Contents

Chapter 1

Einleitung

1.1 Motivation

placeholder

1.2 Zielsetzung

placeholder

Chapter 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. Wheeler [1952] Ira Cotton und Frank Greatorex erwähnten den Begriff erstmalig auf der Fall Joint Computer Con.Cotton und Greatorex Jr [1968] 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.Kress [2020]

> APIs sind wie User Interfaces - nur mit anderen Nutzern im Fokus.
> David Berlind Berlind [2017]

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 Wheeler [1952]. 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 }
```

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 Kress [2020]; Facebook; Sakib; Rakuten

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.

Chapter 3

GraphQL

3.1 Typ-System

Das GraphQL-Typ-System wird zur Definition 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. [Kress, 2020, S. 57-58] [Facebook, Abs. Schemadefinition]

3.2 Schema-Definitions-Sprache SDL

Da GraphQL laut Spezifikation in jeder beliebigen Sprache implementierbar sein soll, wird eine sprachunabhängige Basis für die Definition des GraphQL-Graphen benötigt. Die Grundlage dafür stellt die in der Spezifikation definierte Beschreibungssprache (*SDL*). In GraphQL existieren folgende Typ-Definitionen: *Skalar, Interface, Object, Input Object, Enum* und *Union*. Diese bilden das Rückgrat des Schemas. Diese Typen werden in den nachfolgenden Abschnitten genauer behandelt.

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 [Kress, 2020, S. 60]. 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). [Facebook, Abs. ScalarTypeDefinition] 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 einem Rufzeichen (!) als *not null* deklariert.
In den meisten sprachspezifischen Implementierungen ist es möglich, eigene Skalar-Typen zu definieren. Diese werden verwendet, um beispielsweise verschiedene Datumsformate darzustellen.

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 [Facebook, Abs. 3.9] [Kress, 2020, S. 60-61].

```
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.

Objekt
Wie bereits erwähnt bestehen die Blätter des Baumes in GraphQL aus den Skalar-Typen. Die Knoten wiederum werden von Objekt-Typen definiert. Diese Objekte halten eine Liste von Feldern die einen bestimmten Wert lieferen. Jedes

Feld kann entweder ein Skalar, ein Enum, ein Objekt oder ein 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 [Facebook, Abs. 3.6].

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

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

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.

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.
[Facebook, Abs. 3.7] 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. [Kress, 2020, S.65-66]

Im folgenden Codebeispiel wird die Definition eines Interfaces veranschaulicht:

```
interface Person {
    firstName: String
    lastName: String
}

type Author implements Person {
    id: Int!
```

```
 8     firstName: String
 9     lastName: String
10     books: [Book]
11 }
```

Jedes Interface benötigt eine spezifische Implementierung, im angeführen Beispiel implementiert *Author* das Interface *Person*. Author muss dabei alle Felder von *Person* zur Verfügung stellen.

Input-Objekt
Ein Input-Objekt ist ein spezieller Objekt-Typ. Ein Input-Objekt hält genauso wie ein Objekt-Typ skalare Felder, Enumerationen oder Referenzen. Diese referenzierten Objekt-Typen müssen aber ebenso Input-Objekte sein. Eine Mischung der Objekt-Typen ist laut Spezifikation nicht erlaubt. Ein Input-Objekt unterscheidet sich zu einem Objekt-Typ nur durch das Schlüsselwort *input* anstatt von *type*. Weiters können die Felder eines Input-Objekts keine Argumente beinhalten.

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

```
1 input AuthorCreateInput {
2   firstName: String
3   lastName: String
4   books: [Int!]
5 }
```

Fragementierung

Fragmente helfen dabei den Text von Abfragen zu reduzieren indem sie oft gebrauchte Abfragen von Feldern kapseln. Weiters können Fragmente nur für Objekt-Typen, Interfaces und Union-Typen definiert werden. Fragmente müssen dabei den Objekt-Typen für den sie gelten mit dem Schlüsselwort *on* angeben. Inline-Fragmente müssen nicht mit dem Schlüsselwort *fragment* definiert werden. Sie können mit dem Spread-Operator direkt in der Abfrage definiert werden [Facebook, Abs. 2.8 - 2.8.1]. Fragmente liefern nur dann Daten zurück wenn der Objekt-Typ der sie anfordert auch wirklich dem Typen des Fragments entspricht.
Im folgenden Beispiel wird ein Fragment für den Objekt-Typ Author definiert, weiters wird die Verwendung in einer Query veranschaulicht:

```
1 fragment authorFragment on Author {
2     firstName
3     lastName
4 }
```

Im nachfolgenden Beispiel wird die Nutzung eines Inline-Fragments in einer Query veranschaulicht:

```
1 query{
2     books{
3         title
```

```
4         on Author{
5             firstName
6             lastName
7         }
8     }
9 }
```

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 (siehe **??**) die spezifischen Felder abgefragt werden.

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

Direktive
Direktiven bieten die Möglichkeit die Struktur, des durch eine Query angefragtes Ergebnis zu beeinflussen. Mittels einer Annotation direkt an einem Feld oder einer Objekt-Relation Mit Direktiven ist es möglich mittels Annotationen, direkt an einem Feld oder einem Objekt-Typen, dasselbige entsprechend einer Eingabe zu beeinflussen. GraphQL bietet dafür standardmäßig zwei Arten von Strukturmanipulationen: *@include* und *@skip*, es ist aber auch möglich serverseitig zusätzliche Direktiven zu implementieren. Mittels diesen Direktiven lassen sich Objekt-Typen oder Felder inkludieren oder überspringen. Für die auszuwertende Direktive muss dabei in der Query ein *Boolean* mitübergeben werden.
Im folgenden Beispiel wird eine Direktive gezeigt welche nach Bedarf das Feld *id* ignoriert.

```
1 query {
2     authors($withoutId: Boolean = false){
3         id @include(if: $withoutId)
4         firstName
5         lastName
6     }
7 }
```

3.3 Schema

Im Schema welches mittels der *SDL* definiert wird, werden alle Objekt-Typen des GraphQL-Services definiert. Durch das Schema werden die verwalteten Entitäten durch Objekt-Typen definiert. Weiters werden auch die vom GraphQL zur Verfügung gestellten Interfaces, Unions, Fragments, Direktiven, Enums und Input-Typen definiert. Wenn man die im Schema definierten Objekt-Typen als Baumstruktur betrachtet so sind die referenzierten Objekt-Typen 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 Objekt-Typen. [Kress, 2020, S.60]
Weiters dürfen die im Schema definierte Namen nicht mit doppeltem Unterstrich beginnen. Diese sind für das GraphQL Introspektionsystem reserviert. [Facebook, Abs. 3.3]

3.3.1 Wurzel Operationen

Das Schema definiert die Wurzelknoten der Eingangspunkte der Operationen (Query, Mutation und Subscription) die es unterstützt. Das Schema definiert dadurch den Eingangspunkt dieser Operationen im Typ System. Diese Wurzeloperationen sind spezielle Objekt-Typen. Um ein korrektes Schema zu erstellen muss beachtet werden, dass die Typen und Direktiven eindeutig über ihren Namen identifizierbar sind. Query muss aber zwingend definiert werden. Mutations und Subscriptions sind optional und werden, wenn sie nicht explizit definiert werden, nicht unterstützt. Desweiteren müssen die Objekt-Typen der Wurzel-Operationen unterscheiden und dürfen nicht diesselben sein.

3.3.2 Schema Definition

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

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

type Review {
    userId: Int!
    user: User!
    bookId: Int!
    book: Book!
    rating: Int!
    id: Int!
}

type User {
    firstName: String!
    lastName: String!
    email: String!
    roles: [Role!]!
    reviews: [Review!]!
    id: Int!
}

```

```
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
}

input ReviewUpdateInput {
    id: Int!
    userId: Int!
    bookId: Int!
    rating: Int!
}

input ReviewCreateInput {
    userId: Int!
    bookId: Int!
    rating: Int!
}

input LoginDataInput {
    email: String!
    password: String!
}

input UserInput {
    firstName: String!
    lastName: String!
    email: String!
    password: String!
    id: Int!
}

type Query {
    books(where: BookFilterInput, order: [BookSortInput!]): [Book!]!
```

```
    book(where: BookFilterInput, order: [BookSortInput!]): Book
    authors(where: AuthorFilterInput, order: [AuthorSortInput!]): [Author!]!
    author(where: AuthorFilterInput, order: [AuthorSortInput!]): Author
    reviews(where: ReviewFilterInput, order: [ReviewSortInput!]): [Review!]!
    author(where: ReviewFilterInput, order: [ReviewSortInput!]): Review
}

type Mutation {
    register(input: UserInput): User
    login(input: LoginDataInput): String
    createBook(input: BookCreateInput!): Book!
    updateBook(input: BookUpdateInput!): Book!
    createAuthor(input: AuthorCreateInput!): Author!
    updateAuthor(input: AuthorUpdateInput!): Author!
    createReview(input: ReviewCreateInput!): Review!
    updateReview(input: ReviewUpdateInput!): Review!
}

type Subscription {
    bookAdded: Book
}
```

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

3.3.3 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 angeführten 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. [Kress, 2020, S.62]

3.3.4 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. Variablen 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. [Facebook, Abs. 5.8]

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

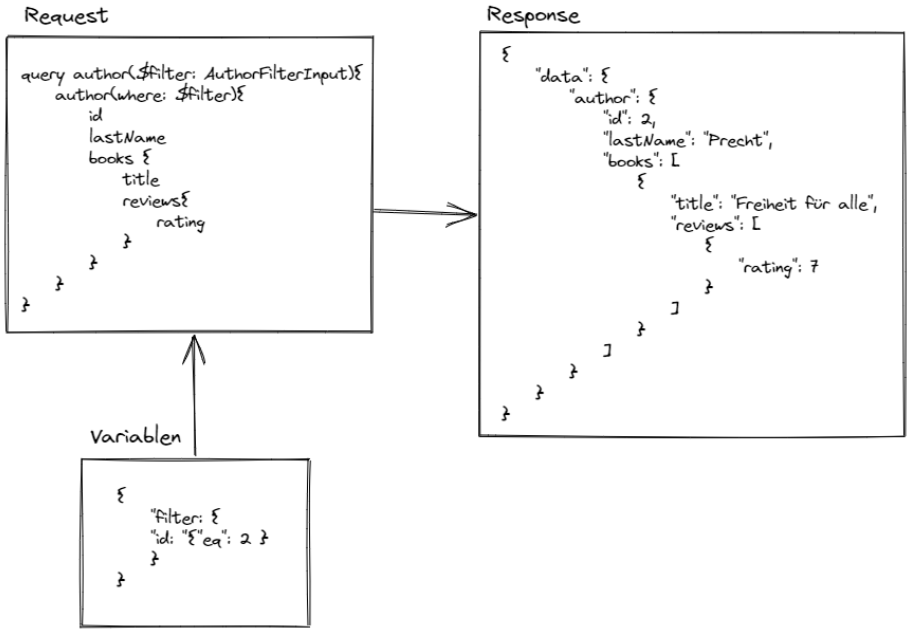

Figure 3.1: Anfrage aller Bücher mit zugehörigen Autoren unter Verwendung einer Variable.

3.4 Zugriff auf den GraphQL-Service

GraphQL liefert keine Spezifikation der 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.

GET und POST-Anfrage unterscheiden sich 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 jede Query,

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 vorgeben. Somit können komplexe Querys nicht funktional sicher über GET-Anfragen abgebildet werden.

Aus diesem Grund werden üblicherweise alle Operationen auf POST-Anfragen abgebildet. 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 Queries 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.

3.5 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 [Kress, 2020, S.40-41].

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.

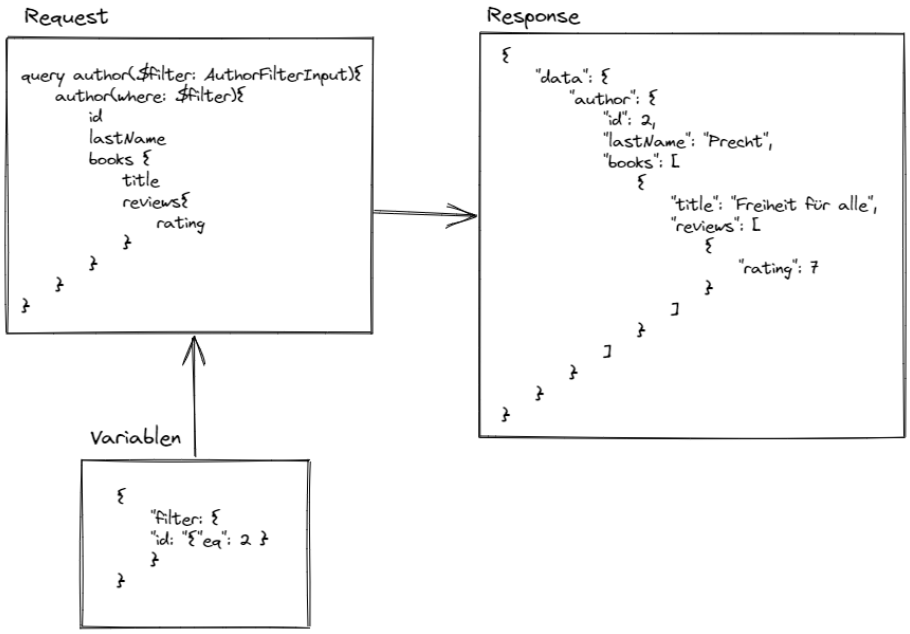

Figure 3.2: Anfrage eines Buches mit ID=2.

Die in der obigen Abbildung ersichtliche Query fragt den Autor mit der ID=2 vom GraphQL-Service ab. Dabei sollen die Bücher die der Autor geschrieben hat und mit dem Buch referenzierte Reviews ebenfalls zurückgegeben werden. Um die in der obigen Abbildung angeführte Query ausführen zu können sind folgende Definitionen im Schema erforderlich:

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

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

type Review {
    userId: Int!
    user: User!
    bookId: Int!
    book: Book!
    rating: Int!
    id: Int!
}
```

```
23
24 input AuthorFilterInput {
25     and: [AuthorFilterInput!]
26     or: [AuthorFilterInput!]
27     firstName: StringOperationFilterInput
28     lastName: StringOperationFilterInput
29     books: ListFilterInputTypeOfBookFilterInput
30     id: ComparableInt32OperationFilterInput
31 }
32
33 type Query {
34     author(where: AuthorFilterInput): Author
35 }
```

Aus dem Schema lassen sich folgende Entitäten herauslesen: *Book*, *Author* und *Review*. Weiters wurde ein *AuthorFilterInput* deklariert, damit serverseitig nach einem bestimmten Buch sortieren werden kann. Die Wurzeloperation Query enthält den Eintrittspunkt *author*. Dieser kann einen Filter entgegennehmen und liefert nach einer erfolgreichen Anfrage einen *Author*, wobei dieser auch *null* sein kann.
Die Anfrage die dem Server für das einholen der Daten geschickt wird enthält dabei folgenden Request-Body:

```
 1 {
 2     "operationName": "author",
 3     "query": "query author($filter: AuthorFilterInput){
 4         authors(where: $filter){
 5             id
 6             lastName
 7             books {
 8                 title
 9                 reviews{
10                     rating
11                 }
12             }
13         }
14     }",
15     "variables": {
16         "filter": {
17             "id": {
18                 "eq": 2
19             }
20         }
21     }
22 }
```

Der Request-Body ist dabei wie bereits erwähnt in 3 Teile unterteilt: *operationName*, *query*, *variables*. Diese Teile werden dann durch den GraphQL-Service zusammengefügt und nach erfolgreicher Validierung an die Resolver zur Evaluierung und Rückgabe der Daten weitergereicht. Betrachtet man die Entitäten und ihre Referenzen als Baumstruktur, so ist bei der Evaluierung dieser Query der Autor die Wurzel, das Buch das Geäst und die Bewertung das Blatt des Baumes. Die Resolver schreiten somit den Baum der angeforderten Entitäten bis zu den Blättern durch und geben dabei die angeforderten Daten an den Client zurück.

3.6 Mutationen

APIs benötigen neben dem lesenden Zugriff auf Daten auch einen schreibenden. Dieser wird in GraphQL mittels Mutationen umgesetzt. Mutationen kapseln die Implementierungen der Geschäftslogik 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. [Kress, 2020, S. 54]
In der folgenden Abbildung ist eine Mutation zum hinzufügen einer Bewertung abgebildet:

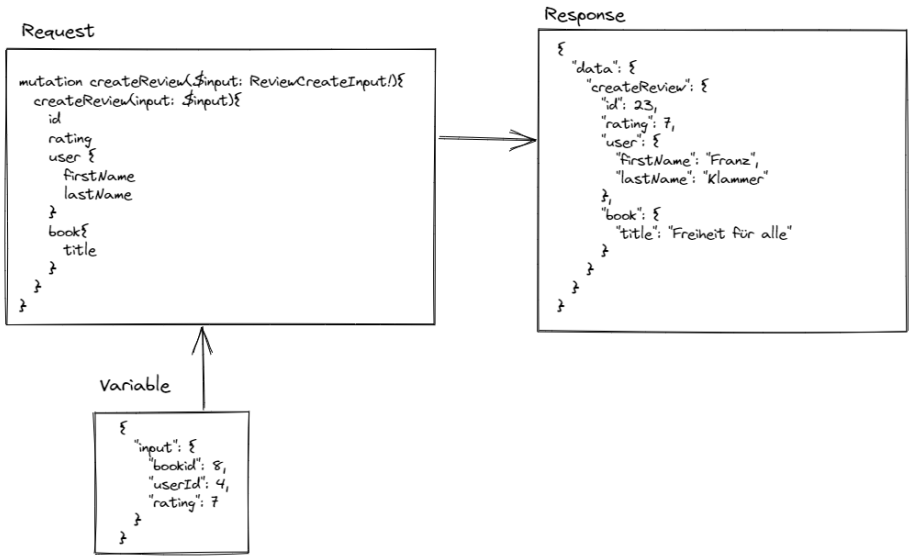

Figure 3.3: Anfrage eines Buches mit ID=2.

Um eine Bewertung zu erstellen wird ein Input-Objekt *ReviewCreateInput* und eine Mutation *createReview* welches ein solches Objekt entgegennimmt benötigt.

```
input ReviewCreateInput {
    userId: Int!
    bookId: Int!
    rating: Int!
}

type mutation {
    createReview(input: ReviewCreateInput!): Review!
}
```

```
{
    "operationName": "createReview",
    "query": "mutation createReview($input: ReviewCreateInput!){
        createReview(input: $input){
```

```
5             id
6             rating
7             user {
8                 firstName
9                 lastName
10            }
11            book{
12                title
13            }
14        }
15    }",
16    "variables": {
17        "input": {
18        "bookId": 8,
19        "userId": 4,
20        "rating": 7
21        }
22    }
23 }
```

Der oben angegebene Request-Body der Anfrage ist wie bei der Query wieder in die drei Teile *operationName*, *query*, *variables* aufgeteilt. Diese werden wieder durch den GraphQL-Service an den verantwortlichen Resolver weitergeleitet.

3.7 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.

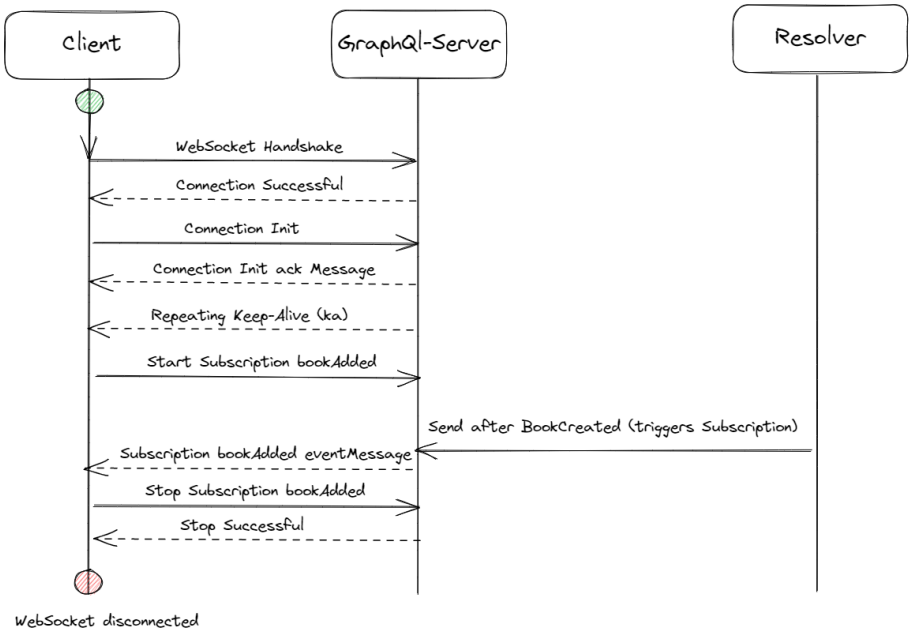

Figure 3.4: Subscription die auf das Erstellen eines Buches wartet.

In der obigen Abbildung
Im folgenden Codebeispiel ist die Definition einer Subscription im Schema ersichtlich.

```
1 type Subscription {
2     bookAdded: Book!
3 }
```

3.8 Bekannte Probleme

Chapter 4

Entwicklung

In diesem Kapitel wird die Entwicklung des Prototypen beschrieben. Der Prototyp ist ein GraphQL-Service welcher mit .NET 6 und der Zuhilfenahme der Bibilothek HotChocolate entwickelt wurde. Desweiteren wird auf die von HotChocolate bereitgestellten Entwicklerhilfsmittel eingegangen.

4.1 Anwendungsszenario

Die Auswahl eines neuen Buches ist oftmals ein sehr schwieriges Unterfangen. Bei der Entscheidungsfindung helfen oft Bewertungen von Lesern die das jeweilige Buch schon gelesen haben. Demnach soll eine Bücher-Bewertungsplattform geschaffen werden. Diese Plattform ermöglicht es Benutzern Bücher zu bewerten und Informationen über Bücher, Autoren oder Bewertungen einzusehen. Weiters sollen Benutzer in der Lage sein sich im System zu registrieren und anzumelden. Angemeldete Benutzer haben, je nach ihren Benutzerrollen, Möglichkeiten im System verwaltete Entitäten zu Erstellen, zu Bearbeiten oder zu Löschen.

4.2 Datenbankschema

Das in der folgenden Abbildung dargestellte ER-Diagram beschreibt die dem Prototypen zu Grunde liegende Datenbank.

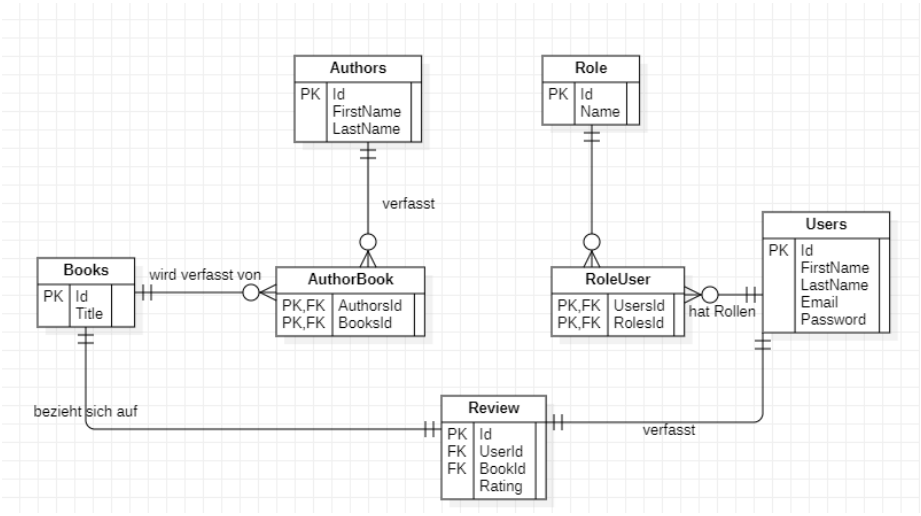

Figure 4.1: Datenbankschema

4.3 Architektur

Der Prototyp wurde mit der für REST-APIs üblichen Drei-Schicht-Architektur umgesetzt. Die Geschäftslogikschicht und Datenbankzugriffsschicht wurden dabei so entwickelt, dass sie eigentlichen Implementierungen je nach Bedarf einfach ausgetauscht werden können. Auf die Geschäftslogik wird in den Unterkapiteln Query, Mutation und Subscription näher eingegangen. Die Datenbankzugriffsschicht wird im Kapitel Entity Framework näher erläutert.

API: Die API die einzige Schnittstelle des Systems zur Außenwelt.

Geschäftslogik: Die Geschäftslogik bildet dabei die Logik der Server-Applikation ab.

Datenbankzugriff: Die darunterliegende Datenbankzugriffsschicht ermöglicht, den darüberliegenden Schichten, CRUD-Operationen auf die angeforderten Daten auszuführen.

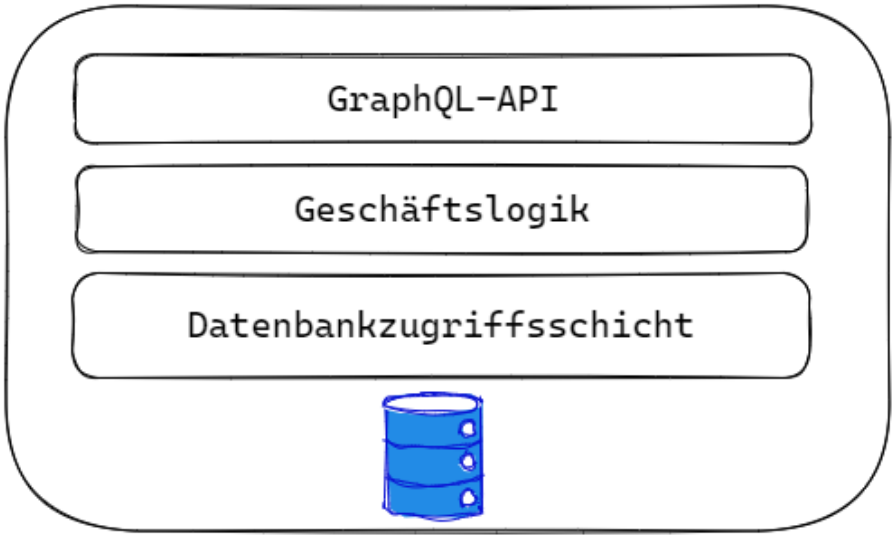

Figure 4.2: Drei-Schicht-Architektur

4.4 Hot Chocolate

Der GraphQL-Service wurde mit dem Framework *Hot Chocolate* umgesetzt. Bei Hot Chocolate handelt es sich um einen Open Source GraphQL-Server welche

4.4.1 Entwurf Schema

4.5 Entity Framework

Die Datenbankzugriffsschicht wurde mit dem *Entity Framework* umgesetzt. Das Entity Framework ist ein *Object Relational Mapper*, es ermöglicht Entwicklern den Fokus auf eine höhere Abstraktionsebene zu legen.

4.6 Querys

4.6.1 Pagination und Sorting

4.7 Mutations

4.8 Subscriptions

4.9 Authentifizierung und Autorisierung

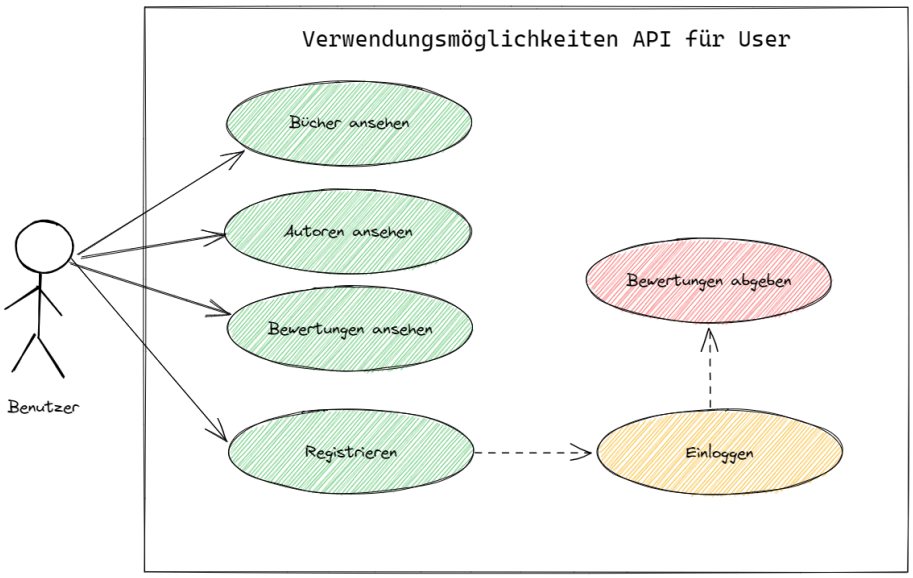

Figure 4.3: Rechte User.

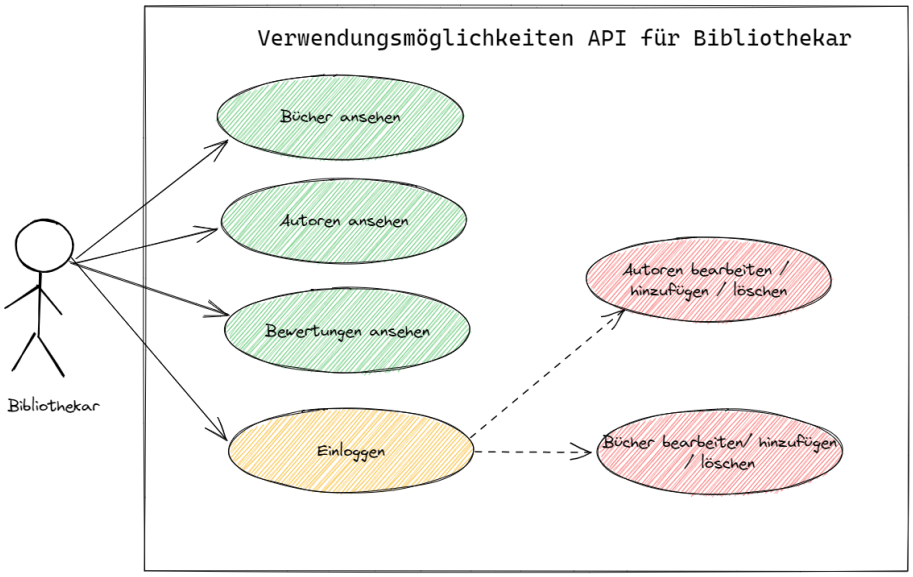

Figure 4.4: Rechte Bibliothekar.

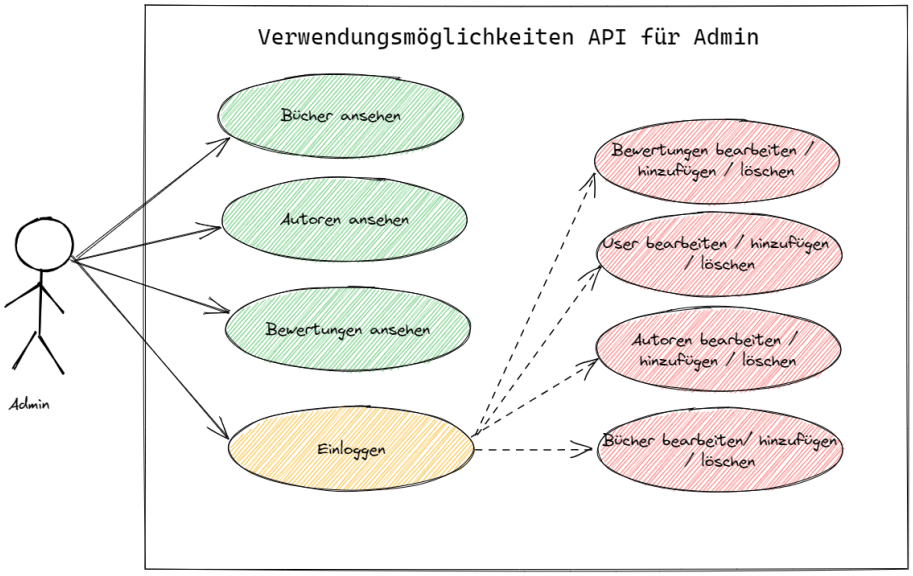

Figure 4.5: Rechte Admin.

4.10 Entwicklertools

4.10.1 GraphiQL

4.11 1 + n Problem

4.12 Fehlermanagement

4.13 Pagination

4.14 Caching

Chapter 5

Conclusio

5.1 Fazit

5.2 Ausblick

Bibliography

[Berlind 2017] Berlind, David: *APIs Are Like User Interfaces–Just With Different Users In Mind*. 2017

[Cotton und Greatorex Jr 1968] Cotton, Ira W. ; Greatorex Jr, Frank S.: 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

[Facebook] Facebook: *Graphql*. – URL `http://spec.graphql.org/June2018/`. – Zugriffsdatum: 2021-11-15

[Kress 2020] Kress, Dominik: *GraphQL: Eine Einführung in APIs mit GraphQL*. dpunkt. verlag, 2020

[Rakuten] Rakuten: *Graphql vs Rest*. – URL `https://blog.api.rakuten.net/graphql-vs-rest/`. – Zugriffsdatum: 2021-11-15

[Sakib] Sakib, Abu: *Graphql vs Rest*. – URL `https://dgraph.io/blog/post/graphql-rest/`. – Zugriffsdatum: 2021-11-15

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