# Filmdatenbank als GraphQL Demo-App

Projektarbeit

David Wallny

20.11.2019

**Zusammenfassung**

GraphQL ist eine Abfragesprache, mit der sich APIs zur Abfrage und Manipulation von Daten strukturieren lassen. In dieser Arbeit wird die Anwendung von GraphQL anhand dem Beispiel einer Filmdatenbank erkundet.

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 GraphQL

GraphQL ist eine Abfragesprache, mit der sich APIs zur Abfrage und Manipulation von Daten strukturieren lassen. Sie wurde 2012 intern von Facebook entwickelt, um in ihren mobilen Apps zum Einsatz zu kommen, und 3 Jahre später veröffentlicht.[1]

## 1.2 Ziel dieser Arbeit

Das Ziel dieser Arbeit ist es, eine beispielhafte Web-Applikation zu entwickeln, die GraphQL als Programmierparadigma für ihre API verwendet. Dadurch sollen Erkenntnisse über den Einsatz von GraphQL, Stärken und Schwächen dieses Ansatzes, und praktische Probleme mit möglichen Lösungen erkundet werden.

Desweiteren soll ein Kontrast zwischen GraphQL und REST (Representational State Transfer) gezogen werden, da REST ein sehr bekanntes, häufig eingesetztes, und bewährtes Programmierparadigma für APIs ist. Dadurch ist es in praktischer Hinsicht der Haupt-„Konkurrent“ zu GraphQL.

### 1.3 Web-Applikation Filmdatenbank

Die Demo-Applikation ist eine simple Filmdatenbank, die über GraphQL mit dem Backend kommuniziert. Sie stellt Filme tabellarisch nach unterschiedlichen Filter- und Sortierungskriteren dar, zeigt nähere Details zu einzelnen Filmen, und erlaubt es Benutzern, Bewertungen zu hinterlassen und einzusehen.

Die Applikation ist ein node.js JavaScript-Projekt, das die von Facebook bereitgestellte graphql-Package benutzt. Der clientseitige Code ist ebenfalls JavaScript.

Die Filmdaten werden aus einer MySQL-Datenbank abgerufen; die GraphQL-Abfragen werden also vom Backend direkt auf SQL-Abfragen abgebildet.

Listing 1: Vollständiges GraphQL-Schema der Filmdatenbank

```
type Query {
  movie(id: ID!): Movie
      movieList(num: Int = 10, offset: Int = 0, order: String = "title",
          search: String, genres: [String]): MovieList!
}

type Mutation {
        postReview(review: ReviewInput!): Boolean!
}

type Movie {
        id: ID!
        title: String!
        origTitle: String
        isAdult: Boolean!
        year: Int
        runtime: Int
        crew: [Person]
        reviews: [Review]
        genres: [String]
        avgRating: Float
        numVotes: Int!
}

type MovieList {
        movies: [Movie]
        rowcount: Int!
}

type Review {
        id: ID!
        forMovie: ID!
        reviewerName: String
        text: String
        score: Int
```

```
}

input ReviewInput {
        forMovie: ID!
        reviewerName: String
        text: String
        score: Int
}

type Person {
        name: String!
        starsIn: [Movie]
        workedOn: [Movie]
        role: String!
}
```

## 2 GraphQL 101

GraphQL benutzt, wie der Name andeutet, Entitätsgraphen zum Austausch von Daten. Die Struktur der Antwort wird in der Anfrage definiert und vom Server mit Daten befüllt.

Visuell erinnern einfache GraphQL-Anfragen an „unfertige“ JSON-Objekte. Antworten werden typischerweise als JSON repräsentiert - allerdings lässt die GraphQL-Spezifikation das dem Entwickler frei.[4] Für Web-Applikationen bietet sich JSON besonders an, da moderne Browser entsprechende Funktionen bereitstellen, und wurde deswegen für die Filmdatenbank ausgewählt.

```
{
  movieList {
    rowcount
    movies {
      title
      year
    }
  }
}
```

Listing 2: Beispielabfrage

```
{
 "data": {
  "movieList": {
   "rowcount": 24808,
   "movies": [
     {
      "title": "Supravietuitorul",
      "year": 2008
     },
     ...
   ]
  }
 }
}
```

Listing 3: JSON-Antwort

Eine Anfrage besteht im Grunde aus einer Beschreibung der Felder, für die sich der Client interessiert.

### 2.1 Endpunkte vs. Felder

Eine REST-API wird typischerweise mit unterschiedlichen Endpunkten entworfen, die unterschiedliche Funktionen oder Ressourcen repräsentieren. In der Web-Entwicklung wird dies durch unterschiedliche URLs realisiert.

GraphQL hingegen sieht nur einen einzigen Endpunkt vor, an den alle Anfragen gesendet werden. Alle Abläufe werden durch Feldnamen definiert. So sind z.B. `title` und `year` Feldnamen des `Movie`-Typs, und `movieList` ist ebenfalls ein Feldname des globalen Query-Root-Objekts. Diese äußersten Felder des Query-Roots ähneln semantisch den Endpunkten des REST-Paradigma, da sie die Einstiegspunkte einer GraphQL-API definieren.

Jedes Feld kann zusätzlich Parameter akzeptieren.

### 2.2 Aufbau einer GraphQL-Abfrage

Eine unabgekürzte GraphQL-Abfrage besteht aus 4 Komponenten: dem Operationstyp, dem Operationsnamen, der Struktur der angeforderten Felder, und optionale Parameter für jedes Feld.

```
query GetMovies {
  movieList(num: 5, order: "random")
     {
    rowcount
    movies {
      title
      year
    }
  }
}
```

Listing 4: Beispiel-Query

```
mutation Poptart {
  postReview(text: "klasse!", score:
     5)
}
```

Listing 5: Beispiel-Mutation

Der Operationstyp **`query`** wird für Abfragen benutzt. Der Operationstyp **`mutation`** ist für Felder vorgesehen, die Daten verändern/schreiben, wenn sie angegeben werden. Diese Konvention dient dazu, dass der Client die Absicht, Daten zu verändern, explizit angeben muss. Der einzige funktionale Unterschied zwischen Query-Feldern und Mutation-Feldern besteht darin, dass Mutations sequentiell abgearbeitet werden[3].

Der Operationsname (z. B. „GetMovies“) ist vom Client frei wählbar und kann für Logging/Profiling-Zwecke benutzt werden.

## 3 Erkenntnisse

### 3.1 Performance & Optimierung

GraphQL bietet Möglichkeiten, die Kommunikation zwischen Client und Server in verschiedenen Hinsichten zu optimieren.

1. **Spezifität der Daten:** Da der Client jedes Feld explizit anfordern muss, werden andere, für den Client nicht interessante Felder natürlich nicht gesendet. Dies spart unnötigen Netzwerk-Traffic.
   (a) Der Server kann dies weiter optimieren, indem nicht benötigte Felder nach Möglichkeit gar nicht erst berechnet/aus der Datenbank abgerufen werden (s. Abschnitt 3.1.1.
2. **Kombination von mehreren Abfragen:** Eine GraphQL-Anfrage kann beliebig viele Felder, oder sogar das selbe Feld mehrere Male anfordern. Dadurch können unnötige Netzwerk-Requests eingespart werden.
   (a) Allerdings ist es **nicht möglich, Abfragen mit Mutations zu kombinieren** (s. Abschnitt 3.3.2).

### 3.1.1 Anmerkung zu Lazy Loading

Die GraphQL-Runtime erlaubt es, Resultate zurückzugeben, die statt konkreten Daten Funktionen enthalten. Diese Funktionen werden just-in-time aufgerufen, und zwar **nur**, wenn das Feld vom Client auch tatsächlich angefordert wurde. Als Code-Beispiel:

```
return {
  numberOne: fetchSQL("SELECT 1 FROM birne;"),
  numberTwo: fetchSQL("SELECT 2 FROM apfel;")
}
```

Listing 6: ohne Lazy Loading

```
return {
  numberOne: () => {
    return fetchSQL("SELECT 1 FROM birne;");
  },
  numberTwo: () => {
    return fetchSQL("SELECT 2 FROM apfel;");
  }
}
```

Listing 7: mit Lazy Loading

Wenn der Client im zweiten Beispiel nur `numberOne` anfordert, wird die zu `numberTwo` dazugehörige Funktion nicht aufgerufen und die andere SQL-Abfrage findet nicht statt.

## 3.2 Komplexität

GraphQL ist an sich eine weitere Abfragesprache, die Frontend-Programmierer zusätzlich beherrschen müssen, um Daten vom Backend zu erhalten. Besonders wenn Variablen zum Einsatz kommen, kann die Abfrage recht komplex wirken, weil jede Variable explizit mit ihrem entsprechenden GraphQL-Typen definiert werden muss.

```
query($search: String, $num: Int, $offset: Int, $genres: [String], $order:
   String) {
  movieList(search: $search, num: $num, offset: $offset, genres: $genres,
     order: $order) {
  rowcount
  movies {
    id
    title
    year
    crew {
      name
      role
    }
  }
}
```

Listing 8: Query mit separat gesendeten Variablen

```
{ "search": "Batm", "num": 10, "offset": 0, genres: null, order: "rating" }
```

Listing 9: Separate Query-Variablen

## 3.3 Praktische Probleme & Lösungen

### 3.3.1 Metadaten / out-of-band Informationen

Es ist nicht direkt möglich, zu einer Antwort Metadaten hinzuzufügen. So wäre es z. B. sinnvoll, zu einer `movieList`-Abfrage, die ein Array von Movie-Objekten zurückgibt, die Gesamtanzahl der gefundenen Filme mitzuliefern. Damit könnten dann Seitenzahlen für eine Navigationsleiste berechnet werden.

Um dies zu lösen, muss die API umstrukturiert und ein neuer Typ eingeführt werden, der die eigentlichen Resultate als Unterfeld enthält.

```
  type MovieList {
   # eigentliches Resultat (Movie-Array)
    movies: [Movie]
   # neue Zusatzinformation
    rowcount: Int!
  }
```

Listing 10: GraphQL-Typ-Definition

### 3.3.2 Query / Mutation nicht kombinierbar

In der Filmdatenbank wäre es sinnvoll, die Liste der Bewertungen neu zu laden, nachdem der Benutzer seine eigene Bewertung hinterlässt. Es ist aber nicht möglich, in einer einzigen Abfrage Mutation-Felder und Query-Felder zu kombinieren.

Um dies zu lösen, können entweder zwei separate Abfragen direkt nacheinander gesendet werden (zusätzlicher Netzwerk-Request), oder der `postReview`-Mutation ein Unterfeld hinzugefügt werden, das eine aktualisierte Liste der Bewertungen, inklusive der soeben kreierten, zurückgibt.

### 3.3.3 Angreifbarkeit

Aus der hohen Flexibilität, die es dem Benutzer erlaubt, Anfragen beliebig zu gestalten und rekursiv zu verschachteln, ergibt sich eine entsprechend große Angriffsfläche. Am Beispiel der Filmdatenbank gibt es z. B. den Typ `Movie` mit einem Array von `People`, die an dem Film gearbeitet haben. `People` hat aber wiederrum ein Unterfeld, das sich auf `Movies` bezieht, an denen diese Person gearbeitet hat.

So kann eine Abfrage beliebig oft verschachtelt werden, um die Ressourcen des GraphQL-Backends maximal zu verschwenden.

```
query CrashTheServer {
  movieList(num: 99999999) {
    movies { crew { workedOn { crew { workedOn { crew { workedOn { crew {
        workedOn { crew { workedOn { crew { workedOn { crew { workedOn {
        title } }}}}}}}}}}}}}
  }
}
```

Listing 11: Angriff gegen GraphQL

Als Lösung kommen mehrere Ansätze in Frage:

- eine Verschachtelungs-Obergrenze, die simple Angriffe wie den obigen verhindern würde
- ein Ressourcen-Limit, das die GraphQL-Engine unterbricht, wenn eine Abfrage zu lange braucht
  - verhindert Angriffe nicht, grenzt den Schaden nur ein
- eine Query-Whitelist, die nur bestimmte Queries zulässt
  - sehr aufwändig; Whitelist muss ständig modifiziert werden, wenn sich irgendeine Abfrage ändert

# 4 Kontrast zu REST

## 4.1 Endpunkte und URLs

Wie in Abschnitt 2.1 erläutert, benutzt GraphQL nur eine einzige URL; die URL hat also anders als bei REST keinen semantischen Nutzen. Auch HTTP-Verben wie GET, POST, PUT,

DELETE werden nicht semantisch verwendet. GET oder POST können beliebig für Queries und Mutations eingesetzt werden.

### 4.2 HTTP-Caching

Da GraphQL nur einen einzigen HTTP-Endpunkt benutzt und Ressourcen nicht eindeutig über URLs identifiziert werden, gestaltet sich HTTP-Caching als umständlich.

Um eine GraphQL-API mit HTTP-Caching zu versehen, müssen folgende Punkte beachtet werden:

- GraphQL-Queries müssen in der URL als GET-Parameter übergeben werden (anstatt als POST-Parameter)
- Query-Variablen müssen ebenfalls als GET-Parameter übergeben werden
- Die Repräsentation der Query und der zugehörigen Variablen müssen bei jedem Netzwerk-Request Byte-für-Byte identisch sein. `query A{movie{title}}` und `query B{movie{title}}` sind funktional identisch, führen aber natürlich zu unterschiedlichen URLs.
- Das GraphQL-Backend **muss** sicherstellen, dass die Query/Parameter von der URL kommen, und nur in diesem Fall entsprechende Caching-Header setzen. Wenn versehentlich der parameterlose Endpunkt (z. B. `/graphql`) gecached wird, kann dies zu schwer auffindbaren Fehlern führen.

Mit komplexen Abfragen oder kann dies schnell zu ungewöhnlich langen URLs führen, die besonders in älteren Browsern zu Problemen führen können.[2]

## 5 Fazit

GraphQL ist ein sehr mächtiges Werkzeug, um APIs zu gestalten. Wenn man aber mit einfachen Daten umgeht und APIs, die sich nicht oder nur selten ändern, rentiert sich die zusätzliche Komplexität von GraphQL eher nicht. Die Filmdatenbank profitiert nicht wirklich von GraphQL und könnte als REST-API zusätzlich von HTTP-Caching profitieren, da sich die meisten Daten nicht ändern.

## 6 Quellen

## Literatur


[1] Lee Byron. GraphQL: A data query language. `https://engineering.fb.com/core-data/graphql-a-data-query-language/`, 2015. [Blog post, accessed 2019-11-28].

[2] Paul Dixon. What is the maximum length of a URL in different browsers? `https://stackoverflow.com/a/417184`, 2019. [StackOverflow question, accessed 2019-12-21].

[3] Facebook. GraphQL Spec, June 2018 Edition - Mutations. `https://graphql.github.io/graphql-spec/June2018/#sec-Mutation`, 2018. [Online].

[4] Facebook. GraphQL Spec, June 2018 Edition - Serialization Format. https://graphql.github.io/graphql-spec/June2018/#sec-Serialization-Format, 2018. [Online].