

**NILS HARTMANN**  
<https://nilshartmann.net>

# GraphQL

Eine praktische Einführung  
am Beispiel Java

Slides (PDF): <https://react.schule/api-summit-2021-graphql>

# NILS HARTMANN

nils@nilshartmann.net

**Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg**

Java  
JavaScript, TypeScript  
React  
GraphQL

**Trainings & Workshops**



<https://reactbuch.de>

**HTTPS://NILSHARTMANN.NET**

# Kurze Umfrage...

<https://www.menti.com>

Code:

38 42 76 47

**Die Website bitte im Browser bis zum Ende der Session offen lassen!**

## AGENDA

### **1. GraphQL Grundlagen: wieso, weshalb, warum**

### **2. GraphQL für Java-Anwendungen**

- API implementieren
- Optimierung

**Jederzeit: Fragen, Diskussionen und Feedback!**

TEIL 1

# GraphQL

# Grundlagen

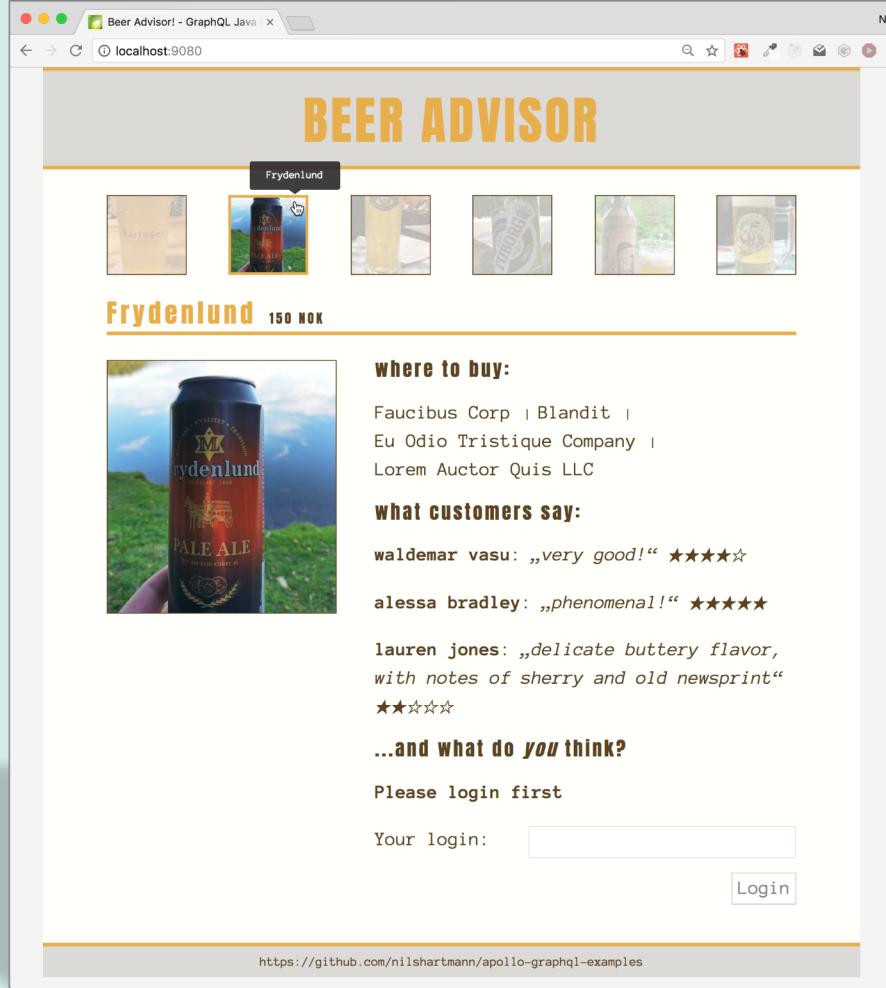
*"GraphQL is a **query language for APIs** and a **runtime for fulfilling those queries** with your existing data"*

- <https://graphql.org>

# GraphQL

*Spezifikation: <https://graphql.org/>*

- Umfasst:
  - Query Sprache und -Ausführung
  - Schema Definition Language
- Kein fertiges Produkt



# Beispiel Anwendung

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

The screenshot shows the GraphiQL interface running on localhost:9000. On the left, a code editor displays a GraphQL query named 'BeerAppQuery'. The query retrieves data for beers, ratings, and a ping. The 'beers' field is highlighted with a blue selection bar. On the right, the results of the query are displayed in a JSON-like format. The results include a list of beers with their names, prices, and associated ratings. A detailed description of each field is provided on the far right.

```
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": "3,80 EUR",  
        "ratings": [  
          {  
            "id": "R1",  
            "beerId": "B1",  
            "author": "Waldemar Vasu",  
            "comment": "Exceptional!"  
          },  
          {  
            "id": "R7",  
            "beerId": "B1",  
            "author": "Madhukar Kareem",  
            "comment": "Awwesome!"  
          },  
          {  
            "id": "R14",  
            "beerId": "B1",  
            "author": "Emily Davis",  
            "comment": "Off-putting buttery nose, laced  
with a touch of caramel and hamster cage."  
          }  
        ],  
        "id": "B2",  
        "name": "Frydenlund",  
        "price": "150 NOK",  
        "ratings": [  
          {  
            "id": "R2",  
            "beerId": "B2",  
            "author": "Andrea Gouyen",  
            "comment": "Very good!"  
          }  
        ]  
      }  
    ]  
  }  
}
```

Schema:

- beers: [Beer!]!
- beer(beerId: String): Beer
- ratings: [Rating!]!
- ping: ProcessInfo!

Query:

- No Description

FIELDS:

- beers: [Beer!]!
- beer(beerId: String): Beer
- ratings: [Rating!]!
- ping: ProcessInfo!

# Demo: GraphiQL

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

<http://localhost:9000>

The screenshot shows the IntelliJ IDEA interface with the "graphql-java-example" project open. The "a.graphql" file is selected in the left sidebar under "Project". The main editor window displays a GraphQL query:

```
query {
  beer(beerId: "B1") {
    price
    name
  }
  shops {
    address
    beers
    id
    name
    ...
    __typename
  }
}
```

A tooltip is displayed over the "beers" field, providing the following information:

- address - Address of the shop
- beers - All Beers this shop sells
- id - Unique ID of this shop
- name - The name of the shop
- ...
- \_\_typename

The bottom panel shows the JSON response from the GraphQL endpoint:

```
{"data": {"beer": {"price": "3,80 EUR", "ratings": [{"author": {"login": "waldemar"}, "rating": 5}, {"author": {"login": "karl"}, "rating": 4}], "name": "Bavarian Lager"}, "shops": [{"address": "Münchener Platz 12", "beers": [{"name": "Bavarian Lager", "rating": 4.5}, {"name": "Premium Pils", "rating": 4.2}, {"name": "Hefe-Weissbier", "rating": 4.8}, {"name": "Dark Porter", "rating": 4.6}, {"name": "IPA", "rating": 4.4}], "id": "S1", "name": "Bavarian Beer Shop"}]}
```

The status bar at the bottom indicates "a.graphql: 93 ms execution time, 140 bytes response".

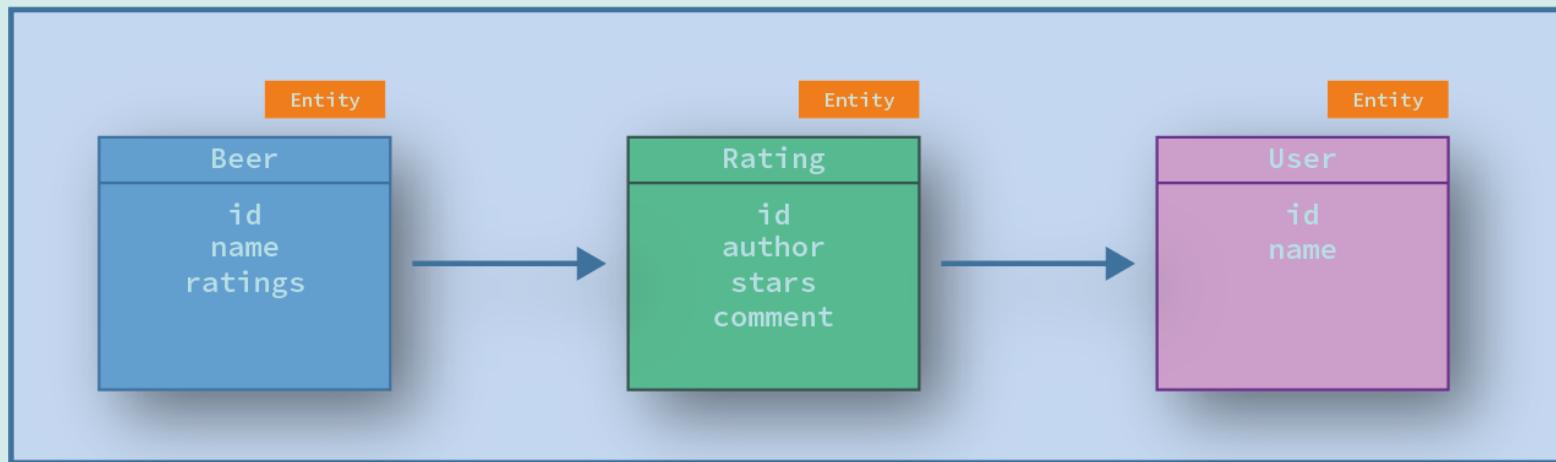
# Demo: IntelliJ

<https://plugins.jetbrains.com/plugin/8097-js-graphq>

# **Vergleich mit REST**

# BEERADVISOR DOMAINE

## "Domain-Model"

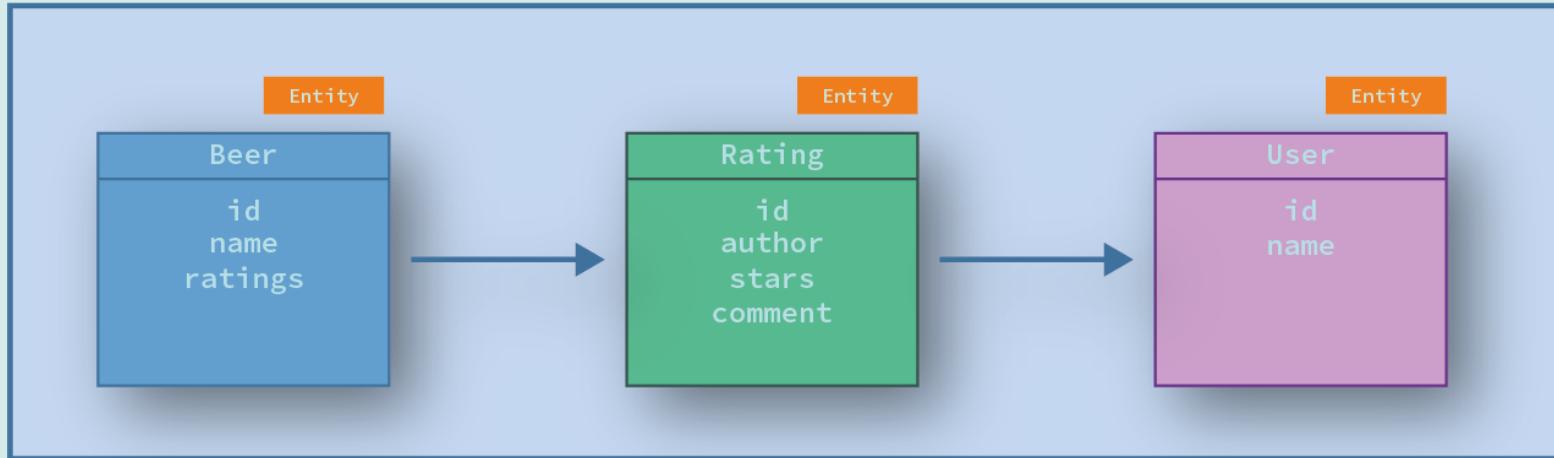


# BEERADVISOR DOMAINE

Beer Advisor: Wie könnte dafür eine **REST-API** aussehen? 🤔



Miro-Board-Link



## BEER ADVISOR

The mobile application interface for Beer Advisor displays a grid of beer reviews. Each review includes a small image of the beer, the name of the beer, its rating (e.g., ★★★★), and a brief customer comment.

Beer Name	Rating	Customer Comment
Barfüßer	★★★★	Barfüßer
Frydenlund	★★★★★	Frydenlund
Grieskirchner	★★★★★	Grieskirchner
TUBORG	★★★★	TUBORG
Baltic Tripple	★★★★★	Baltic Tripple
Viktoria Bier	★★★★	Viktoria Bier

## BEER ADVISOR

**Frydenlund 150 NOK**

This screen shows a detailed product page for Frydenlund Pale Ale. It features a large image of the can, the price (150 NOK), and sections for where to buy and what customers say.

**where to buy:**

Eu Corp. | Blandit Nam LLP |  
Quisque Inc. | In Lorem Donec LLC |  
Eu Odio Tristique Company |  
Lorem Auctor Quis LLC

**what customers say:**

waldemar vasu: „very good!“ ★★★★☆  
alessa bradley: „phenomenal!“ ★★★★★  
lauren jones: „delicate buttery flavor, with notes of sherry and old newsprint“ ★★★★★

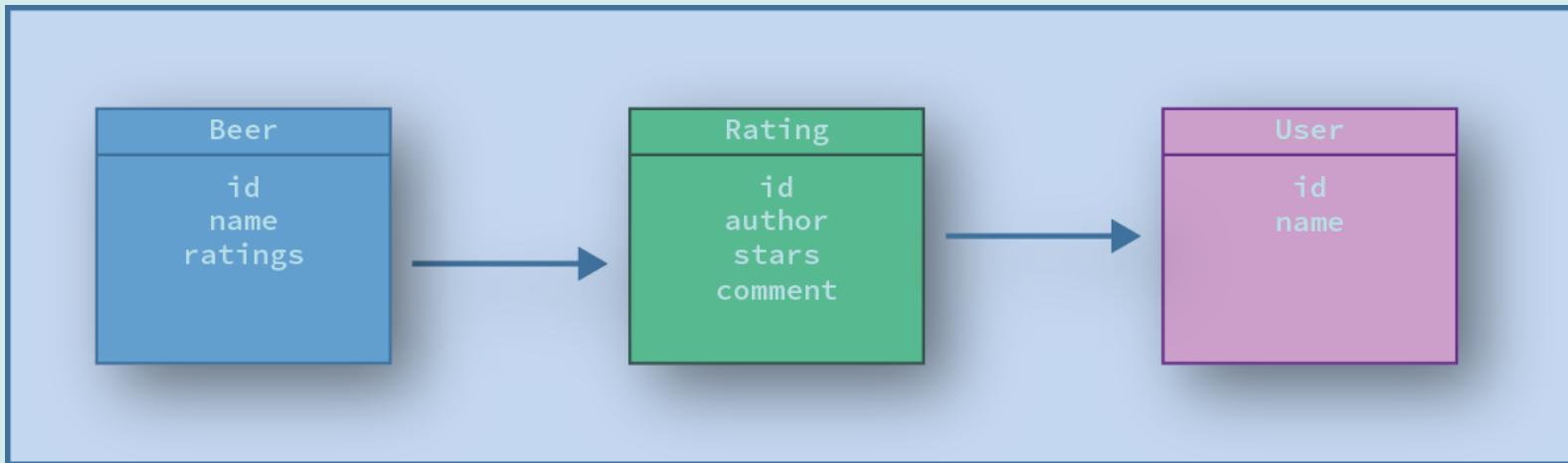
**...and what do you think?**

# ABFRAGEN MIT REST

## REST-Zugriff

- Exemplarisch und vereinfacht: Der **Autor** eines bestimmten **Ratings** eines bestimmten **Biers**

GET /beer/1

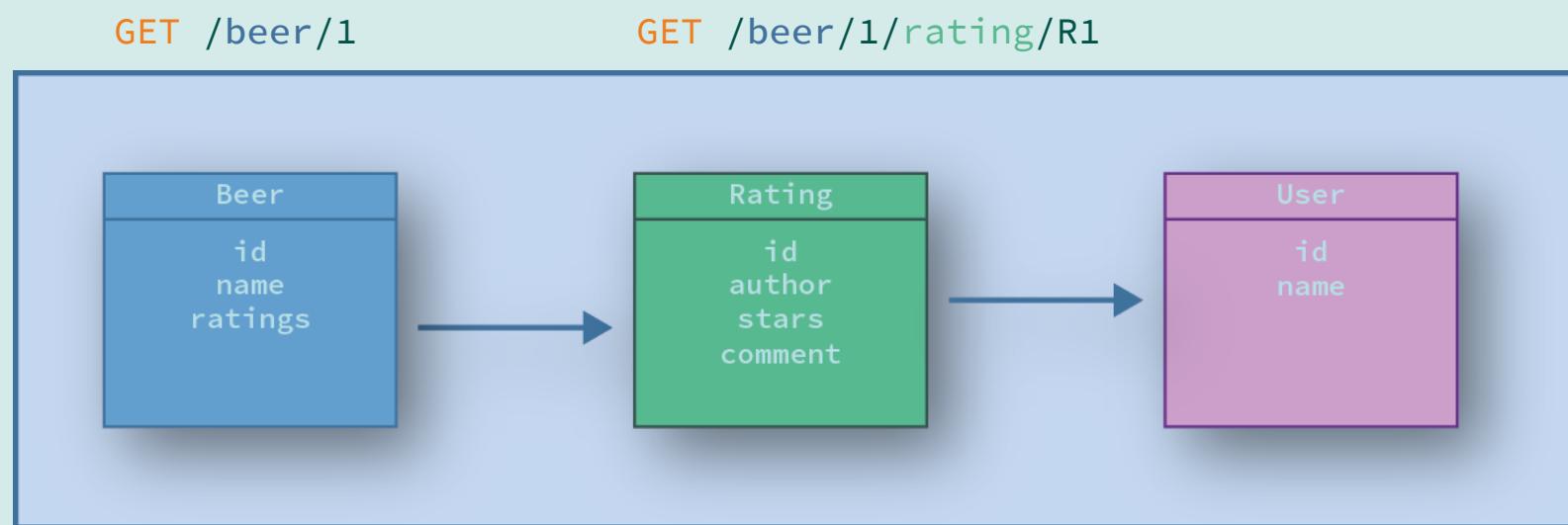


```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

# ABFRAGEN MIT REST

## REST-Zugriff

- Exemplarisch und vereinfacht: Der **Autor** eines bestimmten **Ratings** eines bestimmten **Biers**



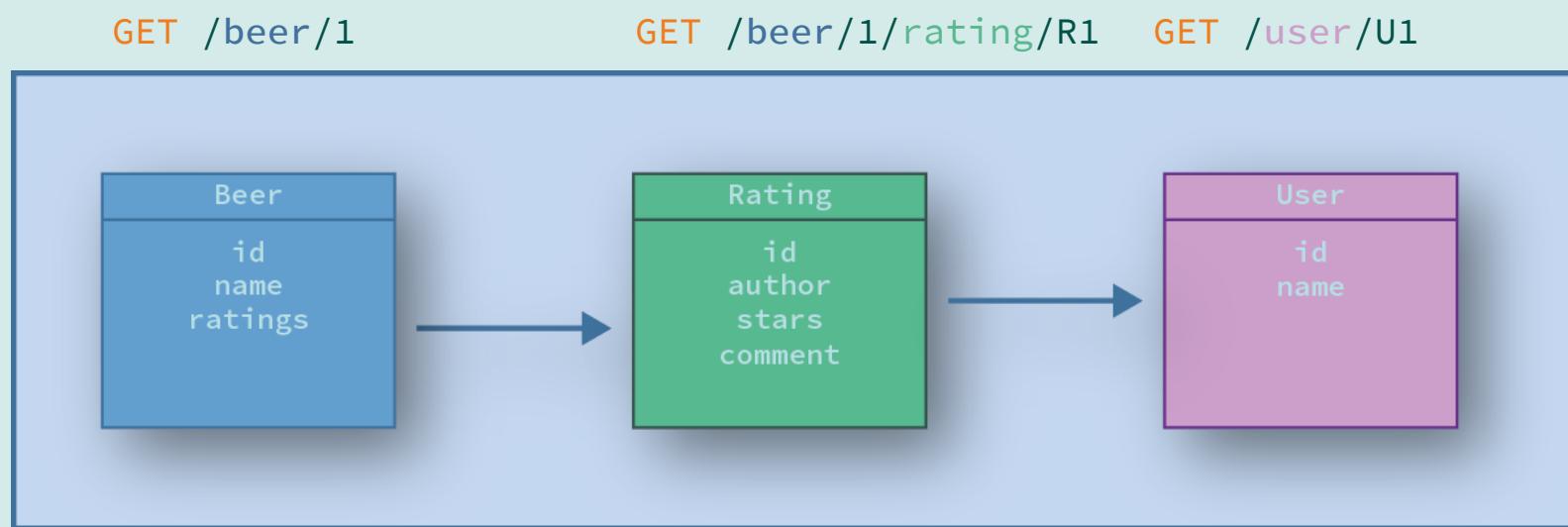
```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

# ABFRAGEN MIT REST

## REST-Zugriff

- Exemplarisch und vereinfacht: Der **Autor** eines bestimmten **Ratings** eines bestimmten **Biers**



```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

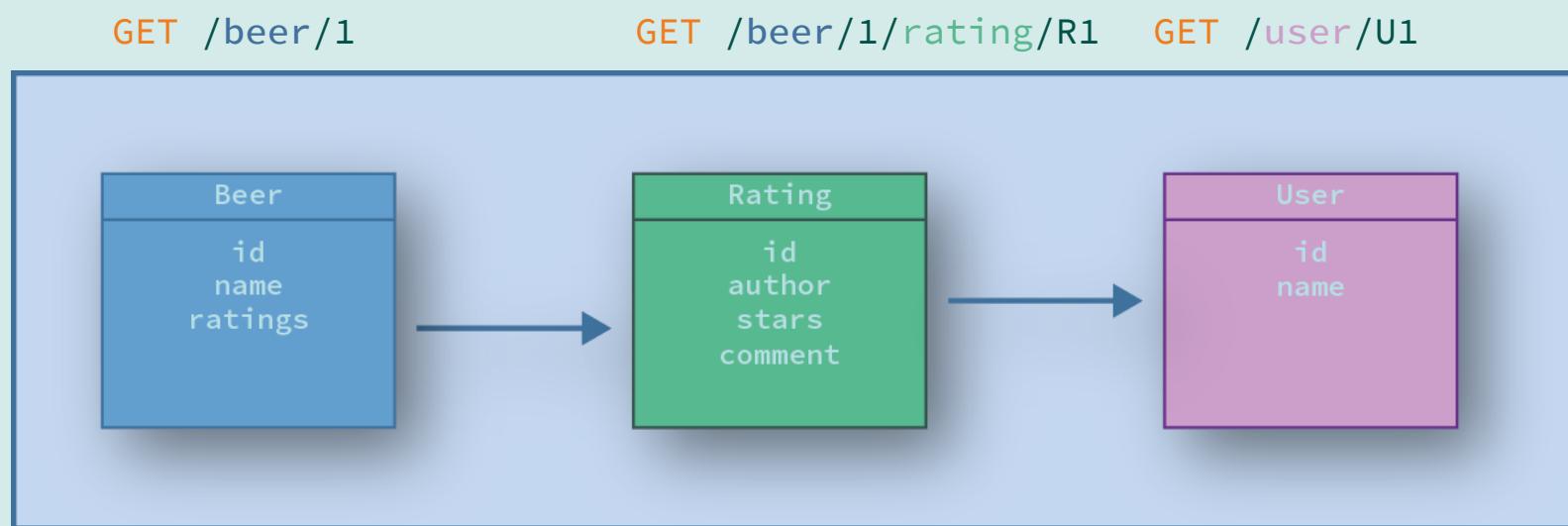
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

```
{  
  "id": "U1",  
  "name": "Klaus",  
}
```

# ABFRAGEN MIT REST

## REST-Zugriff

- Exemplarisch und vereinfacht: Der **Autor** eines bestimmten **Ratings** eines bestimmten **Biers**
- Ebenfalls vereinfacht: es kommt immer ein ganzes Objekt zurück



```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

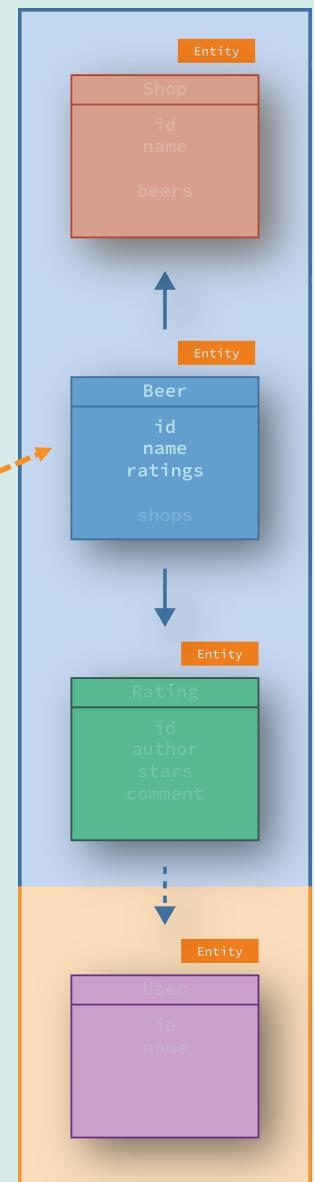
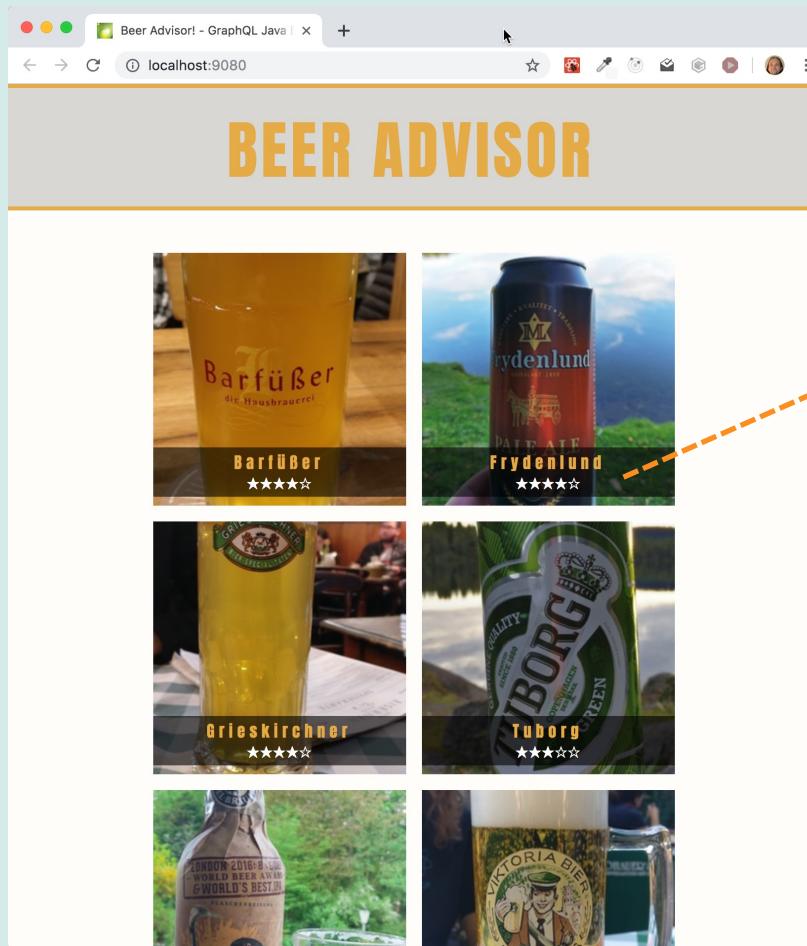
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

```
{  
  "id": "U1",  
  "name": "Klaus",  
}
```

# GRAPHQL QUERIES

## Use-Case spezifische Abfragen – 1

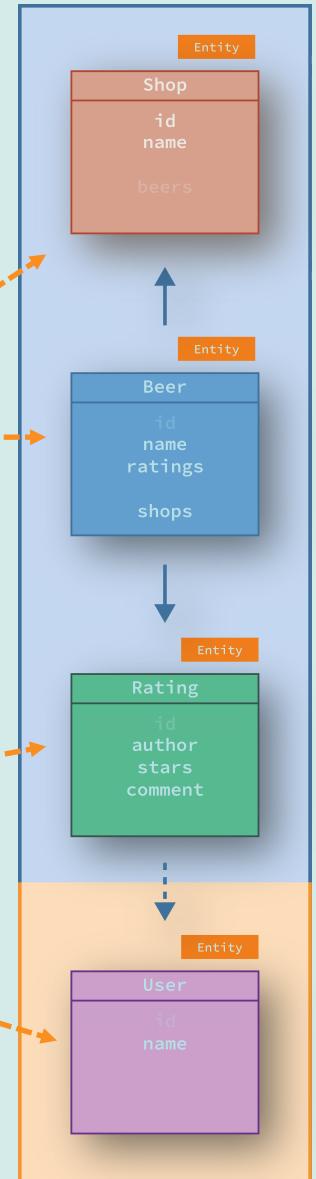
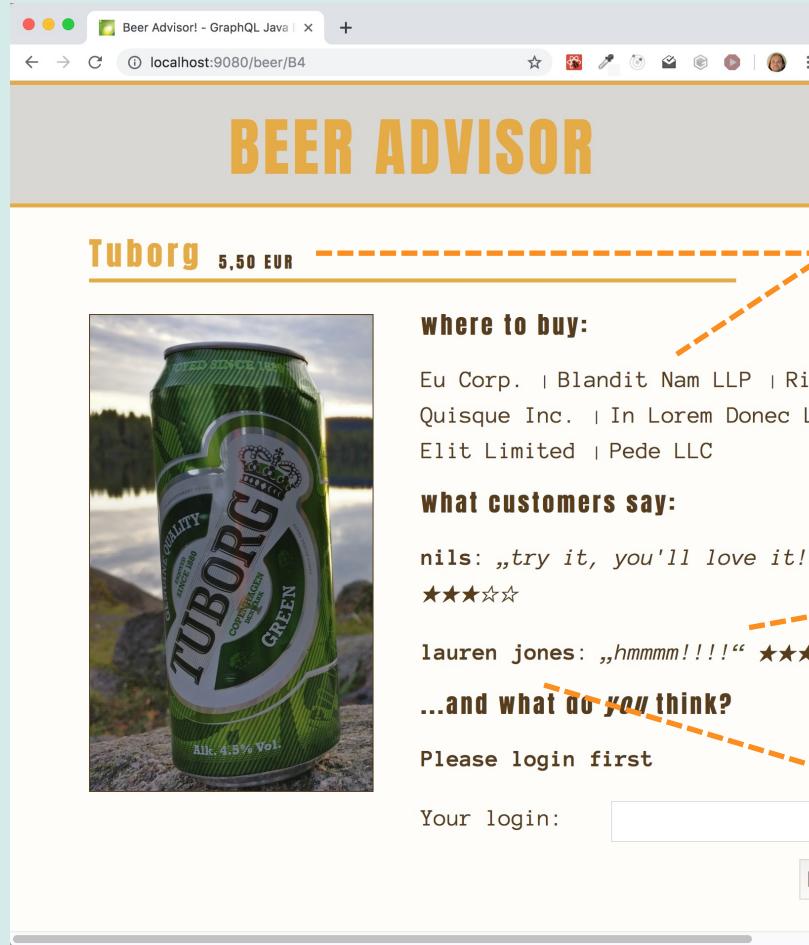
```
{ beer {  
    id  
    name  
    averageStars  
}
```



# GRAPHQL QUERIES

## Use-Case spezifische Abfragen – 2

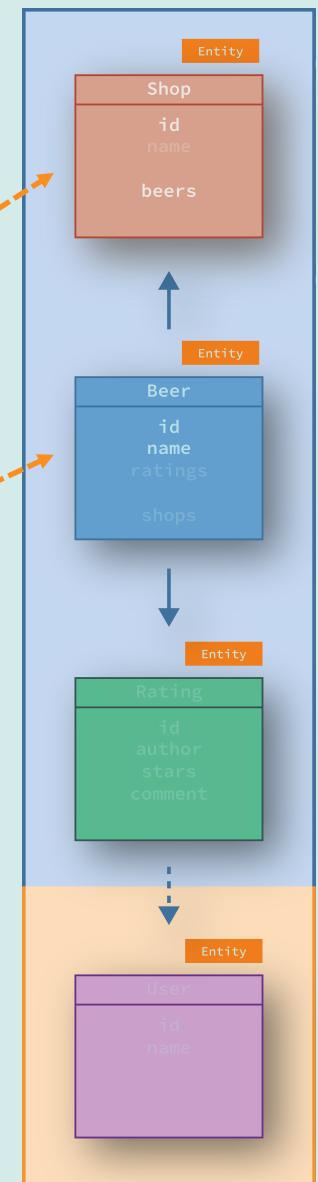
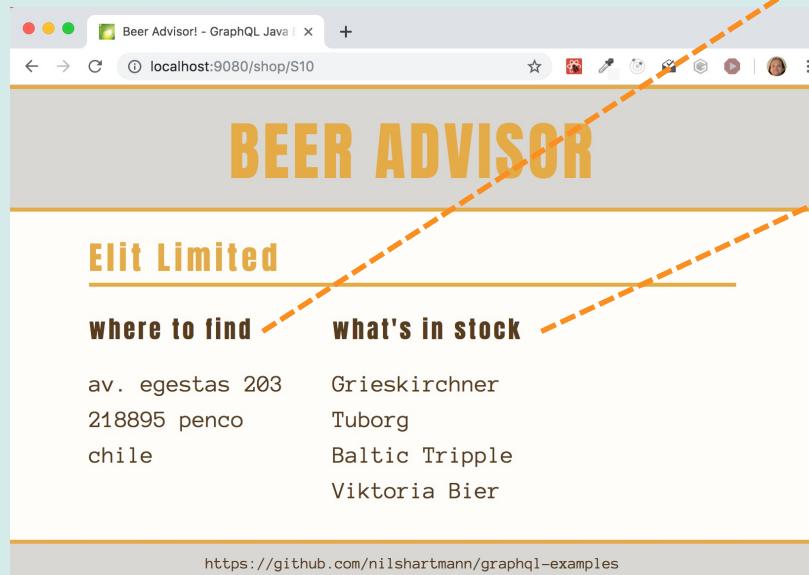
```
{ beer(beerId: "B1" {  
    name  
    price  
    ratings {  
        stars  
        comment  
        author {  
            name  
        }  
    }  
    shops { name }  
}
```



# GRAPHQL QUERIES

## Use-Case spezifische Abfragen – 3

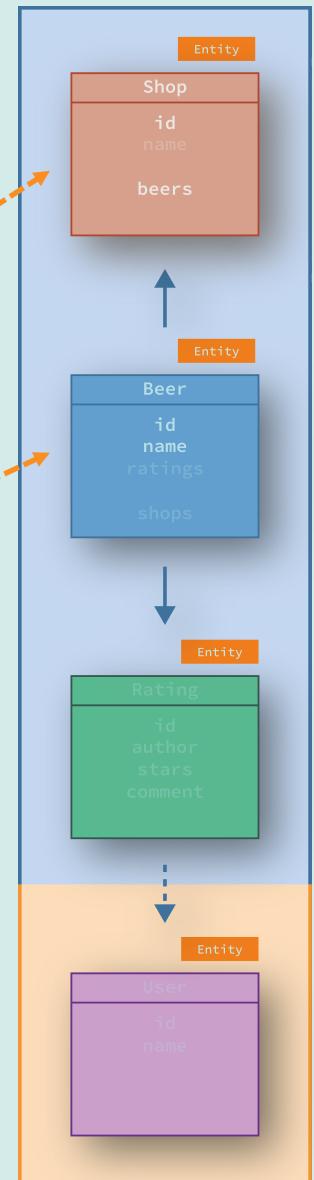
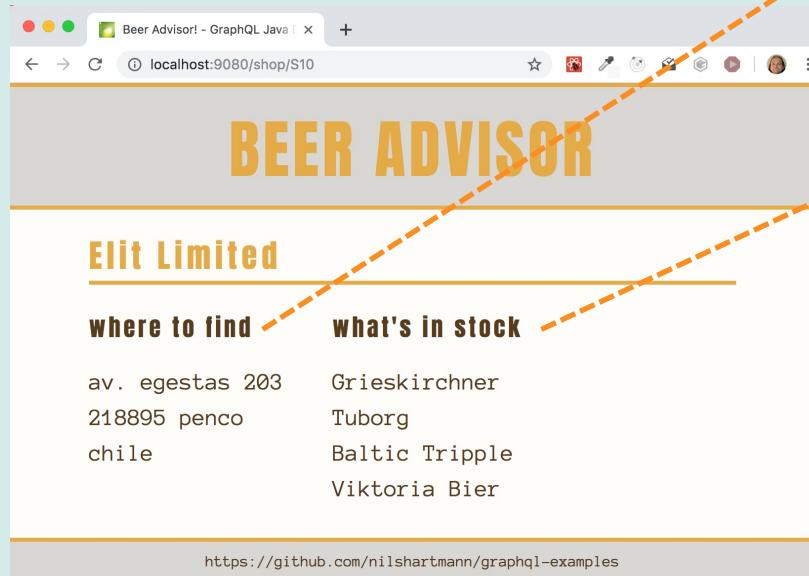
```
{ shop(shopId: "S3") {  
    name  
    address { street city }  
    beers { id name }  
}
```



# GRAPHQL QUERIES

## Use-Case spezifische Abfragen – 3

```
{ shop(shopId: "S3") {  
    name  
    address { street city }  
    beers { id name }  
}
```



Abgefragt werden Daten, nicht Endpunkte 😈

### Wir veröffentlichen mit REST eine fachliche API

- Welche Daten wir zur Verfügung stellen ist unsere Aufgabe
- Wir legen fest, in welcher Form die Daten zur Verfügung gestellt werden

👉 Wir legen damit explizit selbst fest, wie unsere API aussehen soll

### Wir veröffentlichen mit GraphQL eine fachliche API

- Welche Daten wir zur Verfügung stellen ist unsere Aufgabe
- Wir legen fest, in welcher Form die Daten zur Verfügung gestellt werden

👉 Wir legen damit explizit selbst fest, wie unsere API aussehen soll

## Wir veröffentlichen mit GraphQL eine fachliche API

- Welche Daten wir zur Verfügung stellen ist unsere Aufgabe
- Wir legen fest, in welcher Form die Daten zur Verfügung gestellt werden

👉 Wir legen damit explizit selbst fest, wie unsere API aussehen soll

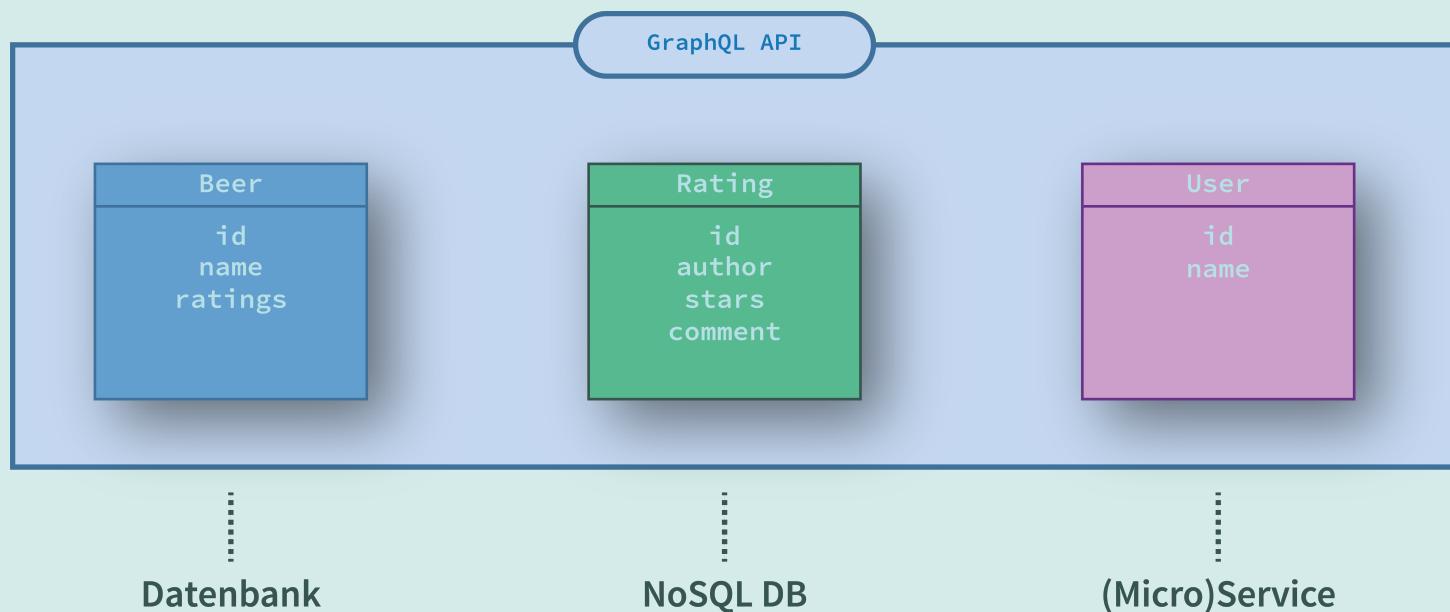
👉 Auch GraphQL erzeugt die API nicht auf „magische“ Weise selbst

- API und API-Zugriffe sind typsicher
- Sehr gutes Tooling vorhanden
- Viel aus einer Hand

# DATEN QUELLEN

**GraphQL macht keine Aussage, wo die Daten herkommen**

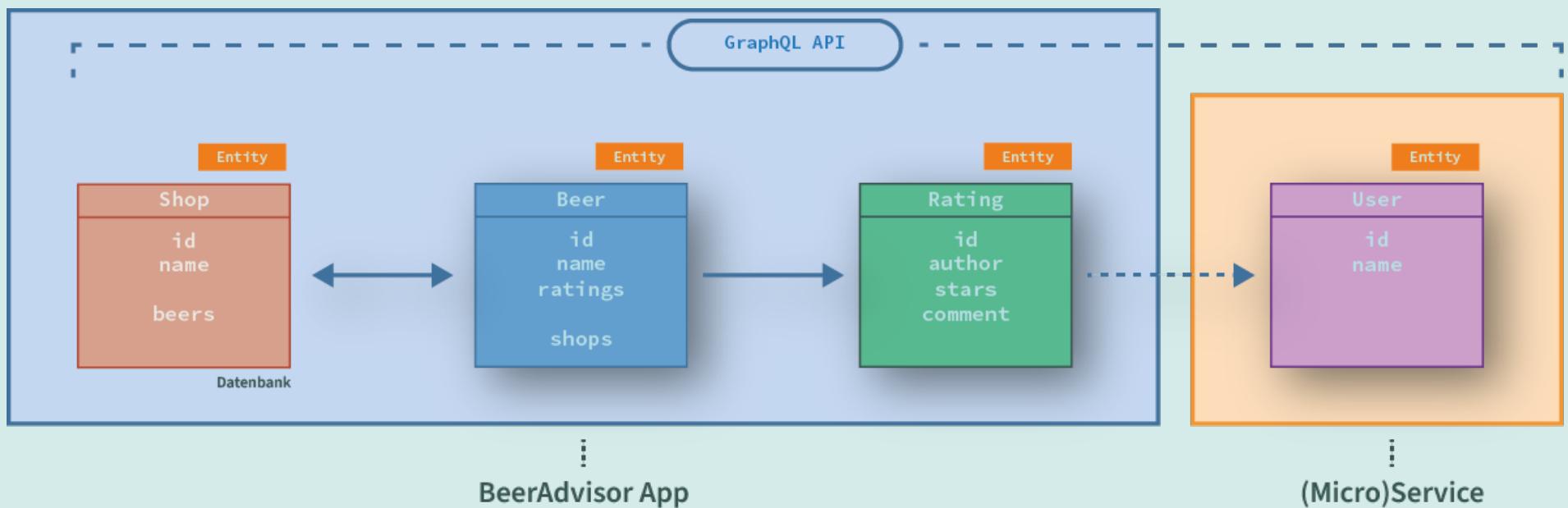
👉 Ermittlung der Daten ist unsere Aufgabe



# HINTERGRUND

## "Architektur" Beer Advisor

erst später, wenn überhaupt auf User Service zugegriffen wird



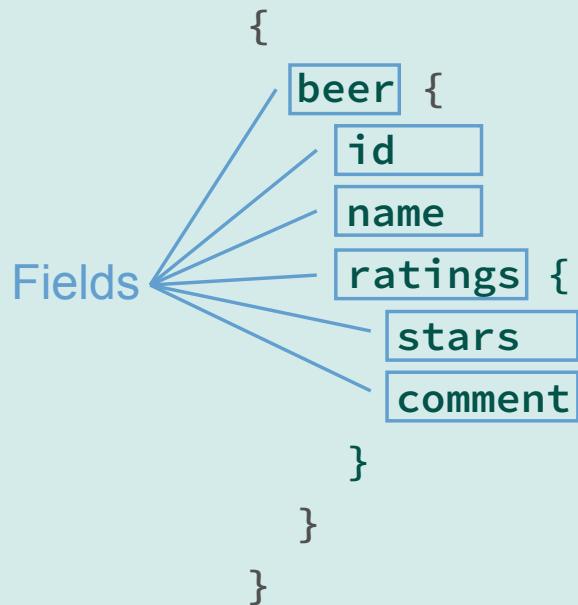
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

# GraphQL

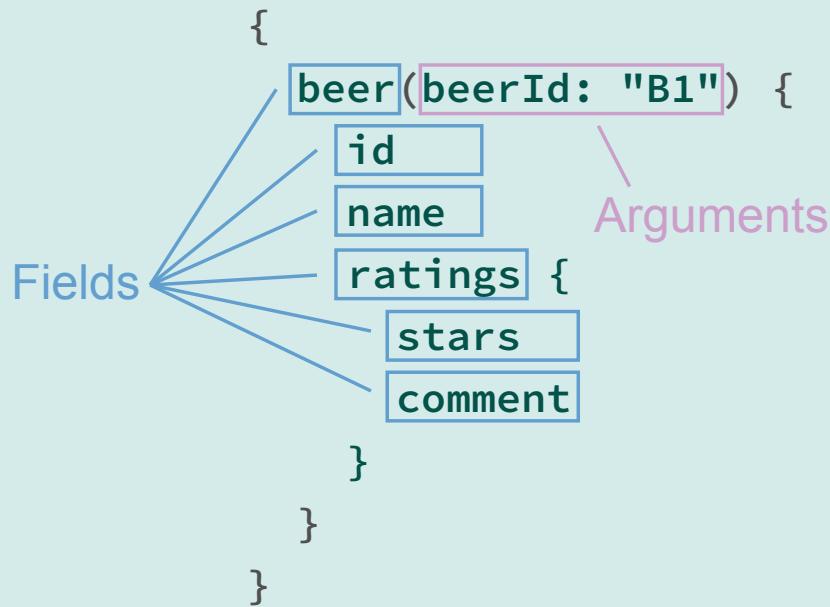
**TEIL 1: ABFRAGEN UND SCHEMA**

# QUERY LANGUAGE



- Strukturierte Sprache, um Daten von der API abzufragen
- Abgefragt werden **Felder** von (verschachtelten) Objekten

# QUERY LANGUAGE



- Strukturierte Sprache, um Daten von der API abzufragen
- Abgefragt werden **Felder** von (verschachtelten) Objekten
- Felder können **Argumente** haben

# QUERY LANGUAGE

## Ergebnis

```
{  
  beer(beerId: "B1") {  
    id  
    name  
    ratings {  
      stars  
      comment  
    }  
  }  
}
```



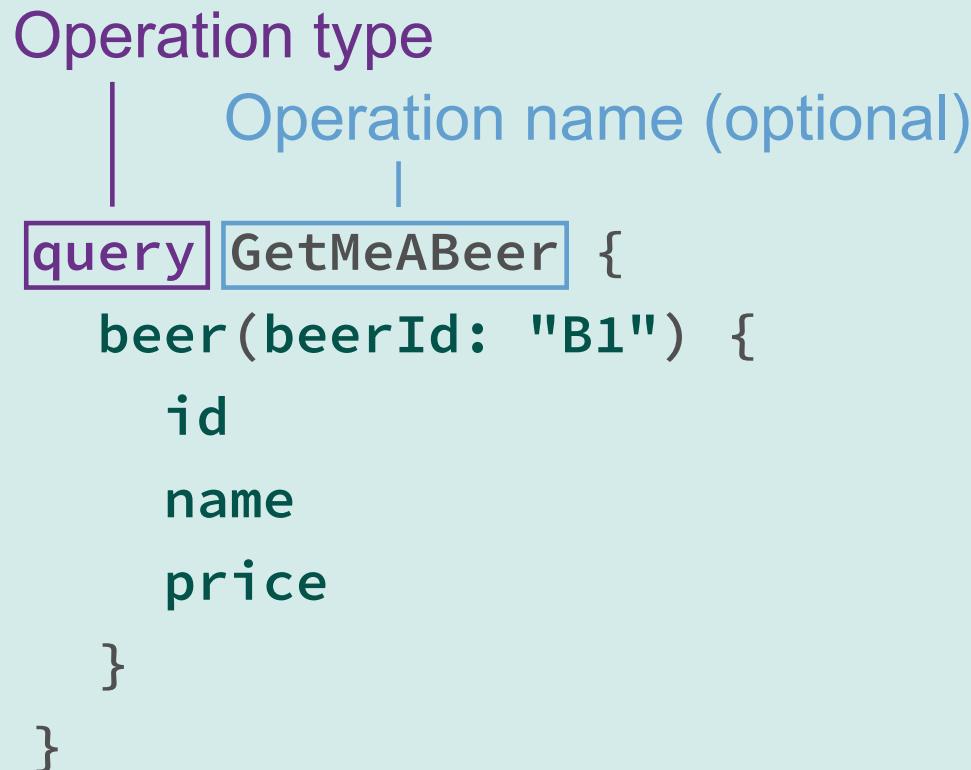
```
"data": {  
  "beer": {  
    "id": "B1"  
    "name": "Barfüßer"  
    "ratings": [  
      {  
        "stars": 3,  
        "comment": "grate taste"  
      },  
      {  
        "stars": 5,  
        "comment": "best beer ever!"  
      }  
    ]  
  }  
}
```

- Identische Struktur wie bei der Abfrage
- *Query ist ein String, kein JSON!*

# QUERY LANGUAGE: OPERATIONS

**Operation:** beschreibt, was getan werden soll

- query, mutation, subscription



# QUERY LANGUAGE: OPERATIONS

## Operation: Variablen

```
query GetMeABeer($bid: ID!) {  
  beer(beerId: $bid) {  
    id  
    name  
    price  
  }  
}
```

Variable Definition  
|  
query GetMeABeer(**\$bid: ID!**) {  
 beer(beerId: **\$bid**) {  
 id  
 name  
 price  
 }  
}  
Variable usage

# QUERY LANGUAGE: MUTATIONS

## Mutations

- Mutation wird zum Verändern von Daten verwendet
- Entspricht POST, PUT, PATCH, DELETE in REST
- Rückgabe Wert kann frei definiert werden (z.B. neue Entität)

Operation type  
| Operation name (optional)      Variable Definition  
|  
`mutation AddRatingMutation($input: AddRatingInput!) {  
 addRating(input: $input) {  
 id  
 beerId  
 author  
 comment  
 }  
}`

`"input": {  
 beerId: "B1",  
 author: "Nils", — Variable Object  
 comment: "YEAH!"  
}`

# QUERY LANGUAGE: MUTATIONS

## Subscription

- Automatische Benachrichtigung bei neuen Daten
- API definiert Events (mit Feldern), aus denen der Client auswählt

Operation type

  |

    Operation name (optional)

    |

    subscription **NewRatingSubscription** {

      newRating: onNewRating {

        |

        Field alias     id

                     beerId

                     author

                     comment

      }

    }

# QUERY LANGUAGE: FRAGMENTS

## Fragments

- Es müssen alle Felder explizit angegeben werden (kein \* möglich)
- Fragmente erlauben wiederverwendbare "Sub-Queries"

Fragment name  
|  
`fragment RatingWithUserAndBeer on Rating {  
 comment  
 beer { name }  
 author { name }  
}`

# QUERY LANGUAGE: FRAGMENTS

## Fragments

- Es müssen alle Felder explizit angegeben werden (kein \* möglich)
- Fragmente erlauben wiederverwendbare "Sub-Queries"

```
Fragment name
|
fragment RatingWithUserAndBeer on Rating {
    comment
    beer { name }
    author { name }
}

query Beer {
    beer(beerId: "B1") {
        ratings {
            ...RatingWithUserAndBeer
        }
    }
}
```

# QUERIES AUSFÜHREN

## Queries werden über HTTP ausgeführt

- Üblicherweise per POST
- Ein *einzelner* Endpoint, z.B. /graphql

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

```
{"data":  
  {"beers": [  
    {"name": "Barfüßer"},  
    {"name": "Frydenlund"},  
    {"name": "Grieskirchner"},  
    {"name": "Tuborg"},  
    {"name": "Baltic Triple"},  
    {"name": "Viktoria Bier"}  
  ]}  
}
```

# QUERIES AUSFÜHREN

## Antwort vom Server

- Grundsätzlich HTTP 200
  - HTTP Status Codes spielen keine Rolle!
- (JSON-)Map mit max drei Feldern

```
{  
  "errors":  
  [  
    { "message": "Could not read User with ID 123",  
      "locations": [ . . . ],  
      "path": [ "beer", "ratings", "author" ]  
    }  
  ],  
  "data": {"beers": [ . . . ] },  
  "extensions": { . . . }  
}
```

## QUERIES AUSFÜHREN

### Queries werden über HTTP ausgeführt

- Der GraphQL-Endpunkt kann parallel zu anderen Endpunkten in der Anwendung bestehen
  - REST und GraphQL kann problemlos gemischt werden
- Wie die Anbindung aussieht hängt vom Framework und Umgebung (Spring / JEE) ab

# ÜBUNG: ARBEITEN MIT GRAPHQL

## GraphQL Queries ausführen

- GraphiQL für den BeerAdvisor läuft unter ... (siehe Chat)
- Öffne GraphiQL, melde dich mit einem der angezeigten User an
- Führe ein paar GraphQL Queries aus, zum Beispiel:
  - Lese alle Biere
  - Erzeuge ein Rating für ein bestehendes Bier
    - Als userId musst Du die ID verwenden, die in GraphiQL oben links neben dem Usernamen angezeigt wird (U...)

TEIL II

# GraphQL Server

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

- <https://graphql.org>

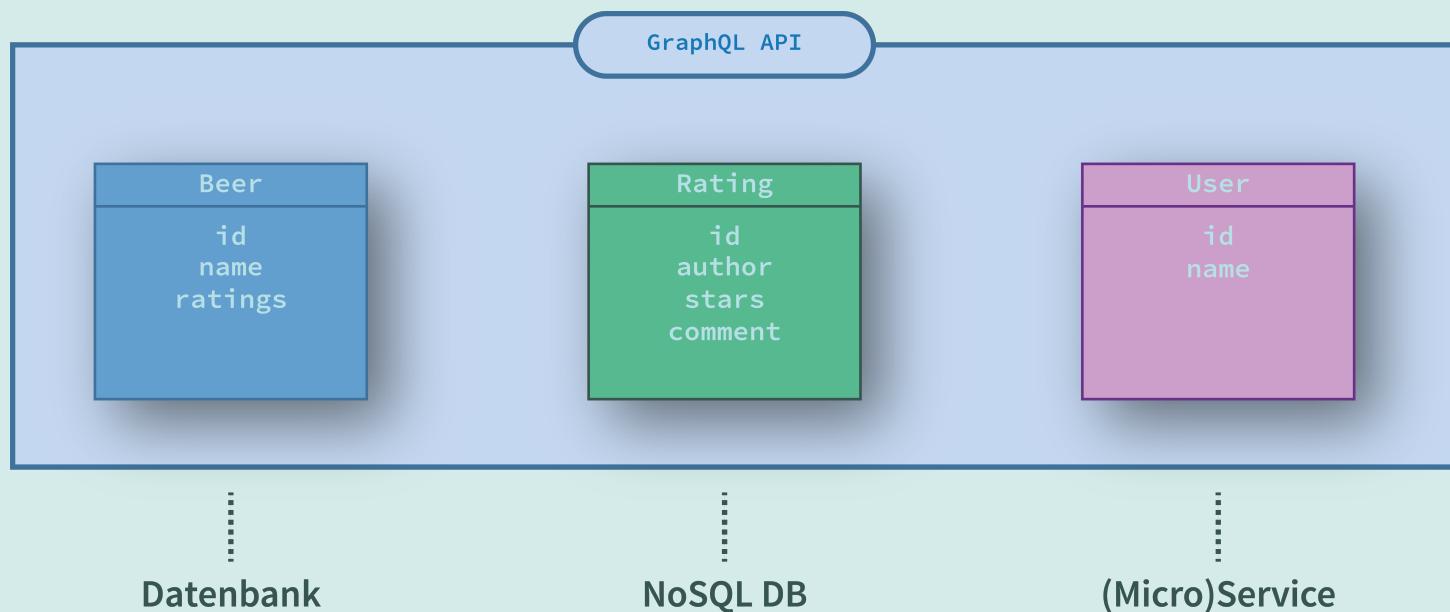
# GraphQL Server

**RUNTIME (AKA: YOUR APPLICATION)**

# GRAPHQL APIs

**GraphQL macht keine Aussage, wo die Daten herkommen**

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



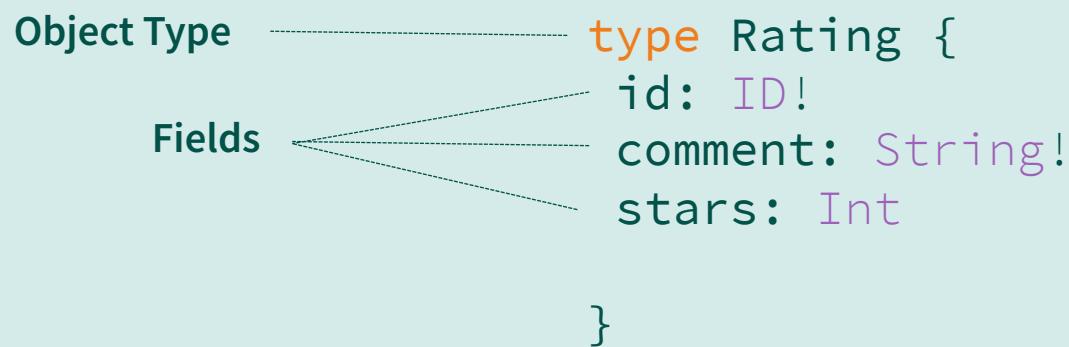
## GRAPHQL SCHEMA

Die GraphQL API muss in einem *Schema* beschrieben werden

- Eine GraphQL API muss mit einem *Schema* beschrieben werden
- Schema legt fest, welche *Types* und *Fields* es gibt
- Nur Anfragen und Ergebnisse, die Schema-konform sind werden ausgeführt bzw. zurückgegeben
- **Schema Definition Language** (SDL)

# GRAPHQL SCHEMA

# Schema Definition per SDL



# GRAPHQL SCHEMA

## Schema Definition per SDL

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

# GRAPHQL SCHEMA

## Schema Definition per SDL

```
type Rating {  
    id: ID!  
    comment: String!  
    stars: Int  
    author: User! ----- Referenz auf anderen Typ  
}  
  
type User {  
    id: ID!  
    name: String!  
}
```



# GRAPHQL SCHEMA

## Schema Definition per SDL

```
type Rating { ←  
  id: ID!  
  comment: String!  
  stars: Int  
  author: User!  
}  
  
type User {  
  id: ID!  
  name: String!  
}  
  
type Beer {  
  name: String!  
  ratings: [Rating!]! ----- Liste / Array  
}  
}
```

# GRAPHQL SCHEMA

## Schema Definition per SDL

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

```
type User {  
    id: ID!  
    name: String!  
}
```

```
type Beer {  
    name: String!  
    ratings: [Rating!]!  
    ratingsWithStars(stars: Int!): [Rating!]!  
}
```

Arguments

# GRAPHQL SCHEMA

## Root-Types: Einstiegspunkte in die API (Query)

Root-Type  
("Query")

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

# GRAPHQL SCHEMA

## Root-Types: Einstiegspunkte in die API (Query)

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

# GRAPHQL SCHEMA

## Root-Types: Einstiegspunkte in die API (Query, Mutation)

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

Root-Type  
("Mutation")

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

# GRAPHQL SCHEMA

## Root-Types: Einstiegspunkte in die API (Query, Mutation)

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

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

```
input NewRating {  
    authorId: ID!  
    comment: String!  
}
```

**Input-Object** -----  
(für komplexe  
Argumente)

# GRAPHQL SCHEMA

## Root-Types: Einstiegspunkte in die API (Query, Mutation, Subscription)

```
type Query {  
    beers: [Beer!]!  
    beer(beerId: ID!): Beer  
}  
  
type Mutation {  
    addRating(newRating: NewRating): Rating!  
}  
  
input NewRating {  
    authorId: ID!  
    comment: String!  
}  
  
type Subscription {  
    onNewRating: Rating!  
}
```

Root-Type  
(Subscription)

## SCHEMA WEITERENTWICKLUNG

**Nur eine Version:** Felder werden immer explizit abgefragt

- Es können "ohne Schaden" neue Felder hinzugefügt werden

Neues Feld .....

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

## SCHEMA WEITERENTWICKLUNG

### Nur eine Version: Felder werden immer explizit abgefragt

- Es können "ohne Schaden" neue Felder hinzugefügt werden
- Alte Felder können 'deprecated' werden
- Verwendung der Felder kann einzeln getrackt werden

**Neues Feld** -----

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

# GRAPHQL SCHEMA

## Schema: Instrospection

- Root-Felder "\_\_schema" und "\_\_type" (Beispiel)

```
query {  
  __type(name: "Beer") {  
    name  
    kind  
    description  
    fields {  
      name description  
      type { ofType { name } }  
    }  
  }  
}
```

```
{  
  "data": {  
    "__type": {  
      "name": "Beer",  
      "kind": "OBJECT",  
      "description": "Representation of a Beer",  
      "fields": [ {  
        "name": "id", "description": "Id for this Beer",  
        "type": { "ofType": { "name": "ID" } }  
      },  
      {  
        "name": "price", "description": "Price of the beer",  
        "type": { "ofType": { "name": "Int" } }  
      },  
      ...  
    ]  
  }  
}
```

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

- <https://graphql.org>

# GraphQL (für Java)

**TEIL 2: RUNTIME-UMGEBUNG (AKA: EURE ANWENDUNG)**

## Grundsätzliche Optionen

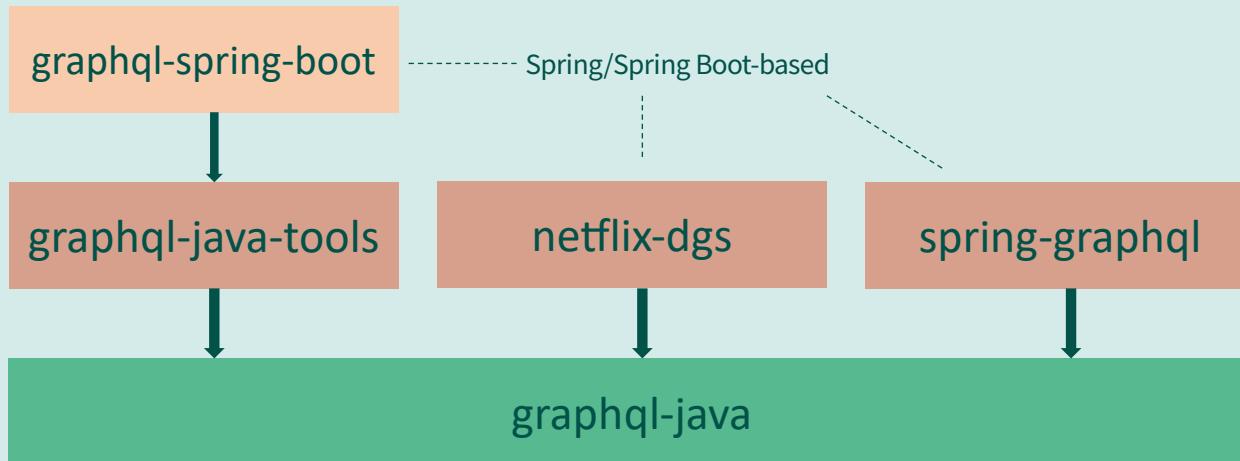
- **graphl-java**: das erste GraphQL Framework für Java (2015!)
  - Low level API, kein Support für Server etc.

graphql-java

# GRAPHQL FÜR JAVA ANWENDUNGEN

## Grundsätzliche Optionen

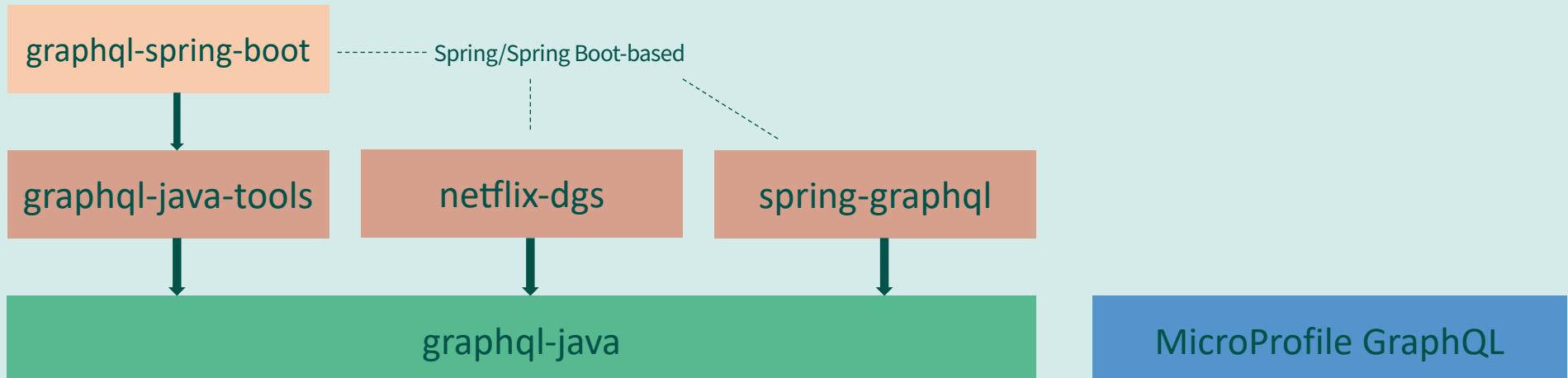
- **graphl-java**: das erste GraphQL Framework für Java (2015!)
  - Low level API, kein Support für Server etc.
  - Verschiede Abstraktionen existieren dafür, insb. für Spring Boot



# GRAPHQL FÜR JAVA ANWENDUNGEN

## Grundsätzliche Optionen

- **graphl-java**: das erste GraphQL Framework für Java (2015!)
  - Low level API, kein Support für Server etc
  - Verschiede Abstraktionen existieren dafür, insb. für Spring Boot
- **MicroProfile**



# GRAPHQL FÜR JAVA ANWENDUNGEN

**graphql-java**

# GRAPHQL FÜR JAVA ANWENDUNGEN

graphql-java



jbellenger 20 days ago (18.10.2021)

...

Hi team! Twitter is wrapping up its migration to graphql-java and I wanted to share some info about how we use graphql and why we chose to migrate our system to graphql-java.

Some background on how we use graphql: we use one unified schema to serve data to first-party twitter clients and to power our rest api. The schema defines the entirety of our api data model and includes roughly 1000 object types, 100 input types, and 300 mutation fields. The schema itself grows daily as several hundred developers across the company add fields and types to the schema to support their projects.

Our graphql api serves around 500k requests per second, and the nature of our api is that some of our largest and most complex queries have the highest request rate. For example, one of our most frequently-executed queries is 2500 lines long and returns 50k fields per request. That's a lot of fields!

# GRAPHQL FÜR JAVA ANWENDUNGEN

**graphql-java** <https://www.graphql-java.com/>

- Reine GraphQL Implementierung
  - Keine Abhängigkeiten auf weitere Libraries
  - Keine Server-Infrastruktur (unabhängig von Spring und JEE)
  - „Nur“ Ausführung von Queries
- API ist sehr low-level

# GRAPHQL SERVER MIT GRAPHQL-JAVA

## Unsere Aufgaben in der Entwicklung

### 1. Das Schema der API festlegen

- graphql-java verfolgt „schema-first“-Ansatz

## Unsere Aufgaben in der Entwicklung

1. Das Schema der API festlegen
  - graphql-java verfolgt „schema-first“-Ansatz
2. *DataFetchers* implementieren, die die angefragten Daten ermitteln
  - Dieser Teil unterscheidet sich von den Frameworks, die auf graphql-java aufbauen

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Schritt 1: Schema definieren

- Wie bereits gesehen

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Rating {  
    id: ID!  
    beer: Beer!  
    author: User!  
    comment: String!  
    stars: Int!  
}  
  
type Beer {  
    id: ID!  
    name: String!  
    price: String!  
    ratings: [Rating!]!  
    ratingsWithStars(stars: Int!): [Rating!]!  
}  
  
type Query {  
    beer(beerId: ID!): Beer  
    beers: [Beer!]!  
}  
  
input AddRatingInput {  
    beerId: ID!  
    userId: ID!  
    comment: String!  
    stars: Int!  
}  
  
type Mutation {  
    addRating(ratingInput: AddRatingInput):  
        Rating!  
}
```

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Schema definieren über Schema-Definition-Language

- Dokumentation kann Zeilenweise mit # hinzugefügt werden
- Oder Blockweise mit """", darin sogar Markdown möglich

```
"""
A **Beer** will be rated with **Ratings**
"""

type Beer {
    # The unique ID of this Beer
    id: ID!
    ...
}

type Query {
    """Get a Beer by its ID or null if not found"""
    beer(beerId: ID!): Beer
}
```

## QUERY BEANTWORTEN: DATA FETCHER

### DataFetcher

- Ein **DataFetcher** liefert einen *Wert* für ein angefragtes Feld
  - Wird von graphql-java für jedes Feld eines Queries aufgerufen

## QUERY BEANTWORTEN: DATA FETCHER

### DataFetcher

- Ein **DataFetcher** liefert einen *Wert* für ein angefragtes Feld
  - Wird von graphql-java für jedes Feld eines Queries aufgerufen
  - Zwingend erforderlich für Root-Types (Query, Mutation)
  - Sonst: per Reflection (getter/setter, Maps, ...)
- (In anderen Implementierungen auch **Resolver** genannt)

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher

- Ein **DataFetcher** liefert einen *Wert* für ein angefragtes Feld
  - Wird von graphql-java für jedes Feld eines Queries aufgerufen
  - Zwingend erforderlich für Root-Types (Query, Mutation)
  - Sonst: per Reflection (getter/setter, Maps, ...)
- (In anderen Implementierungen auch **Resolver** genannt)
- DataFetcher ist funktionales Interface:

```
interface DataFetcher<T> {  
    T get(DataFetchingEnvironment environment);  
}
```

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher implementieren

- Beispiel: Ein einfaches Feld

Schema Definition

```
type Query {  
  beer(id: ID!): Beer  
}
```

Query

```
query { beer(id: "B1")  
  { name price }  
}
```

```
"data": {  
  "beer":  
    { "name": "...", "price": 5.3 }  
}
```

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher implementieren

- Beispiel: Ein einfaches Feld

Schema Definition

```
type Query {  
    beer(id: ID!): Beer  
}
```

Query

```
query { beer(id: "B1")  
    { name price }  
}  
"data": {  
    "beer":  
        { "name": "...", "price": 5.3 }  
}
```

Data Fetcher

```
public class QueryDataFetchers {  
    DataFetcher<Beer> beer = new DataFetcher<>() {  
        public Beer get(DataFetchingEnvironment env) {  
  
        }  
    };  
}
```

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher implementieren

- Beispiel: Ein einfaches Root-Feld

Schema Definition

```
type Query {  
    beer(id: ID!): Beer  
}
```

Query

```
query { beer(id: "B1")  
    { name price }  
}  
"data": {  
    "beer":  
        { "name": "...", "price": 5.3 }  
}
```

Data Fetcher

```
public class QueryDataFetchers {  
    DataFetcher<Beer> beer = new DataFetcher<>() {  
        public Beer get(DataFetchingEnvironment env) {  
            String id = env.getArgument("id");  
            return beerRepository.getBeerById(id);  
        }  
    };  
}
```

# DATAFETCHER

## DataFetcher: Mutations

- technisch wie Queries zu implementieren, aber Daten dürfen verändert werden

Schema Definition

```
input AddRatingInput {  
    beerId: ID!  
    stars: Int!  
}  
  
type Mutation {  
    addRating(input: AddRatingInput!): Rating!  
}
```

Data Fetcher

```
public class MutationDataFetchers {  
    DataFetcher<Rating> addRating = new DataFetcher<>() {  
        public Rating get(DataFetchingEnvironment env) {  
  
        }  
    };  
}
```

# DATAFETCHER

## DataFetcher: Mutations

- technisch wie Queries zu implementieren, aber Daten dürfen verändert werden

Schema Definition

```
input AddRatingInput {  
    beerId: ID!  
    stars: Int!  
}  
  
type Mutation {  
    addRating(input: AddRatingInput!): Rating!  
}
```

Data Fetcher

```
public class MutationDataFetchers {  
    DataFetcher<Rating> addRating = new DataFetcher<>() {  
        public Rating get(DataFetchingEnvironment env) {  
            Map input = env.getArgument("input");  
            String beerId = input.get("beerId");  
            Integer starts = input.get("stars");  
  
        }  
    };  
}
```

# DATAFETCHER

## DataFetcher: Mutations

- technisch wie Queries zu implementieren, aber Daten dürfen verändert werden

Schema Definition

```
input AddRatingInput {  
    beerId: ID!  
    stars: Int!  
}  
  
type Mutation {  
    addRating(input: AddRatingInput!): Rating!  
}
```

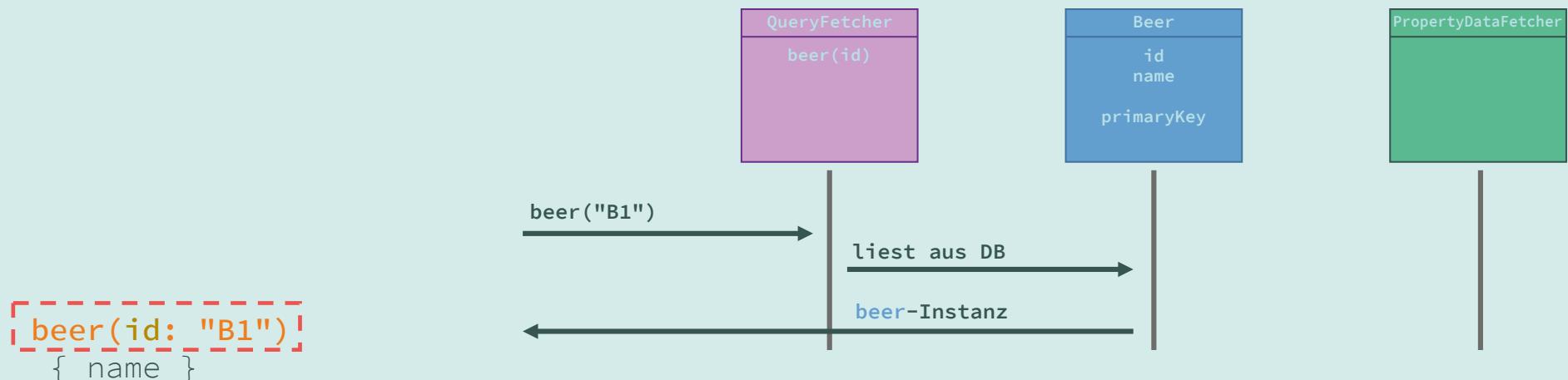
Data Fetcher

```
public class MutationDataFetchers {  
    DataFetcher<Rating> addRating = new DataFetcher<>() {  
        public Rating get(DataFetchingEnvironment env) {  
            Map input = env.getArgument("input");  
            String beerId = input.get("beerId");  
            Integer stars = input.get("stars");  
  
            return ratingService.newRating(beerId, stars);  
        }  
    };  
}
```

# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

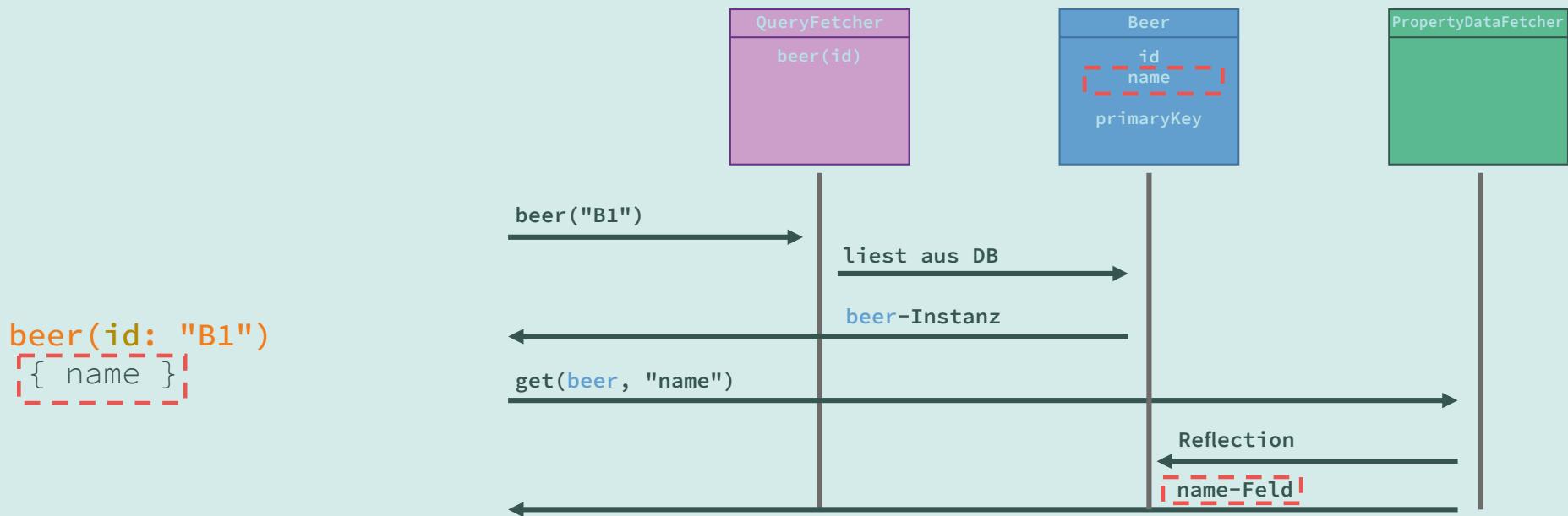
- Für Root-Felder müssen DataFetcher implementiert werden



# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

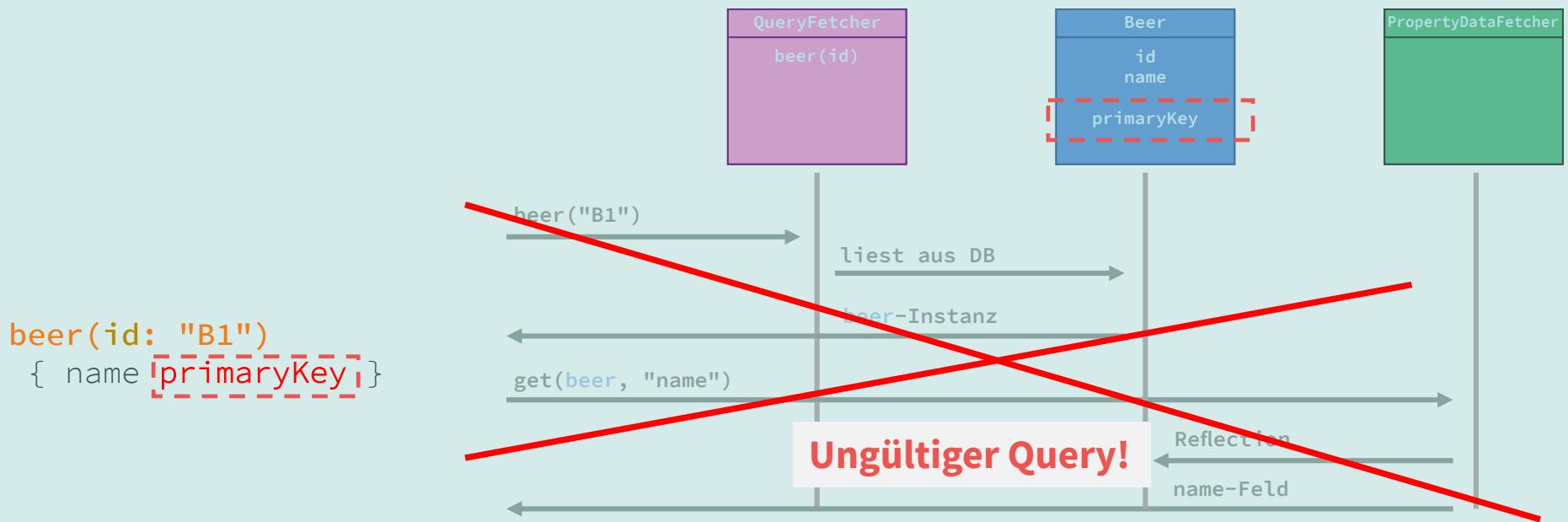
- Für Root-Felder müssen DataFetcher implementiert werden
- Für alle anderen Felder wird ein **PropertyDataFetcher** per Default verwendet
- Der PropertyDataFetcher verwendet Reflection



# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

- Für Root-Felder müssen DataFetcher implementiert werden
- Für alle anderen Felder wird ein **PropertyDataFetcher** per Default verwendet
- Der PropertyDataFetcher verwendet Reflection
  - **es werden nie Daten zurückgeliefert, die nicht im Schema definiert sind**

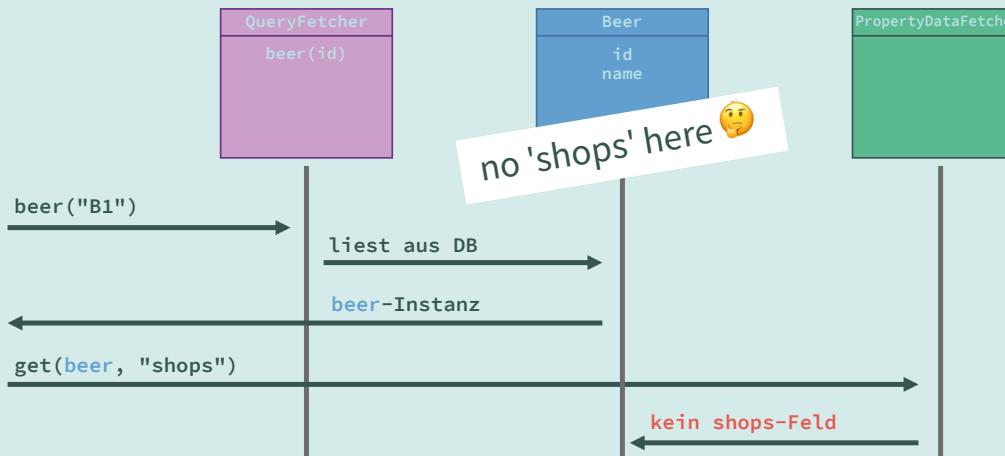


# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

- Problem: Feld existiert gar nicht am Pojo

```
beer(id: "B1")
{ shops { address }
}
```

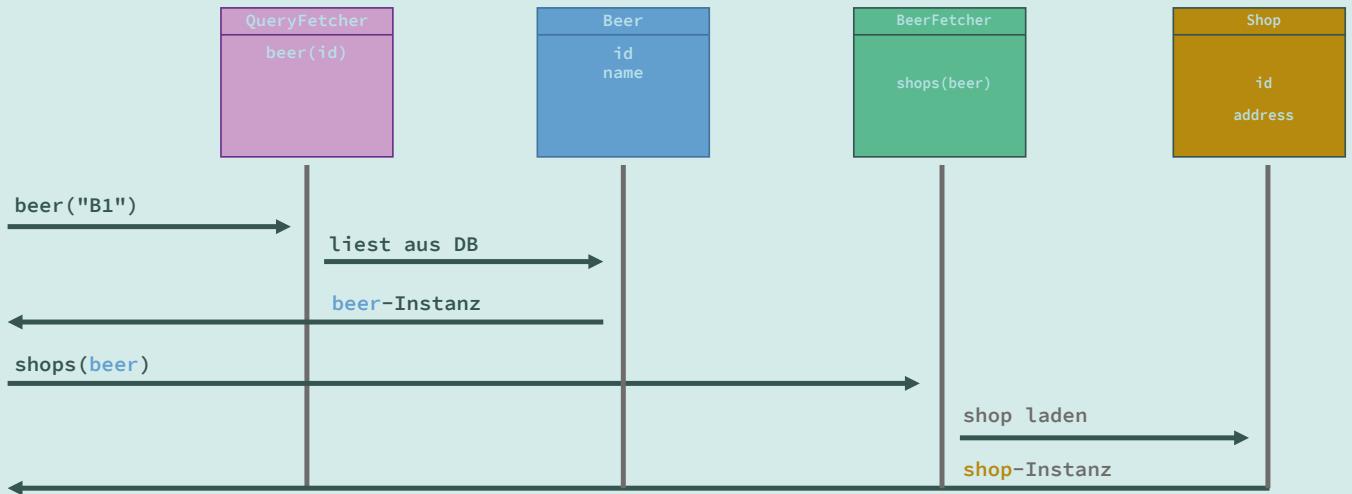


# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

- Eigene DataFetcher können *pro Feld* festgelegt werden
- DataFetcher wird nur ausgeführt, wenn Feld auch im Query abgefragt wird*

```
beer(id: "B1")
{ shops { address
}}
```

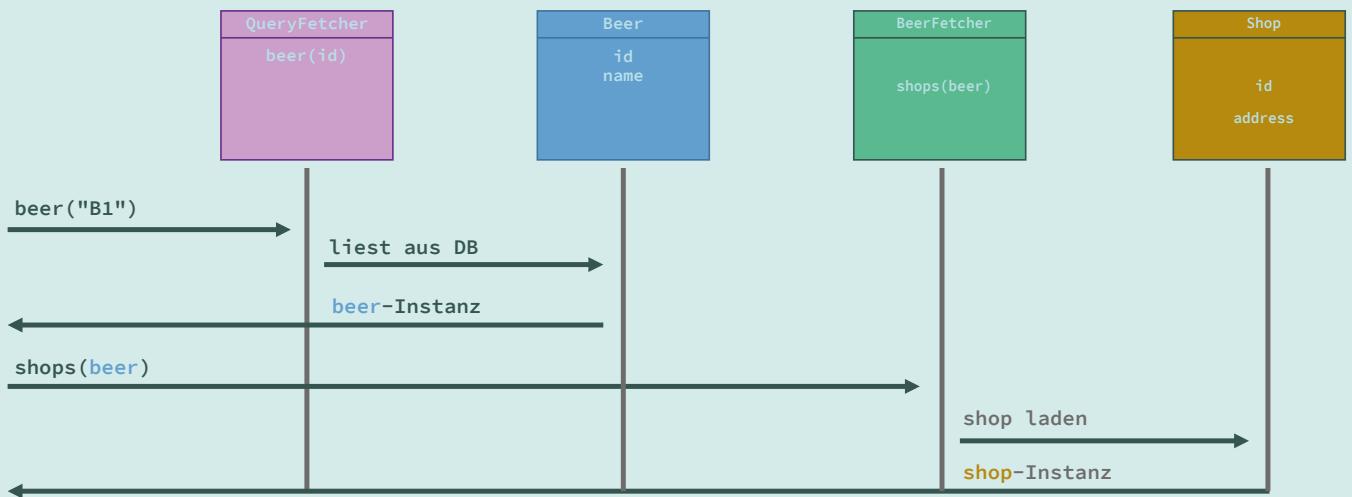


# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

- DataFetcher für nicht-Root-Felder funktionieren wie DataFetcher für Root-Felder
- Sie erhalten das Eltern-Element als "Source"-Property

```
beer(id: "B1")
{ shops { address
}}
```



```
DataFetcher<List<Shop>> shops = new DataFetcher<>() {
    public String get(DataFetchingEnvironment env) {
        Beer parent = env.getSource();
```

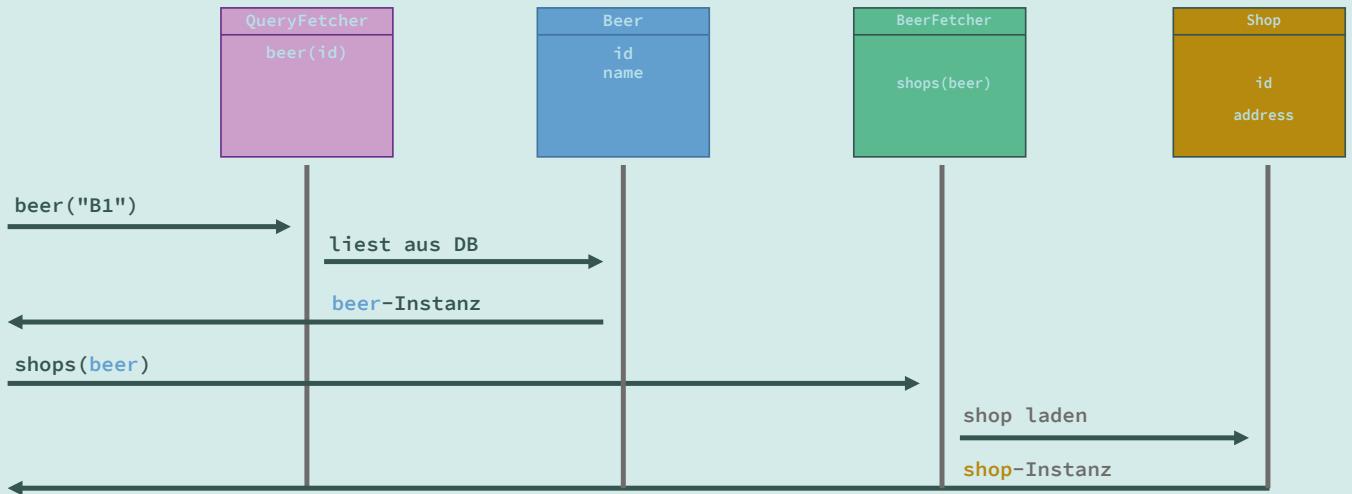
```
    }
};
```

# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

- DataFetcher für nicht-Root-Felder funktionieren wie DataFetcher für Root-Felder
- Sie erhalten das Eltern-Element als "Source"-Property

```
beer(id: "B1")
{ shops { address
}}
```



```
DataFetcher<List<Shop>> shops = new DataFetcher<>() {
    public String get(DataFetchingEnvironment env) {
        Beer parent = env.getSource();

        return shopRepository.findShopsSellingBeer(beerId);
    }
};
```

# AUSFÜHRUNG VON QUERIES

## Ausführen von Queries

# AUSFÜHRUNG VON QUERIES

## Ausführen von Queries

- Zur Ausführung eines Queries wird eine (graphql-java) GraphQL-Instanz benötigt
- Diese verbindet u.a. das Schema mit den DataFetchern
- Der GraphQL-Instanz wird ein Query als String zur Ausführung übergeben

```
GraphQLSchema graphQLSchema = ...;

GraphQL graphql = GraphQL.newGraphQL(graphQLSchema).build();

ExecutionResult result = graphql.execute
    ("{ beers { name price } }");

Map<String, Object> json = result.toSpecification();
```

# AUSFÜHRUNG VON QUERIES

## Ausführen von Queries

- graphql-java enthält keine Anbindung an eine Server-Umgebung (Servlet, Spring), dazu muss man zusätzliche Frameworks nehmen
- Grundsätzlich:
  - Server nimmt Query aus HTTP-Request entgegen
  - führt mit der GraphQL-Instanz den Query aus
  - liefert das Ergebnis in spezifizierter Form zurück

# HIGHER LEVEL FRAMEWORKS

## Auf graphql-java aufbauend

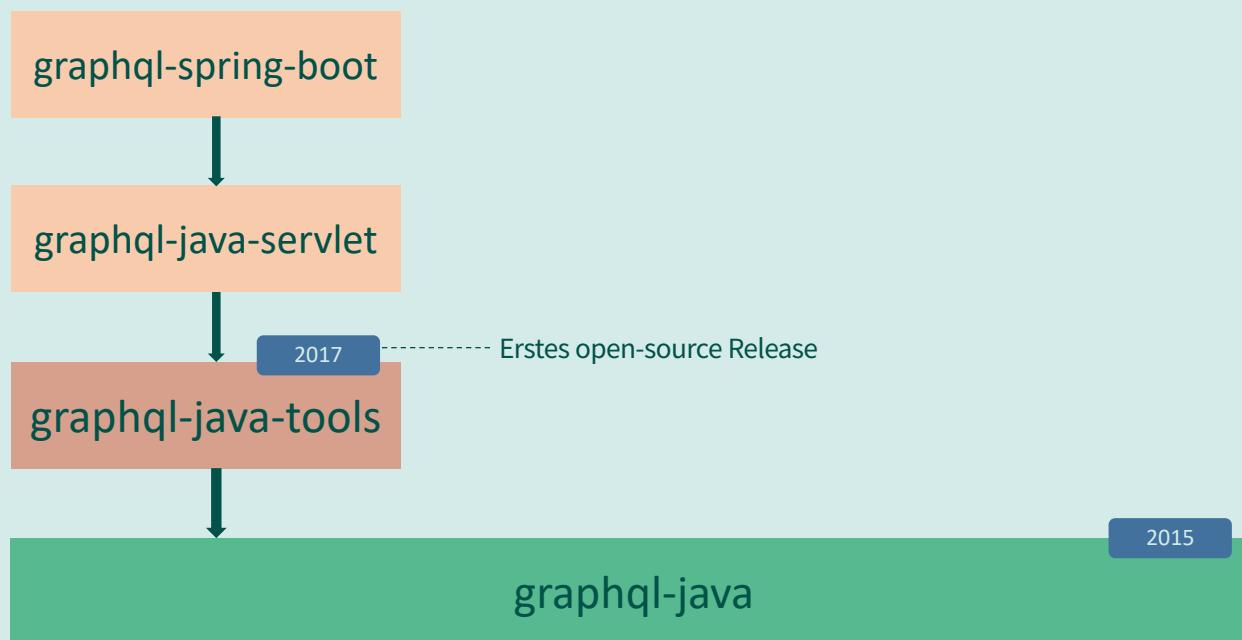
Alle verfolgen dieselbe Idee:

- DataFetcher werden nicht selbst implementiert, es gibt Abstraktionen dafür
  - Unter der Haube werden DataFetcher verwendet
- Zuweisung von DataFetcher an Schema erfolgt automatisch und nicht manuell

# HIGHER LEVEL FRAMEWORKS

## Auf graphql-java aufbauend

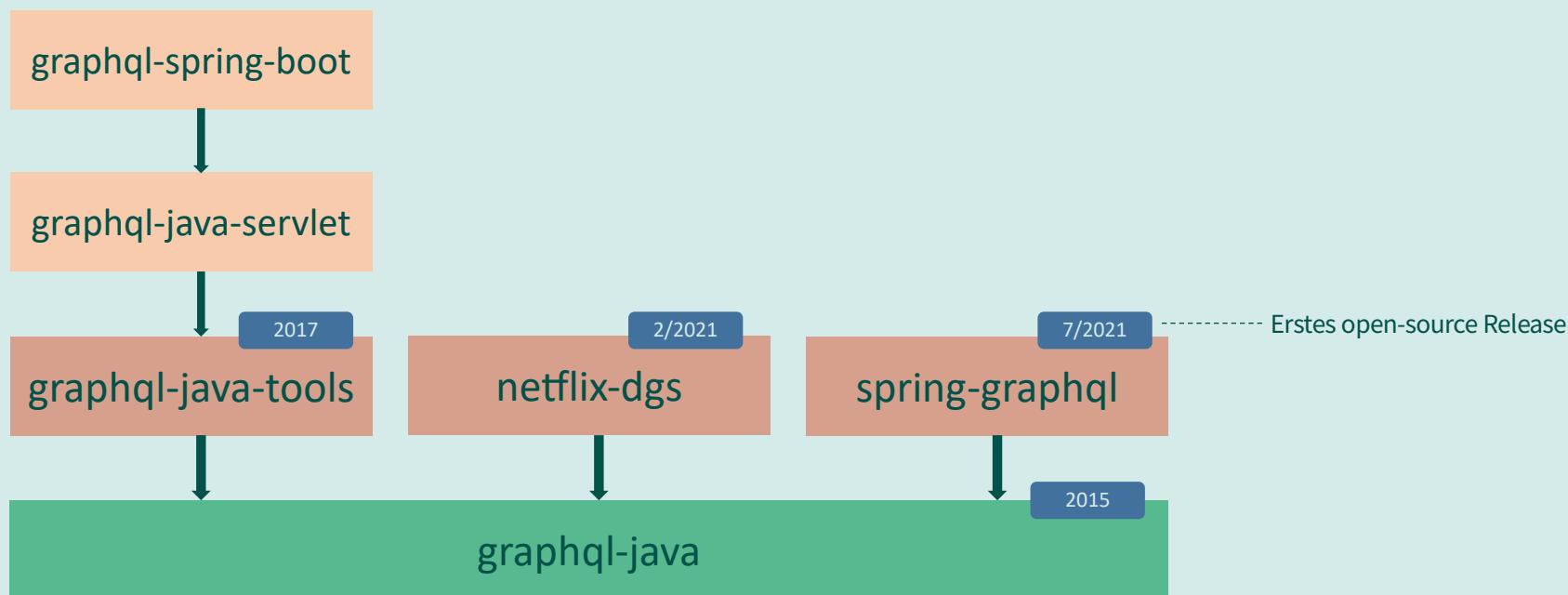
- **graphql-java-tools** ist nicht Spring/JEE-abhängig, aber es gibt Adapter dafür



# HIGHER LEVEL FRAMEWORKS

## Auf graphql-java aufbauend

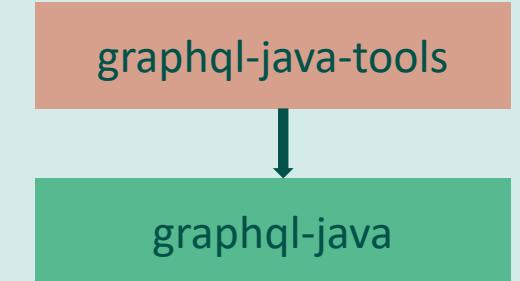
- graphql-java-tools ist nicht Spring/JEE-Abhängig, aber es gibt Adapter dafür
- **Netflix DGS** und **spring-graphql** sehen sehr ähnlich aus
  - Beide sind für Spring Boot



# GRAPHQL-JAVA-TOOLS

## graphql-java-tools

- <https://github.com/graphql-java-kickstart/graphql-java-tools>
- Resolver werden als POJOs implementiert  
Entspricht gewohntem Programmiermodell aus anderen Technologien (z.B. JPA)



## Resolver mit graphql-java-tools

- Beispiel: Resolver mit Root-Field

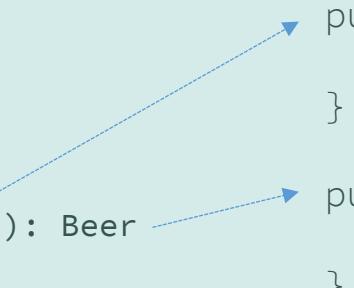
```
type Query {  
  beers: [Beer!]!  
}  
  
public class BeerAdvisorQueryResolver implements  
  GraphQLQueryResolver {  
  
  public List<Beer> beers() {  
    return beerRepository.findAll();  
  }  
}
```



## Resolver mit graphql-java-tools

- Beispiel: Argumente

```
type Query {  
  beers: [Beer!]!  
  beer(beerId: ID!): Beer  
}  
  
public class BeerAdvisorQueryResolver implements  
  GraphQLQueryResolver {  
  
  public List<Beer> beers() {  
    return beerRepository.findAll();  
  }  
  
  public Beer beer(String beerId) {  
    return beerRepository.getBeer(beerId);  
  }  
}
```



## Resolver mit graphql-java-tools

- Beispiel: Mutation und Input Typen
- Komplexe Argumente (GraphQL Input Typen) werden als POJO-Instanzen der Resolver-Methode übergeben

```
input AddRatingInput {  
    beerId: ID!  
    userId: ID!  
    comment: String!  
    stars: Int!  
}
```

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

```
public class AddRatingInput {  
    private String beerId;  
    private int stars;  
    ...  
}
```

```
public class BeerAdvisorMutationResolver  
    implements GraphQLMutationResolver {  
  
    public Rating addRating(AddRatingInput input) {  
        return ratingService.createRating(input);  
    }  
}
```

## Resolver mit graphql-java-tools

- Beispiel: Resolver für Felder, die nicht auf Root-Typen definiert sind
- Das Eltern-Element (getSource in graphql-java) wird als Methoden-Parameter übergeben

```
public class BeerResolver implements GraphQLResolver<Beer> {  
  
    public List<Shop> getShops(Beer parent) {  
        return shopRepository.findShopsSellingBeer(parent.getId());  
    }  
  
}
```

## Bonus: Validierung beim Start

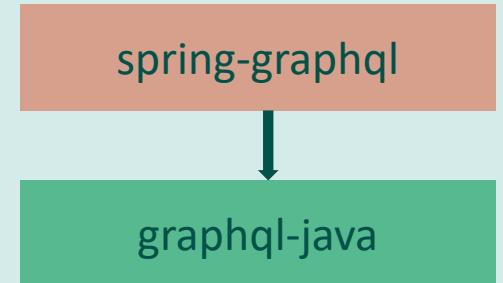
- graphql-java-tools überprüft die Resolver beim Start
- Wenn Resolver fehlen oder fehlerhaft implementiert sind, wird ein Fehler geworfen
- Es gibt also erhöhte Sicherheit, dass die Anwendung auch funktioniert

```
Caused by: graphql.kickstart.tools.resolver.FieldResolverError: No method or field found as defined  
in schema <unknown>:136 with any of the following signatures  
in priority order:
```

```
nh.graphql.beeradvisor.graphql.resolver.BeerResolver.shops(nh.graphql.beeradvisor.domain.Beer)  
nh.graphql.beeradvisor.graphql.resolver.BeerResolver.getShops(nh.graphql.beeradvisor.domain.Beer)  
nh.graphql.beeradvisor.graphql.resolver.BeerResolver.shops  
nh.graphql.beeradvisor.domain.Beer.shops()  
nh.graphql.beeradvisor.domain.Beer.getShops()  
nh.graphql.beeradvisor.domain.Beer.shops  
    at graphql.kickstart.tools.resolver.FieldResolverScanner.missingFieldResolver(...)
```

## spring-graphql

- <https://docs.spring.io/spring-graphql/docs/current-SNAPSHOT/reference/html/>
  - "Offizielle" Spring Lösung für GraphQL
  - Verbindet graphql-java mit with Spring Boot (Konzepten)
  - Stellt GraphQL Endpunkt über Spring WebMVC oder Spring WebFlux zur Verfügung
  - Support für Subscriptions über WebSockets
  - Alle Spring-Features in GraphQL-Schicht wie gewohnt nutzbar
- Noch im Beta-Status (aktuell M3), erste Version erst im Juli veröffentlicht



## Annotated Controllers

- Annotation-basiertes Programmiermodell, ähnlich wie in REST Controllern von Spring

**@Controller**

```
public class BeerQueryController {  
  
    BeerQueryController(BeerRepository beerRepository) { ... }
```

**@QueryMapping**

```
public List<Beer> beers() {  
    return beerRepository.findAll();  
}
```

Mapping auf das Schema mit Namenskonventionen

**@MutationMapping**

```
public Rating addRating(@Argument AddRatingInput input) {  
    return ratingService.createRating(input);  
}
```

## Annotated Controllers

- Annotation-basiertes Programmiermodell, ähnlich wie in REST Controllern von Spring

### @Controller

```
public class BeerQueryController {  
  
    BeerQueryController(BeerRepository beerRepository) { ... }
```

### @QueryMapping

```
public List<Beer> beers() {  
    return beerRepository.findAll();  
}
```

Argumente via Methoden Parameter

### @MutationMapping

```
public Rating addRating(@Argument AddRatingInput input) {  
    return ratingService.createRating(input);  
}
```

## Annotated Controllers

- Annotation-basiertes Programmiermodell, ähnlich wie in REST Controllern von Spring

### @Controller

```
public class BeerQueryController {  
  
    BeerQueryController(BeerRepository beerRepository) { ... }
```

### @QueryMapping

```
public List<Beer> beers() {  
    return beerRepository.findAll();  
}
```

### @MutationMapping

```
public Rating addRating(@Argument AddRatingInput input) {  
    return ratingService.createRating(input);  
}
```

Eltern-Element als Methoden Parameter

### @SchemaMapping

```
public List<Shop> shops(Beer beer) {  
    return shopRepository.findShopsSellingBeer(beer.getId());  
}
```

## Netflix Domain Graph Service Framework (DGS)

- <https://netflix.github.io/dgs/>
- Erste (open-source)-Version veröffentlicht im Februar 2021
- Basiert auf Spring Boot und graphql-java

## Netflix Domain Graph Service Framework (DGS)

- <https://netflix.github.io/dgs/>
- Erste (open-source)-Version veröffentlicht im Februar 2021
- Basiert auf Spring Boot und graphql-java

Features, die es nicht in spring-graphql gibt:

- Code-Generator für Gradle und Maven erzeugt aus dem Schema Java-Klassen
- Support für Apollo Federation (Zusammenfügen verschiedener APIs zu einer)

## Netflix Domain Graph Service Framework (DGS)

- <https://netflix.github.io/dgs/>
- Erste (open-source)-Version veröffentlicht im Februar 2021
- Basiert auf Spring Boot und graphql-java

Features, die es nicht in spring-graphql gibt:

- Code-Generator für Gradle und Maven erzeugt aus dem Schema Java-Klassen
- Support für Apollo Federation (Zusammenfügen verschiedener APIs zu einer)

Aktuell schwer zu sagen, welches das „bessere“ Framework ist, und wie die beiden sich (gemeinsam) weiterentwickeln

## MicroProfile GraphQL

- Annotation-basiertes Programmiermodell
- Code-first: Schema wird aus Code abgeleitet („Entities“ und „Components“)
  - Ähnlich wie in JPA das DB-Schema erzeugt werden kann
  - (im Gegensatz zum Schema-first-Ansatz in graphql-java)
- Aktuell kein Support für Subscriptions
- Unterstützt u.a. von Quarkus

MicroProfile GraphQL

# MICROPROFILE GRAPHQL

**Beispiel:** Aus POJOs wird das Schema abgeleitet

```
@Type  
 @Name("beer")  
 @Description("Represents a Beer that can be rated")  
 public class Beer {  
  
     @Ignore  
     private String primaryKey;    // nicht über API bereitstellen  
  
     @NonNull  
     private String name;  
     private int price;  
     ...  
 }
```

# MICROPROFILE GRAPHQL

**Beispiel:** Mit „Components“ Queries und Mutations implementieren

```
@GraphQLApi
```

```
public class BeerApi {
```

```
    @Inject
```

```
    BeerRepository beerRepository;  
    ShopRepository shopRepository;
```

```
@Query
```

```
@Description("Returns a specific beer, identified by its id")
```

```
public Beer beer(@Name("id") String id) {  
    return beerRepository.getBeerById(id);  
}
```

```
@Query
```

```
public List<Shop> shops(@Source beer) {  
    return shopRepository.findShopsSellingBeer(beer.getId());  
}
```

```
}
```

# GRAPHQL FÜR JAVA ANWENDUNGEN

## Abschliessend: Welches Framework soll ich denn nun nehmen?

- Spring Welt: spring-graphql oder Netflix DGS
  - Die Zukunft wird zeigen, welches „besser“ ist (könnte sein, dass Netflix DGS ein Aufsatz für spring-graphql wird und weitere Features zur Verfügung stellt)

## Abschließend: Welches Framework soll ich denn nun nehmen?

- Spring Welt: spring-graphql oder Netflix DGS
  - Die Zukunft wird zeigen, welches „besser“ ist (könnte sein, dass Netflix DGS ein Aufsatz für spring-graphql wird und weitere Features zur Verfügung stellt)
- JEE
  - graphql-java oder graphql-java-tools und graphql-java-servlet für HTTP Endpunkt

## Abschließend: Welches Framework soll ich denn nun nehmen?

- Spring Welt: spring-graphql oder Netflix DGS
  - Die Zukunft wird zeigen, welches „besser“ ist (könnte sein, dass Netflix DGS ein Aufsatz für spring-graphql wird und weitere Features zur Verfügung stellt)
- JEE
  - graphql-java oder graphql-java-tools und graphql-java-servlet für HTTP Endpunkt
- MicroProfile
  - MicroProfile GraphQL
  - (graphql-java würde aber auch funktionieren)

# Zurück zu unserer Anwendung...

**Welche möglichen Probleme kann es mit unserer API geben?**

👉 Teil 1: Schema

👉 Teil 2: Implementierung

# SCHEMA DESIGN

## Unser Schema

🤔 Welche Probleme könnten wir hiermit haben? Welche Features fehlen?

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Rating {  
    id: ID!  
    beer: Beer!  
    author: User!  
    comment: String!  
    stars: Int!  
}  
  
type Beer {  
    id: ID!  
    name: String!  
    price: String!  
    ratings: [Rating!]!  
    averageStars: Int!  
}  
  
type Query {  
    beer(beerId: ID!): Beer  
    beers: [Beer!]!  
}  
  
input AddRatingInput {  
    beerId: ID!  
    userId: ID!  
    comment: String  
    stars: Int!  
}  
  
type Mutation {  
    addRatingInput(ratingInput: RatingInput!):  
        Rating!  
}
```

# SCHEMA DESIGN

## Unser Schema

🤔 Welche Probleme könnten wir hiermit haben? Welche Features fehlen?

```
type User {  
  id: ID!  
  login: String!  
  name: String!  
}  
  
type Rating {  
  id: ID!  
  beer: Beer!  
  author: User!  
  comment: String!  
  stars: Int!  
}  
  
type Beer {  
  id: ID!  
  name: String!  
  price: String!  
  ratings: [Rating!]!  
  averageStars: Int!  
}  
  
type Query {  
  beer(beerId: ID!): Beer  
  beers: [Beer!]!  
}  
  
input AddRatingInput {  
  beerId: ID!  
  userId: ID!  
  comment: String  
  stars: Int!  
}  
  
type Mutation {  
  addRatingInput(ratingInput: RatingInput!):  
    Rating!  
}
```

Paginierung?  
Sortierung?

# SCHEMA DESIGN

## Unser Schema

🤔 Welche Probleme könnten wir hiermit haben? Welche Features fehlen?

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Rating {  
    id: ID!  
    beer: Beer!  
    author: User!  
    comment: String!  
    stars: Int!  
}  
  
type Beer {  
    id: ID!  
    name: String!  
    price: String!  
    ratings: [Rating!]!  
    averageStars: Int!  
}  
  
type Query {  
    beer(beerId: ID!): Beer  
    beers: [Beer!]!  
}  
  
input AddRatingInput {  
    beerId: ID!  
    userId: ID!  
    comment: String  
    stars: Int!  
}  
  
type Mutation {  
    addRatingInput(ratingInput: RatingInput!): Rating!  
}
```

Fehlerbehandlung?

# SCHEMA DESIGN

## Unser Schema

🤔 Welche Probleme könnten wir hiermit haben? Welche Features fehlen?

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Rating {  
    id: ID!  
    beer: Beer!  
    author: User!  
    comment: String!  
    stars: Int!  
}  
  
type Beer {  
    id: ID!  
    name: String!  
    price: String!  
    ratings: [Rating!]!  
    averageStars: Int!  
}  
  
type Query {  
    beer(beerId: ID!): Beer  
    beers: [Beer!]!  
}  
  
input AddRatingInput {  
    beerId: ID!  
    userId: ID!  
    comment: String  
    stars: Int!  
}  
  
type Mutation {  
    addRatingInput(ratingInput: RatingInput!): Sicherheit?!  
    Rating!  
}
```

# PAGINIERUNG

GraphQL macht keine Aussage über Paginierung, Sortierung, ...

Beispiel: Seiten-basierte Paginierung

```
type Query {  
  beers(  
    page: Int!,  
    pageSize: Int!): BeerList!  
}  
  
type BeerList {  
  page: Int!  
  totalElements: Int!  
  hasNext: Boolean!  
  hasPrev: Boolean!  
  
  beers: [Beer!]!  
}
```

# PAGINIERUNG

GraphQL macht keine Aussage über Paginierung, Sortierung, ...

Beispiel mit Spring Data

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;

public class BeerAdvisorQueryResolver implements
    GraphQLQueryResolver {

type Query {
    beers(
        page: Int!,
        pageSize: Int!): BeerList!
}

type BeerList {
    page: Int!
    totalElements: Int!
    hasNext: Boolean!
    hasPrev: Boolean!
    beers: [Beer!]!
}

    @Inject
    private BeerRepository beerRepository;

    public BeerList beers(int page, int pageSize) {
        Page<Beer> page = beerRepository.find(
            PageRequest.of(page, pageSize)
        );

        return new BeerList(
            page.getNumber(),
            page.getTotalElements(),
            page.hasNext(), page.hasPrevious(),
            page.getContent()
        );
    }
}
```

# PAGINIERUNG

**GraphQL macht keine Aussage über Paginierung, Sortierung, ...**

Sortierung wäre analog über eigene Felder

=> nicht mit der Mächtigkeit von SQL vergleichbar, bzw. muss selbst programmiert werden

```
enum Direction {  
    asc, desc  
}  
  
type BeerOrderCriteria {  
    field: String!  
    direction: Direction!  
}  
  
type Query {  
    beers(  
        page: Int!,  
        pageSize: Int!,  
        orderBy: [BeerOrderCriteria!]  
    ) : BeerList!  
}
```

## SECURITY

**GraphQL macht keine Aussage über Security**

## GraphQL macht keine Aussage über Security

Beispiel mit Spring Security: Absicherung des GraphQL Endpunkts

GraphQL API kann nur verwendet werden, wenn angemeldet, z.B. bei nicht öffentlicher API sinnvoll

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/graphql").authenticated();
    }
}
```

## GraphQL macht keine Aussage über Security

Beispiel mit Spring Security: Absicherung Geschäftslogik  
(Mit JEE Annotations ähnlich)

```
type Mutation {  
    addRatingInput(ratingInput:  
        RatingInput!):  
        Rating!  
}  
  
public class RatingMutationResolver implements  
    GraphQLMutationResolver {  
    // z.B via DI  
    private RatingRepository ratingRepository;  
  
    @PreAuthorize(  
        "isAuthenticated() && #newRating.userId == authentication.principal.id"  
    )  
    public Rating addRating(AddRatingInput newRating) {  
        Rating rating = Rating.from(newRating);  
        ratingRepository.save(rating);  
        return rating;  
    }  
}
```

## GraphQL macht keine Aussage über Security

Fehler landet im 'errors'-Objekt  
(Customization möglich)

The screenshot shows a GraphQL playground interface with the title "Beer Advisor - Nils (U5)". The mutation code is as follows:

```
1 mutation {
2   addRating(ratingInput: {
3     beerId: "B1",
4     userId: "U3",
5     comment: "Darf ich nicht",
6     stars: 3}) {
7     id
8     comment
9     author {
10       id
11     }
12   }
13 }
```

The resulting JSON response includes an "errors" field containing a single error object with the following details:

```
{
  "errors": [
    {
      "message": "Exception while fetching data (/addRating) : Zugriff verweigert",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "addRating"
      ],
      "extensions": {
        "classification": "DataFetchingException"
      }
    ]
  ],
  "data": null
}
```

## ERROR HANDLING

### (Technische) Fehler landen im errors-Objekt

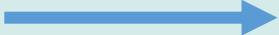
- Fachliche Fehler können auch im fachlichen Error-Objekt untergebracht werden

# ERROR HANDLING

## (Technische) Fehler landen im errors-Objekt

- Fachliche Fehler können auch im fachlichen Error-Objekt untergebracht werden
- Zum Beispiel für Validierungsfehler auf Server-seite

```
type Mutation {  
  addRatingInput  
  (ratingInput:  
   RatingInput!):  
   Rating!  
}
```

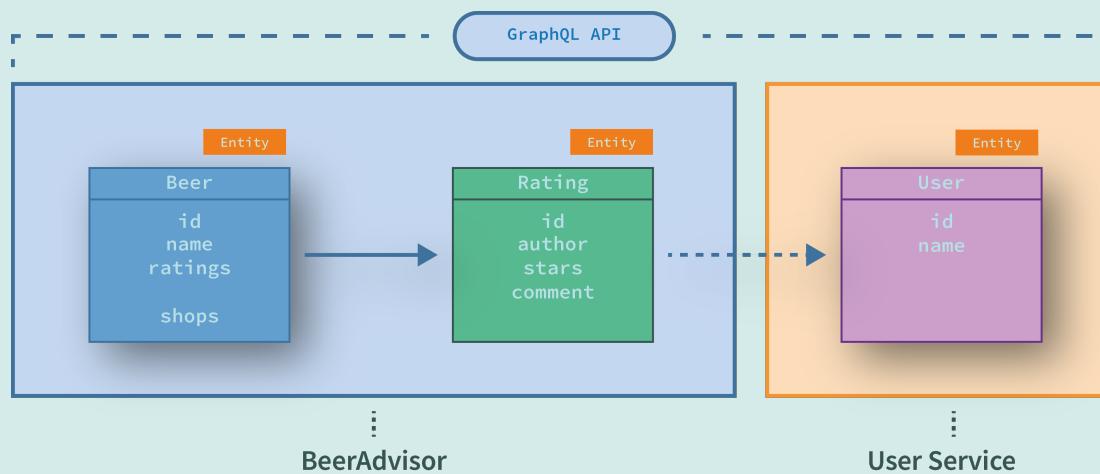


```
type Mutation {  
  addRatingInput  
  (ratingInput:  
   RatingInput!):  
   AddRatingResult!  
}  
  
type AddRatingResult {  
  newRating: Rating  
  validationErrors: [ValidationErrors!]!  
}  
  
type ValidationErrors {  
  field: String!  
  msg: String!  
}
```

# IMPLEMENTIERUNG

💡 Was könnte es in der bestehenden Implementierung für Probleme geben?

Zur Erinnerung ein Ausschnitt aus unserer "Architektur":



## EXKURS: OPTIMIERUNGEN

**Achtung! Optimierungen immer Use-Case-spezifisch**

# IMPLEMENTIERUNG



Was gibt es bei der Ausführung dieses Querys für ein Problem?

```
{  
  beer (beerId: "B3") { id name }  
  shop (shopId: "S1") { name }  
}
```

# IMPLEMENTIERUNG



## Was gibt es bei der Ausführung dieses Querys für ein Problem?

```
{  
  beer (beerId: "B3") { id name }  
  shop (shopId: "S1") { name }  
}
```

1. Felder werden nacheinander ermittelt  
=> was passiert, wenn das lange dauert?  
👉 @slowdown



GraphQLEndpointConfiguration  
„withInstrumentation“ hinzufügen

# IMPLEMENTIERUNG

## Asynchrone Ausführung

```
{  
    beer (beerId: "B3") { id name }  
    shop (shopId: "S1") { name }  
}
```

1. DataFetcher können asynchron ausgeführt werden
2. Dazu liefern sie ein CompletableFuture zurück
3. Dann werden alle DF einer "Ebene" parallel ausgeführt

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

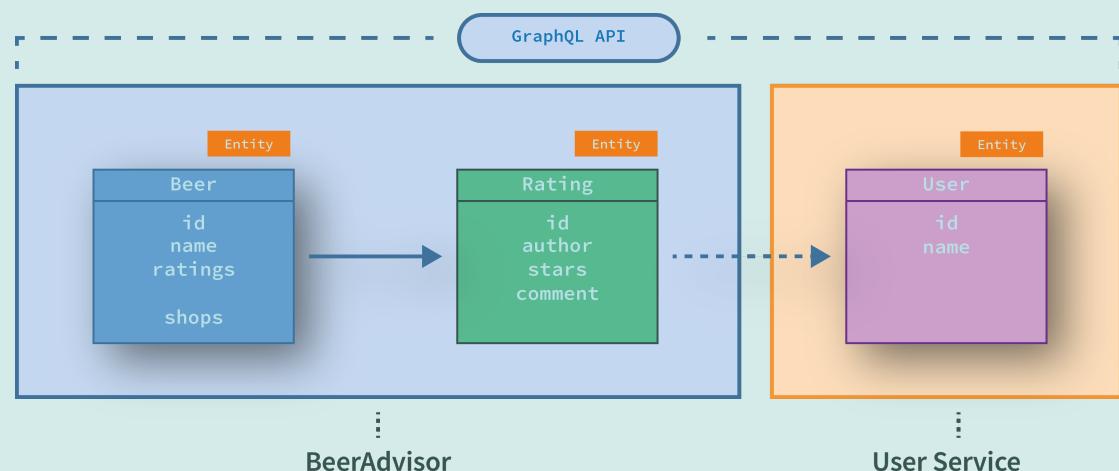
👉 @async

# IMPLEMENTIERUNG



Was gibt es bei der Ausführung dieses Queries für ein Problem?

```
{ beer (beerId: "B3") {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}
```



# DATA LOADER

## Beispiel: Zugriff auf (Remote-)Services

1. Beer-DataFetcher liefert Beer zurück  
(ein DB-Aufruf)

```
{ beer (beerId: "B3") {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}
```

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

# DATA LOADER

## Beispiel: Zugriff auf (Remote-)Services

1. Beer-DataFetcher liefert Beer zurück  
(ein DB-Aufruf)

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

```
{ beer (beerId: "B3") {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}  
}
```

2. Am Beer hängen *n Ratings* (werden im selben SQL-Query aus der DB als Join mitgeladen)

# DATA LOADER

## Beispiel: Zugriff auf (Remote-)Services

1. Beer-DataFetcher liefert Beer zurück  
(ein Aufruf)

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

```
{ beer (beerId: "B3") {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}  
}
```

2. Am Beer hängen n-Ratings (werden im selben SQL-Query aus der DB als Join mitgeladen)
3. author-DataFetcher liefert User *pro Rating* zurück  
**(n-Aufrufe zum Remote-Service)**

```
public DataFetcher<User> authorFetcher() {  
    return env -> {  
        Rating rating = environment.getSource();  
        String userId = rating.getUserId();  
  
        return userService.getUser(userId);  
    };  
}
```

Remote-Call!

# DATA LOADER

## Beispiel: Zugriff auf (Remote-)Services

1. Beer-DataFetcher liefert Beer zurück  
(ein Aufruf)

```
{ beer (beerId: "B3") {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}
```

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

2. Am Beer hängen n-Ratings (werden im selben SQL-Query aus der DB als Join mitgeladen)
3. author-DataFetcher liefert User *pro Rating* zurück  
(n-Aufrufe zum Remote-Service)

```
public DataFetcher<User> authorFetcher() {  
    return env -> {  
        Rating rating = environment.getSource();  
        String userId = rating.getUserId();  
  
        return userService.getUser(userId);  
    };  
}
```

=> 1 (Beer) + n (User)-Calls 😭

## Optimieren und Cachen von Zugriffen mit DataLoader

DataLoader kommen ursprünglich aus der JavaScript-Implementierung

Ein DataLoader kann:

- Aufrufe zusammenfassen (Batching)
- Ergebnisse cachen
- asynchron ausgeführt werden

# 1+N-PROBLEM

## Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück  
(unverändert)

```
{ beer (beerId: "B3") {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}  
}
```

# 1+N-PROBLEM

## Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück  
(unverändert)
2. author-DataFetcher delegiert Ermitteln der Daten  
an den DataLoader.

*GraphQL verzögert das eigentliche Laden der Daten,  
bis alle authorFetcher aufgerufen wurden*

```
{ beer (beerId: "B3") {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}
```

```
public DataFetcher authorFetcher() {  
    return env -> {  
        Rating rating = environment.getSource();  
        String userId = rating.getUserId();  
  
        DataLoader<String, User> dataLoader =  
            env.getDataLoader("user");  
  
        return dataLoader.load(userId);  
    };  
}
```

Sammelt alle load-Aufrufe ein und führt erst dann den DataLoader aus

# 1+N-PROBLEM

## Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück  
(unverändert)
2. author-DataFetcher delegiert Ermitteln der Daten  
an den DataLoader.  
GraphQL verzögert das eigentliche Laden der Daten  
so lange wie möglich.

```
{ beer (beerId: "B3") {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}
```

```
public DataFetcher authorFetcher() {  
    return env -> {  
        Rating rating = environment.getSource();  
        String userId = rating.getUserId();  
  
        DataLoader<String, User> dataLoader =  
            env.getDataLoader("user");  
  
        return dataLoader.load(userId);  
    };  
}
```

 Sammelt alle load-Aufrufe ein und führt erst dann den DataLoader aus

=> 1 (Beer) + 1 (Remote)-Call 😊

# 1+N-PROBLEM

## Optimieren und Cachen von Zugriffen mit DataLoader

Die eigentlichen Daten werden dann gesammelt in einem **BatchLoader** geladen

```
public BatchLoader userBatchLoader = new BatchLoader<String, User>() {  
  
    public CompletableFuture<List<User>> load(List<String> userIds) {  
        return CompletableFuture.supplyAsync(() -> userService.findUsersWithId(userIds));  
    }  
  
};
```

Wird von GraphQL aufgerufen mit einer *Menge* von Ids,  
die aus einer *Menge* von DataFetcher-Aufrufen stammen

# 1+N-PROBLEM

## Beispiel

Ohne DataLoader

(UserService Logs!)

```
query {  
  beers {  
    ratings {  
      author {  
        id  
        name  
      }  
    }  
  }  
}
```

Mit DataLoader

(UserService Logs!)

```
query {  
  beers {  
    ratings {  
      author @useDataLoader {  
        id  
        name  
      }  
    }  
  }  
}
```

# 1+N-PROBLEM

## Beispiel

Ohne DataLoader

(UserService Logs!)

```
query {  
  beers {  
    ratings {  
      author {  
        id  
        name  
      }  
    }  
  }  
}
```

Mit DataLoader

(UserService Logs!)

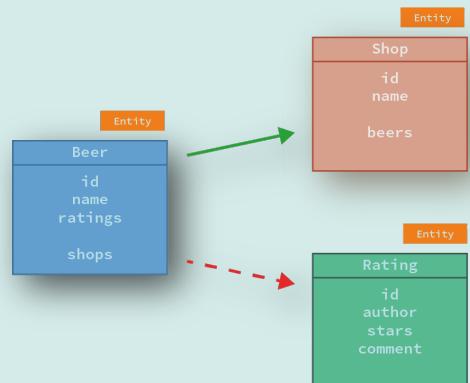
```
query {  
  beers {  
    ratings {  
      author @useDataLoader {  
        id  
        name  
      }  
    }  
  }  
}
```

Im "richtigen" Leben würdet  
ihr das natürlich immer  
einschalten (ohne Direktive)

# EXKURS: OPTIMIERUNGEN

## Problem: optimaler Datenbankzugriff (Beispiel: JPA/JOINs)

```
beers {  
    name  
    shops {  
        name  
    }  
}
```

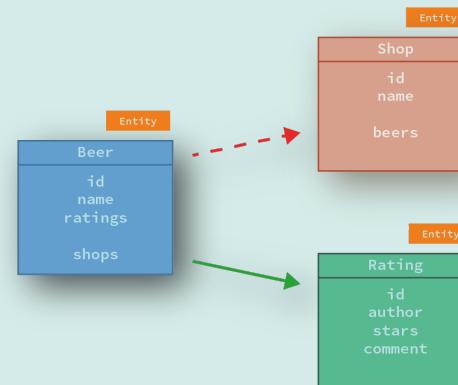
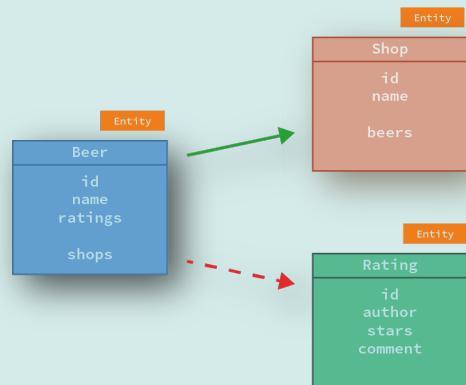


# EXKURS: OPTIMIERUNGEN

## Problem: optimaler Datenbankzugriff (Beispiel: JPA/JOINs)

```
beers {  
    name  
    shops {  
        name  
    }  
}
```

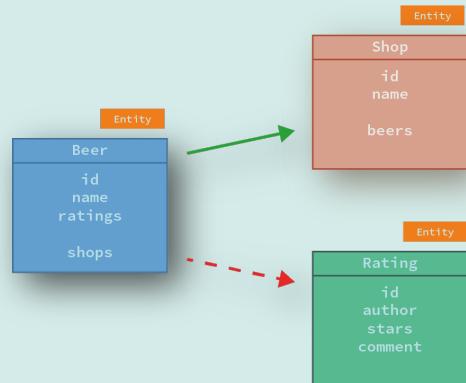
```
beers {  
    name  
    ratings {  
        comment  
    }  
}
```



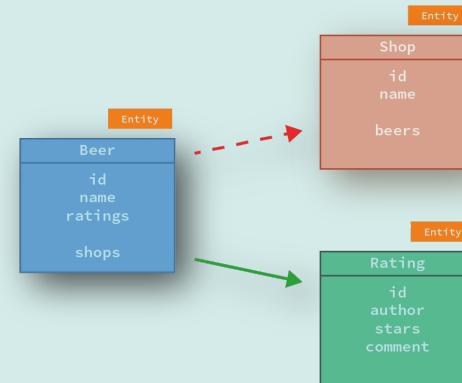
# EXKURS: OPTIMIERUNGEN

## Problem: optimaler Datenbankzugriff (Beispiel: JPA/JOINs)

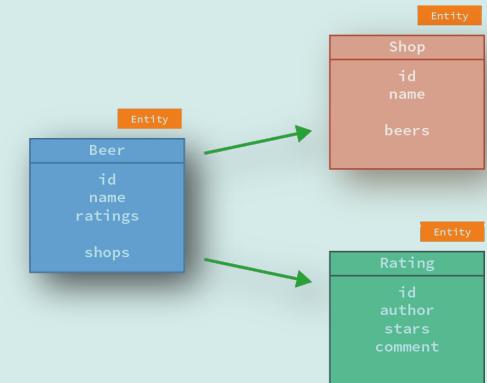
```
beers {  
    name  
    shops {  
        name  
    }  
}
```



```
beers {  
    name  
    ratings {  
        comment  
    }  
}
```



```
beers {  
    name  
    ratings {  
        comment  
    }  
    shops {  
        name  
    }  
}
```

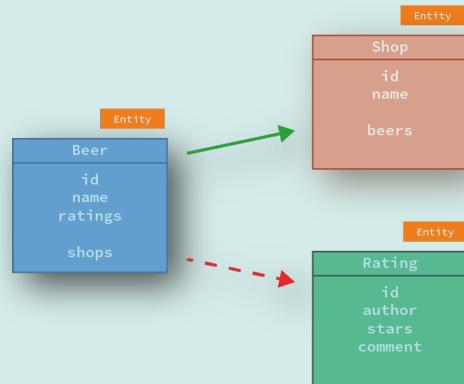


# EXKURS: OPTIMIERUNGEN

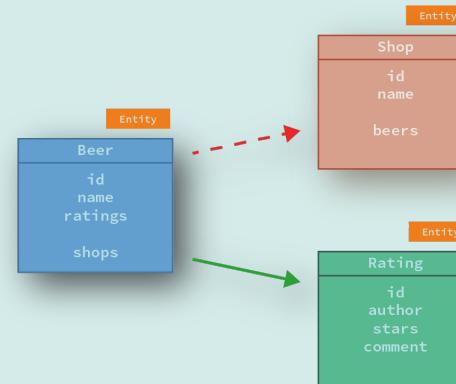
## Problem: optimaler Datenbankzugriff (Beispiel: JPA/JOINs)

- nur zur Laufzeit ermittelbar
- möglichst auf oberstem DataFetcher entscheiden

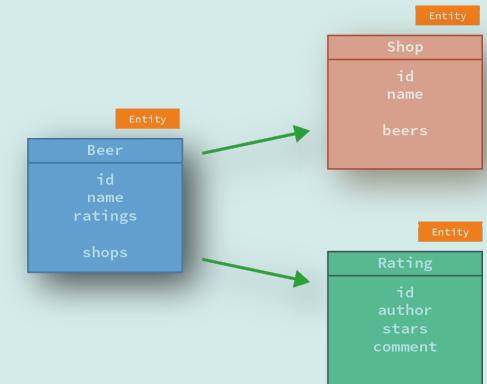
```
beers {  
    name  
    shops {  
        name  
    }  
}
```



```
beers {  
    name  
    ratings {  
        comment  
    }  
}
```



```
beers {  
    name  
    ratings {  
        comment  
    }  
    shops {  
        name  
    }  
}
```



# EXKURS: OPTIMIERUNGEN

## Das SelectionSet

- SelectionSet enthält *alle* abgefragten Felder
- Kann genutzt werden, um Zugriffe auf Datenbank zu optimieren

```
public DataFetcher<Beer> beerFetcher() {  
    return environment -> {  
        DataFetchingFieldSelectionSet selection = environment.getSelectionSet();  
  
        if (selection.contains("ratings")) {  
            // Ratings wurden abgefragt  
        }  
        if (selection.contains("shops")) {  
            // Shops wurden abgefragt  
        }  
  
        String beerId = environment.getArgument("beerId");  
        return beerRepository.getBeer(beerId);  
    };  
}
```

# EXKURS: OPTIMIERUNGEN

## Das SelectionSet

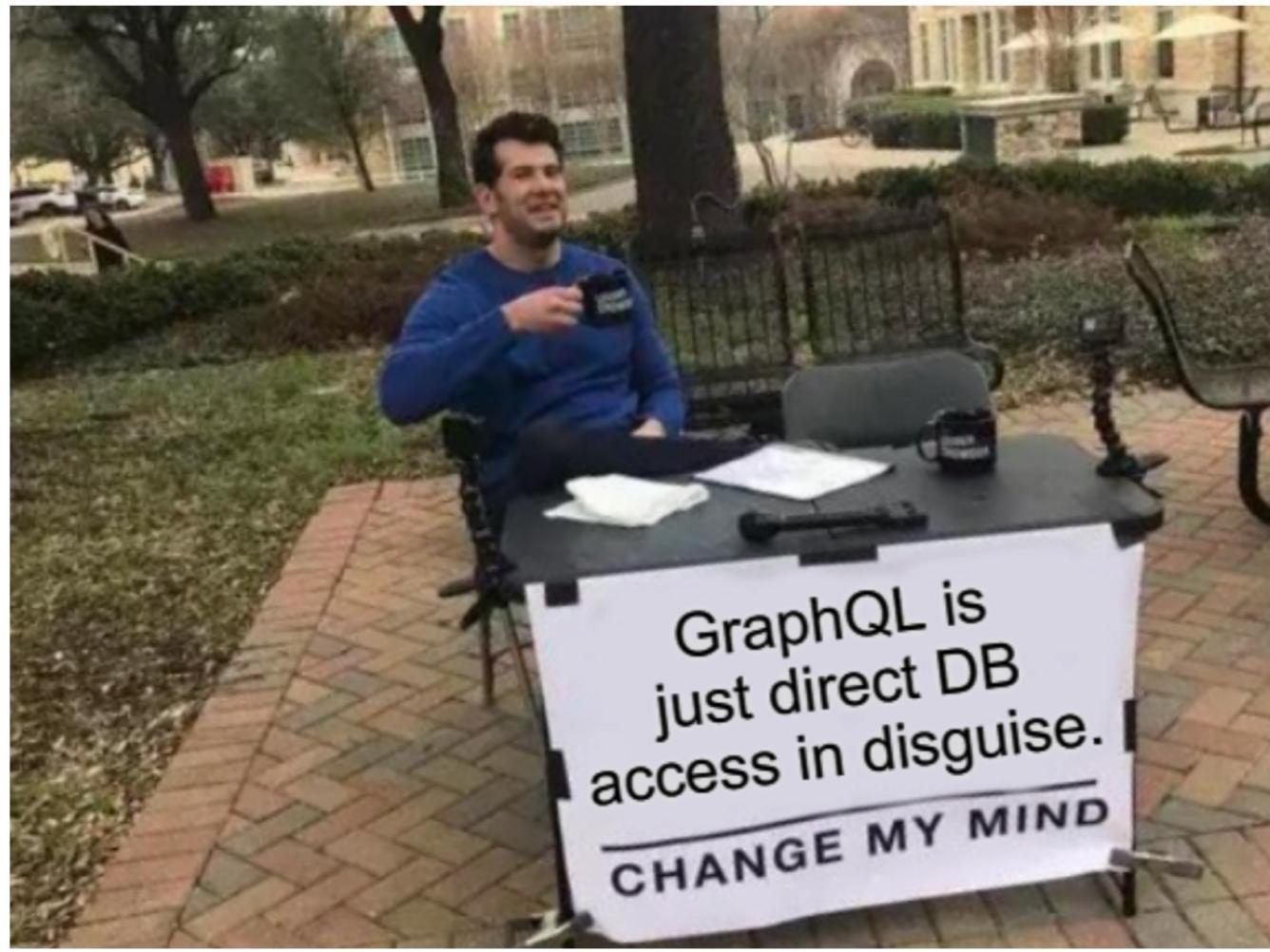
- SelectionSet enthält alle abgefragten Felder
- Kann genutzt werden, um Zugriffe auf Datenbank zu optimieren

### Beispiel: JPA EntityGraph

```
public DataFetcher<Beer> beerFetcher() {  
    return environment -> {  
        DataFetchingFieldSelectionSet selection = environment.getSelectionSet();  
  
        EntityGraph entityGraph = entityManager.createEntityGraph(Beer.class);  
  
        if (selection.contains("ratings")) {  
            entityGraph.addSubgraph("ratings");  
        }  
        if (selection.contains("shops")) {  
            entityGraph.addSubgraph("shops");  
        }  
  
        String beerId = environment.getArgument("beerId");  
        return beerRepository.getBeer(beerId, entityGraph);  
    };  
}
```

# Zusammenfassung

**GRAPHQL – HEILSBRINGER ODER TEUFELSZEUG?**



## GraphQL - Zusammenfassung

- **Ersetzt weder Backend noch Datenbank**
  - Wir definieren eine API
  - Aus dieser API können sich Clients bedienen
- **GraphQL != SQL**
  - kein SQL, keine "vollständige" Query-Sprache
    - z.B. keine Sortierung, keine (beliebigen) Joins etc
  - keine Datenbank!
  - kein Framework!

**...aber, wenn man unbedingt möchte: GraphQL für Datenbanken**

- **GraphQL als ORM Ersatz (JavaScript, Go):**

<https://prisma.io/>

- **Instant GraphQL Schema für PostgresDB (Node.JS):**

<https://www.graphile.org/postgraphile/>

- **Instant GraphQL Schema für PostgresDB:**

<https://hasura.io/>

## GraphQL - Zusammenfassung

- **Interessante, aber noch relativ junge Technologie**
  - Bricht mit einigen Gewohnheiten aus REST
  - Erfordert umdenken
  - REST und GraphQL können zusammen eingesetzt werden
  - Für APIs, die von anderen verwendet werden sollen, vielleicht noch nicht richtig
- **Bibliotheken und Frameworks für viele Sprachen**
  - Prototyp zum Ausprobieren in der Regel schnell gebaut
- **Empfehlung: ausprobieren und weitere Entwicklung verfolgen**



<https://reactbuch.de>

# Vielen Dank!

Beispiel-Code: <https://github.com/nilshartmann/graphql-java-talk>

Slides: <https://react.schule/api-summit-2021-graphql>

Kontakt & Fragen: [nils@nilshartmann.net](mailto:nils@nilshartmann.net)

**HTTPS://NILSHARTMANN.NET | @NILSHARTMANN**