

Contents

1	Einleitung	3
1.1	Motivation	3
1.2	Zielsetzung	4
2	Grundlagen	5
2.1	API	5
2.2	REST-API	6
2.2.1	REST-Bedingungen	7
2.2.2	Kommunikation mit dem Server	8
2.3	GraphQL	8
2.3.1	Entwurfsprinzipien	9
2.4	GraphQL vs. REST	9
2.4.1	Endpunkte	9
2.4.2	Over-fetching und Under-fetching	10
2.4.3	Was ist für welches Szenario besser geeignet?	10
3	GraphQL	11
3.1	Typ-System	11
3.2	Schema-Definitions-Sprache SDL	11
3.3	Schema	15
3.3.1	Wurzel Operationen	16
3.3.2	Schema Definition	16
3.3.3	Parameter	18
3.3.4	Variablen	18
3.4	Zugriff auf den GraphQL-Service	19
3.5	Querys	20
3.6	Mutationen	23
3.7	Subscriptions	24
3.8	Bekannte Probleme	25
4	Entwicklung GraphQL-Service in .NET mit HotChocolate	26
4.1	Anwendungsszenario	26
4.2	Planung Schnittstelle	26
4.3	HotChocolate	28

4.4	Architektur	29
4.4.1	API:	30
4.4.2	Geschäftslogik	30
4.4.3	Datenbankzugriff	30
4.5	Resolver	31
4.6	Field Middleware	32
4.7	Querys	33
4.8	Mutations	38
4.9	Subscriptions	41
4.10	Authentifizierung und Autorisierung	43
4.11	1 + n Problem	48
5	Conclusio	49
5.1	Fazit	49
5.2	Ausblick	49

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 Cambridge, in einem Leitfaden verfasst. Dieser Leitfaden beschreibt das Extrahieren einer Sub-Routinen-Bibliothek mit einheitlichem dokumentierten Zugriff. Wheeler [1952] Ira Cotton und Frank Grestorex erwähnten den Begriff erstmalig auf der Fall Joint Computer Con. Cotton und Grestorex 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 integriert 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 Standardkommunikationsprotokoll 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 Komponente zu haben
3. **Zustandslosigkeit (Stateless):** Zustandslosigkeit bedeutet, dass der Client alle Informationen die der Server benötigt mitsendet. Das resultiert darin, dass der Server keine Overhead Daten des Clients speichern muss um zukünftige 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 Abhängigkeiten 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, wie lange dieser Cache gültig ist. Nach Ablauf der Gültigkeit werden die Daten erneut vom Server geladen. Durch das Zwischenspeichern von Daten wird die Performance 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 realisiert. 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 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 dieselbe 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 benötigten 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

```
1    id: Int!  
2    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].

```
1 enum Category {  
2     Fantasy  
3     Adventure  
4     Mystery  
5     Thriller  
6     Romance  
7 }
```

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 liefern. 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].

```
1 type Book {
2   id: Int!
3   title: String
4   authors: [Author]
5 }
6
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.

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:

```
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 }

```

Jedes Interface benötigt eine spezifische Implementierung, im angeführten 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 realisiert:

```

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

```

Fragmentierung

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 dieselben 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 Introspektionssystem 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

```
1 type Book {
2   id: Int!
3   title: String
4   authors: [Author]
5 }
6
7 type Author {
8   id: Int!
9   firstName: String
10  lastName: String
11  books: [Book]
12 }
13
14 type Review {
15   userId: Int!
16   user: User!
17   bookId: Int!
18   book: Book!
19   rating: Int!
20   id: Int!
21 }
22
23 type User {
24   firstName: String!
25   lastName: String!
26   email: String!
27   roles: [Role!]!
28   reviews: [Review!]!
29   id: Int!
30 }
31
```



```

32 input AuthorUpdateInput {
33     id: Int!
34     firstName: String
35     lastName: String
36     books: [Int!]
37 }
38
39 input AuthorCreateInput {
40     firstName: String
41     lastName: String
42     books: [Int!]
43 }
44
45 input BookUpdateInput {
46     id: Int!
47     title: String
48     authors: [Int!]
49 }
50
51 input BookCreateInput {
52     title: String
53     authors: [Int!]
54 }
55
56 input LoginInput {
57     email: String
58     password: String
59 }
60
61 input ReviewUpdateInput {
62     id: Int!
63     userId: Int!
64     bookId: Int!
65     rating: Int!
66 }
67
68 input ReviewCreateInput {
69     userId: Int!
70     bookId: Int!
71     rating: Int!
72 }
73
74 input LoginDataInput {
75     email: String!
76     password: String!
77 }
78
79 input UserInput {
80     firstName: String!
81     lastName: String!
82     email: String!
83     password: String!
84     id: Int!
85 }
86
87 type Query {
88     books(where: BookFilterInput, order: [BookSortInput!]): [Book!]!

```

```

89     book(where: BookFilterInput, order: [BookSortInput!]): Book
90     authors(where: AuthorFilterInput, order: [AuthorSortInput!]): [Author!]!
91     author(where: AuthorFilterInput, order: [AuthorSortInput!]): Author
92     reviews(where: ReviewFilterInput, order: [ReviewSortInput!]): [Review!]!
93     author(where: ReviewFilterInput, order: [ReviewSortInput!]): Review
94 }
95
96 type Mutation {
97     register(input: UserInput): User
98     login(input: LoginDataInput): String
99     createBook(input: BookCreateInput!): Book!
100    updateBook(input: BookUpdateInput!): Book!
101    createAuthor(input: AuthorCreateInput!): Author!
102    updateAuthor(input: AuthorUpdateInput!): Author!
103    createReview(input: ReviewCreateInput!): Review!
104    updateReview(input: ReviewUpdateInput!): Review!
105 }
106
107 type Subscription {
108     bookAdded: Book
109 }

```

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.

```

1  enum Currency {
2      EUR
3      USD
4      GBP
5  }
6
7  type Book {
8      id: Int!
9      title: String
10     authors: [Author]
11     price: (unit: Currency = EUR): Float
12 }

```

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.

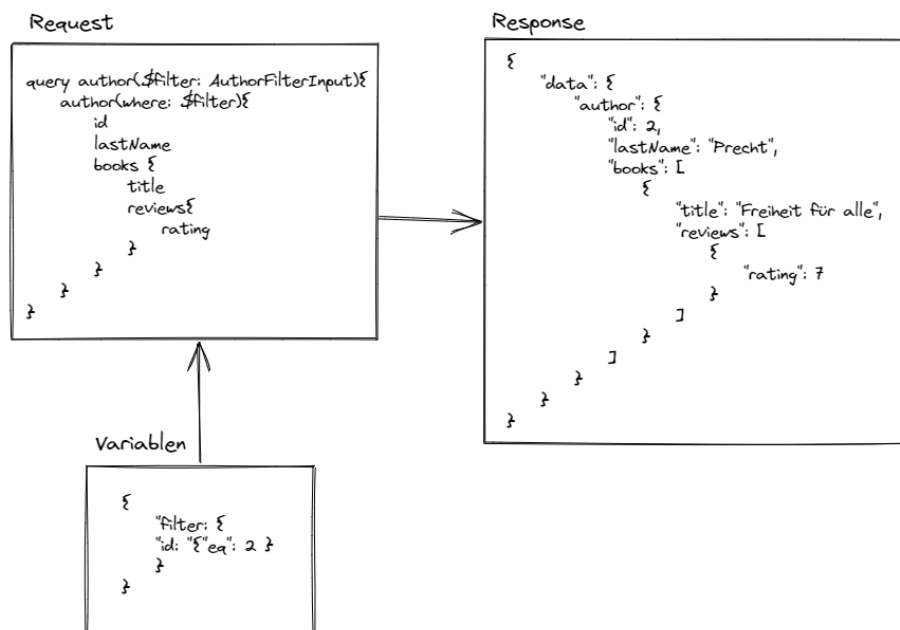


Figure 3.1: Anfrage aller Bücher mit zugehörigen Autoren unter Verwendung einer Variable.

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 Benennung 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.

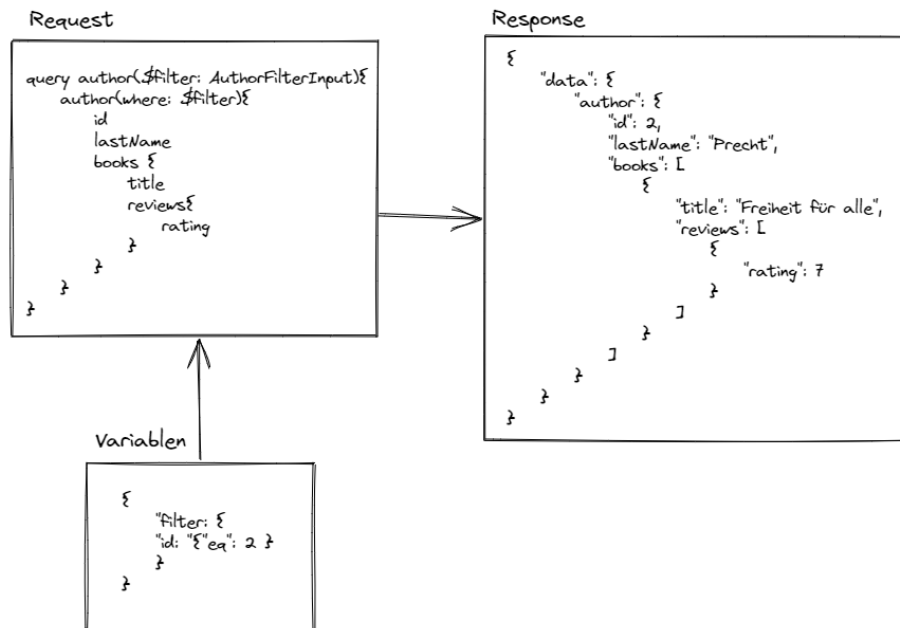


Figure 3.2: Anfrage eines Buches mit ID=2.

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:

```

1 type Author {
2   firstName: String!
3   lastName: String!
4   books: [Book!]!
5   id: Int!
6 }
7
8 type Book {
9   title: String!
10  authors: [Author!]!
11  reviews: [Review!]!
12  id: Int!
13 }
14
15 type Review {
16   userId: Int!
17   user: User!
18   bookId: Int!
19   book: Book!
20   rating: Int!
21   id: Int!
22 }

```

```

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:

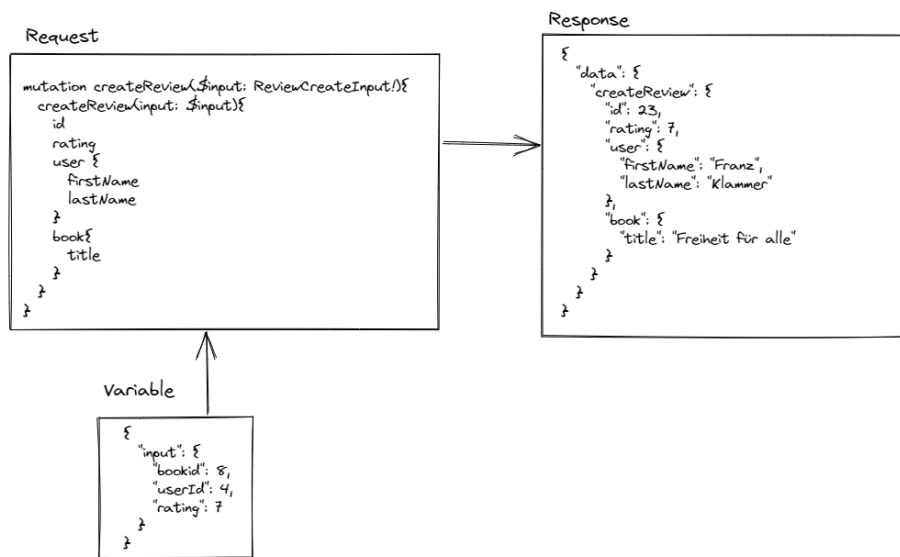


Figure 3.3: Anfrage eines Buches mit ID=2.

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

```
1 input ReviewCreateInput {
2   userId: Int!
3   bookId: Int!
4   rating: Int!
5 }
6
7 type mutation {
8   createReview(input: ReviewCreateInput!): Review!
9 }
```

```
1 {
2   "operationName": "createReview",
3   "query": "mutation createReview($input: ReviewCreateInput!){
4     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 verfügbar ist.

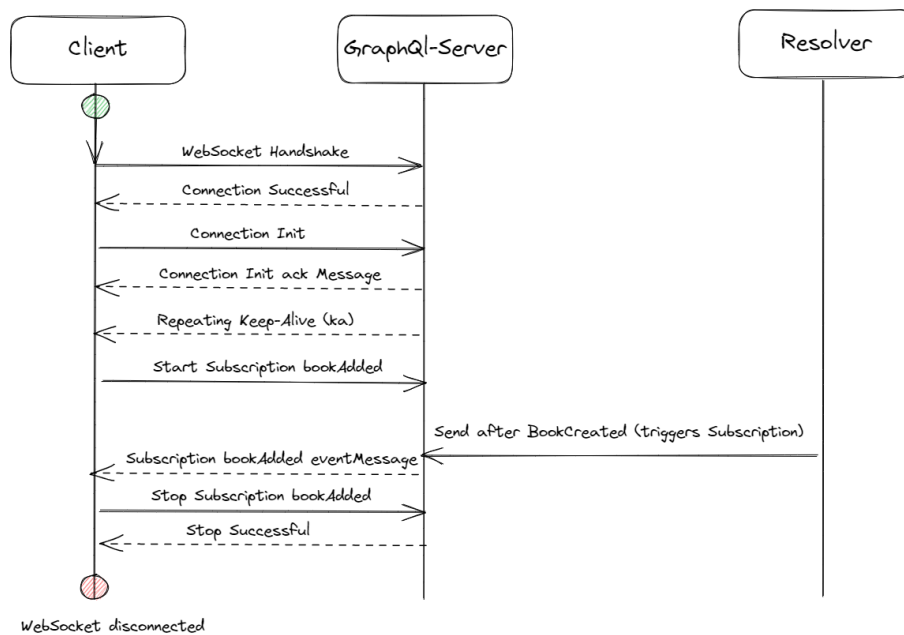


Figure 3.4: Subscription die auf das Erstellen eines Buches wartet.

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 GraphQL-Service in .NET mit HotChocolate

In diesem Kapitel wird die Entwicklung des Prototypen beschrieben. Der Prototyp ist ein GraphQL-Service welcher mit .NET 6 und der Zuhilfenahme der Bibliothek HotChocolate entwickelt wurde.

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 Planung Schnittstelle

Bevor ein Entwickler sich um die technische Umsetzung einer API kümmert, ist es ratsam, dass er sich einen Gesamtüberblick über das zu entwerfende System verschafft. Dafür werden vor Beginn der technischen Umsetzung, unter Zuhilfenahme des Anwendungsszenarios, Use-Cases definiert. Diese Use-Cases beschreiben die benötigten Zugriffe eines Clients auf das System. Zudem werden mit der Definition der Zugriffe auch die benötigten Entitäten definiert. Aus dem

Anwendungsszenario abgeleitete Use-Cases werden in der folgenden Abbildung grafisch dargestellt:

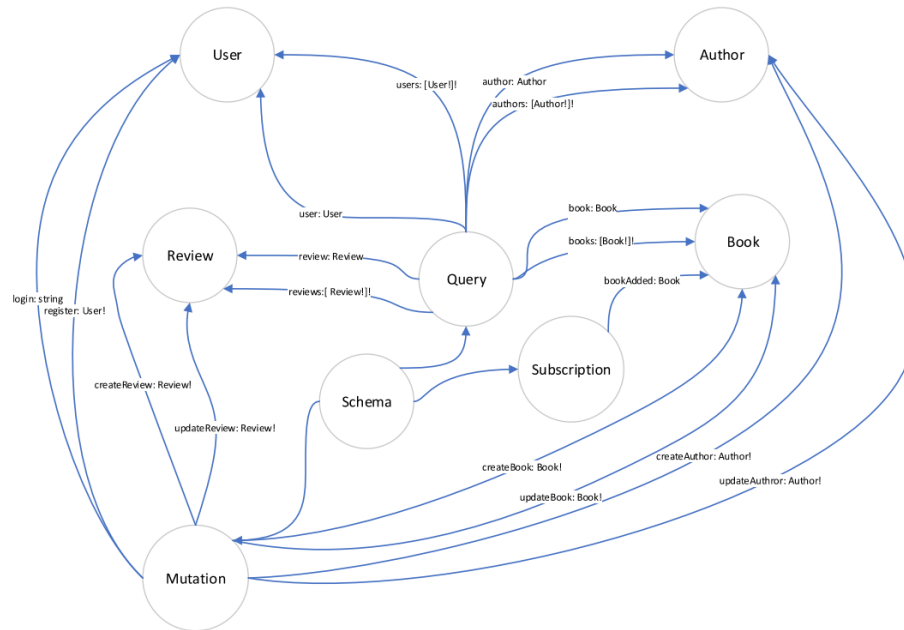


Figure 4.1: Use-Cases des Service.

Die obige Abbildung beschreibt die Zugriffsmöglichkeiten eines Clients auf den Prototypen. Mit der Definition der Use-Cases geht auch die Definition der zu verwaltenden Entitäten einher. Die zu verwaltenden Entitäten und ihre Relationen zueinander sind im folgenden ER-Diagramm dargestellt:

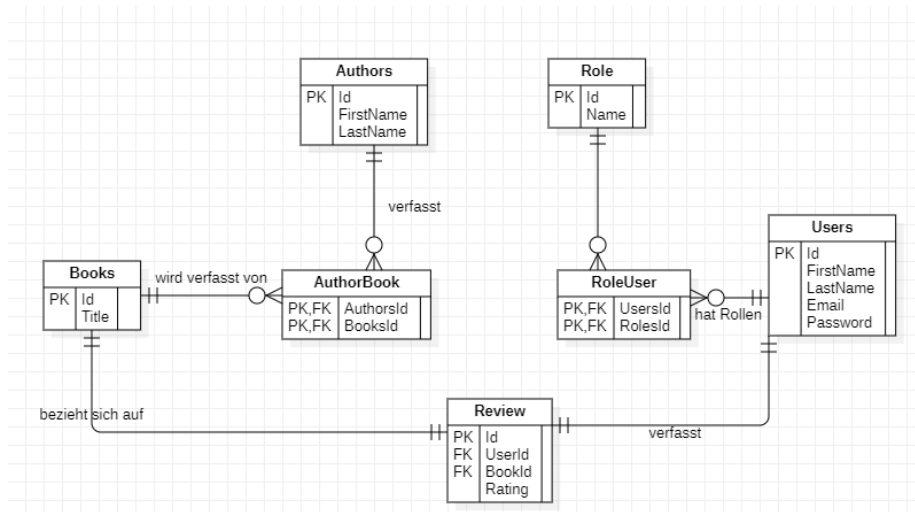


Figure 4.2: Datenbankschema

4.3 HotChocolate

Für die Implementierung des GraphQL-Service wurde das Framework HotChocolate herangezogen. Zusätzlich zum HotChocolate Framework gibt es noch das .NET GraphQL Framework. Die Entscheidung, HotChocolate zu verwenden, fiel aufgrund der besseren Dokumentation des Frameworks und der höheren Aktivität der Community.

HotChocolate Allgemein

HotChocolate ist ein Open-Source Framework zur Implementierung eines GraphQL-Services. Es ist konform, laut den GraphQL-Spezifikationen implementiert. Damit ist HotChocolate kompatibel mit allen anderen GraphQL konform umgesetzten Clients. Ein großer Teil der Komplexität, einen GraphQL-Service zu entwickeln, fällt dabei auf die Entwicklung des Schemas zurück. HotChocolate kümmert sich um die Generierung des Schemas zur Laufzeit und entfernt somit einen großen Teil dieser Komplexität vom Entwickler.

Schema Erstellung in HotChocolate

Um das Schema in HotChocolate zu definieren gibt es 3 Varianten: *Pure-Code-First*, *Code-First* und *Schema-First*. Schema-First übernimmt dabei ein bereits bestehendes Schema und fügt es dem Service zu. Code-First verwendet *Attributes* um das Schema zu definieren. Pure-Code-First verwendet eine Fluent-API. Alle 3 Varianten liefern das selbe Schema, wobei man mit Pure-Code-First am meisten Kontrolle bei der Konfiguration hat. Der Prototyp wurde mittels der Pure-Code-First Vorgehensweise umgesetzt.

Dependency Injection

Dependency-Injection in HotChocolate funktioniert sehr ähnlich wie die Dependency Injection von ASP.NET Applikationen. Die Services werden wie üblich dem Dependency-Injection-Container hinzugefügt:

```
1 builder.Services.AddTransient(typeof(IRepository<>), typeof(Repository<>));
2 builder.Services.AddTransient(typeof(IBookRepository), typeof(BookRepository)
   );
3 builder.Services.AddTransient(typeof(IAuthorRepository), typeof(
   AuthorRepository));
4 builder.Services.AddTransient(typeof(IReviewRepository), typeof(
   ReviewRepository));
5 builder.Services.AddTransient(typeof(IBaseService<>), typeof(BaseService<>));
6 builder.Services.AddTransient<IBookService, BookService>();
7 builder.Services.AddTransient<IAuthorService, AuthorService>();
8 builder.Services.AddTransient<IReviewService, ReviewService>();
9 builder.Services.AddTransient<IAuthService, AuthService>();
```

HotChocolate unterstützt dabei aber nicht die Constructor-Injection. Stattdessen wird die Dependency-Injection in HotChocolate mittels Method-Dependency-Injection umgesetzt. Würde man die Dependency-Injection mittels Constructor-Injection implementieren, so wäre jeder Service automatisch ein Singleton. Für die Method-Injection liefert HotChocolate das Attribut *[Service]*.

Im folgenden Code-Beispiel ist die Verwendung ersichtlich:

```
1 public async Task<IQueryable<Book>> Books([Service] IBookService bookService)
   {
2     //Execute Logic here
3 }
```

4.4 Architektur

Der Prototyp wurde mit der für REST-APIs üblichen Drei-Schicht-Architektur umgesetzt. Die Geschäftslogikschicht und Datenbankzugriffsschicht wurden dabei so entwickelt, dass die eigentlichen Implementierungen, je nach Bedarf, einfach ausgetauscht werden können.

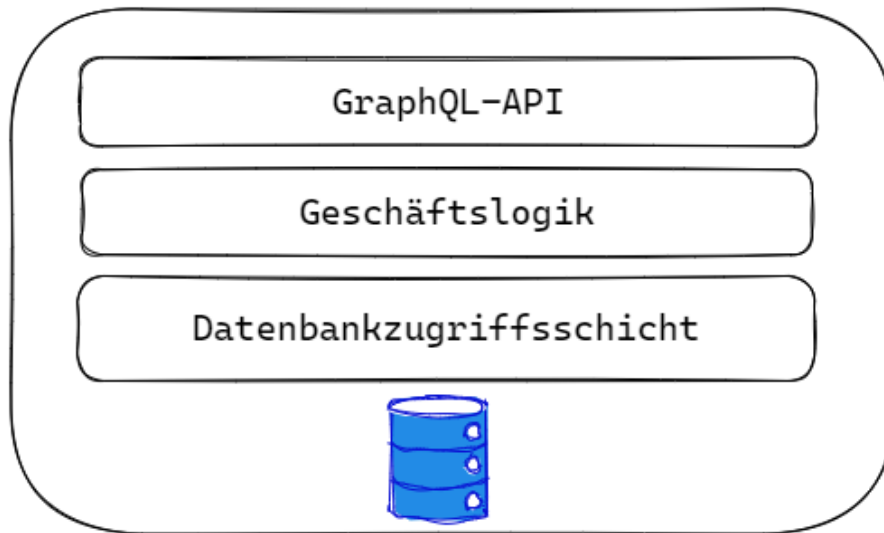


Figure 4.3: Drei-Schicht-Architektur

Die hier abgebildete Architektur dient zur Veranschaulichung, die gezeigten Schichten werden in den folgenden Abschnitten näher erläutert.

4.4.1 API:

Die API bildet die einzige Schnittstelle des Systems zur Außenwelt. Sie wird mittels dem GraphQL-Schema abgebildet. Dieses Schema definiert, wie im GraphQL-Kapitel bereits erwähnt, die verfügbaren Typen sowie die lesenden und schreibenden Zugriffe des GraphQL-Service.

4.4.2 Geschäftslogik

Die Geschäftslogik bildet dabei die Logik der Server-Applikation ab. Diese Logik wird dem GraphQL-Schema mittels Resolvem zur Verfügung gestellt. Genauer zu den Resolvem und zur Geschäftslogik folgt im Unterkapitel Resolver. Die Geschäftslogik wurde mit dem DAO-Pattern umgesetzt und besteht aus *Service* Klassen und zugehörigen Interfaces.

4.4.3 Datenbankzugriff

Die darunterliegende Datenbankzugriffsschicht ermöglicht, den darüberliegenden Schichten, CRUD-Operationen auf die angeforderten Daten auszuführen. Für die Datenbankzugriffsschicht wurde dabei das Entity Framework verwendet. Das Entity Framework ist ein *Object Relational Mapper*, es ermöglicht Entwicklern den Fokus auf eine höhere Abstraktionsebene zu legen. Diese Schicht

wurde mit dem Repository-Pattern umgesetzt. Wobei ein generisches Repository die Grundfunktionalität für die anderen entitätsspezifischen Repositories zur Verfügung stellt.

4.5 Resolver

Das Schema beschreibt, wie bereits im GraphQL-Kapitel erwähnt, nur die verfügbaren Typen, Querys, Mutations und Subscriptions. Über die Generierung der Daten, als auch über die Manipulation dieser, hat das Schema kein Wissen. Für den Datenzugriff bzw. Datenmanipulation sind in GraphQL *Resolver* verantwortlich. Jedes Feld in einer Query ist nichts anderes als eine Methode welches den Wert dieses Typs retourniert. Jedes Feld eines Typs wird dabei einem *Resolver* zugewiesen, diese Resolver sind Zugriffe auf die Geschäftslogik. Wird ein Feld durch eine Query angefordert, liefert der jeweilige *Resolver* die angeforderten Daten zurück.

Umsetzung Resolver mittels Geschäftslogik

Resolver werden im Prototypen durch die Geschäftslogik abgebildet, diese bieten CRUD-Operationen für die jeweilige Entität. Jede Entität verfügt dabei über einen *Service*. Jeder Service wie zum Beispiel der *AuthService*, leitet dabei von einem generisch implementierten *BaseService* ab. Dieser *BaseService* implementiert dabei das Interface *BaseService*.

```
1 public interface IBaseService<TEntity> where TEntity: BaseEntity {
2     public Task<IQueryable<TEntity>> GetAsync(Expression<Func<TEntity, bool>>
        filter = null, params Expression<Func<TEntity, object>>[] includes);
3     public Task<TEntity> GetFirstAsync(Expression<Func<TEntity, bool>> filter
        = null, params Expression<Func<TEntity, object>>[] includes);
4     public Task<TEntity> AddAsync(TEntity entity);
5     public Task<TEntity> UpdateAsync(TEntity entity);
6     public Task<bool> ExistsAsync(int id);
7     public Task RemoveAsync(TEntity entity);
8 }
```

Im obigen CodeBeispiel ist das generische Interface *IBaseService* abgebildet. Es bietet Schnittstellen für die CRUD-Operationen jeder Entität.

```
1 public class BaseService<TEntity> : IBaseService<TEntity> where TEntity :
    BaseEntity {
2     protected readonly IRepository<TEntity> repository;
3
4     public BaseService(IRepository<TEntity> repository) {
5         this.repository = repository;
6     }
7     public virtual async Task<IQueryable<TEntity>> GetAsync(Expression<Func<
        TEntity, bool>> filter = null, params Expression<Func<TEntity, object
        >>[] includes) {
8         return await repository.GetAsync(filter, includes);
9     }
10 }
```

```

11     public virtual async Task<TEntity> GetFirstAsync(Expression<Func<TEntity,
        bool>> filter = null, params Expression<Func<TEntity, object>>[]
        includes) {
12         return await repository.GetFirstAsync(filter, includes);
13     }
14
15     public virtual Task<TEntity> AddAsync(TEntity entity) {
16         return repository.AddAsync(entity);
17     }
18     public virtual async Task<TEntity> UpdateAsync(TEntity entity) {
19         return await repository.UpdateAsync(entity);
20     }
21
22     public virtual async Task<bool> ExistsAsync(int id) {
23         return await repository.ExistsAsync(id);
24     }
25
26     public virtual async Task RemoveAsync(TEntity entity) {
27         await repository.RemoveAsync(entity);
28     }
29 }

```

Im obigen Code-Beispiel ist die Basisimplementierung jedes Service zu sehen. Besonders relevant für das Zusammenspiel mit HotChocolate ist dabei, dass die lesenden Operationen ein *IQueryable* zurückliefern. Warum *IQueryable* so wichtig ist für HotChocolate wird im Abschnitt *Field Middleware* näher erläutert.

Zusammenfassend lassen sich Resolver wie folgt zusammenfassen: Sie bieten den angefragten Feldern die benötigte Logik der Geschäftslogik um eben jene abgefragten Felder bereitzustellen.

4.6 Field Middleware

Field-Middleware erlaubt es, wiederverwendbare Logik vor oder nach der Exekution des Resolvers auszuführen. Funktionalitäten wie Authentifizierung / Autorisierung, Sortieren, Filtern, Pagination und Projection werden dabei von HotChocolate als Middlewarekomponenten zur Verfügung gestellt. Field-Middleware ist dabei *composable*, somit kann man beliebig viele Middleware Komponenten aneinanderreihen. Sie sind eine der fundamentalen Komponenten des Frameworks.

Reihenfolge Exekution Middleware

Middlewarekomponenten werden in der Reihenfolge, in der sie definiert worden sind, ausgeführt. Jede Middleware-Schicht kennt dabei nur die jeweils nächste Middleware. Die letzte auszuführende Middleware ist dabei der eigentliche Resolver. Die Reihenfolge der Anordnung der Middlewarekomponenten ist dabei sehr entscheidend, denn die Komponenten werden in genau dieser Reihenfolge exekutiert. Die folgende Abbildung beschreibt die Aneinanderreihung der Middlewarekomponenten:

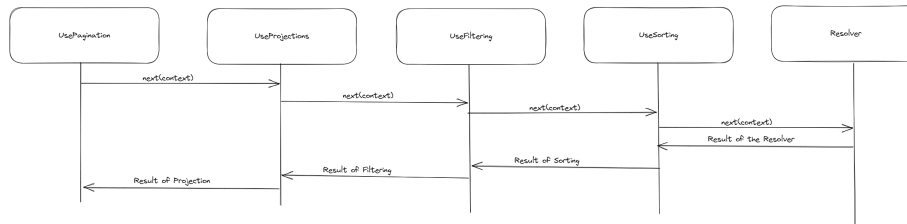


Figure 4.4: Exekutierungsreihenfolge Middleware

In der obigen Abbildung ist zu sehen, dass die Middleware in der Reihenfolge ausgeführt wird, in der sie definiert wurde. Das Resultat, welches von der letzten Middleware (dem Resolver) generiert wurde, wird in der umgekehrten Reihenfolge zurückgereicht. Dabei wird das vom Resolver zurückgelieferte *IQueryable* um die jeweiligen Operationen der restlichen Middlewarekomponenten erweitert.

Im Unterkapitel Query werden die verwendeten Middlewarekomponenten des Prototypen genauer erläutert.

Zusammenführung Middleware

Nachdem jene Middleware, welche als erstes definiert wurde, das Ergebnis der restlichen Middlewarekomponenten erhalten hat wird das *IQueryable* ausgeführt und das Ergebnis an den Client zurückgeliefert.

4.7 Querys

Im folgenden Abschnitt wird die Umsetzung, eines lesenden Zugriffs mittels einer Query, auf die Entität *Author* erläutert. Dabei wird das Zustandekommen der Schema-Definition der Query, als auch der Zugriff auf die Datenbank mittels der Geschäftslogik eingegangen. Weiters wird erläutert wie HotChocolate das Over und Underfetching Problem löst, als auch Filtern, Sortieren und Pagination ermöglicht.

Die folgenden Unterkapitel widmen sich der Umsetzung der Query *authors* welche alle im System gespeicherten Autoren liefert. Dabei kann man diese Entitäten filtern, sortieren und paginieren.

Generierung Schema

Um es einem Client zu ermöglichen auf die Autoren zuzugreifen, ist es erforderlich die Query im Schema zu definieren. Hierbei wird für die Generierung des Schemas, das bereits erwähnte Pure-Code-First verwendet.

```

1 public class AuthorQuery: ObjectType<Query> {
2     protected override void Configure(IObjectTypeDescriptor<Query> descriptor
3         ) {
4         descriptor.Field("authors")
5     }
6 }
  
```

```

4         .ResolveWith<AuthorResolver>(r => r.Authors())
5         .Authorize()
6         .UseProjection()
7         .UseFiltering()
8         .UseSorting()
9         .Type<ListType<NonNullType<AuthorType>>>>();
10    }
11 }

```

Im obigen Code-Beispiel ist zu sehen, dass die Wurzeloperation *Query* um eine *AuthorQuery* erweitert wird. Überschreibt man nun die in der Klasse *ObjectType* definierte Methode *Configure* kann man die Felder der Query mit dem *IObjectTypeDescriptor* erweitern. Dabei wird das Feld *authors* angelegt welches eine Liste von Autoren zurückgibt.

```

1 type Query{
2     authors(where: AuthorFilterInput order: [AuthorSortInput!]): [Author!]
3     @authorize(apply: BEFORE_RESOLVER)
4 }
5 type Author {
6     firstName: String!
7     lastName: String!
8     books: [Book!]!
9     id: Int!
10 }
11
12 input AuthorFilterInput {
13     and: [AuthorFilterInput!]
14     or: [AuthorFilterInput!]
15     firstName: StringOperationFilterInput
16     lastName: StringOperationFilterInput
17     books: ListFilterInputTypeOfBookFilterInput
18     id: ComparableInt32OperationFilterInput
19 }
20
21 input ListFilterInputTypeOfBookFilterInput {
22     all: BookFilterInput
23     none: BookFilterInput
24     some: BookFilterInput
25     any: Boolean
26 }
27
28 input AuthorSortInput {
29     firstName: SortEnumType
30     lastName: SortEnumType
31     id: SortEnumType
32 }
33
34 input StringOperationFilterInput {
35     and: [StringOperationFilterInput!]
36     or: [StringOperationFilterInput!]
37     eq: String
38     neq: String
39     contains: String
40     ncontains: String
41     in: [String]

```

```

42  nin: [String]
43  startsWith: String
44  nstartsWith: String
45  endsWith: String
46  nendsWith: String
47 }
48
49 input ComparableInt32OperationFilterInput {
50   eq: Int
51   neq: Int
52   in: [Int!]
53   nin: [Int!]
54   gt: Int
55   nggt: Int
56   gte: Int
57   ngte: Int
58   lt: Int
59   nlt: Int
60   lte: Int
61   nlte: Int
62 }

```

Der oben gegebene Code erzeugt dabei zur Laufzeit das hier gezeigte Schema. Nur der *AuthorType* als auch die Wurzel-Operation der Query mit dem Feld *authors* wurden explizit generiert. Die Methodenaufrufe *Authorize()*, *UseFiltering()* und *UseSorting()* sind dabei für die Generierung von nicht explizit angegebenen Typen verantwortlich. Diese Methodenaufrufe aktivieren die jeweilige Field-Middleware, diese werden in den nachfolgenden Abschnitten genauer erläutert:

Authorize

Hierbei wird deklariert, dass nur angemeldete Benutzer Zugriff auf dieses Feld haben. Diese Middleware ist, wie in der obigen Schemadefinition ersichtlich, vor dem Resolver auszuführen. Sie stellt damit sicher, dass nur Anfragen mit einer gültigen Authentifizierung an die Geschäftslogik weitergereicht werden. Im Schema wird das Feld *authors* der Query mit der Direktive *@authorize* versehen. Näheres zur Authentifizierung und Autorisierung ist im gleichnamigen Abschnitt ersichtlich.

Projection

Mit *Projections* liefert HotChocolate die Möglichkeit Over und Underfetching zu verhindern. Over und Underfetching zu verhindern bedeutet, dass genau jene Daten, welche vom Client angefordert werden, in der Datenbank selektiert und anschließend an den Client zurückgeliefert werden. Dabei bekommt HotChocolate vom *Resolver* ein *IQueryable*. Dieses *IQueryable* beinhaltet zunächst nur jene Filterungen und Selektionen welche von der Geschäftslogik festgelegt werden. HotChocolate erweitert dieses *IQueryable* nun durch jene Felder welche in der Query angefordert wurden.

Filtering

Aktiviert man die *Filtering* Middleware für ein Query-Feld, so kann man den

implizit von HotChocolate bereitgestellten Filter-Input verwenden. Diese Middleware erweitert das vom Resolver zurückgelieferte *IQueryable* um die gegebene Filterung und liefert das Ergebnis der vorangestellten Middleware zurück. Weiters ist es möglich benutzerdefinierte Typen für die Filterung zu definieren.

Sorting

Aktiviert man die *Sorting* Middleware für ein Query-Feld, so kann man den implizit von HotChocolate bereitgestellten Sortier-Input verwenden. Diese Middleware fügt der vom Resolver zurückgelieferten *IQueryable* die gegebene Sortierung hinzu. Das Ergebnis wird wiederum an die vorangestellte Middleware zurückgeliefert. Weiters ist es möglich benutzerdefinierte Typen für die Sortierung zu definieren.

Zugriff auf Daten

Der *AuthorResolver* kümmert sich dabei um die Bereitstellung der geforderten Daten. Er greift auf den *AuthService*, einer Kindklasse des *BaseService*, zu und dieser mithilfe der Datenbankzugriffsschicht auf die Datenbank. Wichtig dabei ist, dass der *AuthService*, HotChocolate ein *IQueryable* zurückliefert. Das zurückgelieferte *IQueryable* wird vom Resolver an die restlichen Middlewarekomponenten zurückgereicht. Diese erweitern dann das *IQueryable* mit der von ihnen definierten Logik und werten es anschließend aus.

Ausführung Query und Ergebnis

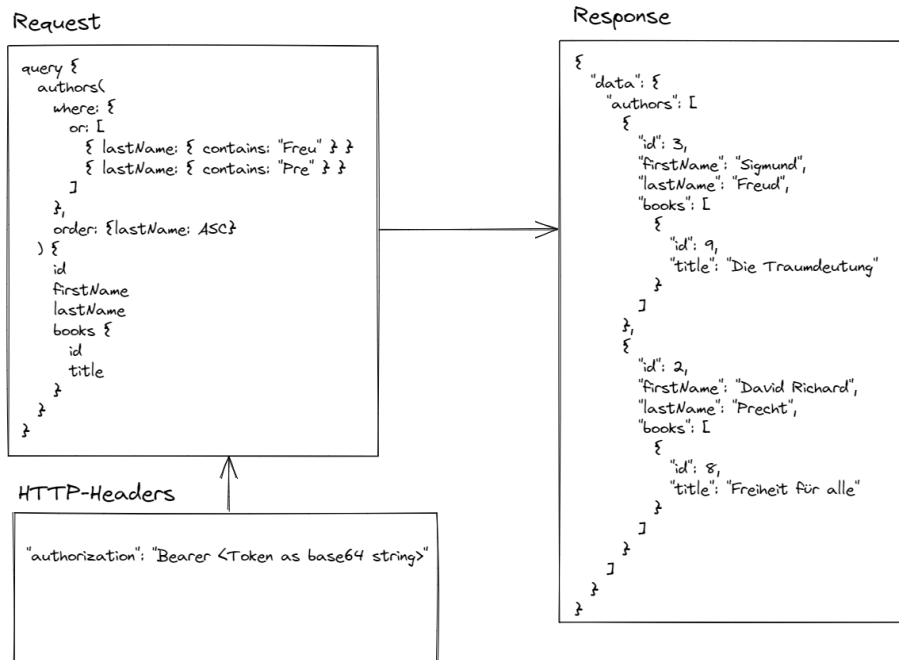


Figure 4.5: Exekutionsreihenfolge Middleware

Die obige Abbildung zeigt eine Query, welche alle Autoren mittels dem Feld *authors* abfragt. Dabei sollen nur jene Autoren zurückgegeben werden deren Nachname entweder "Freu" oder "Pre" enthält. Weiters werden die Autoren nach ihrem Nachnamen sortiert. Bei der Antwort des GraphQL-Service ist zu sehen, dass dieser die Daten genau in jenem Format zurückgibt, in dem sie angefragt worden sind. Mittles HTTP-Parametern wurde der für die Authentifizierung notwendige *JWT-Token* mitübermittelt.

Die oben abgebildete Anfrage hat am Server folgende Abfrage an die Datenbank ausgelöst:

```

1 SELECT [a].[Id], [a].[FirstName], [a].[LastName], [t].[Id], [t].[Title], [t].[AuthorsId], [t].[BooksId]
2 FROM [Authors] AS [a]
3 LEFT JOIN (
4   SELECT [b].[Id], [b].[Title], [a0].[AuthorsId], [a0].[BooksId]
5   FROM [AuthorBook] AS [a0]
6   INNER JOIN [Books] AS [b] ON [a0].[BooksId] = [b].[Id]
7 ) AS [t] ON [a].[Id] = [t].[AuthorsId]
8 WHERE ((@__p_0 LIKE N'') OR (CHARINDEX(@__p_0, [a].[LastName]) > 0)) OR ((
9   @@p_1 LIKE N'') OR (CHARINDEX(@__p_1, [a].[LastName]) > 0))
9 ORDER BY [a].[LastName], [a].[Id], [t].[AuthorsId], [t].[BooksId]

```

4.8 Mutations

Mutations werden wie bereits erwähnt für schreibende Zugriffe auf den GraphQL-Service verwendet. In den folgenden Abschnitten wird die Umsetzung einer Mutation für das Bearbeiten eines bereits bestehenden Buches erläutert. Dabei wird auf das Generieren des Schemas zur Laufzeit, als auch der Zugriff auf die Datenbank mittels der Geschäftslogik erläutert.

Generierung Schema

Damit ein Client die Möglichkeit hat auf eine Mutation zuzugreifen, muss der Wurzeloperation im Schema das gewünschte Feld hinzugefügt werden. Dabei wird, wie bereits bei der Implementierung der Query, die Pure-Code-First Vorgehensweise angewendet.

Im folgenden Code wird die Wurzeloperation Mutation um ein Feld *updateBook* erweitert. Dabei werden nur Benutzern mit den Rollen Admin oder Bibliothekar die Ausführung gestattet. Als Übergabeparameter bekommt die Funktion das *DTO BookUpdateInput*. Der Resolver *BookResolver* kümmert sich dabei um Ausführung der Operation.

```
1 public class BookMutation: ObjectTypeExtension<Mutation>{
2     protected override void Configure(IObjectTypeDescriptor<Mutation>
        descriptor) {
3         descriptor.Field("updateBook")
4             .Authorize(new [] { "Admin", "Librarian" })
5             .Argument("input", a => a.Type<NonNullType<BookUpdateInput>>())
6             .ResolveWith<BookResolver>(r => r.UpdateBook(default, default))
7             .Type<BookType>();
8     }
9 }
```

Der obige Code resultiert in folgendem Schema:

```
1 type mutation{
2     updateBook(input: BookUpdateInput!): Book @authorize(roles: [ "Admin", "
        Librarian" ], apply: BEFORE_RESOLVER)
3 }
4
5 type Book {
6     title: String!
7     authors: [Author!]!
8     reviews: [Review!]!
9     id: Int!
10 }
11
12 type Author {
13     firstName: String!
14     lastName: String!
15     books: [Book!]!
16     id: Int!
17 }
18
19 type Review {
20     userId: Int!
21     user: User!
```

```

22  bookId: Int!
23  book: Book!
24  rating: Int!
25  id: Int!
26 }
27
28 type User {
29   firstName: String!
30   lastName: String!
31   email: String!
32   roles: [Role!]!
33   reviews: [Review!]!
34   id: Int!
35 }
36
37 type Role {
38   name: String!
39   users: [User!]!
40   id: Int!
41 }
42
43 input BookUpdateInput {
44   authors: [Int!]
45   id: Int!
46   title: String!
47 }

```

In dem generierten Schema ist zu erkennen, dass der Objekt-Typ Buch alle Objekt-Typen die dieser explizit oder implizit referenziert, generiert wurden. Desweiteren wurde die Wurzeloperation Mutation durch das Feld *updateBook* erweitert. Das Feld hat eine *@authorize* Direktive, welche nur Benutzern mit den Rollen Admin oder Bibliothekar Zugriff gewährt. Weiters wurde das Input-Objekt *BookUpdateInput* generiert.

Das Input-Objekt *BookUpateInput* wurde dabei zusätzlich wie folgt deklariert:

```

1 public class BookUpdateInput: InputObjectType<BookUpdate> {
2
3     protected override void Configure(IInputObjectTypeDescriptor<BookUpdate>
4         descriptor) {
5         descriptor.Field(f => f.Authors).Type<ListType<NonNullType<IntType
6             >>>>();
7     }
8 }
9
10 public class BookUpdate {
11     public int Id { get; set; }
12     public string Title { get; set; }
13     public ICollection<int> Authors { get; set; }
14 }

```

Bei der Definition des Input-Objekts *BookUpdateInput* bietet die *Configure* Methode der Basisklasse *ObjectType* wiederum die Möglichkeit genauere Definitionen vorzunehmen. Im obigen Code-Beispiel ist ersichtlich, dass das Feld *Authors* eine Liste von *Integer* enthält die nicht *null* sein dürfen.

Anders als die im Datenbankschema beschriebene Entität Buch hält das Input-Objekt *BookUpdateInput* keine Liste des Objekt-Typs Autor, sondern nur eine Liste von *Integer* Werten, welche die IDs widerspiegeln. Mit dieser Vorgehensweise wird einer zyklischen Abhängigkeit vorgebeugt, denn sonst würden die Bücher Autoren referenzieren, welche wiederum Bücher referenzieren und das wiederholt sich endlos. Um diese *DTOs* wieder zu Domänenklassen umzuwandeln, müssen sie gemappt werden. Dafür wird im Prototyp die Bibliothek *AutoMapper* verwendet.

Im folgenden Code-Beispiel wird *BookUpdateInput* wieder zu *Book* umgewandelt:

```
1 public class BookProfile : Profile {
2     public BookProfile() {
3         CreateMap<BookUpdate, Book>()
4             .ForMember(
5                 dest => dest.Authors,
6                 opt => opt.MapFrom(src => src.Authors.Select(id => new Author {
7                     Id = id })));
8     }
9 }
```

Zum Vergleich hier noch die Klasse *Book* welche das Domänenobjekt Buch darstellt und von *BaseEntity* ableitet und damit eine *Id* erbt.

```
1 public class Book: BaseEntity {
2     public string Title { get; set; }
3     public List<Author> Authors { get; set; } = new List<Author>();
4     public List<Review> Reviews { get; set; } = new List<Review>();
5 }
```

Ausführung und Ergebnis

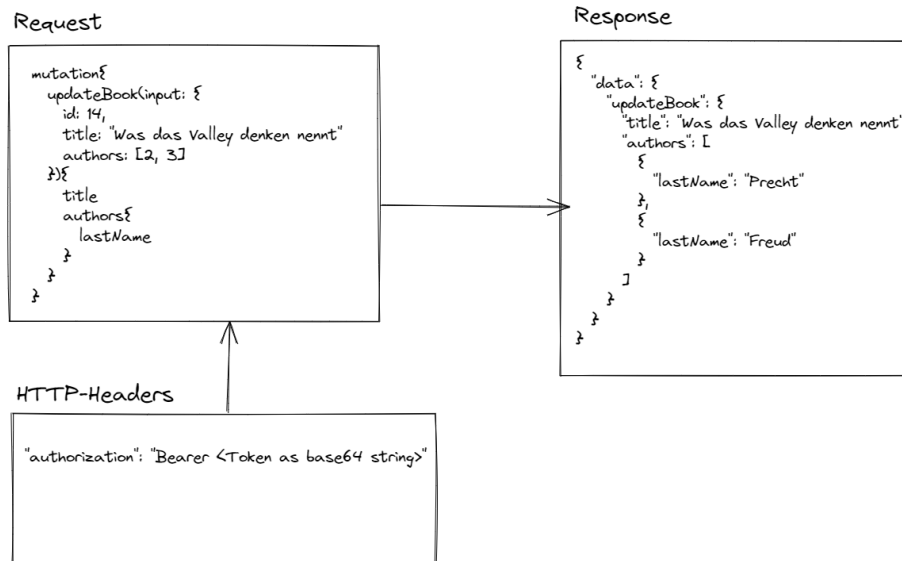


Figure 4.6: Schemadefinition

In dieser Abbildung ist die Ausführung des Feldes *updateBook* der Mutation zu sehen. Der Anfrage muss als HTTP-Header ein *JWT-Token* übergeben werden um zu überprüfen ob der ausführende Benutzer genügend Rechte hat. Die Antwort des GraphQL-Service entspricht dabei wiederum genau dem Schema welches in der Query der Anfrage angegeben wurde.

4.9 Subscriptions

Subscriptions werden wie bereits erwähnt verwendet, um bidirektionale Kommunikation zwischen dem Client und Server zu ermöglichen. Dabei registriert sich der Client auf Events, welche vom Server ausgelöst werden, und bekommt dadurch in Echtzeit Updates. HotChocolate realisiert Subscriptions mittels WebSockets. HotChocolate bietet dabei 2 Subscription Provider: *In-Memory* und *Redis*. Für den Prototypen wurde dabei die *In-Memory* Version verwendet.

Subscription über Event benachrichtigen

Um eine Subscription über ein ausgeführtes Event zu benachrichtigen, stellt HotChocolate den *ITopicEventSender* und den *ITopicEventReceiver* zur Verfügung. Diese Interfaces sind Abstraktionen der Funktionalitäten des Subscription Providers. Diese Abstraktion ermöglicht es dem Entwickler, den Subscription Provider je nach Bedarf, zu einem späteren Zeitpunkt beliebig auszutauschen. Um nun eine Subscription von einem Event zu benachrichtigen, ist folgender Code notwendig:

```

1 public async Task<Book> CreateBook([Service] IBookService bookService,
   BookCreate input, [Service] ITopicEventSender sender) {

```

```

2     var book = await bookService.AddAsync(mapper.Map<Book>(input));
3     await sender.SendAsync("bookAdded", book);
4     return book;
5 }

```

In Zeile 3 des obigen Code-Beispiels wird mittels dem *ITopicEventSender* der *ITopicEventReceiver* von dem neu erstelltem Buch notifiziert. Auf die Verwendung des *ITopicEventReceiver* wird in der folgenden Implementierung einer Subscription, welche auf die Erstellung eines Buches wartet, näher eingegangen:

Schemagenerierung

Für die Generierung des Schemas wird die Pure-Code-First Methode von GraphQL verwendet. Die Wurzeloperationen Subscription wird dabei um ein Feld *bookAdded* erweitert. Die Umsetzung ist dabei in folgendem Code-Beispiel ersichtlich:

```

1 public class BookSubscription: ObjectTypeExtension<Subscription> {
2     protected override void Configure(IObjectTypeDescriptor<Subscription>
3         descriptor) {
4         descriptor
5             .Field("bookAdded")
6             .Type<BookType>()
7             .Resolve(context => context.GetEventMessage<Book>())
8             .Subscribe(async context => {
9                 var receiver = context.Service<ITopicEventReceiver>();
10                return await receiver.SubscribeAsync<string, Book>("bookAdded");
11            });
12 }

```

In Zeile 9 des obigen Code-Beispiels ist ersichtlich, dass der *ITopicEventReceiver* auf eine Benachrichtigung durch den *ITopicEventSender* wartet.

Der oben stehende Code generiert dabei folgendes Schema:

```

1 type subscription{
2     bookAdded: Book
3 }
4
5 type Book {
6     title: String!
7     authors: [Author!]!
8     reviews: [Review!]!
9     id: Int!
10 }
11
12 type Author {
13     firstName: String!
14     lastName: String!
15     books: [Book!]!
16     id: Int!
17 }
18
19 type Review {
20     userId: Int!
21     user: User!
22     bookId: Int!
23     book: Book!

```

```

24  rating: Int!
25  id: Int!
26 }
27
28 type User {
29   firstName: String!
30   lastName: String!
31   email: String!
32   roles: [Role!]!
33   reviews: [Review!]!
34   id: Int!
35 }
36
37 type Role {
38   name: String!
39   users: [User!]!
40   id: Int!
41 }

```

Ausführung und Ergebnis

Auf den Verbindungsaufbau von Subscriptions wurde bereits im Kapitel GraphQL eingegangen. Das folgende Bild zeigt das Ergebnis einer Subscription welche von der Erstellung eines Buches notifiziert wurde.

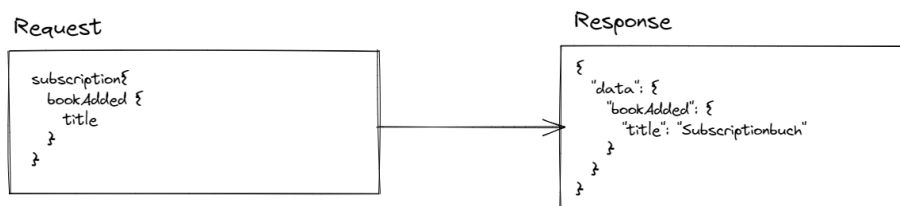


Figure 4.7: Subscription welche von der Erstellung eines Buches benachrichtigt wird.

4.10 Authentifizierung und Autorisierung

Als Authentifizierung wird jener Vorgang bezeichnet, mit dem die Identität eines Benutzers festgestellt wird. Die Autorisierung wiederum ermittelt ob ein Benutzer die erforderlichen Rechte hat, um auf eine bestimmte Ressource zugreifen zu können. Wie im Anwendungsszenario bereits beschrieben, gibt es im System 3 Rollen für Benutzer: User, Librarian und Admin. Jede Rolle kann dabei nur für Sie freigegebene Ressourcen zugreifen.

Die Authentifizierung und Autorisierung für den Prototypen wird mit *JWT-Tokens* und der Zuhilfenahme der *ASP.NET Core-Authentifizierung* umgesetzt. Die folgenden Abbildungen beschreiben die Rollen und die Ressourcen auf die sie Zugriff haben. Grüne Felder bedeuten dabei, dass auch Benutzer ohne valides

JWT-Token Zugriff auf diese Ressource haben. Das gelbe Feld "Einloggen" kann nur nach einer bereits erfolgten Registrierung aufgerufen werden. Rote Felder wiederum verlangen ein valides *JWT-Token* und sind an die Rolle des aufrufenden Benutzers gebunden.

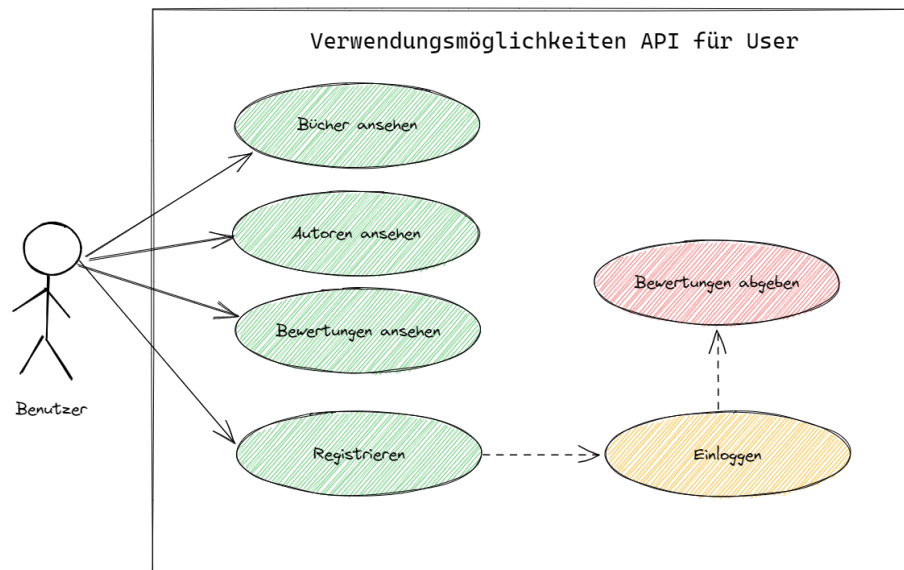


Figure 4.8: Rechte User.

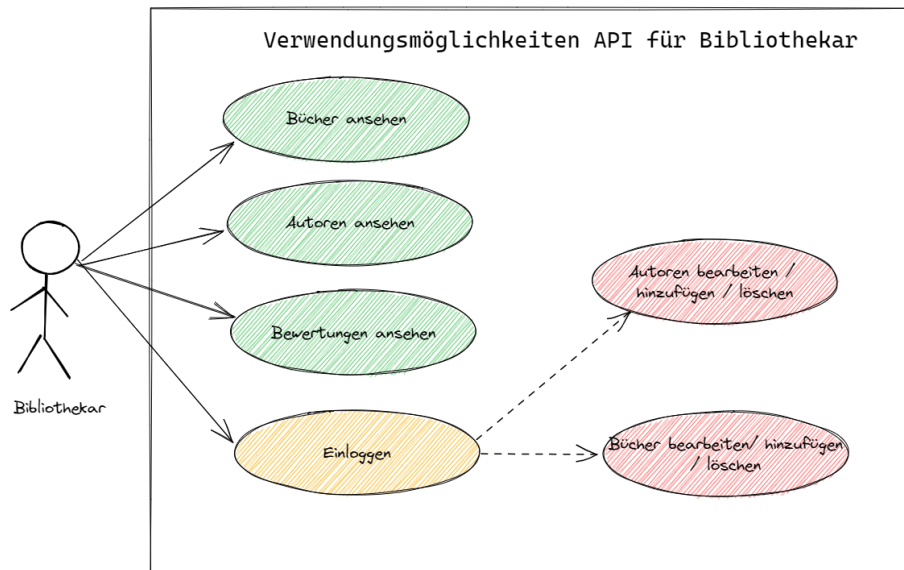


Figure 4.9: Rechte Bibliothekar.

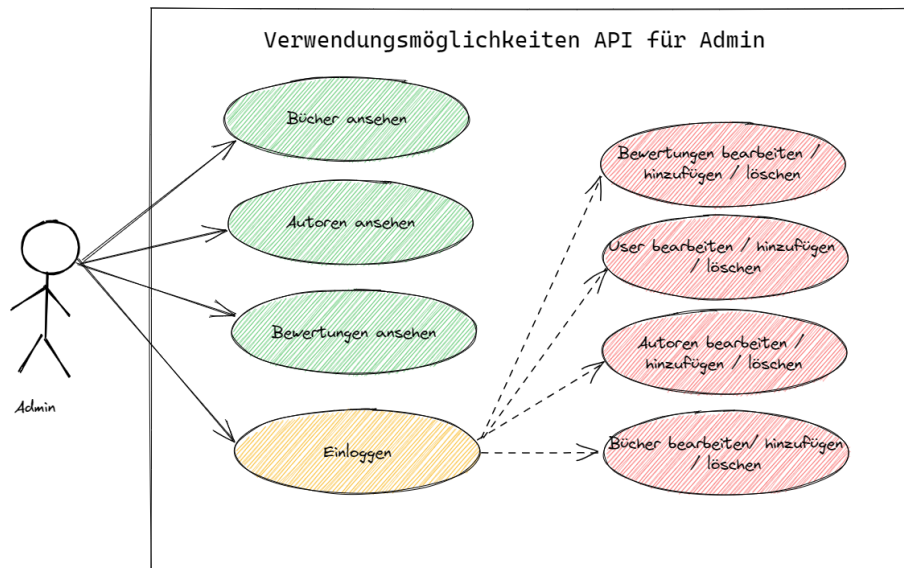


Figure 4.10: Rechte Admin.

Generierung JWT-Token

Um die Authentifizierung mittels JWT-Token zu ermöglichen muss diese Au-

thentifizierungsform erst registriert werden. Diese Registrierung erfolgt, wie in .NET üblich, im *WebApplicationBuilder*.

```
1 builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
2     .AddJwtBearer(options => {
3         var tokenSettings = configuration
4         .GetSection("JWT").Get<TokenSettings>();
5         options.TokenValidationParameters = new TokenValidationParameters {
6             ValidIssuer = tokenSettings.Issuer,
7             ValidateIssuer = true,
8             ValidAudience = tokenSettings.Audience,
9             ValidateAudience = true,
10            IssuerSigningKey = new SymmetricSecurityKey(
11                Encoding.UTF8.GetBytes(tokenSettings.Key)),
12            ValidateIssuerSigningKey = true
13        };
14    });
```

Im obigen Code ist zu sehen, dass dem Server auf dem der GraphQL-Service läuft, eine JWT-Token Authentifizierung hinzugefügt wird. Hierzu wurde eine Klasse *TokenSettings* erstellt welche den *Issuer*, die *Audience* und den *Key* welche die benötigten Daten aus der *appsettings.json* liest und bereitstellt.

Nach der Registrierung, der zu verwendenden Authentifizierungsmethode, erfolgt die eigentliche Generierung des Tokens. Diese wurde im *AuthService* in der Methode *Login* umgesetzt. Diese Methode erhält dabei einen Benutzernamen und ein Passwort als Übergabeparameter. Hervorzuheben ist dabei, dass der Login als Mutation umgesetzt wurde. Zum jetzigen Zeitpunkt finden keine schreibenden Operationen in dieser Methode statt. Login-Operationen beinhalten aber oftmals das Speichern des Letzten erfolgreichen Logins, deswegen wurde für die Implementierung eine Mutation gewählt.

Im folgenden Code-Beispiel wird ein JWT-Token, bei übereinstimmenden User-Credentials, generiert.

```
1 public async Task<string> Login(string email, string password) {
2
3     var user = await userRepository.GetFirstAsync(user => user.Email.Equals(
4         email), user => user.Roles);
5     if (user is not null && BCrypt.Net.BCrypt.Verify(password, user.Password)
6     ) {
7         var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(
8             tokenSettings.Key));
9
10        var credentials = new SigningCredentials(securityKey,
11            SecurityAlgorithms.HmacSha256);
12
13        var claims = new List<Claim>();
14
15        claims.Add(new Claim("FirstName", user.FirstName));
16        claims.Add(new Claim("LastName", user.LastName));
17        claims.Add(new Claim("Email", user.Email));
18        if (user.Roles?.Count > 0) {
19            foreach (var role in user.Roles) {
```

```

16         claims.Add(new Claim(ClaimTypes.Role, role.Name));
17     }
18 }
19
20     var jwtSecurityToken = new JwtSecurityToken(
21         issuer: tokenSettings.Issuer,
22         audience: tokenSettings.Audience,
23         expires: DateTime.Now.AddDays(1),
24         signingCredentials: credentials,
25         claims: claims
26     );
27
28     return new JwtSecurityTokenHandler().WriteToken(jwtSecurityToken);
29 }
30
31     return "";
32 }

```

Der obige Code beinhaltet die Überprüfung der vom Client übergebenen Benutzerdaten. Stimmen diese mit jenen in der Datenbank überein, so wird ein JWT-Token generiert. Der Token beinhaltet dabei folgende Daten des Benutzers: Vorname, Nachname, Email und die zugewiesenen Rollen.

Verwendung Authentifizierung und Autorisierung

Um ein Feld einer Query, Mutation oder Subscription nun abzusichern muss man die Field-Middleware Authentifizierung bei dem jeweiligen Feld aktivieren. Mit der Pure-Code-First Methode von HotChocolate wird die Konfiguration in der *Configure* Methode des jeweiligen *ObjectType* abgebildet.

```

1 public class AuthorQuery: ObjectType<Query> {
2     protected override void Configure(IObjectTypeDescriptor<Query> descriptor
3     ) {
4         descriptor.Field("authors")
5             .ResolveWith<AuthorResolver>(r => r.Authors())
6             .Authorize()
7             .Type<ListType<NonNullType<AuthorType>>>>();
8     }
9 }

```

In dem oben stehenden Code wird festgelegt, dass ein User ein valides *JWT-Token* an die Service übergeben muss um Zugriff auf die Ressource zu haben. Im Umkehrschluss bedeutet dies, dass jeder Benutzer des Systems, wenn er angemeldet ist, Zugriff auf diese Ressource hat.

```

1 public class AuthorQuery: ObjectType<Query> {
2     protected override void Configure(IObjectTypeDescriptor<Query> descriptor
3     ) {
4         descriptor.Field("authors")
5             .ResolveWith<AuthorResolver>(r => r.Authors())
6             .Authorize(new [] { "Admin", "Librarian" })
7             .Type<ListType<NonNullType<AuthorType>>>>();
8     }
9 }

```

Dieser Code erweitert die erste Version um Autorisierung. Es haben somit nur mehr Benutzer mit den Rollen Admin und Bibliothekar Zugriff auf diese Ressource. Ein "normaler" Benutzer würde abgewiesen werden.

4.11 1 + n Problem

Lösung mit DataLoader

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