

3.2 Schema-Definitions-Sprache SDL

Da GraphQL laut Spezifikation in jeder beliebigen Sprache implementierbar sein soll wird eine sprachunabhängige Basis für die Definierung des GraphQL-Graphen benötigt. Diese sprachunabhängige Basis wird durch die in der Spezifikation definierte Beschreibungssprache (*SDL*) gegeben. In GraphQL existieren folgende Typ-Definitionen *Skalar, Interface, Object, Input Object, Enum, Union* diese bilden das Rückgrat des Schemas. Über diese Typen wird in den nachfolgenden Abschnitten eine genauere Übersicht gegeben.

3.2.1 Skalare

Ein Datentyp der nicht mehr weiter vereinfachbar ist wird wie in anderen Programmiersprachen Skalar-Typ genannt. Skalar-Typen repräsentieren die Blätter, also die primitiven Werte des GraphQL-Typ-Systems. GraphQL-Antworten entsprechen der Form eines hierarchisch aufgebauten Baumes.
Grundsätzlich bestehen die Blätter dieses Baumes aus GraphQL-Skalar-Typen (es ist zudem auch möglich, dass die Blätter aus *Null-Werten* oder *Enum-Typen* bestehen). GraphQL beinhaltet folgende vordefinierten Skalar-Typen:

1. Boolean
2. Float
3. Int
4. String
5. ID

```
    id: Int!
    title: String
```

Im oben angeführten Codebeispiel werden die Felder *id* und *title* definiert. Der Name eines Feldes im umgebenden Typ muss dabei eindeutig sein. Die Deklaration erfolgt mit dem Namen als auch dem Typ des Feldes welche mit einem Doppelpunkt getrennt sind. Das Feld *id* wird dabei mit dem *!* als *not null* deklariert.

Benutzerdefinierte Skalare

In den meisten sprachspezifischen Implementierungen ist es möglich eigene Skalar-Typen zu definieren. Diese werden verwendet um beispielsweise verschiedene Datumsformate darzustellen.

3.2.2 Enum

Enum-Typen stellen wie Skalare die Blätter des Typ-Baums dar. Ein Enum-Feld hält ein spezifisches Element aus einer Menge von möglichen Werten.

```
enum Category {
    Fantasy
    Adventure
    Mystery
    Thriller
    Romance
}
```

In diesem Beispiel wurde ein Enum *Category* definiert. Es gilt zu beachten, dass dieser Typ mit dem Schlüsselwort *enum* definiert werden muss.

3.2.3 Objekt

Um auf die Blätter des Baumes zuzugreifen werden in GraphQL Objekte als Knoten verwendet. Diese Objekte halten dabei eine Liste von Feldern die einen bestimmten Wert lieferen. Dabei kann jedes Feld entweder ein Skalar, Enum, Objekt oder Interface sein. Laut Spezifikation, sollten Objekte als eine Menge von geordneten Schlüssel-Wert-Paaren serialisiert werden. Wobei der Name des Feldes der Schlüssel ist und das Ergebnis der Evaluierung des jeweiligen Feldes den Wert abbildet. Um einen Objekt-Typ zu definieren muss das Schlüsselwort *type* verwendet werden.

```
type Book {
    id: Int!
    title: String
    authors: [Author]
}

```

```
 7 type Author {
 8     id: Int!
 9     firstName: String
10     lastName: String
11     books: [Book]
12 }
```

Im oben angeführten Schemaausschnitt werden die beiden Objekte *Author* und *Book* definiert. Um einen Objekttypen zu definieren wird das Schlüsselwort *type* verwendet. Das Objekt Book wird dabei mit den Feldern *id, title* und *authors* definiert. Das Objekt Author erhält die Felder *id, firstName, lastName* und *books*.
Die Felder *id, title, firstName* und *lastName* sind dabei skalare Felder und bilden dabei Blätter des Baumes. Da zwischen den Objekten Author und Book eine n:m Beziehung besteht, halten beide Objekte eine Liste des jeweils anderen Objekt-Typs. Listen werden in der SDL wie im obigen Beispiel ersichtlich mit eckigen Klammern definiert.

3.2.4 Interface

Interfaces sind abstrakte Typen welche eine Liste an Feldern definieren. GraphQL-Interfaces repräsentieren eine Liste von Felder und deren Argumente. Objekte und Interfaces können ein Interface implementieren, dazu muss der Typ, welcher das Interface definieren will, alle Felder des zu implementierenden Interfaces definieren.
Felder eines Interfaces sind an dieselben Regeln wie ein Objekt gebunden. Der Typ eines Feldes kann entweder ein Skalar, Enum, Interface oder Union sein. Zudem ist es möglich, dass ein Typ mehrere Interfaces implementiert.

Im folgenden Codebeispiel wird die Definition eines Interfaces veranschaulicht:

```
 1 interface Person {
 2     firstName: String
 3     lastName: String
 4 }
 5
 6 type Author implements Person {
 7     id: Int!
 8     firstName: String
 9     lastName: String
10     books: [Book]
11 }
```

Ein Interface kann nicht alleine verwendet werden, es braucht also eine spezifische Implementierung welche im Beispiel mit *Author* umgesetzt wurde. Der Objekt-Typ *Author* hat somit nun alle Felder vom Interface *Person*.

3.2.5 Input-Objekt

Ein Input-Objekt-Typ ist ein spezieller Objekt-Typ. Ein Input-Objekt hält genauso wie ein Objekt skalare Felder, Enumerationen oder Referenzen. Diese referenzierten Objekte müssen aber ebenso Input-Objekte sein. Eine Mischung der Objekt-Typen ist laut Spezifikation nicht erlaubt.

Im folgenden Beispiel wird ein Input-Objekt für ein Buch realisert:

```
input AuthorCreateInput {
  firstName: String
  lastName: String
  books: [Int!]
}
```

3.2.6 Union

Union-Typen fügen mehrere Objekte zu einer Gruppe zusammen. Diese Typen sind sehr nützlich, wenn beispielsweise eine Query zwei unterschiedliche Objekt-Typen zurückgeben kann, diese aber nicht diesselben Felder besitzen. In dieser Query können dann mithilfe von Inline-Fragmenten die spezifischen Felder abgefragt werden.

```
union SearchResult = Author | Buch
```

3.3 Schema

Wenn man das Schema als Baumstruktur betrachtet so sind die referenzierten Objekte Verzweigungen des Baumes. Die Blätter am Ende des Baumes enthalten die eigentlichen Daten. Die Astverzweigungen sind von besonderer Bedeutung, denn sie beinhalten die Referenzen zu den anderen Objekten.
Zudem werden die Wurzelknoten der Eingangspunkte der Operationen (Query, Mutation und Subscription) definiert. Um ein korrektes Schema zu erstellen muss beachtet werden, dass die Typen und Direktiven eindeutig über ihren Namen identifizierbar sind. Außerdem dürfen im Schema definierte Namen nicht mit doppeltem Unterstrich beginnen. Diese sind für das GraphQL Introspektionsystem reserviert. spec.graphl.org/SchemaDefinition

3.4 Wurzel Operationen

Das Schema definiert die initialen Wurzelknoten der Operationen die es unterstützt (Query, Mutation, Subscription). Dadurch definiert das Schema den Ort im Typ System wo diese Operationen beginnen. Query ist aber dabei die einzige Operation welche man zwingend definieren muss. Mutations und Subscriptions sind optional und werden, wenn sie nicht explizit definiert werden, nicht unterstützt. Eine Wurzel-Operation muss dabei ein *Object Type* sein. Desweiteren müssen die Objekttypen der Wurzel-Operationen unterscheiden und dürfen nicht diesselben sein.

Bei komplexeren Graphen würde dies zu einer kaum zu lesenden oder adaptierbaren Schemadefinition führen. Das Schema besteht somit aus einzelnen, geschlossenen Objekttyp-Definitionen, die einander oder aber ihre Felder referenzieren. Seite 58 und 59 Ist ein Feld mit ! deklariert so darf dieses nicht null sein.

3.5 Schema Definition

```
type Book {
  id: Int!
  title: String
  authors: [Author]
}

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

input AuthorUpdateInput {
  id: Int!
  firstName: String
  lastName: String
  books: [Int!]
}

input AuthorCreateInput {
  firstName: String
  lastName: String
  books: [Int!]
}

input BookUpdateInput {
  id: Int!
  title: String
  authors: [Int!]
}

input BookCreateInput {
  title: String
  authors: [Int!]
}

input LoginInput {
  email: String
  password: String
}

type Query {
  books: [Book]
  book(id: Int!): Book
  authors: [Author]
  author(id: Int!): Author
}

type Mutation {
  createBook(input: BookCreateInput): Book
  updateBook(input: BookUpdateInput): Book
  delete(id: Int!): Boolean!
  createAuthor(author: AuthorCreateInput): Author
  updateAuthor(author: AuthorUpdateInput): Author
  login(input: LoginInput): String
}

type Subscription {
  bookAdded: Book
}
```

Abbildung 3.1: Schemadefinition

In Abbildung 3.1 ist eine Schemadefinition zu sehen welche Objekt-Typen, Input-Typen und die Wurzeloperationen beinhaltet.

3.6 Zugriff auf den GraphQL-Service

GraphQL liefert keine Spezifikation über die Netzwerkschicht, sondern lediglich eine Empfehlung *HTTP* zu verwenden. Im Gegensatz zu *REST-APIs* beschränkt sich GraphQL auf lediglich zwei HTTP-Methoden: GET und POST. Deswegen ist die Gestaltung und Bennung der Endpunkte nicht so relevant wie bei REST-APIs.

Der Unterschied, ob ein Client mittels GET oder POST-Anfrage auf den Service zugreift, besteht darin, dass mittels GET-Anfrage nur lesende Zugriffe möglich sind. Also können mit GET-Anfragen nur Querys aber keine Mutations umgesetzt werden. Desweiteren müsste man jene Query, welche zum Abfragen der Daten verwendet werden sollte, als URL-Parameter übergeben. Somit gilt es auch zu beachten die Sonderzeichen der Query in *ASCII* umzuwandeln.

Wenn man nun also mittels GET-Anfrage eine Query an den GraphQL-Service schickt, resultiert es in dem Nachteil, dass man eine wesentlich schlechtere Übersicht hat. Denn

die URL-Encodierung verkompliziert die Anfrage und sorgt für Einschränkungen in der Lesbarkeit. Etwaig benötigte Variablen müssten auf dieselbe Art und Weise der Anfrage hinzugefügt werden. Problematischer ist dabei aber jedoch, dass einige Browser eine Maximallänge für URIs definiert haben. Somit können komplexe Querys nicht funktional sicher über GET-Anfragen abgebildet werden.

Aus diesem Grund ist es Standard, dass in GraphQL alle Anfragen per POST-Anfrage abgewickelt werden. Diese werden an den einzigen, nach außen freigegebenen Endpunkt gerichtet. Da für die Anfragen die POST-Methode verwendet wird, können im Body beliebig viele und komplexe Querys an den Service übergeben werden.

Grundsätzlich besteht eine Anfrage an den GraphQL-Service aus der eigentlichen Query und zwei optionalen Parametern: Variablen und der Name der Operation.

3.7 Querys

Querys bieten den Clients die Möglichkeit lesend auf die Objekte, welche vom GraphQL-Service verwaltet werden, zuzugreifen. GraphQL erlaubt es dem Client genau die Daten abzufragen welche er benötigt. Um auf eine Ressource zuzugreifen wird ein POST-Request an den Wurzelknoten der GraphQL-Applikation geschickt. Dieser POST-Request enthält ein in der *GraphQL Query Language* definiertes Objekt. In diesem Objekt werden die auszuführenden Funktionen definiert und welche Felder davon an den Client zu retournieren sind. Das Ergebnis hat genau jenes Format welches in der Anfrage vom Client definiert wurde.
Da GraphQL mit einem Graphenschema arbeitet, welche auf den Beziehungen der Knoten zueinander basiert, ist es möglich diese Relationen zu nutzen um Daten über mehrere Objekte hinweg zu sammeln.

In der folgenden Abbildung ist eine verschachtelte Query zu sehen, welche alle Bücher mit den dazugehörigen Autoren anfordert.

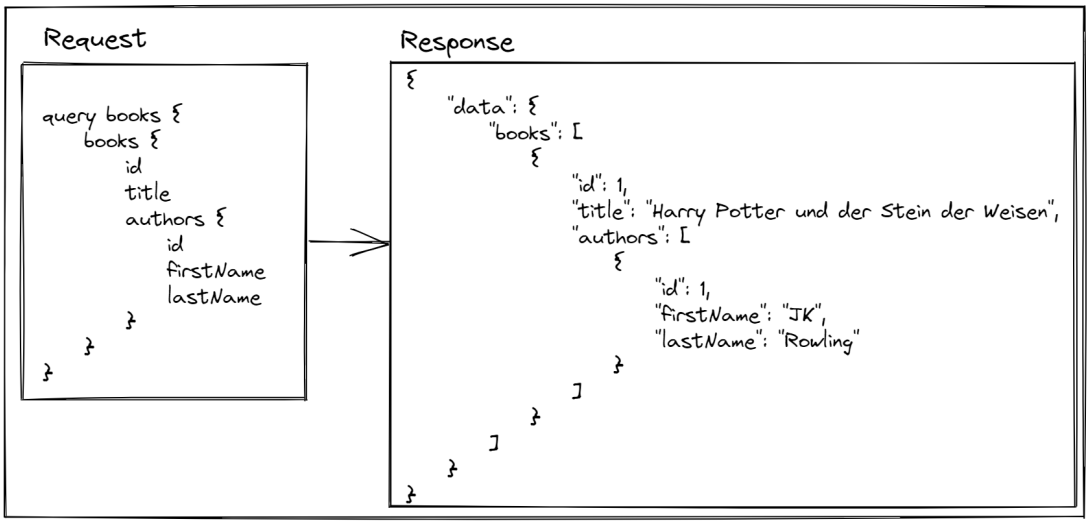

Abbildung 3.2: Anfrage aller Bücher mit zugehörigen Autoren

3.8 Parameter

Felder können zusätzlich noch Parameter halten. Diese Parameter können den Rückgabewert des Feldes ändern um das Feld noch flexibler zu machen.

```
enum Currency {
    EUR
    USD
    GBP
}

type Book {
    id: Int!
    title: String
    authors: [Author]
    price: (unit: Currency = EUR): Float
}
```

Im oben stehenden Code Beispiel wurde der Objekt-Typ Book um ein Feld *price* erweitert. Bei einer Query auf das Feld price des Objekts Book kann ein Argument unit mitgegeben werden um den tatsächlichen Verkaufswert in der richtigen Währung zu bestimmen. Dabei wurde ebenfalls ein Standardwert definiert um unit nicht zwingend als Parameter übergeben zu müssen.

3.8.1 Variablen

Mit Parametern ist es möglich zusätzliche Daten für spezielle Operationen an den GraphQL-Service zu schicken. Diese können aber nur statisch in die Query eingetragen werden. Deswegen kann eine GraphQL Operation zudem mit Variablen erweitert werden. Dadurch hat man verstärkt die Möglichkeit Funktionen wiederzuverwenden. Va-

riablen müssen am Anfang einer Operation definiert werden und befinden sich während der Ausführung im Lebensraum der Operation. Diese Variablen werden unter anderem für die Filterung der angeforderten Objekte verwendet.

In der folgenden Abbildung ist eine Anfrage zu sehen welche das Buch mit der *id = 1* anfordert. Dabei wird die id als Variable übergeben.

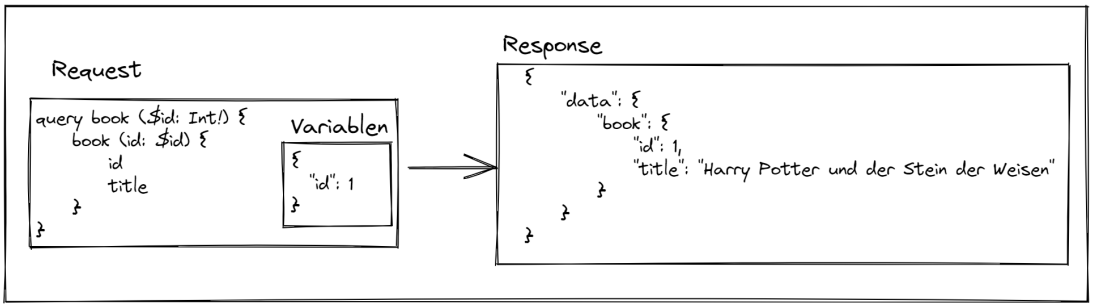

Abbildung 3.3: Anfrage aller Bücher mit zugehörigen Autoren

3.8.2 Aliase

In GraphQL-Querys ist es möglich, dass mehrere miteinander in Konflikt stehende Parameter übergeben werden. Um diesem Problem vorzuwirken besteht die Möglichkeit mit *Aliases* zu arbeiten. Mit ihnen ist es möglich die unterschiedlichen Resultate dieser in Konflikt stehenden Argumente zu benennen und damit zu unterscheiden.

3.8.3 Fragmentierung

Fragmente strukturieren und organisieren komplexe Datenanforderungen. Es ist außerdem möglich Parameter einzuführen, weiters werden bereits in der Query definierte Variablen in den Fragmenten verwendet.

3.9 Mutationen

APIs benötigen neben dem lesenden Zugriff auf Daten auch einen schreibenden. Dieser wird in GraphQL mittels Mutationen umgesetzt. Mutationen kapseln die Implementierungen der Datenbankzugriffsschicht in ein Interface welches die Möglichkeiten zur Manipulation der Daten vorgibt. Manipulierende Anfragen können Daten dadurch nur auf jene Art und Weise ändern, wie es in der Applikation vorgesehen ist.

3.9.1 Designempfehlungen

3.9.2 Subscriptions

Subscriptions sind eine spezielle Form von Query. Sie werden verwendet um den Client vom Server aus über Events zu notifizieren. Notifizierungen werden in der Regel durch

Hinzufügen, Ändern oder Löschen von Datenbankobjekten ausgelöst. Subscriptions halten eine aktive Verbindung zwischen dem GraphQL-Service und dem Client offen. Diese Verbindung wird meistens mit WebSockets umgesetzt. Mit Subscriptions ist es zum Beispiel möglich den aktuellen Bestand von Produkten immer direkt am Client verfügbar zu haben. Mit dieser Information kann auf der Client-Seite gewährleistet werden, dass ein Benutzer nur ein Produkt kaufen kann, welches noch verügbar ist.

Weil sich der Server die Clients merken muss, ändert sich der Status des Systems von einem *statless* zu einem *stateful* API. Dadurch wird wiederum die Komplexität des Systems erhöht und die Skalierbarkeit erschwert.

3.10 Bekannte Probleme

3.10.1 Authentifizierung, Autorisierung und Rollenmanagement

3.10.2 1 + n Problem

3.10.3 Fehlermanagement

3.10.4 Pagination

3.10.5 Caching

Kapitel 4

Entwicklung

4.1 Anwendungsszenario

4.2 Architektur

placeholder

placeholder

4.3 Entwurf Schema

placeholder

4.4 Umsetzung GraphQL mit .NET

4.4.1 Verwendete Bibliotheken

placeholder

4.4.2 Entity Framework

placeholder

placeholder

4.4.3 Umsetzung Authentifizierung, Autorisierung, Rollenmanagement

placeholder

4.4.4 Umsetzung Subscriptions

placeholder

4.4.5 Umsetzung 1 + n Problem

placeholder

4.4.6 Umsetzung Pagination

Kapitel 5

Conclusio

5.1 Fazit

5.2 Ausblick

Literatur

[1] David Berlind. *APIs Are Like User Interfaces–Just With Different Users In Mind*. 2017 (siehe S. 5).

[2] Ira W Cotton und Frank S Greatorex Jr. „Data structures and techniques for remote computer graphics“. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. 1968, S. 533–544 (siehe S. 5).

[3] Dominik Kress. *GraphQL: Eine Einführung in APIs mit GraphQL*. dpunkt. verlag, 2020 (siehe S. 5, 8).

[4] David J Wheeler. „The use of sub-routines in programmes“. In: *Proceedings of the 1952 ACM national meeting (Pittsburgh)*. 1952, S. 235–236 (siehe S. 5, 6).

[5] Facebook. *Graphql*. 15. Nov. 2021. URL: http://spec.graphql.org/June2018/ (besucht am 15. 11. 2021) (siehe S. 8).

[6] Rakuten. *Graphql vs Rest*. 15. Nov. 2021. URL: https://blog.api.rakuten.net/graphql-vs-rest/ (besucht am 15. 11. 2021) (siehe S. 8).

[7] Abu Sakib. *Graphql vs Rest*. 15. Nov. 2021. URL: https://dgraph.io/blog/post/graphql-rest/ (besucht am 15. 11. 2021) (siehe S. 8).