



NILS HARTMANN

<https://nilshartmann.net>



Slides

GraphQL Verheißung oder Verhängnis? Ein praktischer Deep Dive

NILS HARTMANN

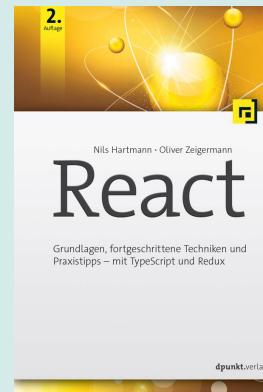
nils@nilshartmann.net

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

Java, Spring, GraphQL, TypeScript, React

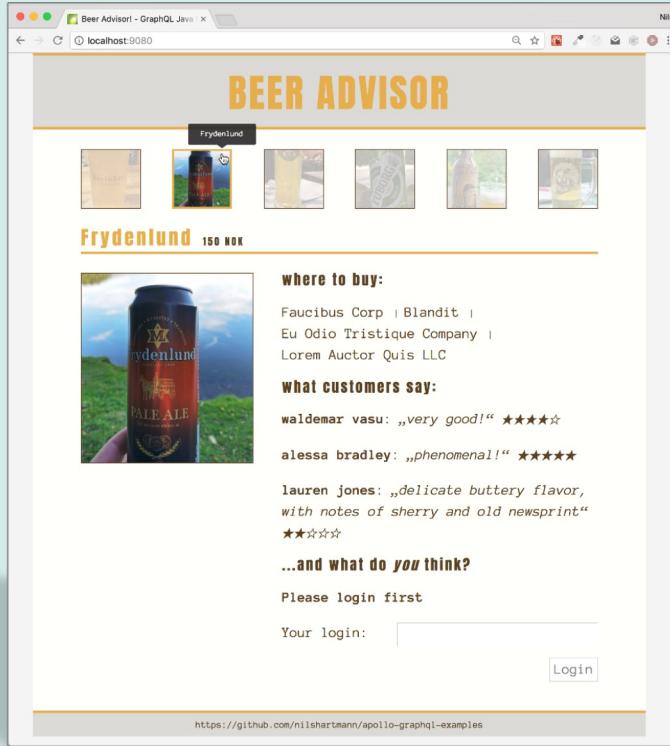


<https://graphql.schule/video-kurs>



<https://reactbuch.de>

HTTPS://NILSHARTMANN.NET



Beispiel Anwendung

Source: <https://github.com/nilshartmann/spring-graphql-talk>

EINE API FÜR DEN BEERADVISOR

Eine API für den Beeradvisor...

EINE API FÜR DEN BEERADVISOR

Ansatz 1: Backend bestimmt Aussehen der Endpunkte / Daten

/api/beer

Beer
id
name
price
ratings
shops

/api/shop

Shop
id
name
street
city
phone

/api/rating

Rating
id
author
stars
comment

EINE API FÜR DEN BEERADVISOR

**Ansatz 1: Backend bestimmt Aussehen der Endpunkte / Daten
REST / HTTP APIs**

/api/beer

Beer
id
name
price
ratings
shops

/api/shop

Shop
id
name
street
city
phone

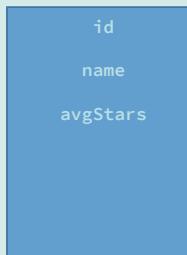
/api/rating

Rating
id
author
stars
comment

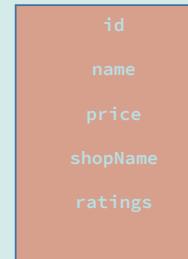
EINE API FÜR DEN BEERADVISOR

Ansatz 2: Client diktiert die API nach seinen Anforderungen

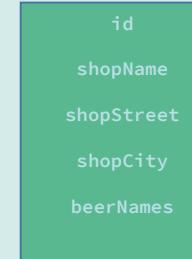
/api/home



/api/beer-view



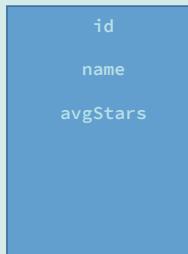
/api/shopdetails



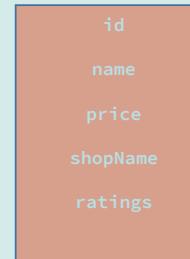
EINE API FÜR DEN BEERADVISOR

Ansatz 2: Client diktiert die API nach seinen Anforderungen Backend for Frontend (BFF)

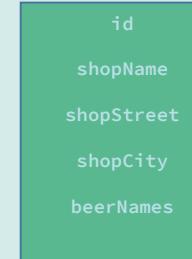
/api/home



/api/beer-view



/api/shopdetails



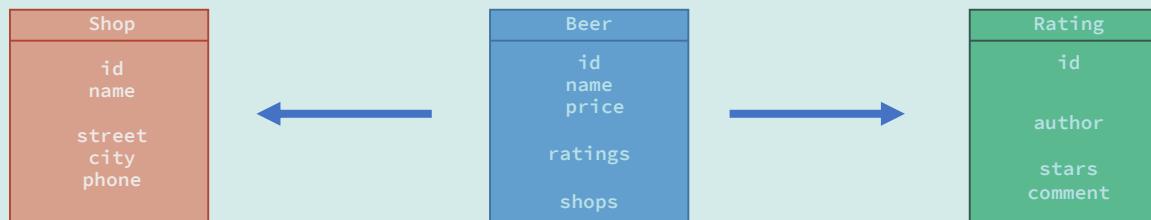
EINE API FÜR DEN BEERADVISOR

Ansatz 3: GraphQL...

EINE API FÜR DEN BEERADVISOR

Ansatz 3: GraphQL...

- Aus Ansatz 1: Server bestimmt, wie Datenmodell aussieht

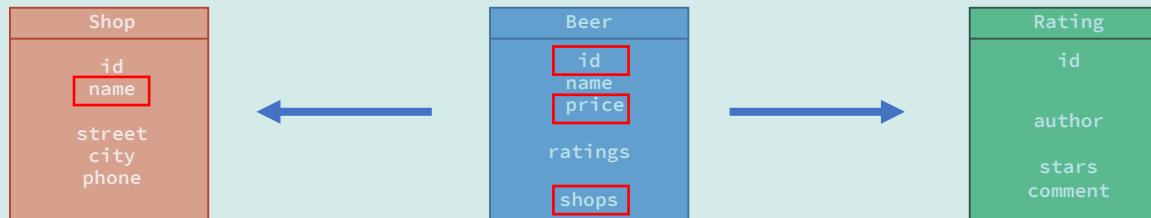


EINE API FÜR DEN BEERADVISOR

Ansatz 3: GraphQL...

- Aus Ansatz 1: Server bestimmt, wie Datenmodell aussieht
- ...aber Client kann pro Ansicht wählen, welche Daten er daraus benötigt

```
{ beer { id price { shops { name } } }
```



*"GraphQL is a **query language for APIs** and a **runtime for fulfilling those queries** with your existing data"*

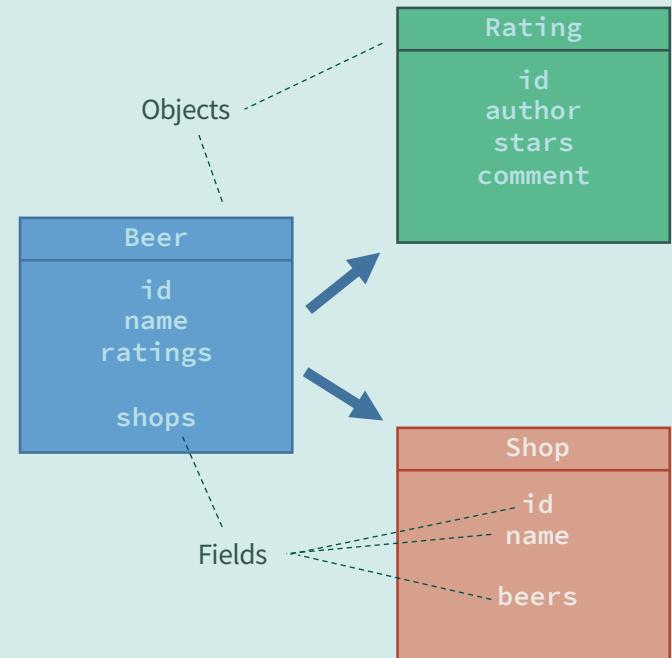
- <https://graphql.org>

GraphQL

Die
Query
Sprache

QUERY LANGUAGE

Über eine GraphQL API werden *Objekte mit Feldern* bereitgestellt

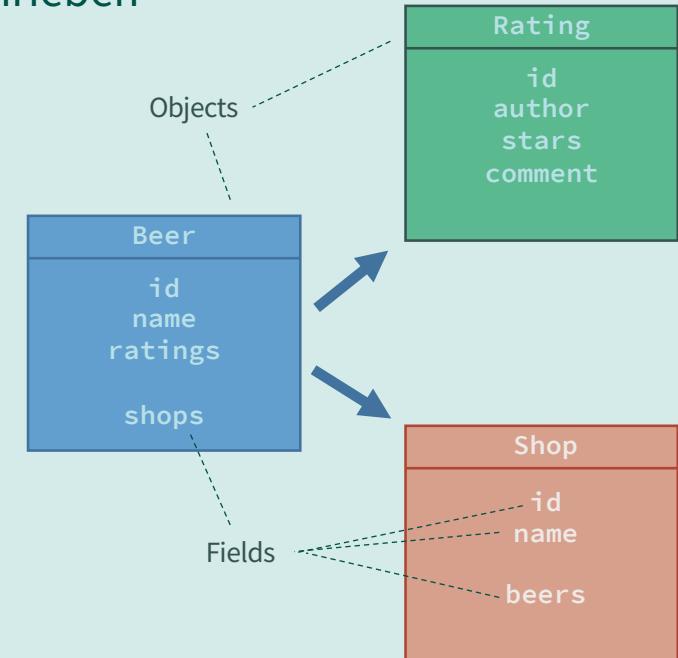


QUERY LANGUAGE

Mit der Query-Sprache werden *Felder von Objekten* abgefragt

- Die Objekte und Felder sind in einem **Schema** beschrieben

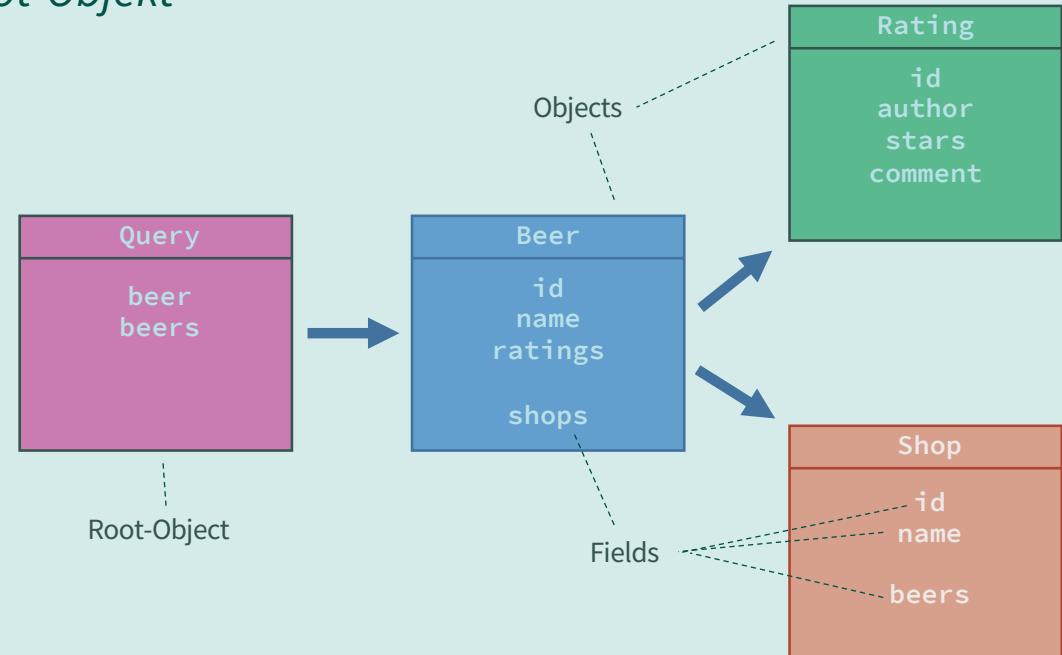
```
type Beer {  
    id: ID!  
    name: String!  
    ratings: [Rating!]!  
    shops: [Shop!]!  
}  
  
type Rating {  
    id: ID!  
    author: String!  
    stars: Int!  
    comment: String!  
}  
  
type Shop {  
    id: ID!  
    name: String!  
    beers: [Beer!]!  
}
```



QUERY LANGUAGE

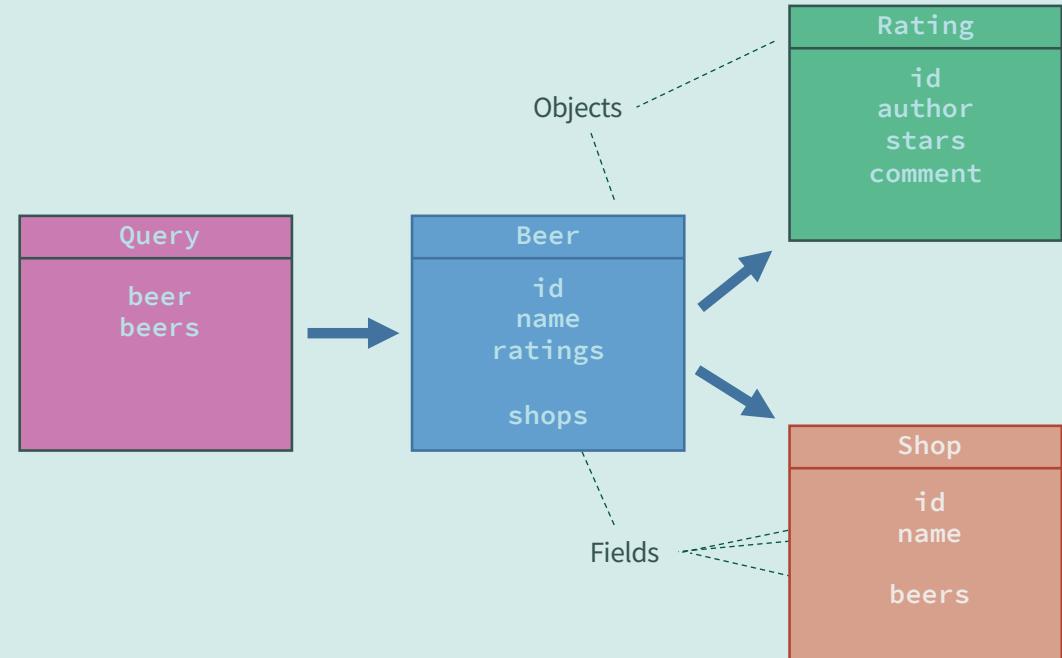
Mit der Query-Sprache werden *Felder von Objekten abgefragt*

- Der Graph beginnt bei einem *Root-Objekt*



QUERY LANGUAGE

Mit der *Query-Sprache* werden aus dem Graphen *Felder* ausgewählt

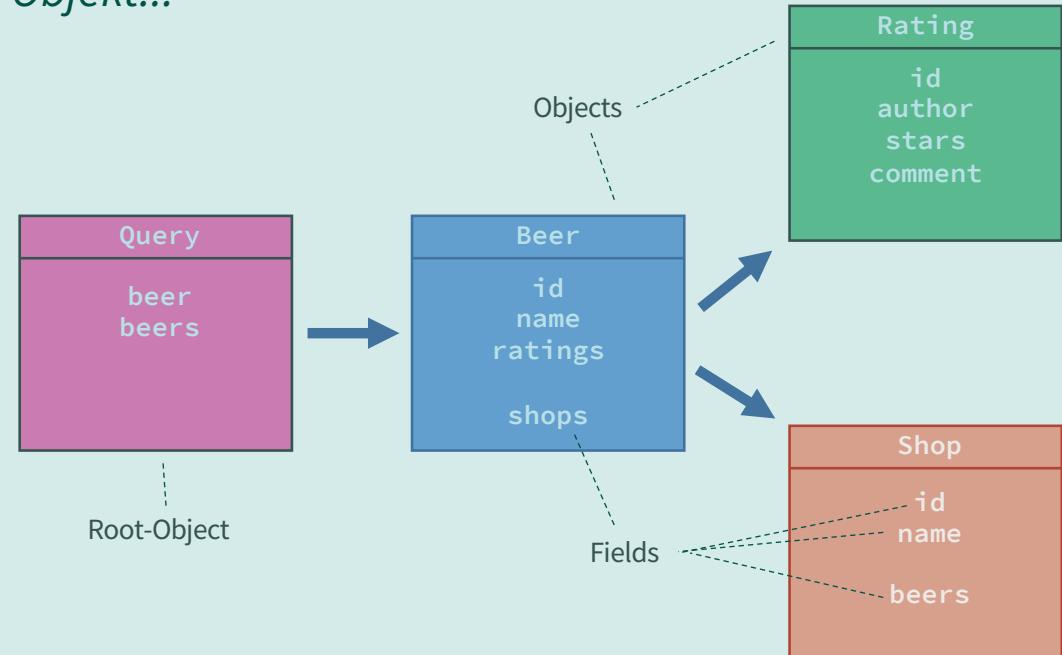


```
query { }
```

QUERY LANGUAGE

Mit der Query-Sprache werden *Felder von Objekten abgefragt*

- Der Query beginnt bei dem *Root-Objekt*...

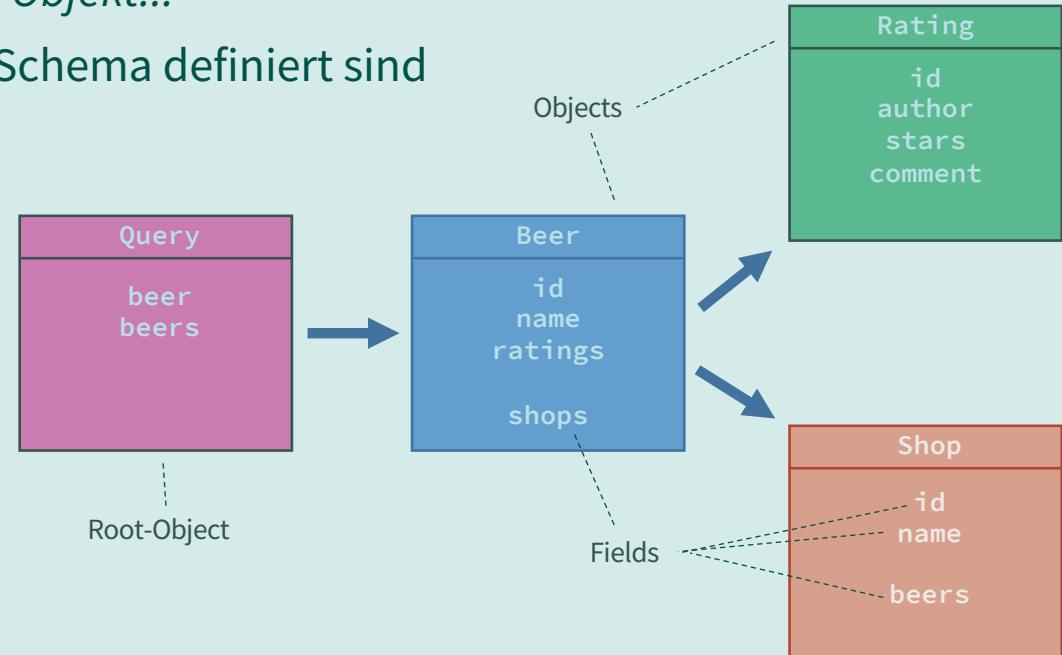


```
query { beers }
```

QUERY LANGUAGE

Mit der Query-Sprache werden *Felder von Objekten abgefragt*

- Der Query beginnt bei dem *Root-Objekt*...
- ...folgt dann den Pfaden, die im Schema definiert sind

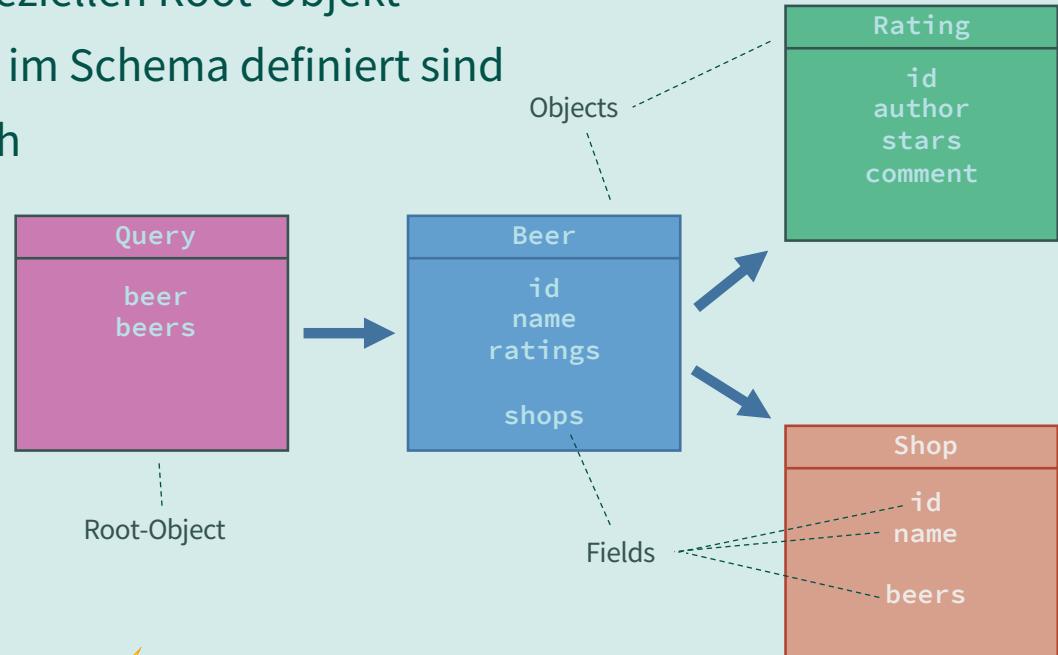


```
query { beers { ratings { comment } } }
```

QUERY LANGUAGE

Mit der Query-Sprache werden *Felder von Objekten abgefragt*

- Alle Queries starten an einem speziellen Root-Objekt
- Man kann nur Pfade folgen, die im Schema definiert sind
- Andere "joins" sind nicht möglich



query { **shops** { id } }

The screenshot shows the GraphiQL interface running on localhost:9000. On the left, the query editor contains a GraphQL query for a 'BeerAppQuery'. The results pane on the right displays the JSON response, which includes a list of beers with their details like id, name, price, and ratings.

```
query BeerAppQuery {
  beers {
    id
    name
    price
    ratings {
      id
      beerId
      author
      comment
    }
  }
}

beers
beer
ratings
ping
__schema
__type
>Returns all beers in our store
```

```
{
  "data": {
    "beers": [
      {
        "id": "B1",
        "name": "Barfüßer",
        "price": "3,88 EUR",
        "ratings": [
          {
            "id": "R1",
            "beerId": "B1",
            "author": "Waldemar Vasu",
            "comment": "Exceptional!"
          },
          {
            "id": "R7",
            "beerId": "B1",
            "author": "Madhukar Kareem",
            "comment": "Awesome!"
          },
          {
            "id": "R14",
            "beerId": "B1",
            "author": "Emily Davis",
            "comment": "Off-putting buttery nose, laced with a touch of caramel and hamster cage."
          }
        ]
      },
      {
        "id": "B2",
        "name": "Frøyenlund",
        "price": "158 NOK",
        "ratings": [
          {
            "id": "R2",
            "beerId": "B2",
            "author": "Andrea Gouyen",
            "comment": "Very good!"
          }
        ]
      }
    ]
  }
}
```



Ein Query



Mutation



Subscription



Dokumentation



Netzwerkverkehr (evtl.)

Demo: Query-Sprache

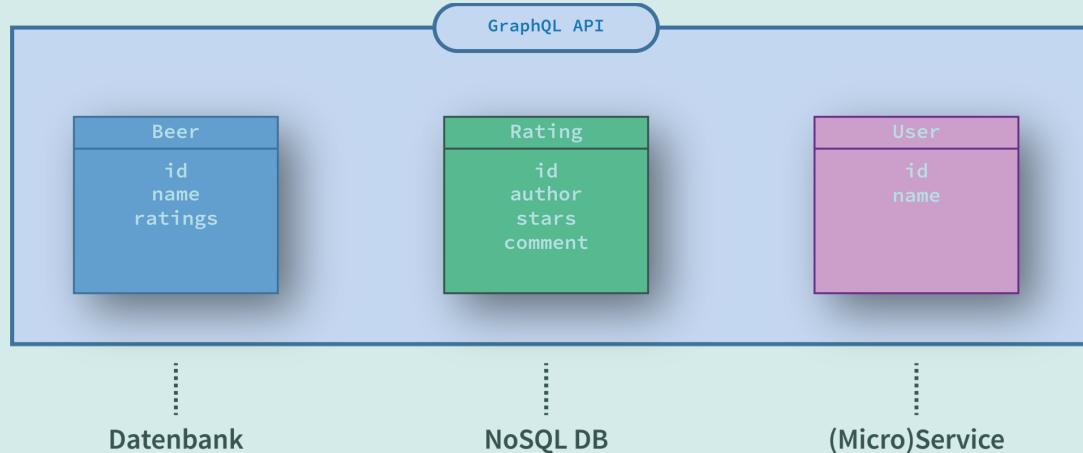
<https://github.com/graphql/graphiql>

GraphQL Server Implementierung

GRAPHQL APIS

GraphQL macht keine Aussage, wo die Daten herkommen

- 👉 Ermittlung der Daten ist unsere Aufgabe
- 👉 Müssen nicht aus einer Datenbank kommen



Implementieren von GraphQL APIs

- In der Regel muss man die Logik selbst implementieren
- Es gibt Frameworks für fast alle Programmiersprachen

IMPLEMENTIERUNG

Die Spec schreibt nicht vor, wie eine GraphQL Implementierung aussehen muss

- Es gibt aber deutliche Hinweise
- Fast alle Frameworks arbeiten danach

IMPLEMENTIERUNG

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an

IMPLEMENTIERUNG

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)

IMPLEMENTIERUNG

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt
8. Die Antwort wird an den Client gesendet

Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und **Schema**)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
- 5. Die Resolver-Funktionen ermitteln die Daten für ein Feld**
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt
8. Die Antwort wird an den Client gesendet

👉 Die Definition des Schemas ist unsere Aufgabe

👉 Die Implementierung der Resolver-Funktionen ist unsere Aufgabe

Live Beispiel : Schema + Resolver

- 👨‍💻 workspace-Verzeichnis
- 👨‍💻 schritt_01: initiales Schema
- 👨‍💻 schritt_02: initiale Resolver-Funktionen (BeerAdvisorGraphQlController)

QUERIES AUSFÜHREN

Die Spec schreibt nicht vor, wie Queries auszuführen sind

- Typisch: über HTTP API
- Request ist dann (meist) ein HTTP Post Request
- Response ein JSON-Objekt mit den gelesenen Daten
- Transportschicht ist eher Implementierungsdetail

QUERIES AUSFÜHREN

Die Spec schreibt nicht vor, wie Queries auszuführen sind

- Typisch: über HTTP API
- Es gibt nur einen Endpunkt für die ganze API (z.B. /graphql)
- Problemlos parallel zu (bestehender) REST API zu betreiben

Live Beispiel: Schema Evolution

🕵️‍♂️ Query #1 ausführen

🕵️‍♂️ schritt_03: Rating im Schema ergänzen

- bestehende Implementierung funktioniert weiter
- (Die Mutation ist nicht implementiert, nur als Beispiel)

🕵️‍♂️ Query #1 ausführen => funktioniert immer noch unverändert

🕵️‍♂️ Query #2 mit Rating ausführen => funktioniert

GRAPHQL SCHEMA

Das Schema

- Es gibt nur eine Version

```
type Beer {  
    name: String!  
    price: Int!  
    type: BeerType!  
    created: DateTime  
}
```

GRAPHQL SCHEMA

Das Schema

- Es gibt nur eine Version
- Schema kann jederzeit erweitert werden, ohne bestehende Clients zu beeinflussen

```
type Beer {  
    name: String!  
    price: Int!  
    type: BeerType!  
    created: DateTime  
}
```

Evolution

```
type Beer {  
    name: String!  
    price: Int!  
    type: BeerType!  
    created: DateTime  
  
    ratings: [Rating!]  
}
```

```
type Rating {  
    id: ID!  
    comment: String  
    likes: Int  
}
```

Live Beispiel: Fehlerbehandlung

💡 Beispiel technischer Fehler: addRating-Mutation ausführen (oder Query mit fehlerhaftem Feld)

"Fachliche" Fehlerbehandlung

💡 schritt_04: addRating-Mutation bewerten (Rückgabe-Typ)

💡 Exemplarisch: "@deprecated"

💡 schritt_04: Union-Type (02_mutation_union_type.txt)

Fehlerbehandlung

- Fehler(-Antworten) werden in der Regel im Schema explizit vorgesehen
- Oder im error-Feld
- HTTP Status Codes spielen eher keine Rolle

PERFORMANCE TUNING

Performance von GraphQL Queries

Live Beispiel: Asynchrone Funktionen

- ⌚ Material von 05_async "aktivieren"
- ⌚ TracingInstrumentation einschalten
- ⌚ Query mit beers und averageStars

- ⌚ Problem beschreiben
- ⌚ Umstellen auf calculateAverageStars_async im Controller

Live-Coding Beispiel: Data Loader

- 💡 Material von 06_data_loader: Schema und RatingController
- 💡 Query mit user, wird aus MicroService abgefragt
- 💡 Console-Log vom UserService => doppelte Ids

- 💡 DataLoader im RatingController aktivieren

Performance von GraphQL Queries

- Resolver Funktionen werden per Default nacheinander ausgeführt

Performance von GraphQL Queries

- Resolver Funktionen werden per Default nacheinander ausgeführt
- Frameworks unterstützen typischerweise:
 1. Asynchrone Verarbeitung

Performance von GraphQL Queries

- Resolver Funktionen werden per Default nacheinander ausgeführt
- Frameworks unterstützen typischerweise:
 1. Asynchrone Verarbeitung
 2. "Data Loader" zum Zusammenfassen von Aufrufen externer Dienste (DB, ...)



Vielen Dank!

Slides & Source: <https://graphql.schule/myposter> (PDF)

Kontakt: nils@nilshartmann.net

Twitter: [@nilshartmann](https://twitter.com/nilshartmann)



Die GraphQL API muss in einem *Schema* beschrieben werden

- Das Schema legt fest, welche *Types* und *Fields* es gibt
- **Schema Definition Language (SDL)**

GRAPHQL SCHEMA

Schema Definition per SDL
Objekte

```
type Rating {  
}  
          ^----- Type (Object)
```

GRAPHQL SCHEMA

Schema Definition per SDL Objekte und Felder

```
type Rating {  
    id  
    comment  
    approved  
    stars  
}
```

Type (Object)

Felder

The diagram illustrates a GraphQL schema definition. It shows a single type named 'Rating' which contains four fields: 'id', 'comment', 'approved', and 'stars'. Dashed lines connect the type name 'Rating' to the label 'Type (Object)' and each of the four fields to the label 'Felder'.

GRAPHQL SCHEMA

Schema Definition per SDL

Objekte und Felder

```
type Rating {  
    id: ID!  
    comment: String! ----- Return Type (non-nullable)  
    approved: Boolean  
    stars: Int ----- Return Type (nullable)  
}
```

GRAPHQL SCHEMA

Schema Definition per SDL

Objekte und Felder

```
type Rating {  
    id: ID!  
    comment: String! ----- Return Type (non-nullable)  
    approved: Boolean  
    stars: Int ----- Return Type (nullable)  
}
```

👉 Felder sind konzeptionell **Funktionen**, die Werte zurückliefern

GRAPHQL SCHEMA

Schema Definition per SDL

Datentypen

```
type Rating {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    stars: Int  
}
```

Skalare Datentypen (Blätter)

GRAPHQL SCHEMA

Schema Definition per SDL Datentypen

```
type Rating {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    stars: Int  
}  
  
enum BeerType { pils, ipa, pale_ale }  
  
scalar DateTime  
  
type Beer {  
    name: String!  
    description: String!  
    type: BeerType!  
    created: DateTime  
}
```

Aufzählungstypen

GRAPHQL SCHEMA

Schema Definition per SDL Datentypen

```
type Rating {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    stars: Int  
}  
  
enum BeerType { pils, ipa, pale_ale }  
  
scalar DateTime  
  
type Beer {  
    name: String!  
    description: String!  
    type: BeerType!  
    created: DateTime  
}
```

Eigene Skalare

GRAPHQL SCHEMA

Schema Definition per SDL Datentypen

```
type Rating {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    stars: Int  
}  
  
enum BeerType { pils, ipa, pale_ale }  
  
scalar DateTime  
  
type Beer {  
    name: String!  
    description: String!  
    type: BeerType!  
    created: DateTime  
    ratings: [Rating!] }  
}
```

Listen

GRAPHQL SCHEMA

Schema Definition per SDL Argumente

```
type Rating {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    stars: Int  
}  
  
enum BeerType { pils, ipa, pale_ale }  
  
scalar DateTime  
  
type Beer {  
    name: String!  
    description: String!  
    type: BeerType!  
    created: DateTime  
    ratings: [Rating!]!  
    ratingsWithStars(start: Int!): [Rating!]!  
}
```

Argumente

GRAPHQL SCHEMA

Schema Definition per SDL Dokumentation

```
"""
A `Beer` is the main object in our service.

A `Beer` can have multiple **ratings**.

"""

type Beer {
    name: String!
    descriptions: String!
    ratings: [Rating!]!

    "Returns ratings with a minimal amount of stars."
    ratings(
        "Specifies number of stars"
        minStars: Int!
    ): [Rating]!
}
```

GRAPHQL SCHEMA

Root-Typen

Einstiegspunkte in den Graphen

Root-Type
("Query")

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

GRAPHQL SCHEMA

Root-Typen

Einstiegspunkte in den Graphen

Root-Type -----
("Query")

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

```
input NewRating { beerId: Id! stars: Int! }
```

Root-Type -----
("Mutation")

```
type Mutation {  
    addRating(newRating: NewRating!): Rating  
}
```

GRAPHQL SCHEMA

Root-Typen

Einstiegspunkte in den Graphen

Root-Type
("Query")

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

```
input NewRating { beerId: Id! stars: Int! }
```

Root-Type
("Mutation")

```
type Mutation {  
    addRating(newRating: NewRating!): Rating  
}
```

Root-Type
("Subscription")

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