GraphQL für Java-Anwendungen

Nils Hartmann

GraphQL ist eine Sprache zur Abfrage von Daten, die mit dem Versprechen antritt, sowohl einfach in der Entwicklung als auch effizient in der Laufzeit zu sein. Dieser Artikel stellt die Sprache vor und zeigt, wie man dafür ein API für die eigene Java-Anwendung bauen kann.

Die Abfragesprache GraphQL steht seit 2015 als Open-Source-Lösung zur Verfügung. Ursprünglich von Facebook konzipiert und veröffentlicht, wird die Sprache seit Ende 2018 von einem Konsortium, der GraphQL Foundation *[1]*, weiterentwickelt und spezifiziert. GraphQL selbst ist kein fertiges Produkt; in der Spezifikation ist im Wesentlichen beschrieben, wie Abfragen aussehen müssen und wie diese von einem Server bearbeitet und beantwortet werden müssen. Möchte man für die eigene Anwendung eine GraphQL-Schnittstelle bereitstellen, kann man auf Frameworks und Tools für diverse Sprachen zurückgreifen, die einem bei der Implementierung helfen. Für Java gibt es das „graphql-java"-Projekt *[2]*, das später in diesem Artikel vorgestellt wird.

Zunächst aber noch zu den konzeptionellen und implementierungsunabhängigen Aspekten von GraphQL. Die zentrale Idee von GraphQL ist, dass ein Client immer genau die Daten von einer Anwendung abfragen kann, die er für einen Use-Case, zum Beispiel eine Ansicht im Frontend, benötigt. Dadurch können Netzwerk-Requests und übertragenes Datenvolumen optimiert werden, da der Client prinzipiell niemals zu viele oder zu wenige Daten abfragen muss. Gleichzeitig soll durch den Einsatz der Sprache aber auch die Entwicklung möglichst einfach sein, da sie es erlaubt, Anbieter und Verwender des API zu entkoppeln. Der Server kann nämlich einfach beliebige Daten zur Verfügung stellen, der Client ist aber nicht gezwungen, diese ganz zu konsumieren. Vielmehr pickt sich der Client je nach Anwendungsfall genau die Teile der Daten heraus, die er für den Anwendungsfall benötigt. Der Server kann somit Daten „auf Vorrat" zur Verfügung stellen und auch bestehende Daten erweitern, ohne dass davon bestehende Clients betroffen wären.

Ein praktisches Beispiel

Um die Konzepte von GraphQL praktisch zu verdeutlichen, soll eine kleine Beispiel-Anwendung, der BeerAdvisor, dienen. Der Quellcode der Anwendung steht auf GitHub zur Verfügung. Mit dieser Anwendung können Nutzer eine Reihe von Bieren bewerten und sich ansehen, wo diese Biere erhältlich sind. Andere Nutzer können die abgegebenen Bewertungen einsehen *(siehe Abbildung 1)*.

Der BeerAdvisor besteht clientseitig aus drei Ansichten, die jeweils auf unterschiedliche Ausschnitte aus dem Domain Model zugreifen. Die Übersichtsseite zeigt beispielsweise nur den Namen der Biere (Entity „Beer") und deren durchschnittliche Bewertung an. Die Detailansicht eines Bieres stellt neben diesen Informationen auch den Preis des Bieres dar, darüber hinaus aber auch dessen Bewertungen (aus den Entities „Rating" und „User" im Domain Model) und die Namen der Geschäfte (Entity „Shop"), in denen es erhältlich ist. Die Detail-Seite eines Geschäfts wiederum zeigt Informationen zu einem Geschäft (etwa dessen Adresse) sowie die dort erhältlichen Biere – davon aber nur jeweils den Namen und nicht deren Preis, Bewertungen etc. Mit GraphQL kann die Anwendung für jede der Ansichten genau die Informationen aus dem Model abfragen, die sie dafür jeweils benötigt.

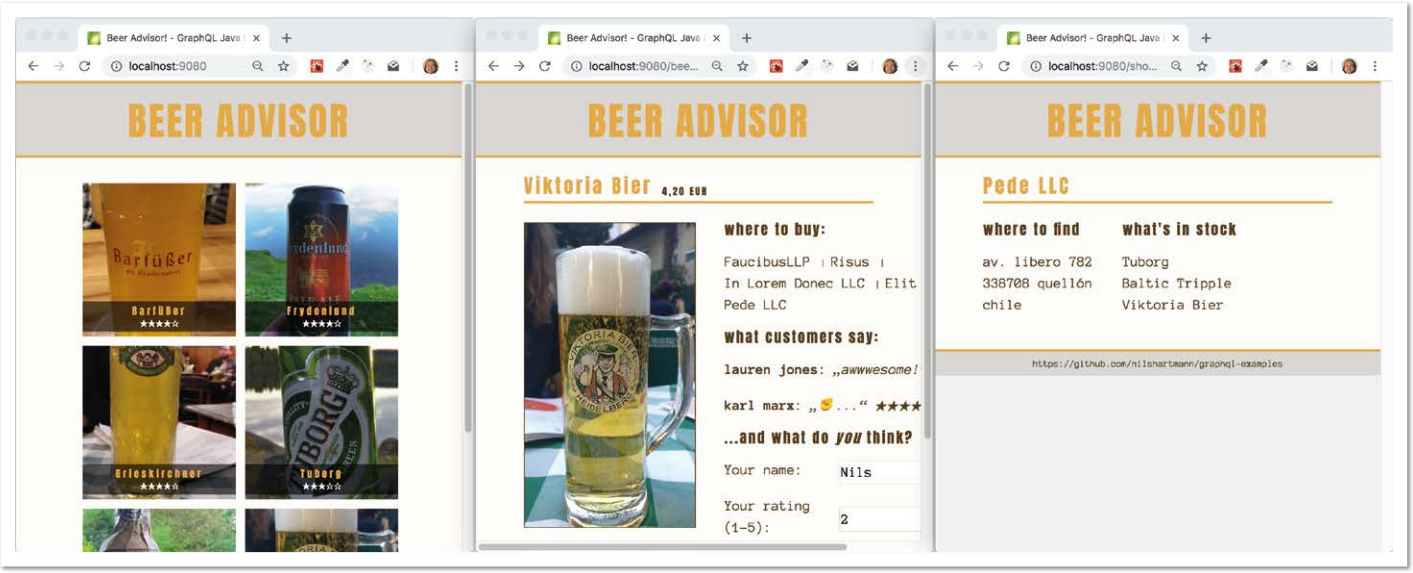

Abbildung 1: Die Beispielanwendung BeerAdvisor

```
query OverviewPageQuery {
  beers {
     name
     averageStars
  }
}
```

Listing 1

```
query BeerViewQuery {
  beer(beerId: "B1") {
     name
     ratings {
        comment
        author {
           name
        }
     }
     shops { . . . }
  }
}
```

Listing 2

```
curl -X POST -H "Content-Type: application/json" \
     -d ,{"query":"{ beers { id name } }"}‘
     http://localhost:9000/graphql
```

Listing 3

Mit GraphQL Daten abfragen

Eine GraphQL-Abfrage besteht immer aus einer oder mehreren „Operationen".

Eine Operation kann eine „Query" sein, um Daten zu lesen, eine „Mutation", um Daten zu schreiben bzw. zu verändern, oder eine „Subscription", wenn der Client sich für Events registrieren möchte, die der Server veröffentlicht.

Die Abfrage wird in einer JSON-artigen Notation formuliert, in der Daten vom API selektiert werden, in GraphQL „Felder" genannt.

Listing 1 zeigt die Query, die Namen (Feld „name") und durchschnittliche Bewertung (Feld „averageStars") für alle „Beer"-Objekte lädt, um die Übersichtsseite der Anwendung darzustellen:

Wie dort beschrieben, benötigt die Detailansicht eines Bieres andere Daten, beispielsweise die Bewertungen sowie deren Autoren. Die folgende Query *(Listing 2)* fragt diese Daten ab. Neu in dieser Query ist, dass Felder aus einer Hierarchie von Objekten abgefragt werden. Außerdem erhält das Feld „beer" ein Argument (ähnlich wie das in Java bei Methoden möglich ist).

Eine Abfrage in GraphQL besteht folglich aus einer hierarchischen Struktur von Feldern, die an einen GraphQL-fähigen Server geschickt werden muss. Das erfolgt in der Regel per HTTP-POST-Request an einen zentralen Endpunkt; eine Versionierung des API gibt es nicht, da dieses abwärtskompatibel erweitert werden kann. Der Endpunkt nimmt die Anfrage entgegen, verarbeitet sie und liefert das Ergebnis zurück. *Listing 3* zeigt beispielhaft, wie man mit dem Kommandozeilen-Tool „curl" eine GraphQL-Abfrage ausführen kann.

Das Ergebnis der Abfrage ist ein JSON-Objekt, das auf Root-Ebene ein „data"-Feld enthält. Dieses Feld enthält die abgefragten Daten, deren Hierarchie identisch mit der Abfrage ist. Der Client weiß daher durch die Formulierung seiner Abfrage, wie die Antwort vom Server strukturell aussieht. Eine exemplarische Antwort auf die oben gezeigte „BeerViewQuery" ist in *Abbildung 2* zu sehen – die Hierarchie unterhalb von „data" entspricht genau der abgefragten Struktur der Query.

Um Daten auf dem Server zu verändern, führt der Client eine Mutation aus. Die Abfrage dazu ist fast identisch zu einer Query, nur dass sie als Operation-Typ „mutation" angibt. Als Argumente übergibt der Client alle Informationen, die der Server benötigt, um die Aktion erfolgreich auszuführen. Aus dem spezifischen Ergebnis einer Mutation kann der Client wiederum – genau wie bei einer Query – einzelne Felder auswählen, die er vom Server als Antwort benötigt. Im BeerAdvisor können Nutzer über eine Mutation eine neue Bewertung („Rating") für ein Bier abgeben. Dazu muss der Client Informationen wie den Kommentar und den Benutzer übergeben. Der Ser-

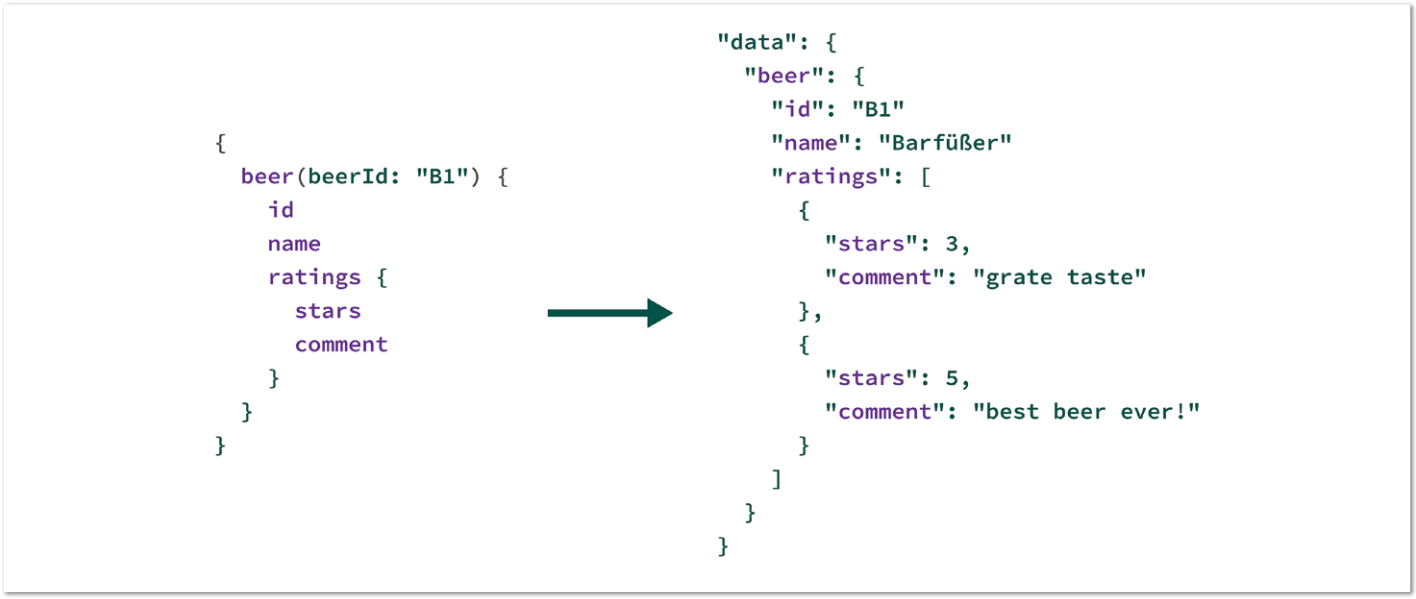

Abbildung 2: Frage und Antwort einer GraphQL Query

ver liefert für die „addRating"-Mutation das neue, auf dem Server angelegte Rating-Objekt zurück. Aus diesem Objekt kann sich der Client erneut die Felder herauspicken, die er benötigt – in unserem Fall ist das lediglich das „id"-Feld, da alle anderen Informationen der Bewertung auf dem Client schon bekannt sind *(siehe Listing 4)*.

```
mutation AddRatingMutation {
  addRating(beerId: „B1", userId: „U2", comment:
"tasty", stars: 3) {
      id
  }
}
```

Listing 4

Die API-Definition

Woher weiß der Client (oder ein Entwickler) nun, welche Objekte und Felder es auf dem Server überhaupt gibt? Welche Queries er ausführen kann und welche Mutations mit welchen Parametern vorhanden sind? Dazu muss man das GraphQL API mit einem Schema beschreiben.

Ein gängiger Weg zur Beschreibung des Schemas ist die Schema Definition Language (SDL), die im Juni 2018 den Weg in die GraphQL-Spezifikation geschafft hat und die auch in Java-Anwendungen verwendet wird.

Ein Schema beschreibt die Objekte (in GraphQL „Object Types" genannt) mitsamt ihren Feldern, die Clients abfragen können. Da GraphQL eine typsichere Sprache ist, muss man die Felder mit Typen versehen. Es gibt einige eingebaute Typen wie ID, String, Int, Boolean und außerdem Arrays beziehungsweise Listen. Die Definition eines eigenen, fachlichen Objekttyps könnte wie in *Listing 5* aussehen.

```
type Rating {
  id: ID!
  beer: Beer!
  author: User!
  comment: String!
  stars: Int!
}
```

Listing 5

```
type Query {
  beer(beerId: ID!): Beer
  beers: [Beer!]!
}

type Mutation {
  addRating(ratingInput: AddRatingInput): Rating!
}

type Subscription {
  onNewRating(beerId: ID!): Rating!
}
```

Listing 6

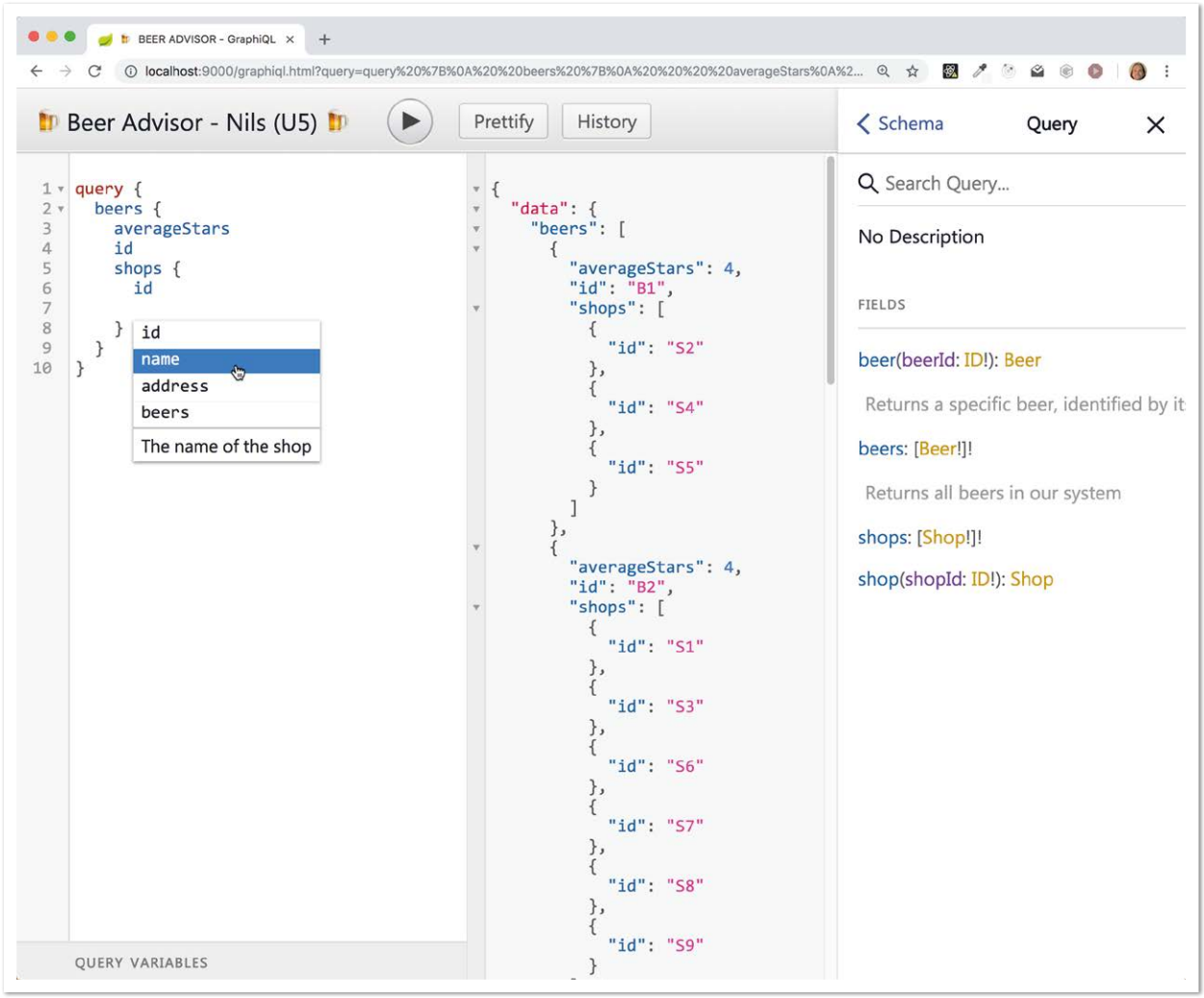

Abbildung 3: Der GraphQL API Explorer „GraphiQL"

Listing 5 definiert den Typ „Rating", der aus fünf Feldern besteht. Jedes Feld erfordert neben dem Namen den jeweiligen Typ. Dazu gehören auch die Angaben, ob das Feld ein Pflichtfeld ist oder nicht (gekennzeichnet mit einem Ausrufezeichen) oder ob es sich um ein Array handelt (dann wäre der Feld-Typ mit eckigen Klammern umschlossen). Die Felder „beer" und „author" zeigen auf weitere fachliche Objekttypen.

Neben den Objekttypen besteht ein Schema noch aus den Operationen, die über das API ausgeführt werden können – das sind die angesprochenen Queries, Mutations und Subscriptions. Die Operationen heißen „Root Operation Types", werden syntaktisch genauso wie Object Types in der SDL beschrieben und bestehen ebenfalls aus einer Menge von Feldern *(siehe Listing 6)*.

Am Query Type ist ein Feld „beer" definiert. Daran ist die Beschreibung von Argumenten für Felder zu sehen. Für die einzelnen Argumente müssen ebenfalls Name und Typ angegeben werden, ähnlich wie bei Argumenten in Java-Methoden.

Während Mutation und Subscription in einem Schema optional sind, ist zumindest die Definition des Query Type notwendig.

GraphQL-Abfragen beginnen immer bei einem Feld, das innerhalb eines der Root Operation Types definiert ist. Von dort aus kann dann weiter transitiv über alle referenzierten Types gewandert werden.

Bei der Ausführung einer Query stellt die GraphQL-Laufzeitumgebung auf dem Server sicher, dass nur Queries, die mit dem Schema kompatibel sind, verarbeitet werden. Queries, die ungültig sind, weil sie zum Beispiel nichtexistierende Felder abfragen, werden vom Server zurückgewiesen. Ähnlich verhält es sich mit dem Ergebnis der Query: Der Client bekommt die Daten nur zurückgeschickt, wenn die Antwort dem Schema entspricht. Ebenfalls wird überprüft, ob Pflichtfelder wirklich befüllt sind und ob die zurückgegebenen Felder den erwarteten Typen entsprechen. Dadurch können sowohl Server als auch Client bei der Verarbeitung von Queries und deren Ergebnissen sicher sein, dass sie es nur mit syntaktisch korrekten Daten zu tun haben.

Entwickler-Werkzeuge

Das Schema selbst kann zur Laufzeit ebenfalls durch eine reguläre GraphQL Query abgefragt werden, ähnlich wie dies in Java mittels Reflection möglich ist.

Die Möglichkeit, das Schema zur Laufzeit abzufragen, hat eine Reihe von Werkzeugen für die Arbeit und Entwicklung von GraphQL-Anwendungen hervorgebracht. Ein prominentes Beispiel ist GraphiQL. Dabei handelt es sich um einen Editor, der im Browser läuft und mit dem Nutzer GraphQL-Abfragen schreiben und ausführen können. GraphiQL fragt das jeweilige Schema eines API ab und kann dadurch Features wie Code-Completion und Syntax-Highlighting anbieten, so wie man sie auch aus der Java-Entwicklung in IDEs gewohnt ist. Der Editor lässt sich auch in die eigene GraphQL-Anwendung integrieren, um zum Beispiel zur Entwicklungszeit Queries zu testen *(siehe Abbildung 3)*.

Auch für die JetBrains-Produktfamilie (u.a. IDEA, WebStorm) stehen Plug-ins zur Verfügung, mit denen sich ebenfalls aus der IDE heraus Queries gegen ein GraphQL API absetzen lassen können und die Unterstützung bei der Formulierung des Schemas in der SDL bieten (Refactorings etc.).

GraphQL-Server in Java

Wir haben jetzt gesehen, wie Abfragen formuliert und ausgeführt werden und wie das Schema des API definiert wird. Nun gilt es, das API serverseitig in Java zu implementieren und für Clients bereitzustellen.

Grundlage dafür ist das Open-Source-Projekt graphql-java. Mit diesem Projekt wird zunächst das Schema des API definiert (per SDL oder programmatisch). Dann werden „DataFetcher" implementiert, die festlegen, welche Daten für welches Feld des API zurückgeliefert werden. Das Projekt hat keine Abhängigkeit auf Spring oder Java EE, kann aber sehr einfach in beiden Umgebungen eingesetzt werden.

```
class BeerAdvisorDataFetchers {
  // gesetzt z.B. durch DI-Mechanismus
  private BeerRepository beerRepository;

  public DataFetcher beerDataFetcher() {
     return new DataFetcher<List<Beer>>() {
        @Override
           public List<Beer> get(DataFetchingEnvironment env) {
              return beerRepository.findAll();
        }
     }
  }

  public DataFetcher beerDataFetcher() {
     return new DataFetcher<Beer>() {
        @Override
        public Beer get(DataFetchingEnvironment env) {
           // Argument "beerId" aus der gerade ausgeführten
           // Query auslesen
           String beerId = env.getArgument("beerId");
           return beerRepository.getBeer(beerId);
        }
     }
  }
}
```

Listing 7

Für die Integration in Spring und Spring Boot stehen zwei weitere Projekte, graphql-java-spring *[2]* und graphql-spring-boot *[2]* zur Verfügung.

Das Schema des API wird üblicherweise in einer oder mehreren Text-Dateien abgelegt und so eingebunden, dass es zur Laufzeit über den Klassenpfad eingelesen werden kann (zum Beispiel im „src/main/resources"-Ordner).

Damit der Server zur Laufzeit weiß, welche Informationen für welche Anfrage geliefert werden sollen, müssen „DataFetcher" implementiert und an das Schema gebunden werden. Ein DataFetcher liefert dabei immer den Wert für ein Feld aus dem Schema. Es handelt sich dabei um ein Interface, das nur über eine einzige Methode verfügt, die implementiert werden muss. Zur Laufzeit bekommt diese Methode über den Parameter „environment" Informationen über die aktuelle Abfrage übergeben. Darüber lassen sich dann zum Beispiel die in der Query angegebenen Argumente eines Feldes auslesen.

In *Listing 7* sind exemplarisch zwei DataFetcher zu sehen. Zum einen der DataFetcher für das „beers"-Feld (das eine Liste aller Biere zurückliefert) und zum anderen der Fetcher für das „beer"-Feld (das ein einzelnes Bier, das über das Argument „beerId" vom Aufrufer angegeben wird, zurückliefert).

Da es sich bei dem DataFetcher um ein funktionales Interface handelt, kann die Implementierung auch über Lambdas erfolgen *(siehe Listing 8)*.

Die DataFetcher müssen zwingend für alle Felder angelegt werden, die in einem der Root Operation Types (also Query, Subscription oder Mutation) definiert sind. Für alle darunterliegenden bzw. referenzierten Objekte sind die DataFetcher optional. Ist für ein Feld kein DataFetcher angegeben, wird per Default der „PropertyDataFetcher" verwendet. Dieser probiert ein Feld entweder über Reflektion und Namenskonventionen zu ermitteln (bei POJOs) oder liest den Wert aus einer Map aus. Aus diesem Grund muss für die Felder „name", „id" und „price", die im Schema für den „Beer"-Typ definiert sind, auch kein DataFetcher angegeben werden. Das Root-Objekt (Beer) wird über einen der beiden oben gezeigten DataFetcher ermittelt, die jeweils ein „Beer"-Objekt (bzw. eine Liste davon) zurückgeben. Da in der Java-Klasse „Beer" ebenfalls die Felder „name", „id" und „price" vorhanden sind, muss dafür kein expliziter DataFetcher angegeben werden, da der PropertyDataFetcher diese Felder automatisch findet und abfragen kann. (Dabei ist übrigens sichergestellt, dass nur auf solche Felder zugegriffen wird, die auch im Schema definiert sind. Ein Feld, das im Schema nicht definiert ist, aber an einer POJO-Instanz existiert, ist trotzdem nicht abfragbar.)

Anders sieht die Sache für das Feld „averageStars" am „Beer"-Objekt aus. Dieses Feld ist in der Java-Klasse „Beer" nicht definiert, da der Wert dynamisch berechnet werden soll. Also muss dafür ein eigener DataFetcher implementiert werden. DataFetcher für (Unter-)Objekte sehen genauso aus wie die vorher gezeigten DataFetcher für die Root Operation Types. Über die Methode „getSource" am übergebenen Environment kann die Instanz des Objektes abgefragt werden, das der übergeordnete Fetcher ermittelt hat. In diesem Beispiel geht es um ein Feld, das im API-Schema auf dem „Beer"-Objekt definiert ist, folglich liefert „getSource" eine Instanz der Java-Klasse „Beer" *(siehe Listing 9)*.

```
class BeerAdvisorDataFetchers {
  private BeerRepository beerRepository;

  public DataFetcher<Beer> beerFetcher() {
     return environment -> {
        String beerId = environment.getArgument("beerId");
        return beerRepository.getBeer(beerId);
     };
  }

  public DataFetcher<List<Beer>> beersFetcher() {
     return environment -> beerRepository.findAll();
  }
}
```

Listing 8

```
import nh.graphql.beeradvisor.domain.Beer;

public class BeerDataFetchers {
  public DataFetcher<Integer> averageStarsFetcher() {
     return environment -> {
        // Instanz ermitteln, von der das Feld 'averageStars'
        // abgefragt wurde
        Beer beer = environment.getSource();

        // Wert für das Feld berechnen und zurückliefern
        return (int)
           Math.round(beer.getRatings().stream()
              .mapToDouble(Rating::getStars)
              .average()
              .getAsDouble());
     };
  }
}
```

Listing 9

```
SchemaParser schemaParser = new SchemaParser();

TypeDefinitionRegistry typeRegistry =
  schemaParser.parse(/* (Klassen-)Pfad zur SDL-Datei */);

// Zuordnung von Typen/Feldern zu DataFetchern
RuntimeWiring runtimeWiring = RuntimeWiring.newRuntimeWiring()
  .type(newTypeWiring("Query")
     .dataFetcher("beer", beerAdvisorDF.beerFetcher())
     .dataFetcher("beers", beerAdvisorDF.beersFetcher()))
  .type(newTypeWiring("Beer")
     .dataFetcher("averageStars",
        beerDFs.averageStarsFetcher()))
  .type(newTypeWiring("Mutation")
     .dataFetcher("addRating",
        beerAdvisorDF.addRatingFetcher()))
  .build();

// Das "ausführbare" GraphQL Schema
GraphQLSchema schema =
  new SchemaGenerator().makeExecutableSchema
     (typeRegistry, runtimeWiring);
```

Listing 10

```
GraphQL graphQL = GraphQL.newGraphQL(schema).build();

ExecutionInput executionInput =
  ExecutionInput.newExecutionInput()
     .query
     ("query { beers { id name ratings { stars } }
}").build();

ExecutionResult executionResult =
  graphQL.execute(executionInput);

Map<String, Object> data = executionResult.getData();
```

Listing 11

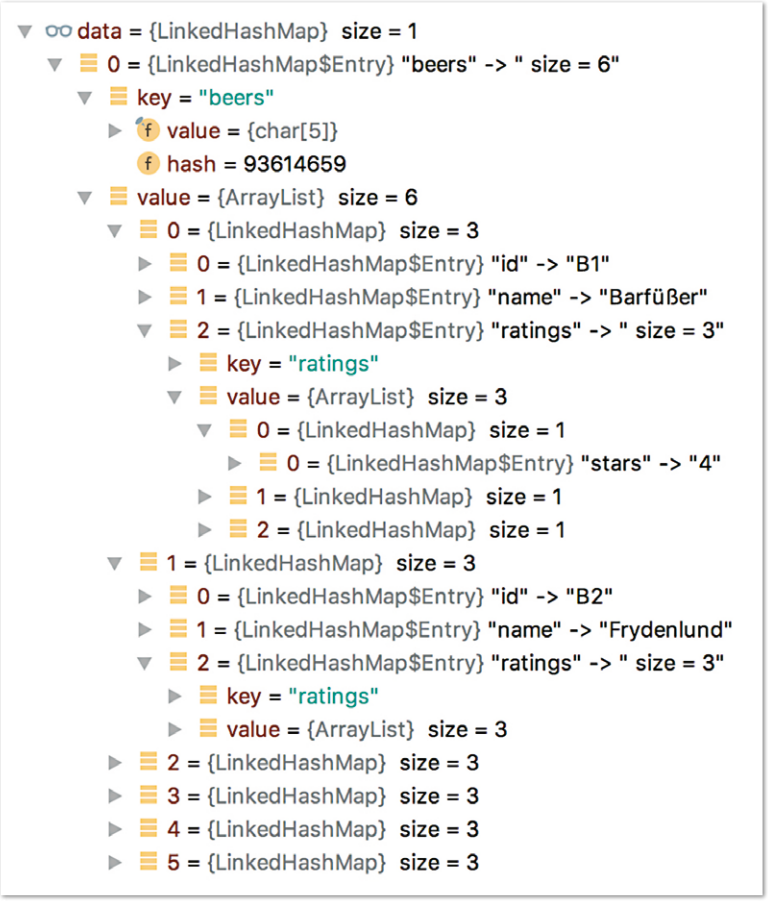

Abbildung 4: Das Ergebnis eine GraphQL-Abfrage in Java

DataFetcher für Mutations und Subscriptions werden technisch identisch implementiert, sie unterscheiden sich lediglich in dem, was sie fachlich machen, von den gezeigten Fetchern. Eine Mutation etwa könnte ein Domain-Objekt verändern und über ein Repository in der Datenbank speichern. Bei Subscriptions ist zu beachten, dass der vom DataFetcher zurückgelieferte Wert ein Reactive-Streams-Publisher-Objekt sein muss. Beispielhafte Implementierungen dafür finden sich jeweils im BeerAdvisor Source Code.

Bereitstellen des API

In der Anwendung ist jetzt das Schema des API definiert und mit den DataFetchern ist beschrieben, wie Daten zu einzelnen Feldern ermittelt werden können. Im letzten Schritt muss das Schema mit den DataFetchern verknüpft werden. Dies erfolgt beim Start der Anwendung über das „RuntimeWiring". Mit diesen Informationen wird dann eine Instanz von „GraphQLSchema" erzeugt, mit der es dann möglich ist, Abfragen auszuführen. Das gekürzte *Listing 10* zeigt das exemplarisch.

Über die erzeugte Schema-Instanz lassen sich jetzt aus der Java-Anwendung heraus GraphQL Queries ausführen. Das Ergebnis der (erfolgreichen) Queries *(Listing 11)* ist in der Regel eine verschachtelte Map mit den Werten der abgefragten Felder *(siehe Abbildung 4)*.

Das Projekt graphql-java-tools *[2]* bietet darauf aufbauend eine Abstraktionsschicht an, mit der es möglich ist, GraphQL-Abfragen und -Antworten auf Java Beans zu mappen.

In der Regel werden GraphQL-Abfragen aber nicht aus der Java-Anwendung heraus ausgeführt, sondern sollen über einen HTTP-Endpunkt angeboten werden. Dazu muss die oben gezeigte Logik in ein Servlet eingebunden werden, das die Query aus dem HTTP Request entgegennimmt und an die GraphQL-Instanz weiterleitet. Ein fertiges Servlet steht in den Projekten graphql-java-servlet *[2]* und graphql-java-spring *[2]* zur Verfügung.

Fazit

GraphQL bietet eine interessante Möglichkeit, Daten gezielt vom Server zu laden, ohne dass sich Client und Server sehr eng aneinander koppeln müssen. Auf der anderen Seite ist die Sprache aber von der Mächtigkeit her nicht mit SQL oder Ähnlichem zu vergleichen: Sortierungen, Aggregationen oder eine Suche etwa fehlen der Sprache. Werden solche Features benötigt, müssen sie manuell programmiert werden, zum Beispiel durch entsprechende Felder oder Argumente an Feldern.

Ein GraphQL API für die eigene Anwendung anzubieten, ist mit dem graphql-java-Projekt relativ einfach. Darüber hinaus macht das typisierte Schema sehr gute Entwicklertools möglich, sodass sich eigene bestehende Anwendungen zügig mit einem GraphQL API versehen lassen. Es lohnt sich auf jeden Fall, die Sprache zumindest einmal auszuprobieren und die weitere Entwicklung in diesem Bereich im Auge zu behalten.

Referenzen

[1] *https://foundation.graphql.org/*
[2] *https://www.graphql-java.com/*
[3] *https://github.com/nilshartmann/graphql-examples/tree/master/java/beeradvisor_graphql-java/backend*

Nils Hartmann
nils@nilshartmann.net

Nils Hartmann ist freiberuflicher Softwareentwickler, Architekt und Trainer aus Hamburg. Er programmiert sowohl in Java als auch in JavaScript/TypeScript und beschäftigt sich mit der Entwicklung von Single-Page-Anwendungen. Nils ist Co-Autor des Buchs „React – Die praktische Einführung" (dpunkt-Verlag) und gibt sein Wissen auf Konferenzen, Schulungen und Workshops weiter.