Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

GraphQL im Produktiveinsatz

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Franz-Filip Schörghuber

Begutachtet von FH-Prof. DI Johann Heinzelreiter

Hagenberg, Mai 2021

software engineering
IKM Fakultät Hagenberg

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Die vorliegende, gedruckte Bachelorarbeit ist identisch zu dem elektronisch übermittelten Textdokument.

Datum

31.05.2022

Unterschrift

Kurzfassung

Für die Realisierung Ressourcen-orientierter Kommunikation sind REST-Services oftmals das Mittel der Wahl. Dabei kommuniziert das Frontend mit dem Backend, um die für die Visualisierung benötigen Daten abzufragen. Dabei ist die Datenabfrage mittels REST jedoch sehr unflexibel. Bei komplexeren Datenstrukturen reicht eine einzelne Abfrage oftmals nicht aus, um alle für den Client relevanten Daten bereitzustellen. Weiters können Probleme auftreten, die auf die Unflexibilität von REST zurückzuführen sind. Um der Unflexibilität von REST und den auftretenden Problemen entegenzuwirken, wurde von Facebook GraphQL entwickelt. GraphQL bietet eine alternative Möglichkeit, Web-APIs zu realiseren und dabei die Probleme von REST zu minimieren. Weiters bietet GraphQL eine flexiblere und effizientere Möglichkeit, Daten abzufragen.

In dieser Arbeit werden die konzeptionellen Grundlagen von GraphQL aufgearbeitet. Weiters wird der Entwicklungsprozess eines GraphQL-Service beschrieben. Der aus der Entwicklung resultierende Prototyp wird mit .NET 6 unter Zuhilfenahme des Frameworks HotChocolate umgesetzt. Für den Datenbankzugriff wird das Entity Framework herangezogen. Die Umsetzung beschäftigt sich zudem mit der Lösung von bekannten Problemen bei der Implementierung von GraphQL-Services wie dem „1+n Problem“ oder dem Verhindern von Underfetching und Overfetching. Weiters wird die Absicherung vom GraphQL-Service behandelt. Zudem wird auf die bidirektionale Kommunikation zwischen Backend und Frontend mittels Subscriptions eingegangen.

Abstract

For the realisation of resource-oriented communication, REST services are often the means of choice. The frontend communicates with the backend to retrieve the data needed for the visualisation. Data retrieval via REST is very inflexible. With complex data structures, a single query is often not sufficient to provide all the data relevant for the client. Furthermore, problems can occur due to the inflexibility of REST. To counteract the inflexibility of REST and the problems that arise, Facebook developed GraphQL. GraphQL offers an alternative option to realise Web-APIs and to minimise the problems of REST. Furthermore, GraphQL offers a more flexible and efficient way to retrieve data.

In this thesis the conceptual basics of GraphQL are presented. Moreover, the development process of a GraphQL-Service is described. The resulting prototype is implemented with .NET 6 and the HotChocolate framework. The Entity Framework is used for database access. The implementation also deals with the solution of known problems such as the 1+n problem or the prevention of under and overfetching. Additionally, the GraphQL service is secured against unauthorised access from outside. Last but not least, it is shown that bidirectional communication between backend and frontend can be realised by means of subscriptions.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Kommunikation von verteilten Systemen über ein Netzwerk ist in der heutigen Zeit von enormer Bedeutung. Daten werden zwischen den Systemen zur visuellen Präsentation und zur maschinellen Weiterverarbeitung ausgetauscht. Es ist üblich, diese Systeme mittels einer Client-Server Architektur umzusetzen. Der Server stellt dem Client Ressourcen zur Verfügung, welche je nach Bedarf abgefragt werden können. Heutzutage erreichen den Server Anfragen von vielen unterschiedlichen Clients. Ein Client kann dabei beispielsweise eine mobile Applikation als auch eine Webapplikation sein. Clients haben wiederum verschiedene Anforderungen an die Ressourcen welcher der Server bereitstellt.

1.1 Motivation

Verteilte Systeme beschränken sich meist nicht nur auf einen einzigen Client. Ein Client kann biespielsweise eine Webanwendung in Form einer Single-Page-Webapplikation oder eine mobile Applikation sein. Momentan werden überwiegend REST-APIs für den Datenaustausch zwischen Client und Server verwendet. Eine REST-API bietet einem Client Ressourcen in unterschiedlichen Repräsentationen an. Diese Ressourcen werden dem Client mittels Uniform Resource Identifiers (URIs) zur Verfügung gestellt. Da nun unterschiedliche Clients, unterschiedliche Anforderungen an die Ressourcen haben, führt dies unweigerlich zu einer höheren Komplexität der REST-API. Die Unübersichtlichkeit der REST-Schnittstelle folgt aus den vielen verschiedenen Anforderungen der verschiedenen Clients. Weiters wird die REST-Schnittstelle bei vielen verschiedenen Anforderungen schnell sehr umfangreich. Ein Client hat keine Möglichkeit die Struktur der Daten, welche er benötigt, genauer zu spezifizieren. Dadurch kann der Client entweder zu wenig Daten oder zu viele Daten erhalten. In beiden Fällen führt dies zu einem erhöhten Netzwerkverkehr. Da der Client entweder Daten nachladen muss oder Daten verwirft, da er sie garnicht benötigt. Besonders bei mobilen Clients führt dies zu Performanz-Einbußen. Um diesen Problemen entgegenzuwirken, wurde von Facebook GraphQL entwickelt. GraphQL ist eine Datenabfragesprache, welche es Clients ermöglicht die Struktur, der abgefragten Daten, genau festzulegen. Die Struktur der Daten wird in der Abfrage festgelegt und anschließend in genau dieser Form vom Server re-

tourniert. GraphQL bietet, anders als REST-APIs, genau einen Endpunkt. An diesen Endpunkt kann der Client lesende und schreibende Zugriffe senden. Weiters bietet GraphQL die Möglichkeit, bidirektionale Kommunikation zwischen dem Client und dem Server für Echtzeitdaten zu schaffen.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, dem Leser die konzeptionellen Grundlagen und Basiswissen von GraphQL zu vermitteln. Weiters wird dem Leser eine konkrete Umsetzung eines GraphQL-Service in .NET und unter Zuhilfenahme der Bibilothek HotChocolate vermittelt. Dabei bietet der entwickelte Prototyp Lösungen für bekannte Probleme, etwa dem 1 + n Problem oder Over- und Underfetching. Weiters dient die Arbeit dazu, die Vorteile von GraphQL gegenüber REST aufzuzeigen.

Kapitel 2

Grundlagen

2.1 API

Der Begriff API steht für *Application Programming Interface* (auf Deutsch: Anwendungs-Programmier-Schnittstelle). Die Grundlagen heutiger APIs wurden 1952 von David Wheeler, einem Informatikprofessor an der Universität Cambrigde, in einem Leitfaden verfasst. Dieser Leitfaden beschreibt das Extrahieren einer Bibliothek von Funktionen und Prodzeduren mit einheitlichem, dokumentierten Zugriff [Wheeler 1952, S.5-6]. Ira Cotton und Frank Greatorex erwähnten den Begriff erstmalig auf der Fall Joint Computer Konferenz [Cotton und Greatorex Jr 1968, S.5]. Dabei erweiterten sie den Leitfaden von David Wheeler um einen wesentlichen Punkt: Der konzeptionellen Trennung der Schnittstelle und Implementierung der Bibliothek. Somit kann die Implementierung, welche auf die Schnittstelle zugreift, ohne Einfluss auf die Verwender ausgetauscht werden [Kress 2020, S.2]. APIs sind wie Benutzerschnittstellen, nur ist ein anderer Nutzen dabei im Fokus [Berlind 2017, S.5]. APIs sind also wie Benutzerschnittstellen für die Interaktion mit Benutzern gedacht. Der wesentliche Unterschied zwischen Benutzerschnittstellen und einer API liegt aber in der Art der Nutzer, die auf das Interface zugreifen. Bei Benutzerschnittstellen spricht man von einem *human readable interface*, was bedeutet, dass 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.

2.2 REST-API

REST steht für *Representational State Transfer* [Wheeler 1952, S.235-236]. REST wurde aufgrund des Erfolgs von SOAP entworfen [Ganatra 2021, Abs. REST APIs]. REST ist 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 XML verwendet [Kress 2020, S.11]. Anders als bei SOAP können REST-APIs aber auch noch andere Datenübertragungsformate wie JSON oder YAML

anbieten [Ganatra 2021, Abs. REST APIs].

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

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

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

In diesem Beispiel ist ersichtlich, dass JSON (134 Bytes) wesentlich weniger Daten als XML (238 Bytes) beinhaltet. Zusammenfassend kann man sagen, dass JSON deutlich leichtgewichtiger als XML ist, was sich positiv auf die Datenübertragungszeit auswirkt.

2.2.1 REST-Bedingungen

Als REST-API versteht man eine API welche die REST-Bedingungen erfüllt [Ganatra 2021, Abs. REST-APIs]. Die REST-Bedingungen werden wie folgt definiert:

Einheitliche Schnittstelle

Der zentrale Unterschied zwischen REST und anderen netzwerkbasiertern Architekturstilen, ist die Definition eines einheitlichen Interfaces [Fielding 2000, S.81]. Den beteiligten Komponenten wird es ermöglicht sich unabhängig voneinander weiterzuentwickeln. Um diese unabhängige Weiterentwicklung zu gewährleisten werden Implementierungen von den Services, die sie zur Verfügung stellen, entkoppelt.

Client-Server-Architektur

Das Client-Server-Prinzip beschreibt verteilte Systeme. In diesen verteilten Systemen sendet der Client über eine netzwerkbasierte Verbindung Anfragen an den Server. 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.

Zustandslosigkeit

Zustandlosigkeit bedeutet, dass der Client alle Informationen die der Server benötigt, um eine Anfrage auszuführen, mitsendet. Diese REST-Bedingung gibt einer REST-API eine Verbesserung der Sichtbarkeit, der Zuverlässigkeit und der Skalierbarkeit [Fielding 2000, S. 79].

Schichtenarchitektur

Geschichtete Systeme sind Systeme, die aus mehreren hierarchischen angeordneten Schichten bestehen. Dadurch wird eine Architektur geschaffen werden, in der die einzelnen

Komponenten nicht über die unmittelbare Schicht, mit der sie interagiert hinaussehen kann. Mit dieser Architektur wird die Komplexität der Abhängigkeiten innerhalb des Systems reduziert. Weiters wird die Unabhängigkeit der beteiligten Komponenten gefördert [Fielding 2000, S. 79].

Zwischenspeicher

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 Performanz des Servers gesteigert. Ein Nachteil besteht dahingehend, dass dem Client womöglich nicht immer die aktuellsten Daten zur Verfügung stehen [Fielding 2000, S. 79].

2.2.2 Kommunikation mit dem Server

Die Kommunikation zwischen Server und Client wird bei einer REST-API mittels HTTP und dessen Methoden realisert. Eine REST-Anfrage besteht aus einem Endpunkt, der HTTP-Methode, den HTTP-Headern und dem HTTP-Body. Ein Endpunkt enthält eine *URI* welche die Ressource, auf die zugegriffen wird, identifiziert.

Die HTTP-Methode beschreibt die CRUD-Operation, welche auf die Ressource ausgeführt werden soll. Um CRUD (Create, Read, Update, Delete) Operationen auf Ressourcen abzusetzen, werden folgende HTTP-Methoden verwendet:

1. **GET:** Wird als lesender Zugriff auf eine Ressource verwendet.
2. **POST:** Wird verwendet, um eine neue Ressource zu erstellen.
3. **PUT:** Aktualisiert oder ersetzt eine bestehende Ressource.
4. **PATCH:** Wird verwendet, um eine Ressource zu bearbeiten.
5. **DELETE:** Wird für das Löschen einer Ressource verwendet.

Der HTTP-Header übermittelt bei einer Anfrage zusätzliche Informationen, die für das Caching oder die Authentifizierung benötigt werden. Der HTTP-Body beinhaltet die Daten, welcher der Client an den Server schicken will.

2.2.3 Antwort des Servers auf eine Anfrage

Da REST-APIs auf dem HTTP-Standard beruhen, wird der zurückgegebene HTTP-Statuscode einer Anfrage verwendet, um den Erfolg oder das Fehlschlagen der Anfrage festzustellen. Jeder Status-Code gehört zu einer Klasse, jede Klasse wird durch die erste Ziffer des dreistelligen Statuscodes definiert. Die Statuscodes von 100 bis 199 bedeuten, dass die Antwort eine vorläufige Informationsantwort enthält. Ein Statuscode Bereich von 200 bis 299 signalisiert eine erfolgreiche Anfrage. 300 bis 399 gibt an den Client die Information zurück, dass die Anfrage an eine andere Ressource weitergeleitet werden muss. Liegt der zurückgegebene Statuscode zwischen 400 und 499 so hat der Client

bei der Anfrage etwas falsch gemacht. 500 bis 599 bedeutet, dass der Fehler am Server passiert ist [Fielding 2000, S. 120].

2.3 GraphQL

GraphQL wurde 2015 von Facebook für seine eigene Facebook-API veröffentlicht und ist die Spezifikation einer plattformunabhängigen Query-Sprache für APIs [Kress 2020, S. 18]. GraphQL wurde unter der MIT-Lizenz veröffentlicht. Die Kommunikation erfolgt wie bei REST über ein Request-Response-Schema. Für eine Ressource kann es in ERST mehrere Endpunkte geben. In GraphQL gibt es aber genau einen Endpunkt [Kress 2020, S. 18]. An diesen Endpunkt werden entweder Querys oder Mutations gesendet. Querys sind lesende Zugriffe auf Ressourcen, Mutations beschreiben schreibende Zugriffe auf die Ressourcen. Weiters ermöglicht GraphQL eine bidirektionale Verbindung zwischen Client und Server mittels Subscriptions. Die Zugriffsmöglichkeiten (Query, Mutation, Subscription) und die von GraphQL verwalteten Objekte werden mittels der *Schema Definition Language SDL* in einem Schema definiert. In GraphQL werden Ressourcen im Kontext von Graphen gesehen. Knoten, welche mittels des GraphQL-Schema-Systems definiert werden, repräsentieren Objekte. Kanten beschreiben die Beziehungen von Objekten im Graphen [Rakuten 2021, Abs. Basics of a GraphQL API].

GraphQL erlaubt es Clients, Daten von mehereren Ressourcen in einer einzigen Abfragequery abzufragen. Dabei müssen nicht mehr wie bei REST mehrere Abfragen an den Server geschickt werden, um dasselbe Ergebnis zu erzielen.

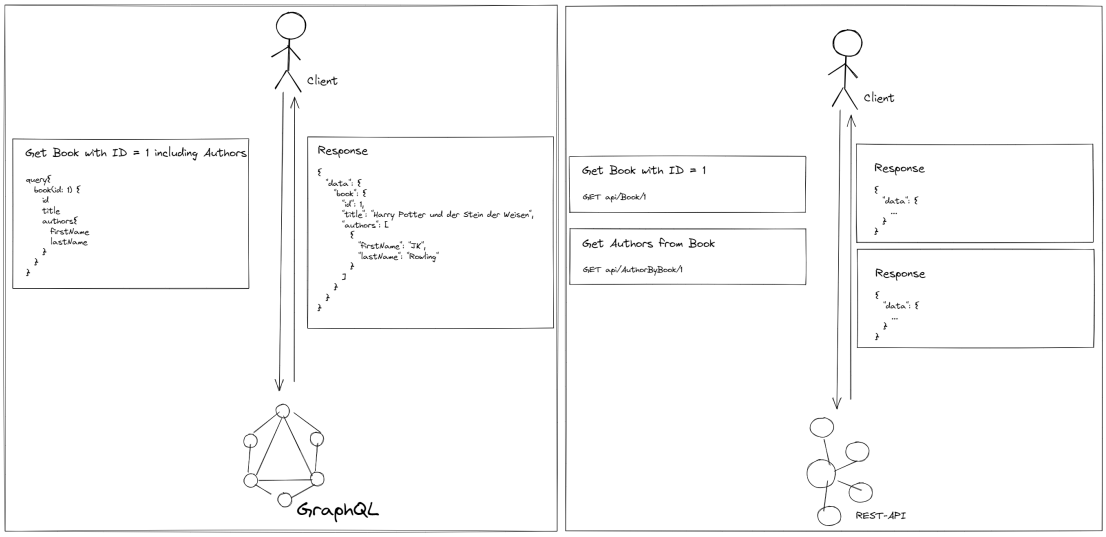

Abbildung 2.1: Gegenüberstellung REST-Anfrage / GraphQL-Anfrage

In Abbildung 2.1 ist zu sehen, dass ein Client zwei Anfragen an eine REST-API stellen muss, während er bei einem GraphQL-Service nur eine einzige Anfrage benötigt. Weiters ist zu sehen, dass GraphQL die Daten, welche in der Query abgefragt wurden, genau

in der Struktur zurückliefert, in der sie abgefragt wurden. Dadurch hat GraphQL den Vorteil, dass Antworten auf Anfragen immer vorhersehbar sind [Rakuten 2021, Abs. Basics of a GraphQL API].

2.3.1 Entwurfsprinzipien

Da GraphQL sprachunabhängig ist und auch nicht an ein Transportprotokoll gekoppelt ist, werden in der Spezifikation keine Implementierungsdetails definiert, sondern nur Entwurfsprinzipien. Diese Entwurfsprinzipien werden in den folgenden Abschnitten näher erläutert:

Produktzentriert

Die Anforderungen der Clients (Darstellung der Daten) stehen im Mittelpunkt. GraphQL bietet mit einer Abfragesprache dem Client die Möglichkeit, genau die Daten abzufragen, die er tatsächlich benötigt. Die Hierarchie der Abfrage wird durch eine Menge ineinander geschachtelter Felder abgebildet [Facebook 2021, Abs. 1].

Hierarchisch

Eine GraphQL-Anfrage ist hierarchisch strukturiert. Jede Anfrage ist so geformt wie die Daten die zurückgelifert werden. Das bedeutet, dass der Client die Daten genau in dem Format erhält, wie diese in der Anfrage spezifiziert wurden. Das ist ein intuitiver Weg für den Client, um seine Datenanforderungen zu definieren [Facebook 2021, Abs. 1].

Strenge Typisierung

Jeder GraphQL-Service definiert ein anwendungsspezifisches Typsystem. Anfragen werden im Kontext dieses Typsystems ausgeführt. Mit Tools wie zum Beispiel *GraphiQL* kann man vor Ausführung der Anfrage sicherstellen, dass sie syntaktisch und semantisch korrekt sind [Facebook 2021, Abs. 1].

Benutzerdefinierte Antwort

Mittels des Typsystems definiert der GraphQL-Service ein Schema, welches er veröffentlicht. In diesem Schema sind die Zugriffsarten als auch die verwalteten Ressourcen definiert. Der Client ist dafür zuständig, zu definieren, welche Daten er wie abfragen will. Die meisten anderen Client-Server-Applikationen geben die Form der Daten, welche sie zurückgeben, selbst vor. Ein GraphQL-Service retourniert exakt die Daten, welche der Client angefordert hat, nicht mehr und nicht weniger [Facebook 2021, Abs. 1].

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 [Facebook 2021, Abs. 1].

2.4 GraphQL vs. REST

In den folgenden Abschnitten werden die grundlegenden Unterschiede zwischen REST und GraphQL erläutert.

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 diesen kann der Client eine Abfrage mit einer Query oder einer Mutation senden, um auf die jeweilige Ressource zuzugreifen.

2.4.2 Overfetching und Underfetching

Als Overfetching wird das Laden von zu vielen oder nicht benötigten Daten bezeichnet. Underfetching 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 weniger 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 einem erhöten Programmieraufwand und damit auch einer höheren Komplexität der Anwendung.

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

Kapitel 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 2021, 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 2021, 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 eindeutig sein. Die Deklaration erfolgt mit dem Namen als auch dem Typ des Feldes, welche mit einem Doppelpunkt getrennt sind. Das Feld *id* wird 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 2021, 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 2021, 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 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 skalare Felder und bilden die 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 2021, 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!
    firstName: String
    lastName: String
    books: [Book]
}
```

Jedes Interface benötigt eine spezifische Implementierung, im angeführen Beispiel implementiert *Author* das Interface *Person*. Author muss 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:

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

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 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 2021, 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:

```
fragment authorFragment on Author {
    firstName
    lastName
}
```

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

```
query{
    books{
        title
        on Author{
            firstName
            lastName
        }
    }
}
```

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.

```
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 in der Query ein *Boolean* mitübergeben werden.

Im folgenden Beispiel wird eine Direktive gezeigt welche nach Bedarf das Feld *id* ignoriert.

```
query {
    authors($withoutId: Boolean = false){
        id @include(if: $withoutId)
        firstName
        lastName
    }
}
```

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 2021, 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.

Eine ausführliche Schema-Definition erfolgt im Abschnitt 4.7.

3.3.2 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]

3.3.3 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 2021, 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.

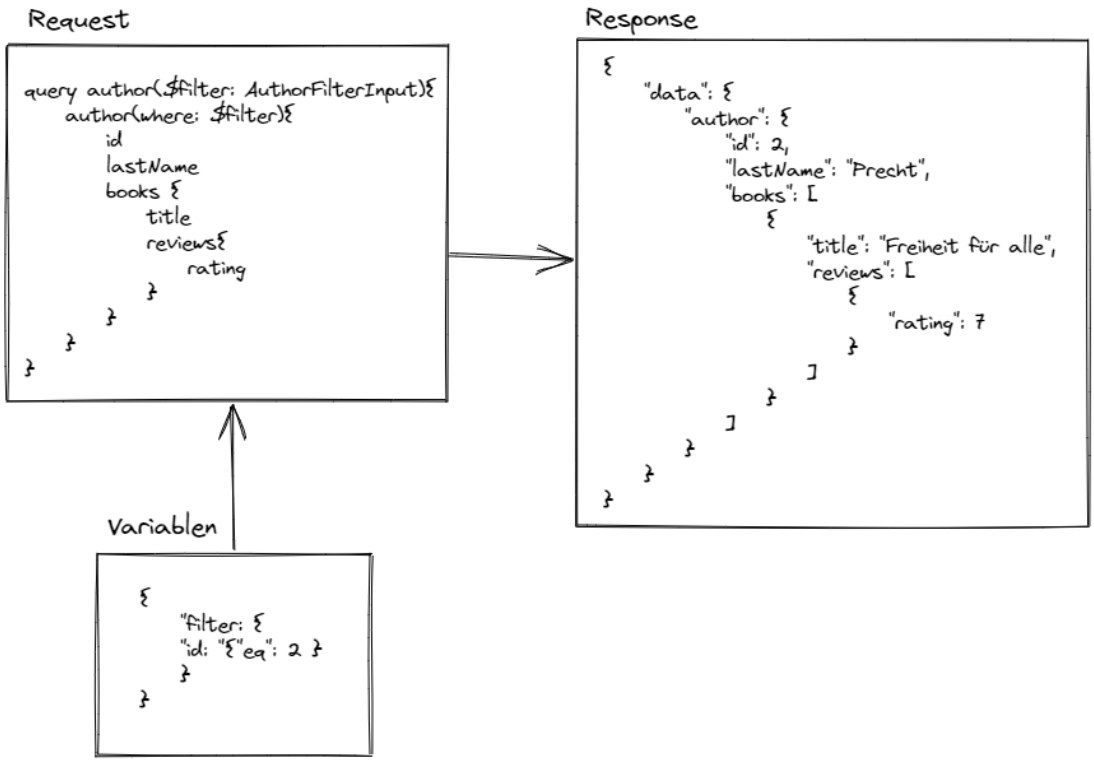

Abbildung 3.1: Anfrage aller Bücher mit zugehörigen Autoren unter Verwendung einer Variable.

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 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.

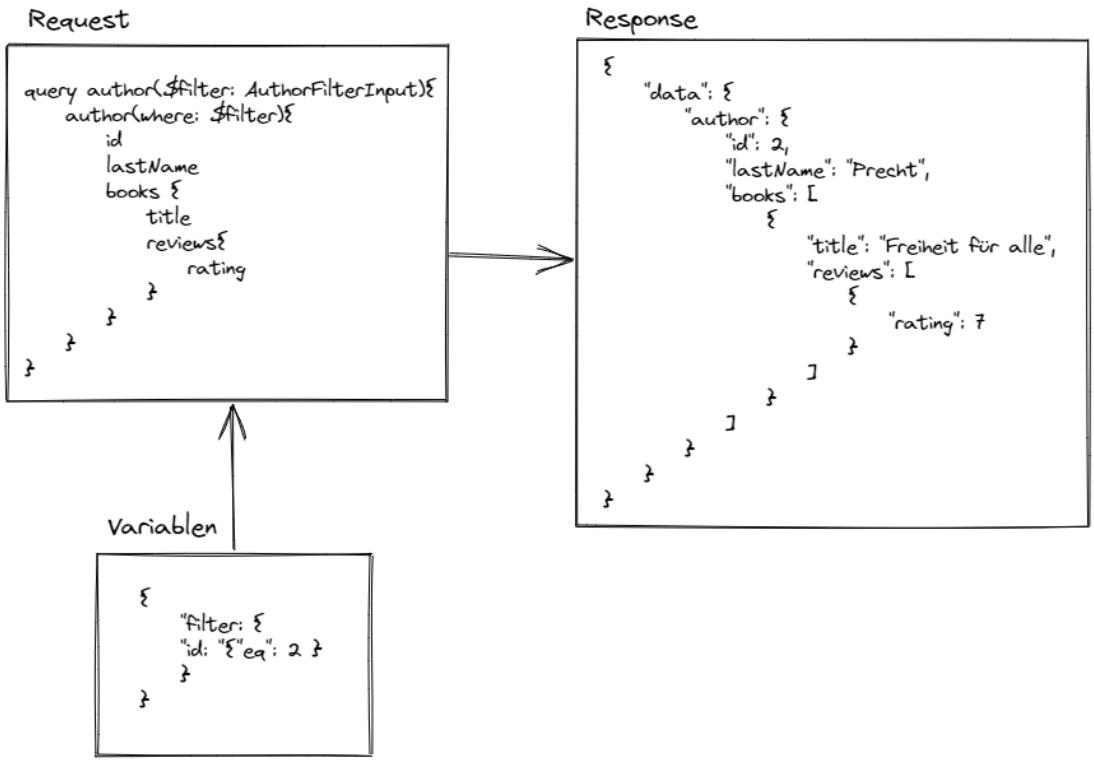

Abbildung 3.2: Anfrage eines Buches mit ID=2.

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!
```

```
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]. Das Ergebnis einer Mutation wird genauso behandelt wie das Ergebnis einer Query.
In der folgenden Abbildung ist eine Mutation zum Hinzufügen einer Bewertung abgebildet:

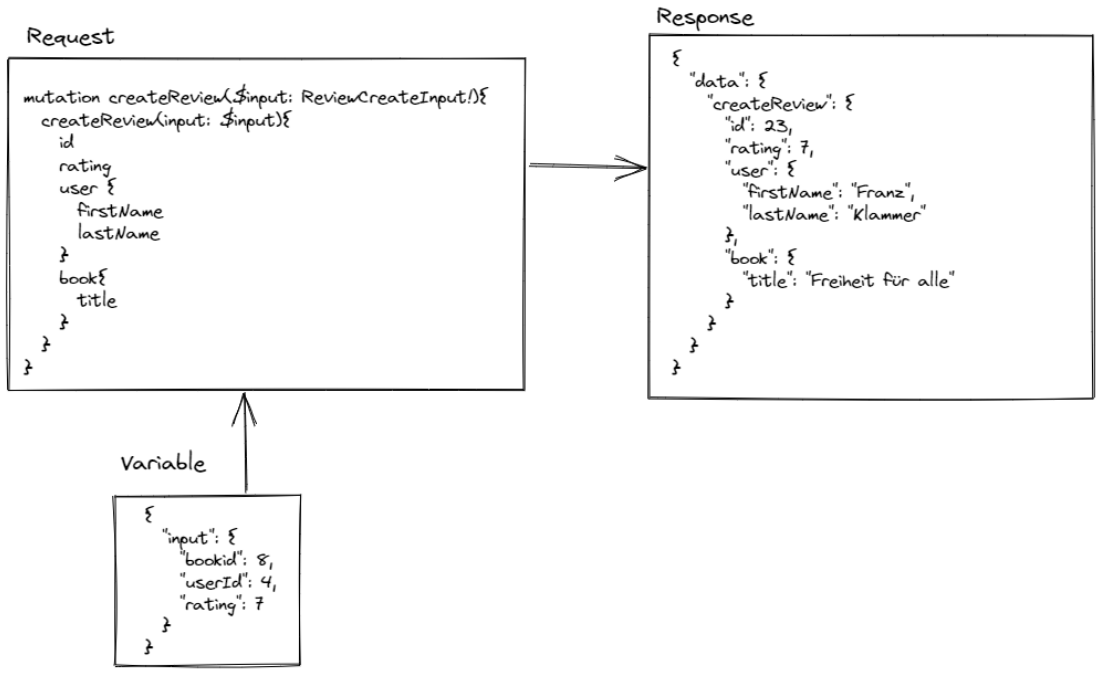

Abbildung 3.3: Anfrage eines Buches mit ID=2.

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

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 Datenobjekten 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.
Im folgenden Codebeispiel ist die Definition einer Subscription im Schema ersichtlich.

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

Die folgende Abbildung beschreibt die Verbindung eines Clients mittels einer Subscription zum GraphQL-Service.

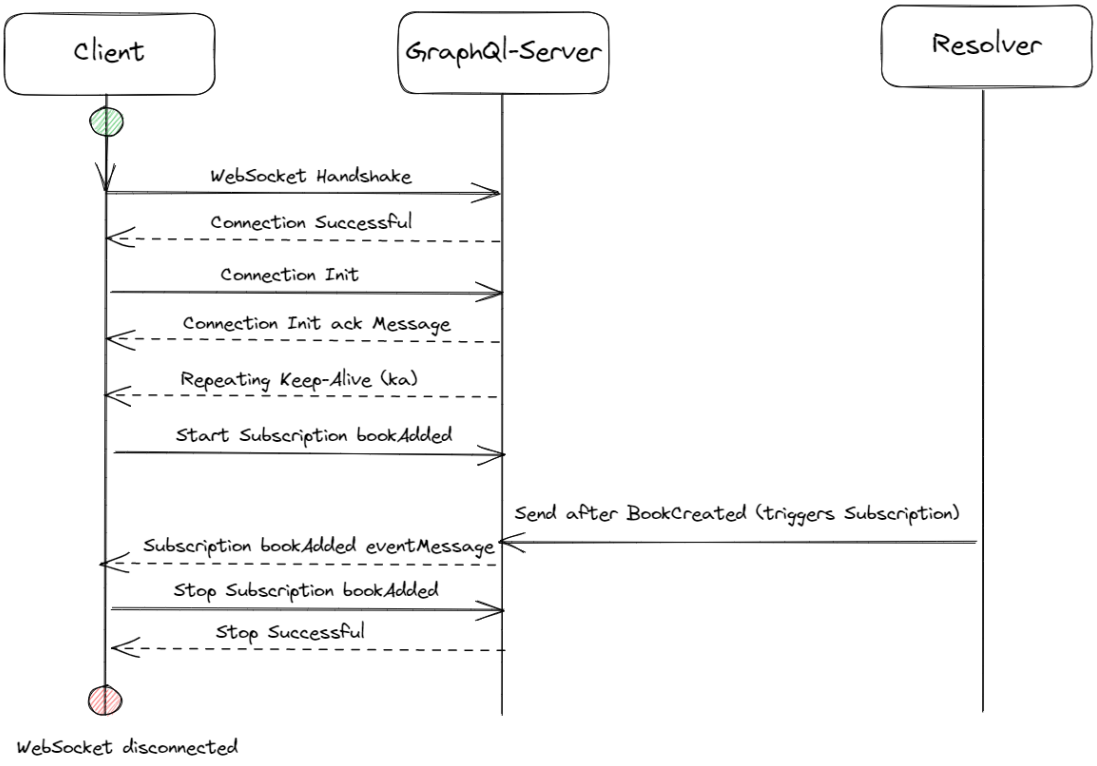

Abbildung 3.4: Subscription, die auf das Erstellen eines Buches wartet.

3.8 1 + n Problem

Das klassische Beispiel für Probleme mit Bezug auf die Performanz in GraphQL ist das „1 + n Problem“. In GraphQL sind Relationen einzelner Objekte zueinander in Graphen angeordnet. Das bedeutet, dass jedes Kindelement sein Elternelement kennt, jedoch keine weiteren. Gleichzeitig kennt jedes Elternelement nur die jeweiligen Kindelemente, nicht jedoch die Kinder der anderen Knoten auf der gleichen Ebene [Kress 2020, S. 104-105].

Das Problem wird nachfolgend anhand eines Beispiels näher erläutert. Ein GraphQL-Service bietet den Zugriff auf eine Liste von Büchern. Diese Bücher halten wiederum jeweils eine Liste von Bewertungen. Möchte ein Client nun alle Bücher und dessen Bewertungen ausgeben, würden die auf der Datenbank ausgeführten SQL-Statements folgendermaßen aussehen:

```
1 SELECT * from Books;
2 SELECT * from Rating WHERE BookId = 1;
3 SELECT * from Rating WHERE BookId = 2;
4 ...
5 SELECT * from Rating WHERE BookId = n;
```

Es wird also zuerst ein Datenbankzugriff auf die Tabelle Book ausgeführt. Anschließend werden n zusätzliche Datenbankzugriffe auf die referenzierten Bewertungen ausgeführt, was insgesamt in $1 + n$ Abfragen resultiert, wobei n der Anzahl der Bewertungen entspricht.

DataLoader

Mit dem Konzept des „DataLoaders“ kann dieses Problem gelöst werden. DataLoader ist eine Zusatzbibliothek von GraphQL und wurde ebenfalls von Facebook entwickelt [Kress 2020, S. 105]. Diese Bibliothek ist offiziell für die Performanzoptimierung von GraphQL-Services empfohlen. Sie ermöglicht das Bündeln von Resolver-Aufrufen zu einer einzelnen Anfrage. Der DataLoader sammelt bei einer Anfrage an den GraphQL-Endpunkt die IDs aller angeforderten Elemente eines Ressourcentyps [Kress 2020, S. 105]. Anschließend werden alle Ressourcen mit IDs aus dieser Liste geladen. Die Datenbankabfrage würde in unserem Beispiel wie folgt aussehen:

```
1 SELECT * from Books;
2 SELECT * from Rating WHERE BookId in (1, 2);
```

Damit können mithilfe des DataLoaders $n + 1$ Zugriffe auf zwei Datenbankzugriffe reduziert werden.

Kapitel 4

Entwicklung eines GraphQL-Service in .NET mit HotChocolate

In diesem Kapitel wird die prototypische Entwicklung eines GraphQL-Service beschrieben. Der Prototyp wurde mit .NET 6 und unter Zuhilfenahme der Bibilothek HotChocolate entwickelt wurde.

4.1 Anwendungsszenario

Die Auswahl eines neuen Buches ist oftmals ein sehr schwieriges Unterfangen. Bei der Entscheidungsfindung helfen oft Bewertungen von Kunden, die das jeweilige Buch schon gelesen haben. Daher 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 Design der 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 auf Grundlage des Anwendungsszenarios, Anwendungsfälle definiert. Diese Anwendungsfälle beschreiben die benötigten Zugriffe eines Clients auf das System. Zudem werden mit der Definition der Zugriffe auch die benötigen Entitäten definiert. Aus dem Anwendungsszenario abgeleitete Use-Cases werden in der folgenden Abbildung grafisch dargestellt:

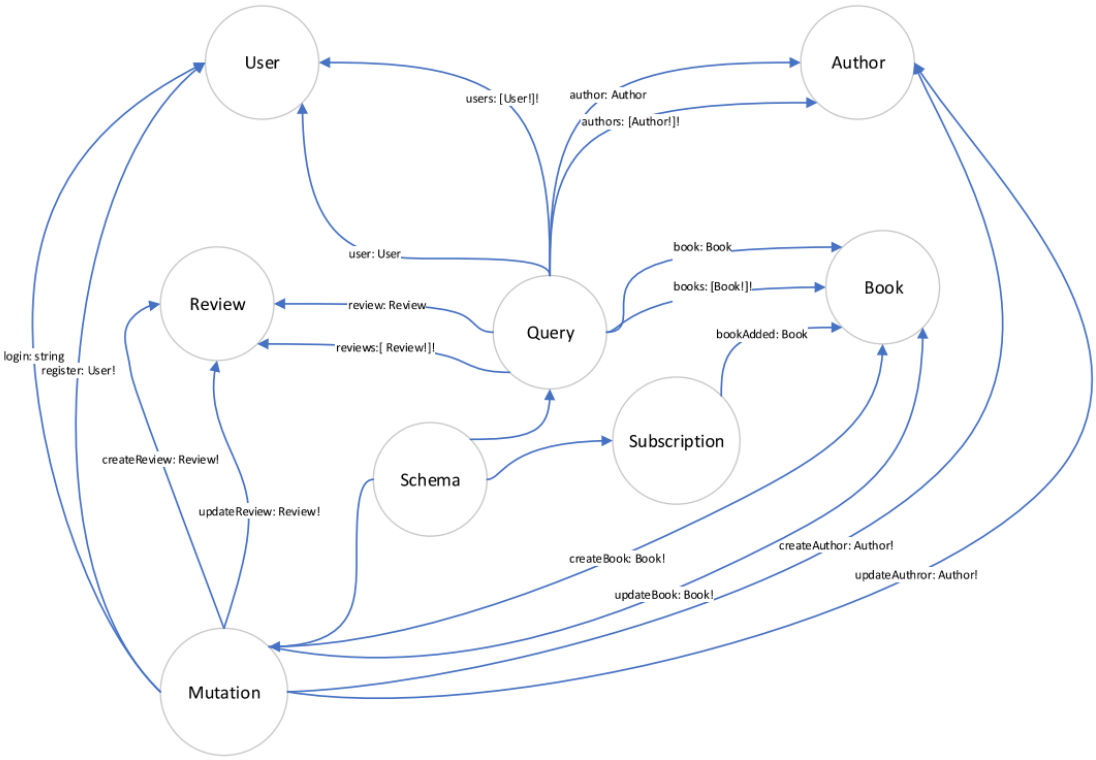

Abbildung 4.1: Use-Cases für das Beispielszenario.

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

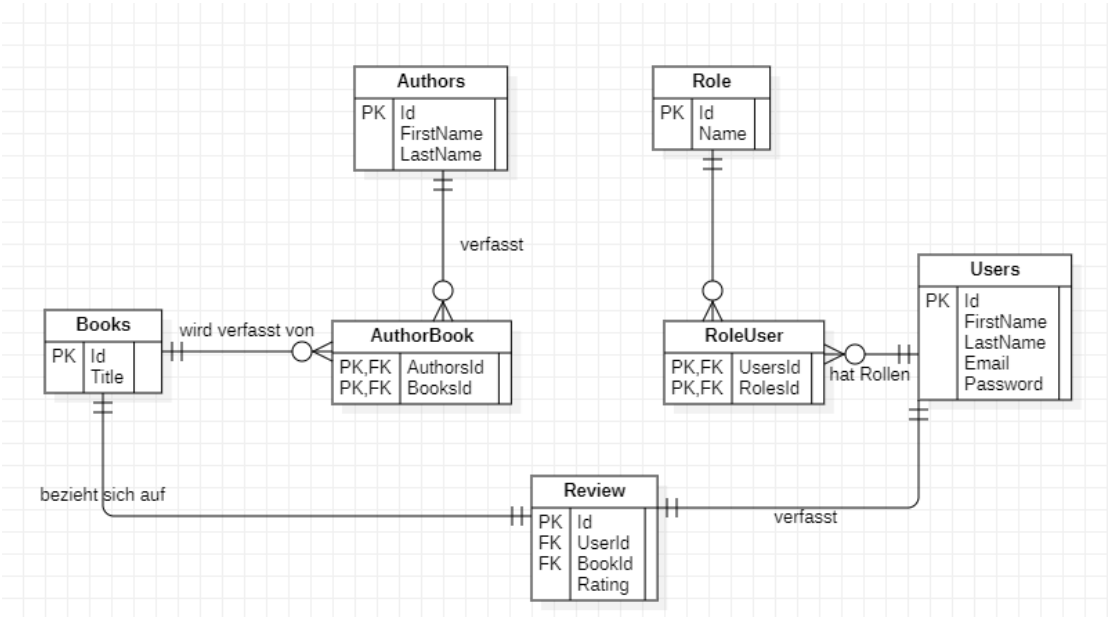

Abbildung 4.2: Datenbankschema

4.3 HotChocolate

Für die Implementierung des GraphQL-Service wurde das Framework HotChocolate herangezogen. Neben HotChocolate wird auch noch das Framework .NET GraphQL häufig eingesetzt. Die Entscheidung fiel auf HotChocolate aufgrund der besseren Dokumentation des Frameworks und der höheren Aktivität der Community.

HotChocolate allgemein

HotChoclate ist ein quelloffenes Framework zur Implementierung eines GraphQL-Service. Es ist konform zur GraphQL-Spezifikation implementiert. Damit ist HotChocolate kompatibel mit allen GraphQL konformen Clients. Der Entwurf des Schemas gehört zu den Kernaufgaben bei der Entwicklung eines GraphQL-Service.

Schema-Erstellung in HotChocolate

Es gibt drei Varianten, in HotChoclate ein Schema zu entwickeln: *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 Attribute um das Schema zu definieren. Pure-Code-First verwendet eine Fluent-API. Alle drei Varianten liefern dasselbe Schema, wobei man mit Pure-Code-First-Ansatz am meisten Kontrolle bei der Konfiguration hat. Der Prototyp wurde mittels der Pure-Code-First-Vorgehensweise umgesetzt.

Abhängigkeitsinjektion

Abhängigkeitsinjektion in HotChocolate funktioniert sehr ähnlich wie die Abhängig-

keitsinjektion von ASP.NET-Applikationen. Die Services werden, so wie die anderen Komponenten der Anwendung, zum Service-Container hinzugefügt:

```
builder.Services.AddTransient(typeof(IRepository<>), typeof(Repository<>));
builder.Services.AddTransient(typeof(IBookRepository), typeof(BookRepository));
builder.Services.AddTransient(typeof(IAuthorRepository), typeof(AuthorRepository));
builder.Services.AddTransient(typeof(IReviewRepository), typeof(ReviewRepository));
builder.Services.AddTransient(typeof(IBaseService<>), typeof(BaseService<>));
builder.Services.AddTransient<IBookService, BookService>();
builder.Services.AddTransient<IAuthorService, AuthorService>();
builder.Services.AddTransient<IReviewService, ReviewService>();
builder.Services.AddTransient<IAuthService, AuthService>();
```

HotChocolate unterstüzt aber nicht die Konstruktor-Injektion. Stattdessen wird die Abhängigkeitsinjektion in HotChoclate mittels Methoden-Injektion umgesetzt. Würde man die Abhängigkeitsinjektion mittels Konstruktor-Injektion implementieren, wäre jeder Service automatisch ein Singleton. Für die Methoden-Injektion liefert HotChocolate das Attribute [Service].

Im folgenden Quelltextausschnitt ist ersichtlich, wie Methoden-Injektion verwendet wird:

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

4.4 Architektur

Der Prototyp wurde mit der für REST-APIs üblichen Drei-Schicht-Architektur umgesetzt. Die Geschäftslogikschicht und Datenbankzugriffsschicht wurden laut dem *Program to an Interface not an Implementation*-Prinzip entwickelt, somit können die konkreten Implementierungen, je nach Bedarf, einfach ausgetauscht werden.

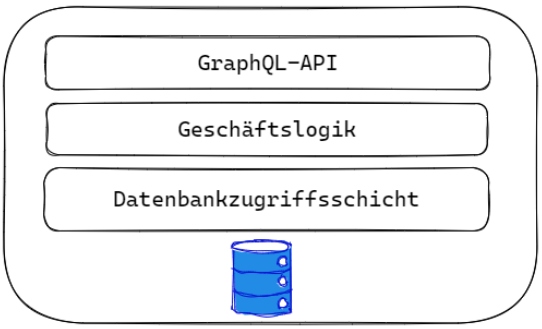

Abbildung 4.3: Drei-Schicht-Architektur

Abb 4.4. gibt einen Überblick über die Gesamtarchitektur. 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 ist ein Abbild des GraphQL-Schemas. Dieses Schema definiert, wie in Abschnitt 3.3 beschrieben, die verfügbaren Typen sowie die lesenden und schreibenden Zugriffe des GraphQL-Service.

4.4.2 Geschäftslogik

Die Geschäftslogik wird dem GraphQL-Schema mittels Resolvern zur Verfügung gestellt. Genaueres zu den Resolvern und zur Geschäftslogik folgt im Abschnitt 4.5. Die Geschäftslogik besteht aus *Service*-Klassen und zugehörigen Interfaces welche mit der Datenbankzugriffsschicht interagieren.

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*, der es Entwicklern ermöglicht, den Fokus auf eine höhere Abstraktionsebene zu legen. Diese Schicht wurde mit dem Repository-Pattern umgesetzt. Ein generisches Repository stellt die Grundfunktionalität für die anderen entitätsspezifischen Repositories zur Verfügung. Der folgende Quelltextausschnitt zeigt einen Teil der Implementierung der Grundfunktionalitäten:

```
1  public class Repository<TEntity> : IRepository<TEntity> where TEntity : BaseEntity {
2      protected readonly LibraryContext context;
3      public Repository(IDbContextFactory<LibraryContext> contextFactory) {
4          this.context = contextFactory.CreateDbContext();
5      }
6
7      public async Task<IQueryable<TEntity>> GetAsync(
8          Expression<Func<TEntity, bool>> filter = null,
9          params Expression<Func<TEntity, object>>[] includes)
10     {
11         IQueryable<TEntity> query = context.Set<TEntity>();
12
13         foreach (var include in includes) {
14             query = query.Include(include);
15         }
16
17         if (filter != null) {
18             query = query.Where(filter);
19         }
20
21         return query.AsQueryable();
22     }
23
24     public virtual async Task<TEntity> AddAsync(TEntity entity) {
25         if (entity is null) {
26             throw new ArgumentNullException(nameof(entity));
27         }
28
29         await context.AddAsync(entity);
30         await context.SaveChangesAsync();
```

```
31          return entity;
32      }
33
34      public virtual async Task<TEntity> UpdateAsync(TEntity entity) {
35          if (entity is null) {
36              throw new ArgumentNullException(nameof(entity));
37          }
38
39          context.Entry(entity).State = EntityState.Modified;
40
41          await context.SaveChangesAsync();
42          return entity;
43      }
44  }
```

Im folgenden Quelltextausschnitt wird die Implementierung des Repository für die Domänenklasse Book gezeigt. Es zeigt die notwendigen Überschreibungen der Basisimplementierung:

```
 1  public class BookRepository : Repository<Book>, IBookRepository {
 2      public BookRepository(IDbContextFactory<LibraryContext> contextFactory)
 3          : base(contextFactory) { }
 4
 5      public override async Task<Book> AddAsync(Book book) {
 6          book.Authors = await context.Authors
 7              .Where(author => book.Authors.Select(x => x.Id)
 8              .ToList().Contains(author.Id)).ToListAsync();
 9          return await base.AddAsync(book);
10      }
11
12      public override async Task<Book> UpdateAsync(Book book) {
13          var bookToUpdate = await GetFirstAsync(
14              b => b.Id == book.Id, b => b.Authors);
15          if (bookToUpdate == null) {
16              throw new ArgumentNullException(nameof(bookToUpdate));
17          }
18
19          bookToUpdate.Title = book.Title;
20
21          if (book.Authors.Count != bookToUpdate.Authors.Count
22              || !bookToUpdate.Authors.All(book.Authors.Contains))
23          {
24              bookToUpdate.Authors.UpdateManyToMany(book.Authors, b => b.Id);
25              bookToUpdate.Authors = await context.Authors
26              .Where(author => book.Authors.Select(a => a.Id)
27                              .ToList().Contains(author.Id))
28                              .ToListAsync();
29          }
30
31          return await base.UpdateAsync(bookToUpdate);
32      }
33  }
```

4.5 Resolver

Das Schema beschreibt, wie in Abschnitt 3.3 erwähnt, nur die verfügbaren Typen, Querys, Mutations und Subscriptions. Aus dem Schema ist aber die Generierung der Daten als auch die Manipulation dieser nicht abbildbar. 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

Resolver delegieren im Prototyp an entsprechende Methoden der Geschäftslogik, welche CRUD-Operationen der jeweiligen Entität zur Verfügung stellt. Jede Entität verfügt über einen *Service*. Jeder Service, wie zum Beispiel der *AuthorService*, leitet von einem generisch implementierten *BaseService* ab. Der *BaseService* implementiert dabei das Interface *IBaseService*.

```
1 public interface IBaseService<TEntity> where TEntity: BaseEntity {
2     public Task<IQueryable<TEntity>> GetAsync(
3         Expression<Func<TEntity, bool>> filter = null,
4         params Expression<Func<TEntity, object>>[] includes);
5     public Task<TEntity> GetFirstAsync(
6         Expression<Func<TEntity, bool>> filter = null,
7         params Expression<Func<TEntity, object>>[] includes);
8     public Task<TEntity> AddAsync(TEntity entity);
9     public Task<TEntity> UpdateAsync(TEntity entity);
10    public Task<bool> ExistsAsync(int id);
11    public Task RemoveAsync(TEntity entity);
12 }
```

Im obigen Quelltextausschnitt 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 {
3     protected readonly IRepository<TEntity> repository;
4
5     public BaseService(IRepository<TEntity> repository) {
6         this.repository = repository;
7     }
8     public virtual async Task<IQueryable<TEntity>> GetAsync(
9         Expression<Func<TEntity, bool>> filter = null,
10        params Expression<Func<TEntity, object>>[] includes)
11    {
12        return await repository.GetAsync(filter, includes);
13    }
14
15    public virtual async Task<TEntity> GetFirstAsync(
16        Expression<Func<TEntity, bool>> filter = null,
17        params Expression<Func<TEntity, object>>[] includes)
18    {
```

```
19          return await repository.GetFirstAsync(filter, includes);
20      }
21
22      public virtual Task<TEntity> AddAsync(TEntity entity) {
23          return repository.AddAsync(entity);
24      }
25      public virtual async Task<TEntity> UpdateAsync(TEntity entity) {
26          return await repository.UpdateAsync(entity);
27      }
28
29      public virtual async Task<bool> ExistsAsync(int id) {
30          return await repository.ExistsAsync(id);
31      }
32
33      public virtual async Task RemoveAsync(TEntity entity) {
34          await repository.RemoveAsync(entity);
35      }
36  }
```

Im obigen Quelltextausschnitt 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 4.6 näher erläutert.

4.6 Field Middleware

Field-Middleware erlaubt es, wiederverwendbare Logik vor oder nach der Ausführung des Resolvers auszuführen. Funktionalitäten wie Authentifizierung / Autorisierung, Sortieren, Filtern, Pagination und Projektionen werden dabei von HotChocolate als Middleware-Komponenten zur Verfügung gestellt. Man kann beliebig viele Middleware-Komponenten aneinanderreihen. Sie sind eine der fundamentalen Komponenten des Frameworks.

Reihenfolge Exekution Middleware

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

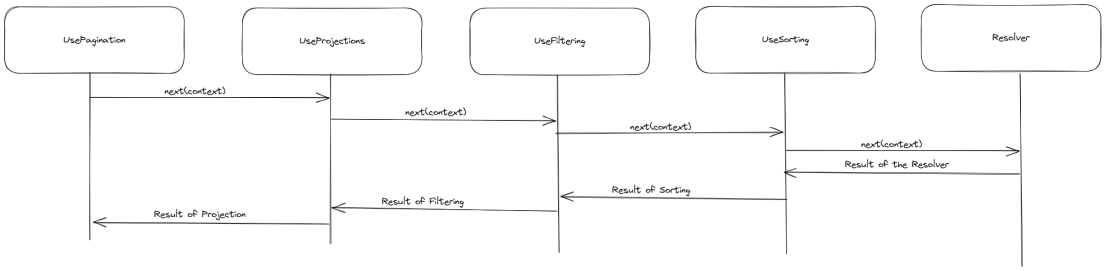

Abbildung 4.4: Ausführungsreihenfolge Middleware

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 Middleware-Komponenten erweitert.

Im Abschnitt 4.7 werden die verwendeten Middleware-Komponenten des Prototypen genauer erläutert.

Ausführung Middleware

Nachdem jene Middleware, welche als erstes definiert wurde, das Ergebnis der restlichen Middleware-Komponenten erhalten hat, wird die *LINQ*-Abfrage 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 Problem des Over- und Underfetchings löst und Filtern, Sortieren und Pagination ermöglicht.
Die folgenden Unterabschnitte 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 der bereits erwähnte Pure-Code-First-Ansatz verwendet.

```
1 public class AuthorQuery: ObjectType<Query> {
2   protected override void Configure(IObjectTypeDescriptor<Query> descriptor){
3     descriptor.Field("authors")
4         .ResolveWith<AuthorResolver>(r => r.Authors())
5         .Authorize()
6         .UseProjection()
7         .UseFiltering()
8         .UseSorting()
9         .Type<ListType<NonNullType<AuthorType>>>();
10    }
11 }
```

Im obigen Quelltextausschnitt ist zu sehen, dass die Wurzeloperation vom Typ ObjectType< *Query* > um 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. Die angeforderten Daten des Clients werden vom Resolver zur

Verfügung gestellt. Resolver werden im Abschnitt 4.5 näher eräutert.

In den folgenden Quelltextausschnitten wird das zur Laufzeit generierte Schema gezeigt:

```
type Query{
    authors(where: AuthorFilterInput order: [AuthorSortInput!]): [Author!]
    @authorize(apply: BEFORE_RESOLVER)
}

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

Nur der Objekt-Typ *Author* als auch die Wurzel-Operation der Query mit dem Feld *authors* wurden explizit generiert. Die Methodenaufrufe *Authorize()*, *UseFiltering()* und *UseSorting()* sind für die implizite Generierung, der im nachfolgenden Schema angegebenen Typen, verantwortlich. Diese Methodenaufrufe aktivieren die jeweilige Field-Middleware, diese werden in den nachfolgenden Abschnitten genauer erläutert.

```
input AuthorFilterInput {
    and: [AuthorFilterInput!]
    or: [AuthorFilterInput!]
    firstName: StringOperationFilterInput
    lastName: StringOperationFilterInput
    books: ListFilterInputTypeOfBookFilterInput
    id: ComparableInt32OperationFilterInput
}

input ListFilterInputTypeOfBookFilterInput {
  all: BookFilterInput
  none: BookFilterInput
  some: BookFilterInput
  any: Boolean
}

input AuthorSortInput {
    firstName: SortEnumType
    lastName: SortEnumType
    id: SortEnumType
}

input StringOperationFilterInput {
  and: [StringOperationFilterInput!]
  or: [StringOperationFilterInput!]
  eq: String
  neq: String
  contains: String
  ncontains: String
  in: [String]
  nin: [String]
  startsWith: String
  nstartsWith: String
  endsWith: String
```

```
35   nendsWith: String
36 }
37
38 input ComparableInt32OperationFilterInput {
39     eq: Int
40     neq: Int
41     in: [Int!]
42     nin: [Int!]
43     gt: Int
44     ngt: Int
45     gte: Int
46     ngte: Int
47     lt: Int
48     nlt: Int
49     lte: Int
50     nlte: Int
51 }
```

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 Abschnitt 4.10 ersichtlich.

Projektionen

Mit Projektionen liefert HotChocolate die Möglichkeit Over- und Underfetching zu verhindern. Das bedeutet, dass genau jene Daten, welche vom Client angefordert werden, in der Datenbank selektiert und anschließend an den Client zurückgeliefert werden. Dazu muss der Resolver ein Abfrageobjekt vom Typ IQueryable aber an das Framework übergeben. Dieses Abfrageobjekt beinhaltet zunächst nur jene Filterungen und Selektionen, welche von der Geschäftslogik festgelegt werden. HotChocolate erweitert dieses Abfrageobjekt nun durch jene Felder welche in der Query angefordert wurden.

Filtering

Aktiviert man die *Filtering*-Middleware für ein Query-Field, so kann man den implizit von HotChocolate bereitgestellten Filter-Input verwenden. Diese Middleware erweitert vom Resolver zurückgelieferte Abfrageobjekt um die gegebene Filterung und liefert das Ergebnis der vorangestellten Middleware zurück. Im obigen Quelltextausschnitt ist ersichtlich, dass die Filtering-Middleware zur Laufzeit das Input-Objekt AuthorFilterInput und die darin referenzierten Input-Objekte generiert. Weiters ist es möglich benutzerdefinierte Typen für die Filterung zu definieren.

Sorting

Aktiviert man die *Sorting*-Middleware für ein Query-Field, so kann man den implizit von HotChocolate bereitgestellten Sortier-Input verwenden. Diese Middleware fügt vom Resolver zurückgelieferten Abfrageobjekt 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* greift dabei auf die Geschäftslogik zu. Er greift auf den AuthorService, einer Kindklasse von BaseService, zu und dieser mithilfe der Datenbankzugriffsschicht auf die Datenbank. Wichtig dabei ist, dass der AuthorService, HotChocolate ein Abfrageobjekt zurückliefert. Das zurückgelieferte Abfrageobjekt wird vom Resolver an die restlichen Middleware-Komponenten zurückgereicht. Diese erweitern dann das Abfrageobjekt mit der von ihnen definierten Logik und werten es anschließend aus. Der AuthorResolver ist im folgenden Quelltextausschnitt ersichtlich:

```
1 public class AuthorResolver {
2     private readonly IAuthorService authorService;
3     private readonly IMapper mapper;
4 
5     public AuthorResolver(IAuthorService authorService, IMapper mapper) {
6         this.authorService = authorService;
7         this.mapper = mapper;
8     }
9 
10    public async Task<IQueryable<Author>> Authors() {
11        return await authorService.GetAsync();
12    }
13 
14    public async Task<Author> CreateAuthor([Service] IAuthorService authorService,
      AuthorCreate input) {
15        return await authorService.AddAsync(mapper.Map<Author>(input));
16    }
17 
18    public async Task<Author> UpdateAuthor([Service] IAuthorService authorService,
      AuthorUpdate input) {
19        return await authorService.UpdateAsync(mapper.Map<Author>(input));
20    }
21 
22    public async Task<Author> AuthorByDataLoader(int id, AuthorByIdDataLoader
      dataLoader, CancellationToken cancellationToken) {
23        return await dataLoader.LoadAsync(id, cancellationToken);
24    }
25 }
```

Ausführung Query und Ergebnis

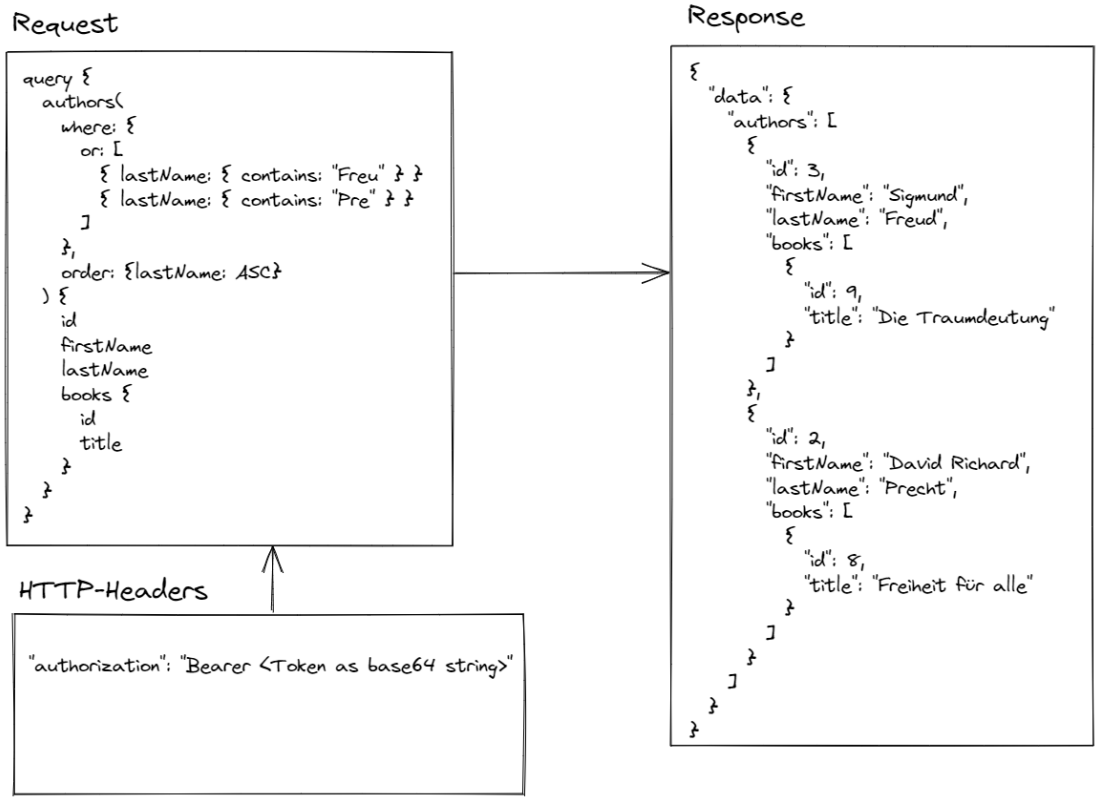

Abbildung 4.5: Ausführungsreihenfolge Middleware

Die obige Abbildung zeigt eine Query, welche alle Autoren mittels dem Feld *authors* abfrägt. Dabei sollen nur jene Autoren zurückgegeben werden deren Nachname entweder "Freuöder "Preënthält. 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 ((@__p_1
      LIKE N'') OR (CHARINDEX(@__p_1, [a].[LastName]) > 0))
9 ORDER BY [a].[LastName], [a].[Id], [t].[AuthorsId], [t].[BooksId]
```

Dieses SQL-Statement verdeutlicht, dass nur die tatsächlich abgefragten Daten in der Datenbank selektiert wurden.

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 BookUpateInput. Der Resolver BookResolver kümmert sich dabei um Ausführung der Operation.

```
1 public class BookUpdateInput: InputObjectType<BookUpdate> {
2     protected override void Configure(
3         IInputObjectTypeDescriptor<BookUpdate> descriptor)
4     {
5         descriptor.Field(f => f.Authors).Type<ListType<NonNullType<IntType>>>();
6     }
7 }
8
9 public class BookMutation: ObjectTypeExtension<Mutation>{
10     protected override void Configure(IObjectTypeDescriptor<Mutation> descriptor) {
11         descriptor.Field("updateBook")
12             .Authorize(new [] {"Admin", "Librarian"})
13             .Argument("input", a => a.Type<NonNullType<BookUpdateInput>>())
14             .ResolveWith<BookResolver>(r => r.UpdateBook(default, default))
15             .Type<BookType>();
16     }
17 }
```

Der obige Code hat die Generierung des folgenden Schemas zur Folge:

```
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!]!
```

```
  reviews: [Review!]!
  id: Int!
}

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

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!
}

type Role {
  name: String!
  users: [User!]!
  id: Int!
}

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

Im generierten Schema ist zu erkennen, dass der Typ *Book* alle 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 wie folgt deklariert:

```
public class BookUpdateInput: InputObjectType<BookUpdate> {
    protected override void Configure(
        IInputObjectTypeDescriptor<BookUpdate> descriptor)
    {
        descriptor.Field(f => f.Authors).Type<ListType<NonNullType<IntType>>>();
    }
```

```
 7 }
 8
 9 public class BookUpdate {
10     public int Id { get; set; }
11     public string Title { get; set; }
12     public ICollection<int> Authors { get; set; }
13 }
```

Bei der Definition des Input-Objekts *BookUpdateInput* bietet die Methode *Configure* der Basisklasse *ObjectType* wiederum die Möglichkeit, genauere Definitionen vorzunehmen. Im obigen Quelltextausschnitt ist ersichtlich, dass das Feld *authors* eine Liste mit *Int*-Objekten enthält, die nicht *null* sein dürfen.

Anders als die im Datenbankschema beschriebene Entität Book hält das Input-Objekt BookUdpateInput keine Liste des Typs Author, sondern nur eine Liste von *Int*-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 diese auf Datenübertragungsklassen abgebildet werden. Dafür wird im Prototyp die Bibliothek *AutoMapper* verwendet.

Im folgenden Quelltextausschnitt 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(
 7                     src => src.Authors.Select(
 8                         id => new Author { Id = id })));
 9     }
10
11 }
```

Zum Vergleich hier noch die Klasse Book, welche das Domänenobjekt Book 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

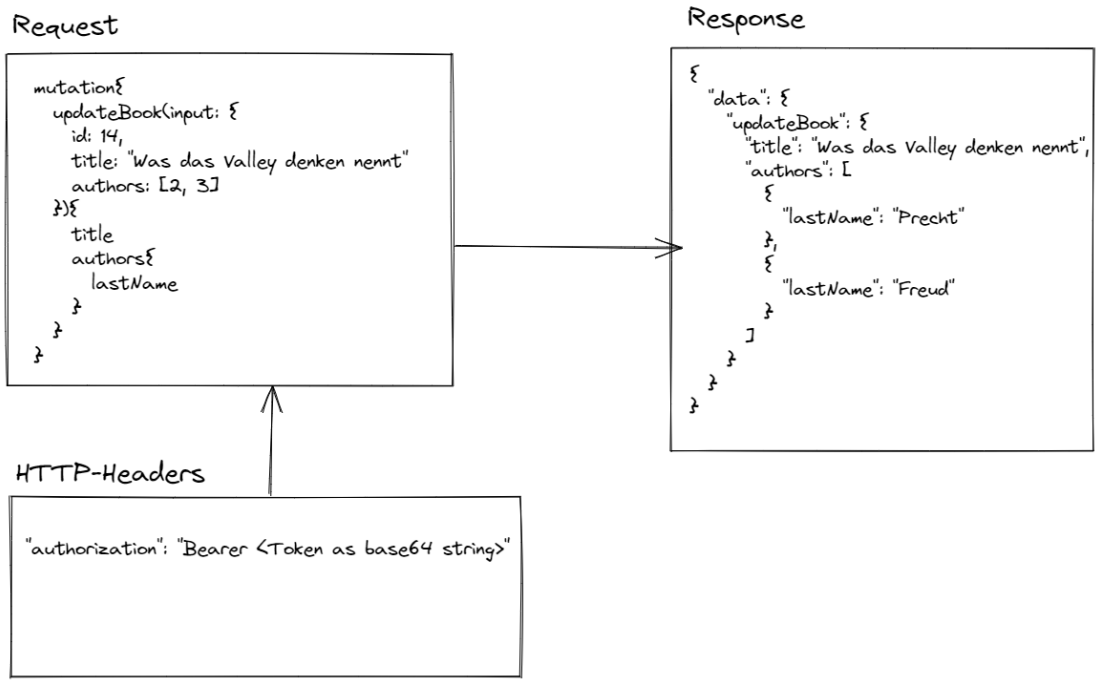

Abbildung 4.6: Schemadefinition

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 zwei 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, muss folgende Methode implementiert werden:

```
1 public async Task<Book> CreateBook(
2     [Service] IBookService bookService,
3     BookCreate input,
4     [Service] ITopicEventSender sender)
5 {
6     var book = await bookService.AddAsync(mapper.Map<Book>(input));
7     await sender.SendAsync("bookAdded", book);
8     return book;
9 }
```

In Zeile 3 des obigen Quelltextausschnittes 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 der Pure-Code-First-Ansatz von GraphQL verwendet. Die Wurzeloperationen Subscription wird dabei um ein Feld *bookAdded* erweitert. Die Umsetzung ist dabei in folgendem Quelltextausschnitt ersichtlich:

```
1 public class BookSubscription: ObjectTypeExtension<Subscription> {
2     protected override void Configure(
3         IObjectTypeDescriptor<Subscription> descriptor)
4     {
5         descriptor
6         .Field("bookAdded")
7         .Type<BookType>()
8         .Resolve(context => context.GetEventMessage<Book>())
9         .Subscribe(async context => {
10            var receiver = context.Service<ITopicEventReceiver>();
11            return await receiver.SubscribeAsync<string, Book>("bookAdded");
12        });
13    }
14 }
```

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

Der oben angeführte Code generiert 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 in Abschnitt 3.7 eingegangen. Das folgende Bild zeigt das Ergebnis einer Subscription, welche von der Erstellung eines Buches notifiziert wurde.

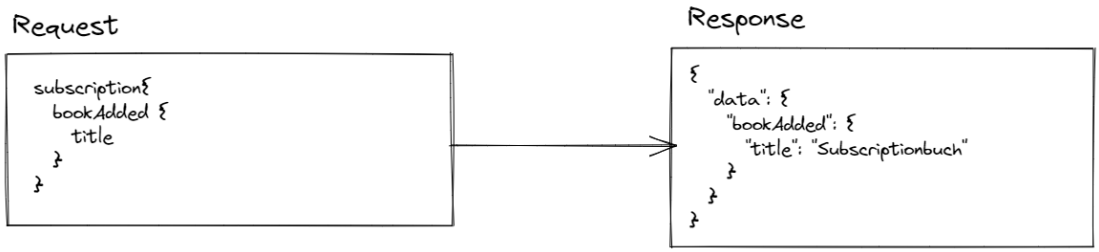

Abbildung 4.7: Subscription welche von der Erstellung eines Buches benachrichtigt wird.

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 drei Rollen für Benutzer: *User*, *Librarian* und *Admin*. Jede Rolle kann nur auf für Sie freigegebene Ressourcen zugreifen.

Die Authentifizierung und Autorisierung für den Prototyp 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.

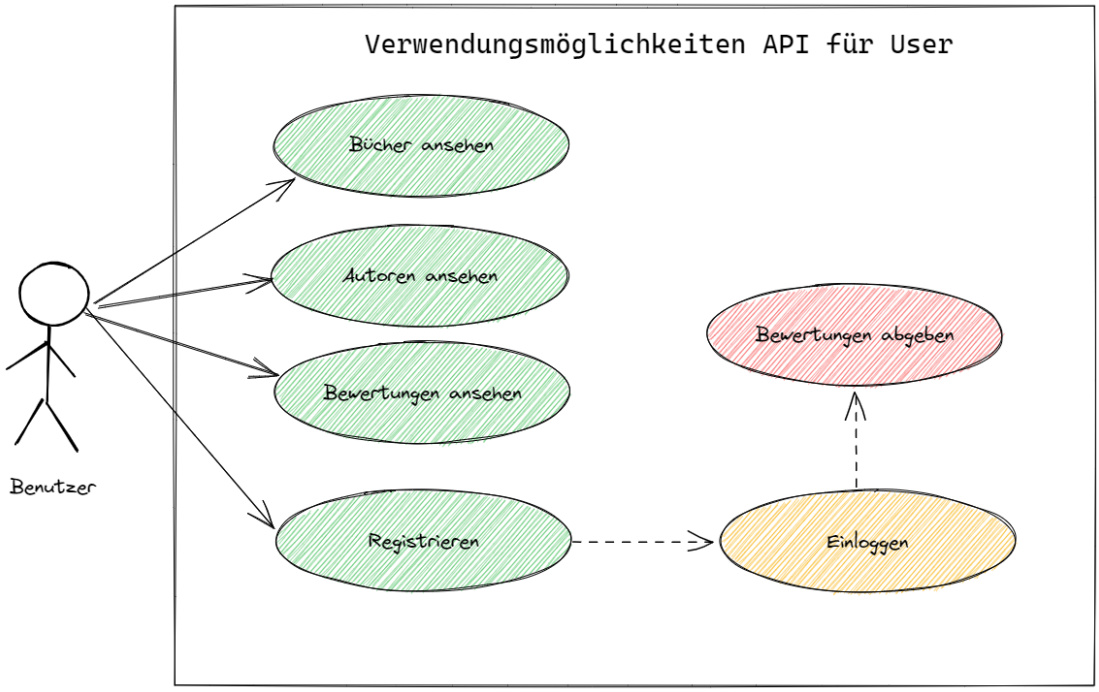

Abbildung 4.8: Rechte der Rolle *User*.

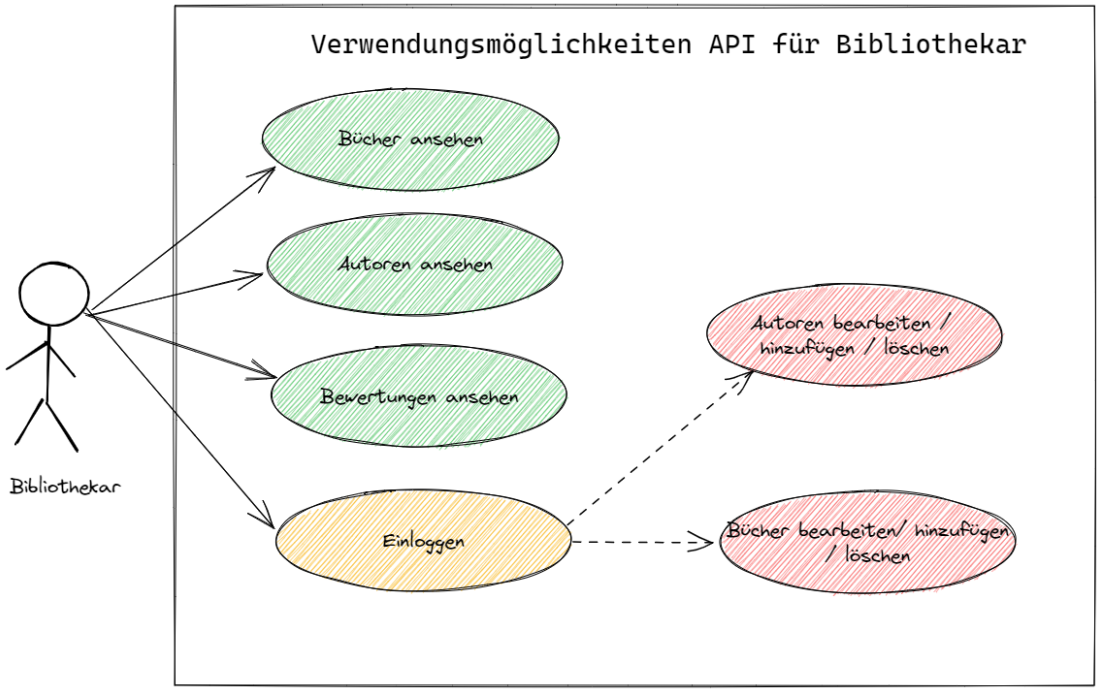

Abbildung 4.9: Rechte der Rolle *Librarian*.

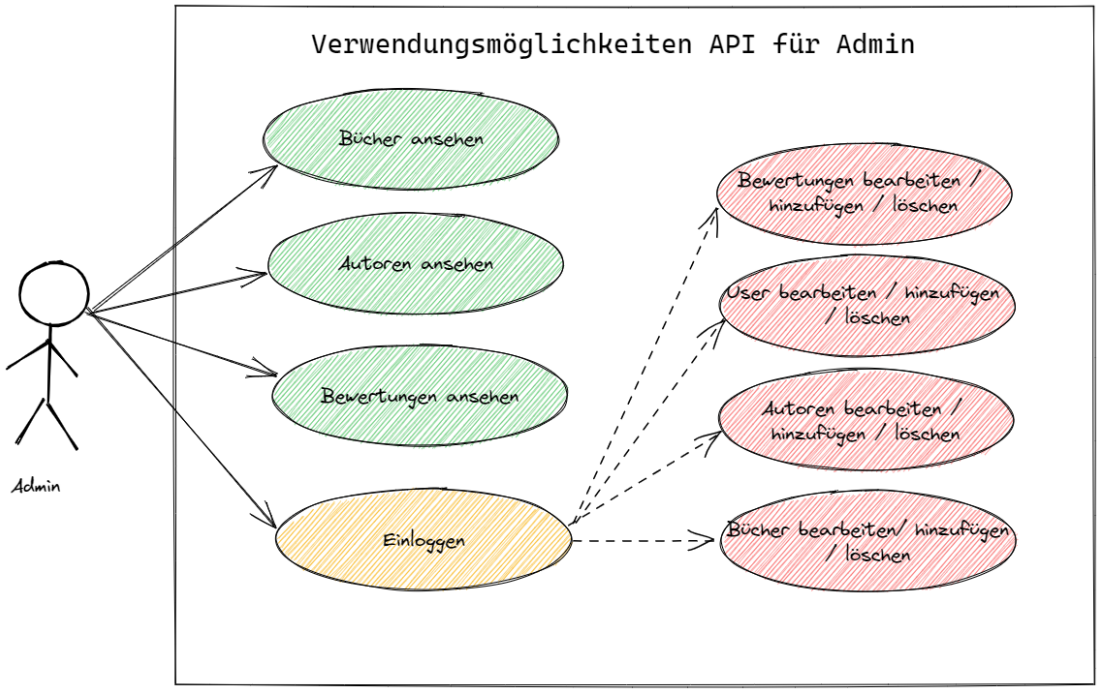

Abbildung 4.10: Rechte der Rolle *Admin*.

Generierung JWT-Token

Um die Authentifizierung mittels JWT-Token zu ermöglichen, muss diese Authentifizierungsform erst registriert werden. Diese Registrierung erfolgt, wie in .NET üblich, im *WebApplicationBuilder*.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(options => {
    var tokenSettings = configuration.GetSection("JWT").Get<TokenSettings>();
    options.TokenValidationParameters = new TokenValidationParameters {
        ValidIssuer = tokenSettings.Issuer,
        ValidateIssuer = true,
        ValidAudience = tokenSettings.Audience,
        ValidateAudience = true,
        IssuerSigningKey = new SymmetricSecurityKey(
                                Encoding.UTF8.GetBytes(tokenSettings.Key)),
        ValidateIssuerSigningKey = true
    };
});
```

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, 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 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, weswegen für die Implementierung eine Mutation gewählt wurde.

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

```
public async Task<string> Login(string email, string password) {
   var user = await userRepository
            .GetFirstAsync(user => user.Email.Equals(email), user => user.Roles);
   if (user is not null&&BCrypt.Net.BCrypt.Verify(password, user.Password)){
       var securtityKey = new SymmetricSecurityKey(
                                Encoding.UTF8.GetBytes(tokenSettings.Key));

       var credentials = new SigningCredentials(
                                securtityKey, SecurityAlgorithms.HmacSha256);

       var claims = new List<Claim>();

       claims.Add(new Claim("FirstName", user.FirstName));
       claims.Add(new Claim("LastName", user.LastName));
       claims.Add(new Claim("Email", user.Email));
       if (user.Roles?.Count > 0) {
           foreach (var role in user.Roles) {
```

```
18                claims.Add(new Claim(ClaimTypes.Role, role.Name));
19            }
20        }
21
22        var jwtSecurityToken = new JwtSecurityToken(
23            issuer: tokenSettings.Issuer,
24            audience: tokenSettings.Audience,
25            expires: DateTime.Now.AddDays(1),
26            signingCredentials: credentials,
27            claims: claims
28        );
29
30        return new JwtSecurityTokenHandler().WriteToken(jwtSecurityToken);
31    }
32
33 return "";
34 }
```

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: *FirstName*, *LastName*, *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         descriptor.Field("authors")
4             .ResolveWith<AuthorResolver>(r => r.Authors())
5             .Authorize()
6             .Type<ListType<NonNullType<AuthorType>>>();
7     }
8 }
```

In dem oben stehenden Code wird festgelegt, dass ein User ein valides *JWT-Token* an den 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.

4.11 1 + n Problem

Die Definition dieses Problems wurde im Abschnitt 3.8 erläutert. Dieser Abschnitt implementiert eine Lösung des 1 + n Problems mittels eines DataLoaders, welcher von HotChoclate zur Verfügung gestellt wird.

Die folgende Abfrage ermöglicht es dem Client, die Daten von zwei Autoren mittels einer GraphQL-Abfrage zu ermitteln.

```
query {
    a: authorByDataLoader(id: 1) {
        id
        firstName
        lastName
    }

    b: authorByDataLoader(id: 2) {
        id
        firstName
        lastName
    }
}
```

Das Problem, welches der GraphQL-Service hat, besteht darin, dass der Resolver kein Wissen über die Abfrage-Query als Ganzes hat. Der Resolver besitzt also nicht die Information, ob er mehrfach parallel aufgerufen wird, um ähnliche oder gleiche Daten aus derselben Datenquelle abzufragen. Der DataLoader hat die Aufgabe, diese einzelnen Resolveraufrufe in einen einzigen Datenbankzugriff zusammenzuführen.

Implementierung DataLoader

In diesem Abschnitte wird ein DataLoader für Autoren implementiert. Für die Implementierung wird die Pure-Code-First-Vorgehensweise verwendet. Für die Realsierung sind folgende Schritte erforderlich:

Um einen DataLoader mit HotChoclate zu implementieren muss eine neue Klasse erstellt werden welche von *BatchDataLoader* ableitet:

```
public class AuthorByIdDataLoader : BatchDataLoader<int, Author> {
    private readonly IAuthorService authorService;

    public AuthorByIdDataLoader
        (IBatchScheduler batchScheduler, IAuthorService authorService,
        DataLoaderOptions? options = null) : base(batchScheduler, options)
    {
        this.authorService = authorService;
    }

    protected override async Task<IReadOnlyDictionary<int, Author>> LoadBatchAsync(
        IReadOnlyList<int> keys, CancellationToken cancellationToken)
    {
        return (await authorService.GetAsync(a => keys.Contains(a.Id)))
               .ToDictionary(t => t.Id);
    }
}
```

In der Methode *LoadBatchAsync* werden alle Autoren anhand ihrer IDs von der *GetAsync*-Methode des *AuthorService* selektiert. Die *GetAsync*-Methode des *AuthorService* erwartet dabei einen LINQ-Ausdruck.

Zusätzlich muss im *AuthorResolver* der DataLoader registriert werden:

```
public async Task<Author> AuthorByDataLoader(
    int id,
    AuthorByIdDataLoader dataLoader,
    CancellationToken cancellationToken)
{
    return await dataLoader.LoadAsync(id, cancellationToken);
}
```

Anschließend muss die *AuthorQuery* in der *Configure*-Methode um das folgende Feld erweitert werden:

```
descriptor.Field("authorByDataLoader")
    .UseProjection()
    .ResolveWith<AuthorResolver>(
        g => g.AuthorByDataLoader(default, default, default))
    .Argument("id", a => a.Type<NonNullType<IntType>>())
    .Type<AuthorType>();
```

Würde die oben angeführte Anfrage ohne DataLoader ausgeführt werden, so würde sie in zwei seperaten Datenbankabfragen münden. Der DataLoader kümmert sich darum, dass daraus eine einzelene Datenbankabfrage wird. Dafür werden die Daten, nicht wie in Abschnitt 4.7 direkt vom Resolver abgefragt, sondern vom DataLoader. Dieser kümmert sich darum, dass die einzeln abgefragten Autoren zu einer Liste aus Schlüssel-Wert-Paaren zusammengeführt werden. Anschließend führt er diese Abfrage mittels des *AuthorService* auf der Datenbank aus.

Die Implementierung des *AuthorByDataLoader* mündet anschließend in folgendem Datenbankzugriff:

```
SELECT [a].[Id], [a].[FirstName], [a].[LastName]
FROM [Authors] AS [a]
WHERE [a].[Id] IN (1, 2)
```

Wie an diesem SQL-Statement ersichtlich ist, werden die einzelnen Abfragen zu einer zusammengeführt.

Kapitel 5

Zusammenfassung

In diesem Kapitel werden die behandelten Themen und die wichtigsten Erkenntnisse zusammengefasst.
Die Spezifikation von GraphQL definiert Entwurfsprinzipien, macht aber keine Vorgaben zur Implementierung von GraphQL-Services. GraphQL beschreibt eine konkrete Architektur, REST wiederum beschreibt laut Fielding einen Architekturstil.

Wenn es darum geht komplexe strukturierte Daten zu verwalten, schneidet GraphQL besser ab. Hier ist vor allem anzuführen, dass GraphQL Over- und Underfetching per Design verhindert. GraphQL gibt, anders als REST-APIs, die Struktur der Daten nicht selbst vor, sondern lässt den Client die Struktur der benötigten Daten festlegen. Dadurch bekommt der Client immer genau jene Daten, die er benötigt. Weiters beeinflusst GraphQL die Performanz des Clients positiv, indem der Netzwerkverkehr auf das Notwendigste reduziert wird. Davon profitieren vor allem mobile Applikationen.

Die Zugriffsmöglichkeiten (Query, Mutation, Subscription) und die von GraphQL verwalteten Objekte werden in einem Schema definiert. Aus dem Schema ist weder die Generierung der Daten noch die Manipulation dieser ableitbar. Die Generierung und Manipulation der Daten übernehmen in GraphQL-Resolver, die je nach Frameowrk unterschiedlich implementiert werden. Die Resolver stellen das Bindeglied zur Geschäftslogik dar. Dadurch ist es möglich, einen GraphQL-Service neben einem bereits bestehenden REST-Service zu betreiben. Die Resolver verwenden dabei die bereits bestehende Geschäftslogik.

Es konnte ein prototypischer GraphQL-Service in .NET 6 mittels dem Framework Hot-Chocolate erfolgreich umgesetzt werden. Dieser Prototyp bietet Clients die Möglichkeit lesend als auch schreibend auf die verwalteten Daten zuzugreifen. Weiters bietet der Prototyp bidirektionale Kommunikation zwischen Client und Server mittels Subscriptions. Der Endpunkt des GraphQL-Client wurde gegen unbefugte Zugriffe mit einem JWT-Token abgesichert. Dabei wurde auch die Autorisierung mittels Rollen, die im

JWT-Token gespeichert werden, realisiert. Der Prototyp implementiert eine Lösung für das „1 + n Problem“ mittels eines Dataloaders.

Quellenverzeichnis

Literatur

Berlind, David (2017). *APIs Are Like User Interfaces–Just With Different Users In Mind* (siehe S. 3).

Cotton, Ira W und Frank S Greatorex Jr (1968). „Data Structures and Techniques for Remote Computer Graphics“. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, S. 533–544 (siehe S. 3).

Fielding, Roy Thomas (2000). *Architectural Styles and the Design of network-based Software Architectures*. University of California, Irvine (siehe S. 4–6).

Kress, Dominik (2020). *GraphQL: Eine Einführung in APIs mit GraphQL*. dpunkt. verlag (siehe S. 3, 6, 9–11, 13, 14, 16, 19, 21).

Wheeler, David J (1952). „The use of sub-routines in programmes“. In: *Proceedings of the 1952 ACM national meeting (Pittsburgh)*, S. 235–236 (siehe S. 3).

Online-Quellen

Facebook (15. Nov. 2021). *Graphql*. URL: http://spec.graphql.org/June2018/ (besucht am 15. 11. 2021) (siehe S. 7, 9–14).

Ganatra, Ronak (11. Nov. 2021). *GraphQL Vs. REST APIs*. URL: https://graphcms.com/blog/graphql-vs-rest-apis (besucht am 27. 05. 2022) (siehe S. 3, 4).

Rakuten (15. Nov. 2021). *Graphql vs Rest*. URL: https://blog.api.rakuten.net/graphql-vs-rest/ (besucht am 15. 11. 2021) (siehe S. 6, 7).