

**NILS HARTMANN**

<https://nilshartmann.net>

**Heilsbringer oder Teufelszeug?**

# GraphQL

**Eine Einführung**

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

# NILS HARTMANN

nils@nilshartmann.net

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

Java  
JavaScript, TypeScript  
React  
GraphQL

**Trainings & Workshops**

**...auch online bzw. remote!**



<https://reactbuch.de>

**HTTPS://NILSHARTMANN.NET**

# Kurze Umfrage...

<https://www.menti.com>

Code:

61 72 41 1

## AGENDA

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

### 2. GraphQL für Java-Anwendungen

- API implementieren
- Optimierung
- Alternativen zu graphql-java

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

(Per Audio oder per Chat – ganz wie ihr mögt)

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/>*

- 2015 von Facebook erstmals veröffentlicht
- Weitere Entwicklung seit 2018 in GraphQL Foundation
- Umfasst:
  - Query Sprache und -Ausführung
  - Schema Definition Language
  - Nicht: Implementierung
    - Referenz-Implementierung: graphql-js

## *GraphQL != Mainstream*

- Implementierungen und Einsatz noch "bleeding edge" (?)
- Wenig erprobte Best-Practices (?)
- ...dennoch wird es von einigen verwendet!



**tom**

@tgvashworth

Folgen

Heh. Twitter GraphQL is quietly serving more than 40 million queries per day. Tiny at Twitter scale but not a bad start.

Original (Englisch) übersetzen

RETWEETS

**93**

GEFÄLLT

**244**



22:59 - 9. Mai 2017

4

93

244

<https://twitter.com/tgvashworth/status/862049341472522240>

**TWITTER**



Folge ich



Announcing GitHub Marketplace and the official releases of GitHub Apps and our GraphQL API

Original (Englisch) übersetzen

# GitHub

## GitHub

GitHub is where people build software. More than 23 million people use GitHub to discover, fork, and contribute to over 64 million projects.

[github.com](https://github.com)

11:46 - 22. Mai 2017

<https://twitter.com/github/status/866590967314472960>

GITHUB

A screenshot of a web browser displaying the GitLab GraphQL API documentation. The page is titled "GraphQL API" and includes sections for "Getting Started", "Quick Reference", "GraphiQL", and "What is GraphQL?". The URL in the address bar is <https://docs.gitlab.com/ee/api/graphql/>.

**GraphQL API**

Version history [...](#)

- Introduced in GitLab 11.0 (enabled by feature flag `graphql`).
- Always enabled in GitLab 12.1.

## Getting Started

For those new to the GitLab GraphQL API, see [Getting started with GitLab GraphQL API](#).

## Quick Reference

- GitLab's GraphQL API endpoint is located at </api/graphql>.
- Get an [introduction to GraphQL from graphql.org](#).
- GitLab supports a wide range of resources, listed in the [GraphQL API Reference](#).

## GraphiQL

Explore the GraphQL API using the interactive [GraphiQL explorer](#), or on your self-managed GitLab instance on <https://<your-gitlab-site.com>/-/graphql-explorer>.

See the [GitLab GraphQL overview](#) for more information about the GraphiQL Explorer.

## What is GraphQL?

[GraphQL](#) is a query language for APIs that allows clients to request exactly the data they need, making it possible to get all required data in a limited number of requests.

The GraphQL data (fields) can be described in the form of types, allowing clients to use [client-side GraphQL libraries](#) to consume the API and avoid manual parsing.

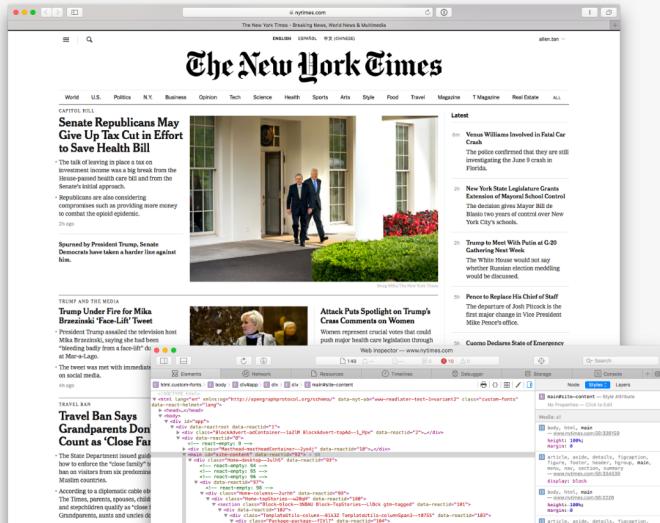
<https://docs.gitlab.com/ee/api/graphql/>



Scott Taylor [Follow](#)

Musician. Sr. Software Engineer at the New York Times. WordPress core committer. Married to Allie.  
Jun 29 · 5 min read

# React, Relay and GraphQL: Under the Hood of the Times Website Redesign



A look under the hood.

The New York Times website is changing, and the technology we use to run it is changing too.

<https://open.nytimes.com/react-relay-and-graphql-under-the-hood-of-the-times-website-redesign-22fb62ea9764>

NEW YORK TIMES



Lee Byron

@leeb

Folgen



While most discussion of [@GraphQL](#) centers around web apps, for the last 7 years Facebook only really used GraphQL for mobile.

Very excited for the new “FB5” version of [fb.com](#), powered entirely by React, GraphQL, and of course: Relay.

Tweet übersetzen

22:41 - 30. Apr. 2019

<https://twitter.com/leeb/status/1123326647552266241>

## FACEBOOK 5

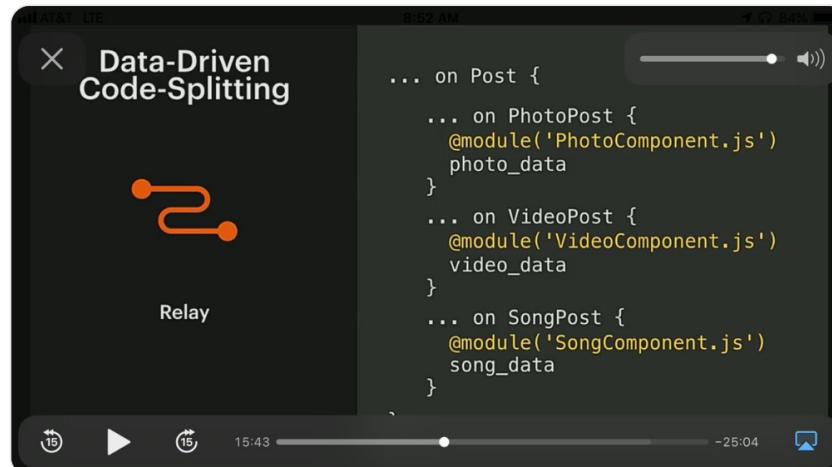


**Nick Schrock**  
@schrockn

Folgen

From the talk about the rewrite of fb using Relay and GraphQL. This feature is so amazing and intuitive. Deliver js only if the graphql query returns data that requires that js.

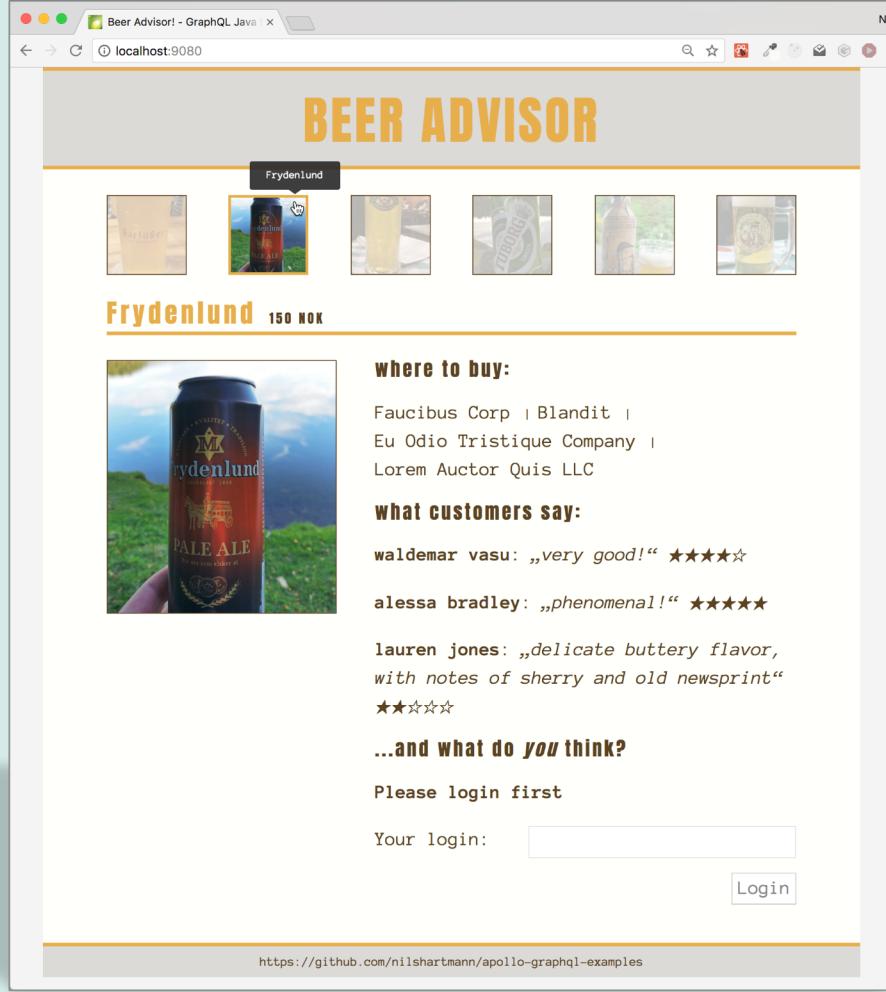
Tweet übersetzen



18:06 - 1. Mai 2019

<https://twitter.com/schrockn/status/1123619660732047360>

## NEXT GEN GRAPHQL?



# GraphQL praktisch

Source-Code: <https://nils.buzz/graphql-java-example>

The screenshot shows the GraphiQL interface running at [localhost:9000/graphiql](http://localhost:9000/graphiql). The left panel displays a GraphQL query for a "BeerAppQuery" type:

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

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

The right panel shows the schema definition and descriptions for each field:

- FIELDS**
- beers: [Beer]!**  
Returns all beers in our store
- beer(beerId: String): Beer**  
Returns the Beer with the specified Id
- ratings: [Rating]!**  
All ratings stored in our system
- ping: ProcessInfo!**  
Returns health information about the running process

# Demo: GraphiQL

<http://localhost:9000/>

A screenshot of the IntelliJ IDEA IDE interface. The main editor window shows a GraphQL query file named `BeerPage.tsx`. The code is as follows:

```
const BEER_RATING_APP_QUERY = gql`query BeerRatingAppQuery {
  backendStatus: ping {
    name
    nodeJsVersion
    uptime
  }
  beer {
    id
    beerId
    author
    comment
  }
}
```

A tooltip is displayed over the `beer` field, listing the following suggestions:

- f **beer** - Returns the Beer with the specified Id [Beer!]!
- f **beers** - Returns all beers in our store [Beer!]!
- f **ping** - Returns health information about t... [ProcessInfo!]
- f **ratings** - All ratings stored in our system [Rating!]!
- f **\_schema** - Access the current type schema of... [\_\_Schema!]
- f **\_type** - Request the type information of a sing... [\_\_Type]

Below the tooltip, a note says: "Dot, space and some other keys will also close this lookup and be inserted into editor".

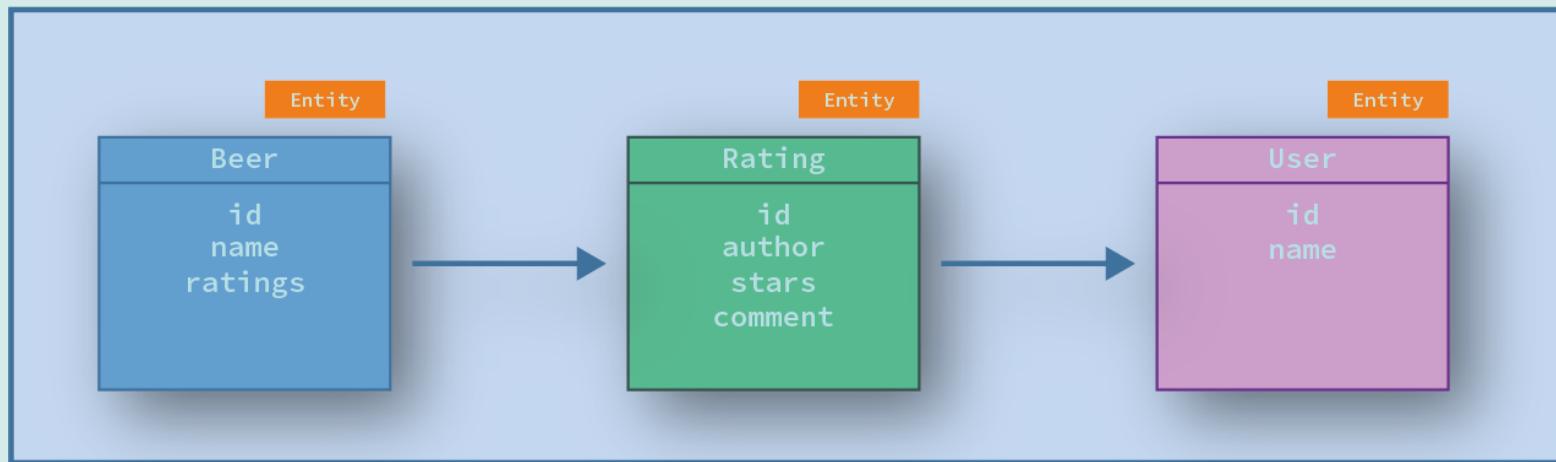
# Demo: IDE Support

Beispiel: IntelliJ IDEA

# **Vergleich mit REST**

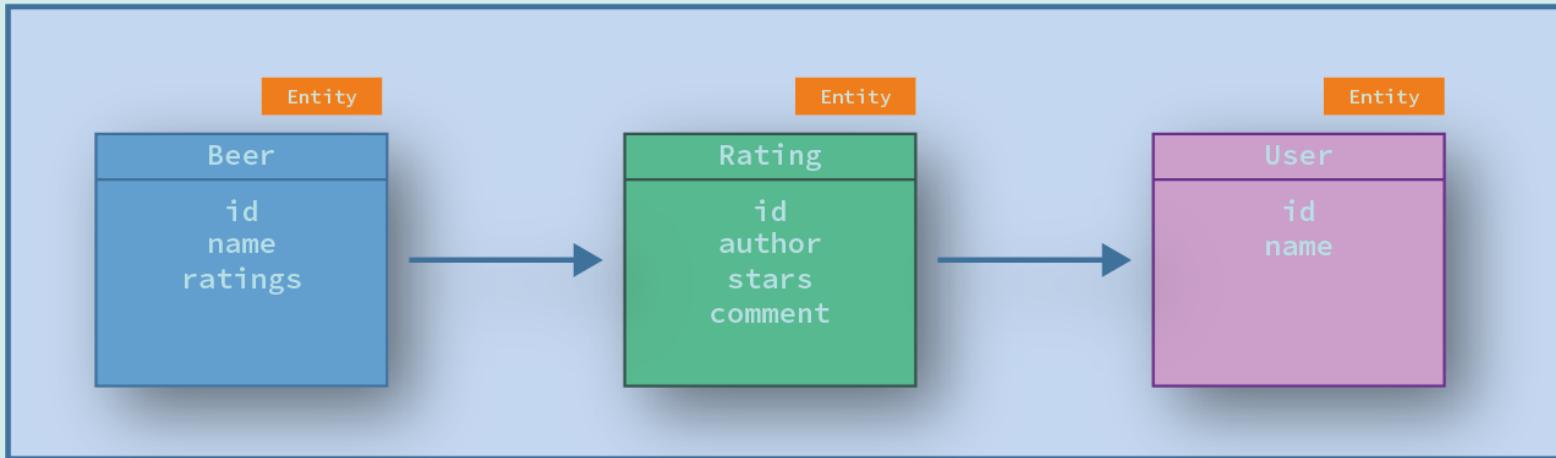
# BEERADVISOR DOMAINE

## "Domain-Model"



# BEERADVISOR DOMAINE

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



## BEER ADVISOR

A grid of six beer-related images, each with a star rating:

- Barfüßer:** A glass of beer with the brand name on it. Rating: ★★★★☆.
- Frydenlund:** A can of Pale Ale with the brand name on it. Rating: ★★★★☆.
- Grieskirchner:** A glass of beer with the brand name on it. Rating: ★★★★☆.
- Tuborg:** A can of Tuborg beer. Rating: ★★★★☆.
- Baltic Trippel:** A bottle of Baltic Trippel beer. Rating: ★★★★☆.
- Viktoria Bier Heidelberg:** A mug of beer with the brand name on it. Rating: ★★★★☆.

## BEER ADVISOR

**Frydenlund 150 NOK**

**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“ ★★★★★

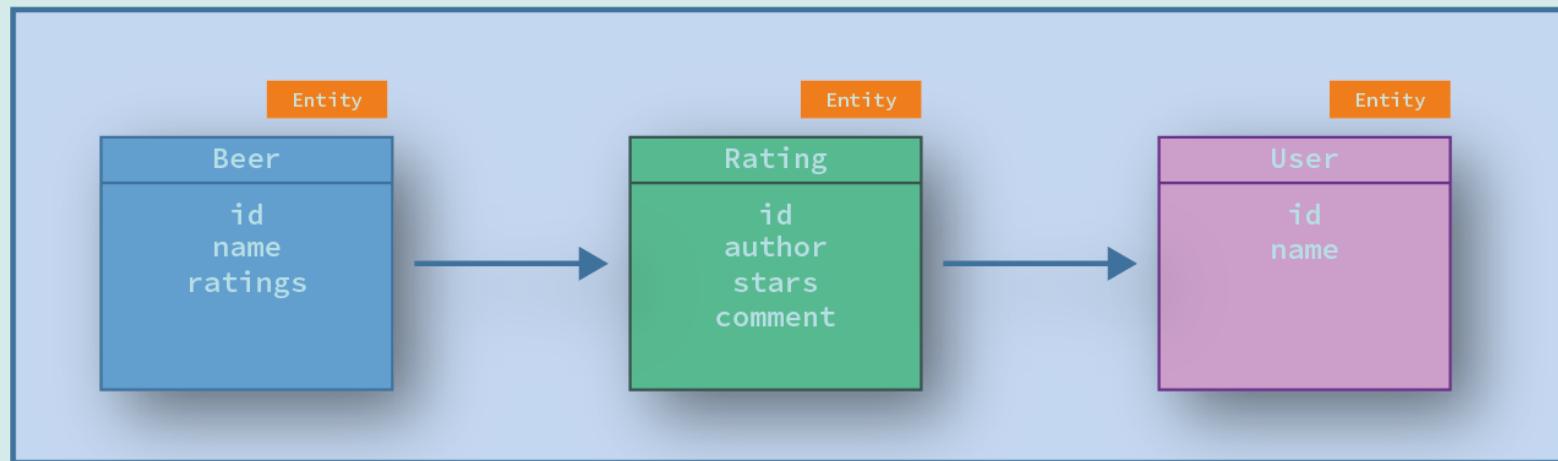
**...and what do you think?**

# ABFRAGEN MIT REST

## REST-Zugriff

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

GET /beers/1



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

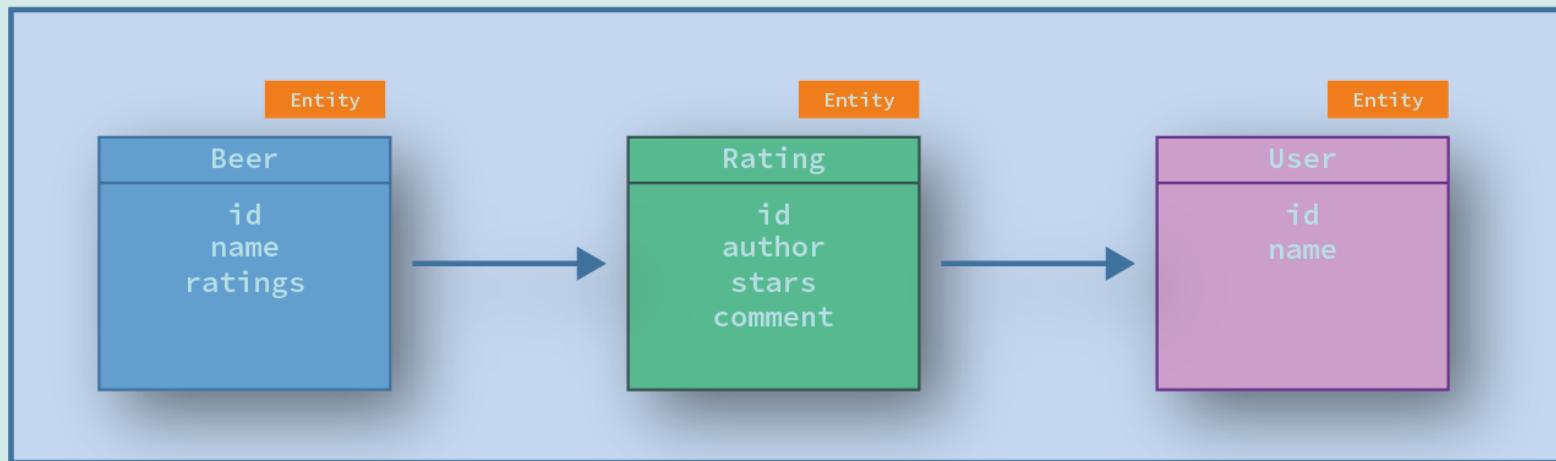
# ABFRAGEN MIT REST

## REST-Zugriff

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

GET /beers/1

GET /beers/1/ratings/R1



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

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

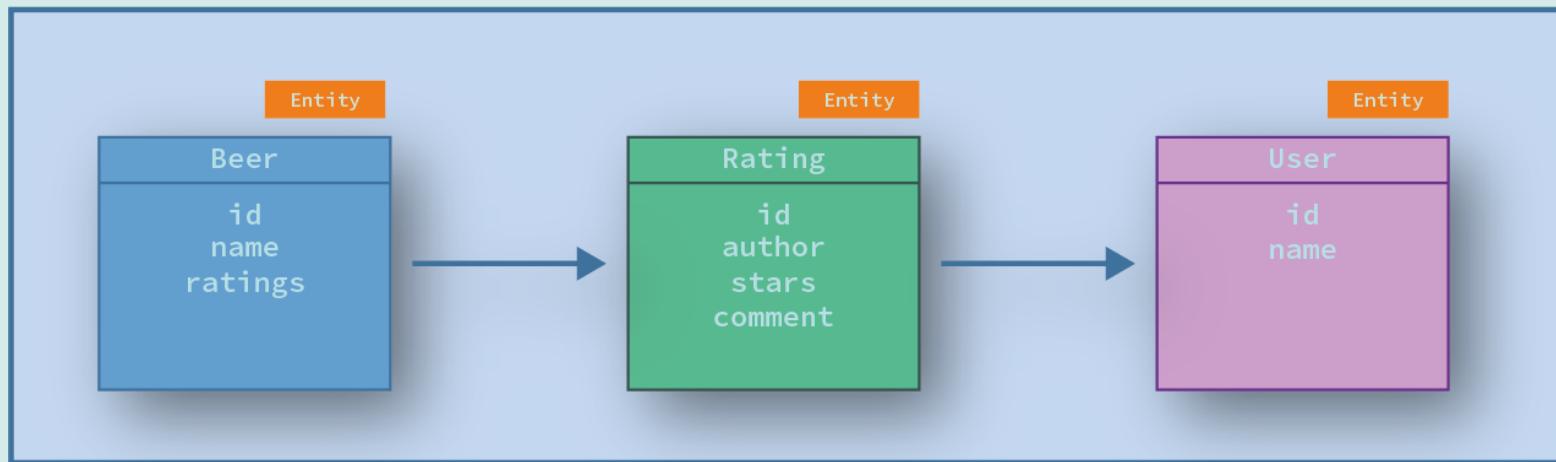
# ABFRAGEN MIT REST

## REST-Zugriff

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

GET /beers/1

GET /beers/1/ratings/R1 GET /users/U1



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

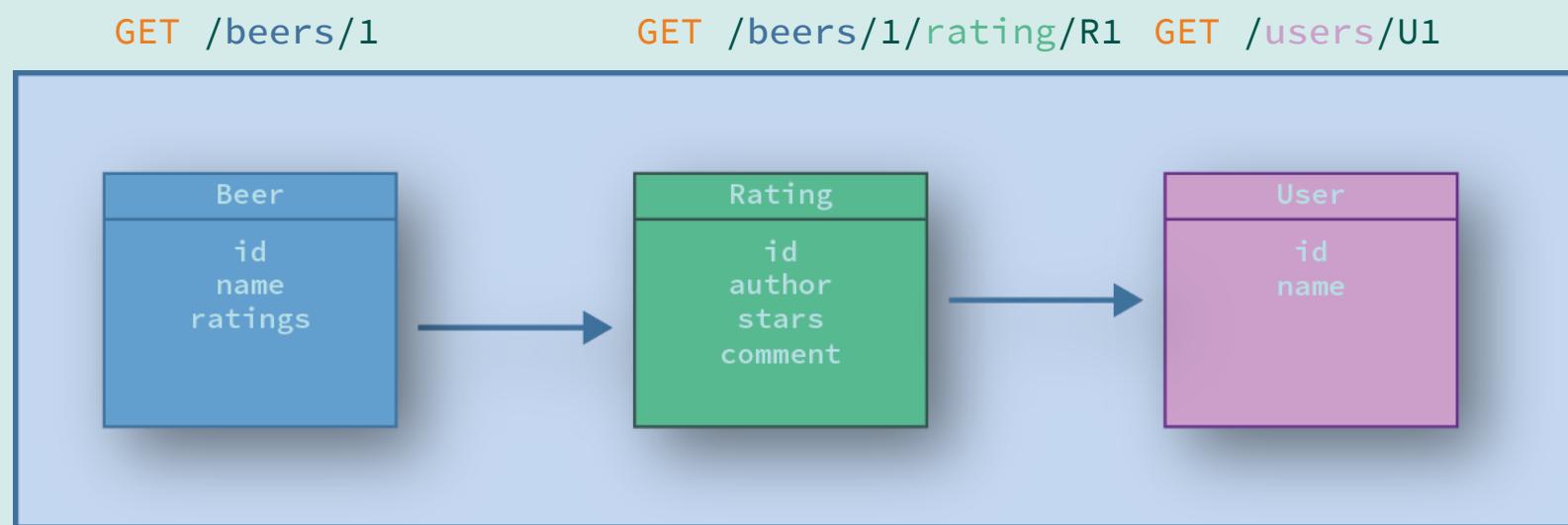
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

```
{  
  "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"  
}
```

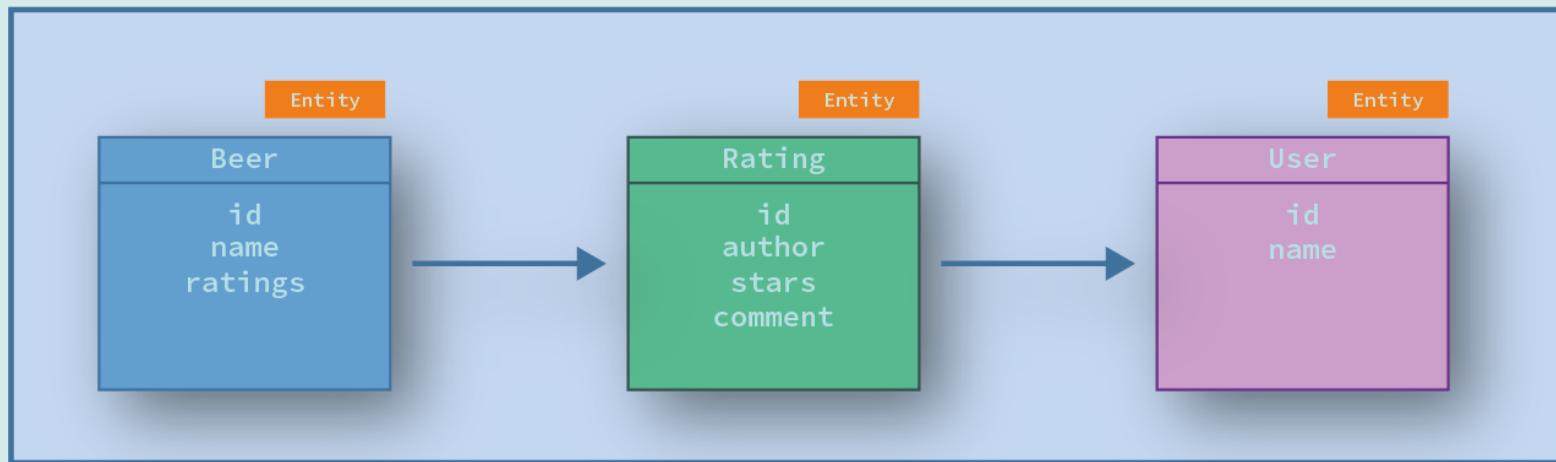
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

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

# ABFRAGEN MIT GRAPHQL

## GraphQL

```
query { beer
  { name ratings(rid: "R1")
    { stars author { name } }
  }
}
```

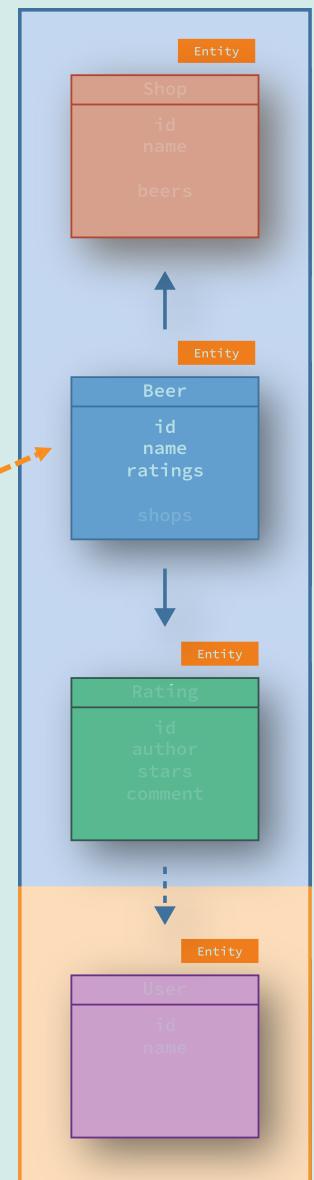
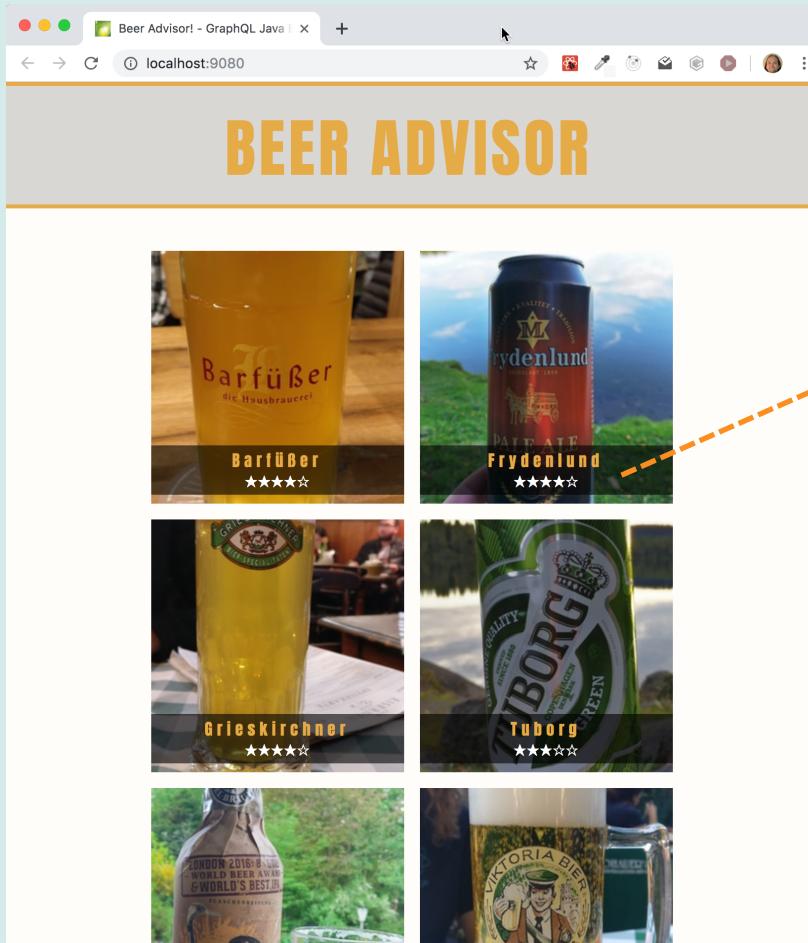


```
{
  "name": "Barfüßer",
  "ratings": {
    "stars": 3,
    "comment": "good",
    "author": { "name": "Klaus" }
  }
}
```

# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 1

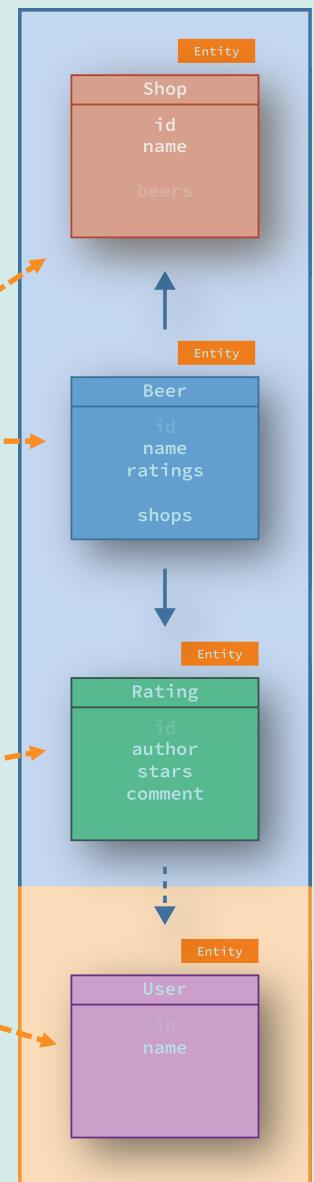
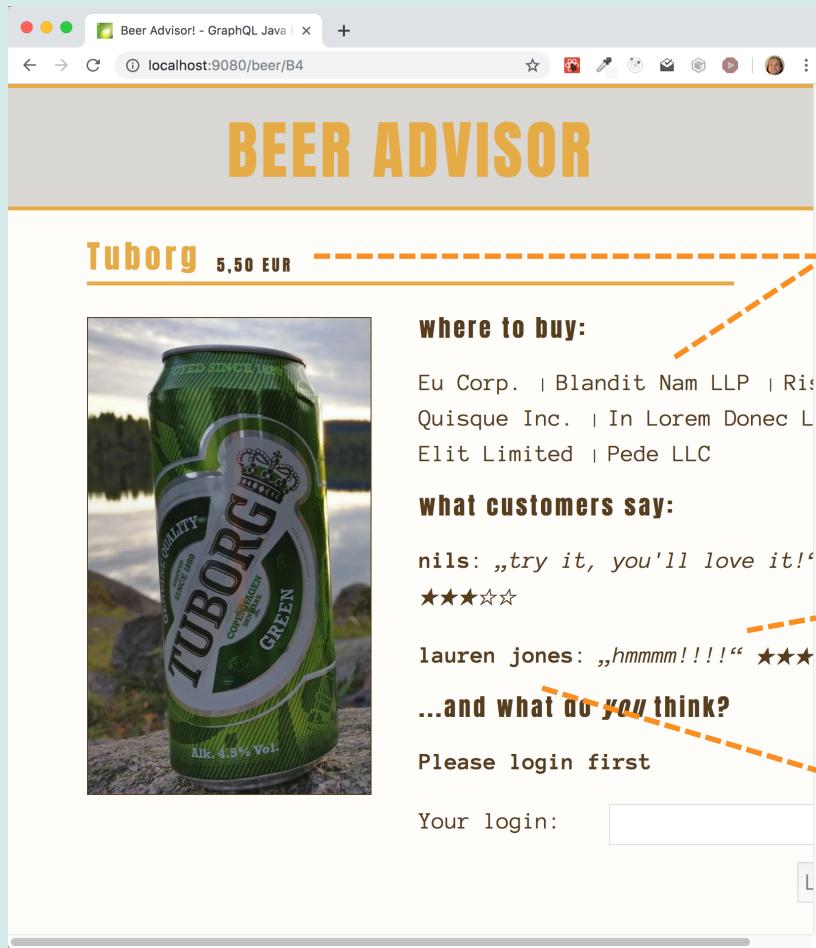
```
{ beer {  
    id  
    name  
    averageStars  
}
```



# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 2

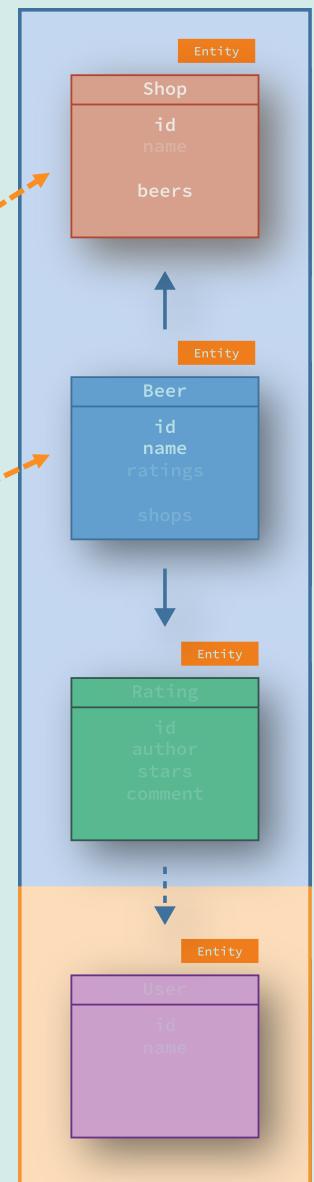
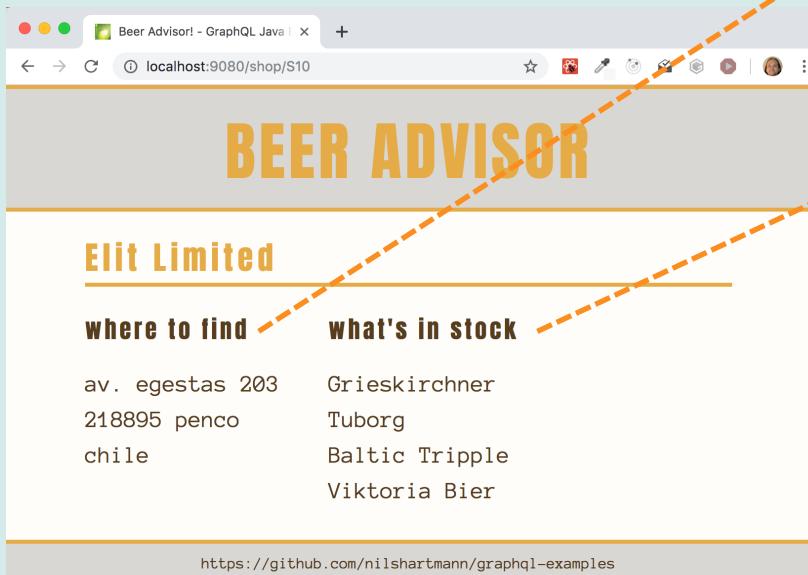
```
{ beer(beerId: "B1" {  
    name  
    price  
    ratings {  
        stars  
        comment  
        author {  
            name  
        }  
    }  
    shops { name }  
}
```



# GRAPHQL EINSATZSzenarien

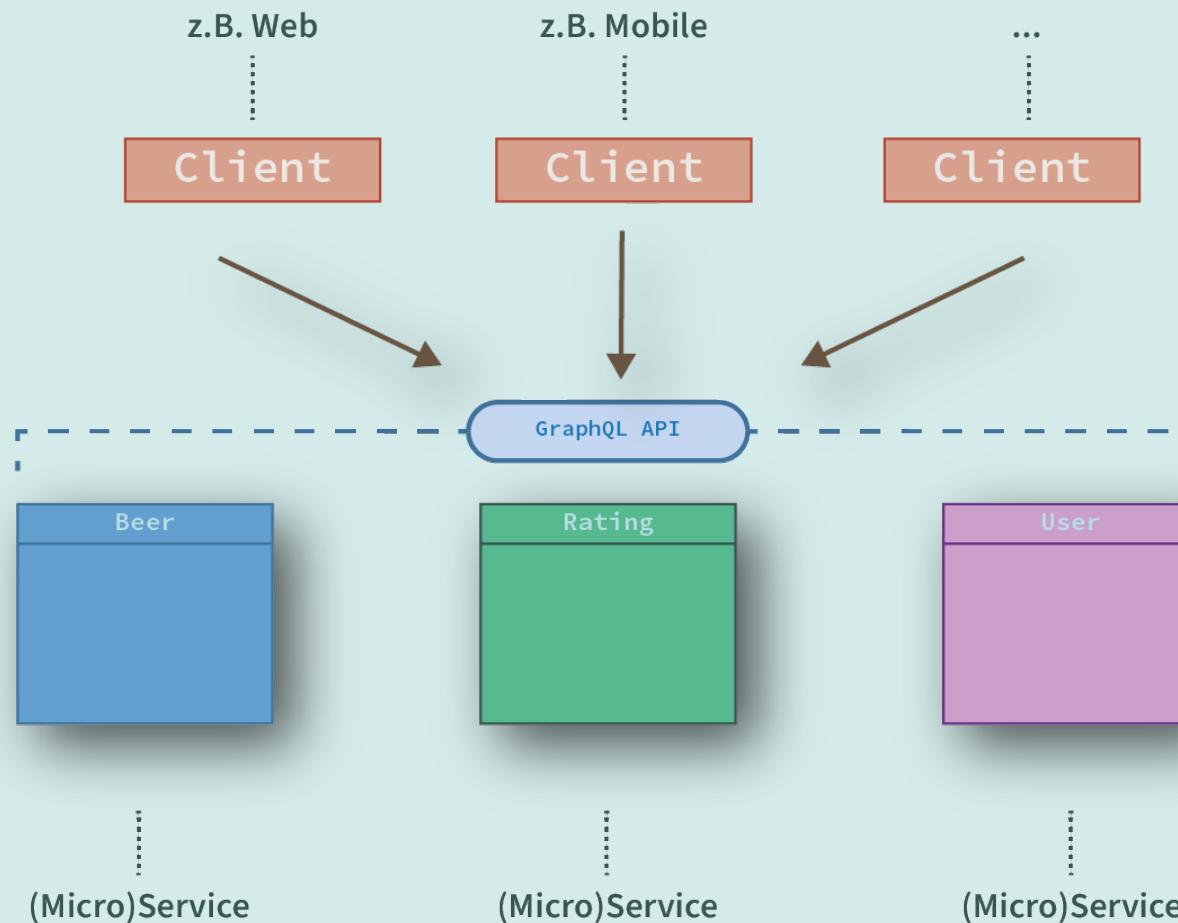
## Use-Case spezifische Abfragen – 3

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



# DATEN QUELLEN

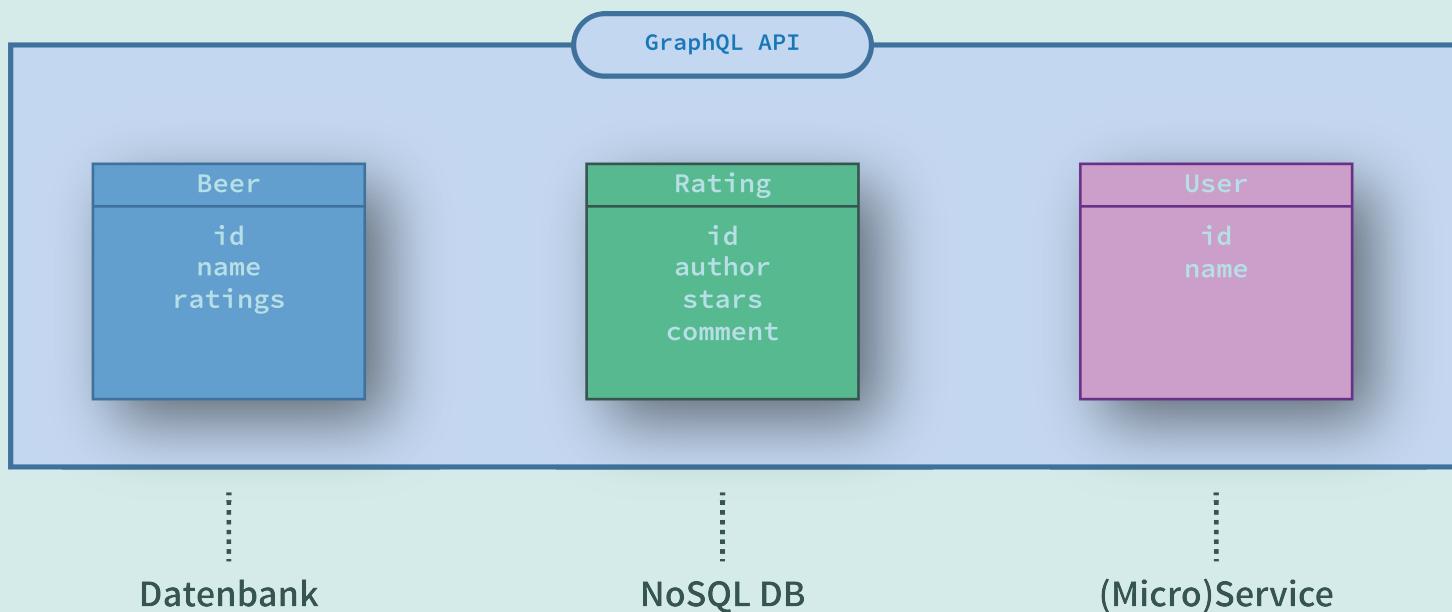
## GraphQL: Architekturbild



# DATEN QUELLEN

## GraphQL macht keine Aussage, wo die Daten herkommen

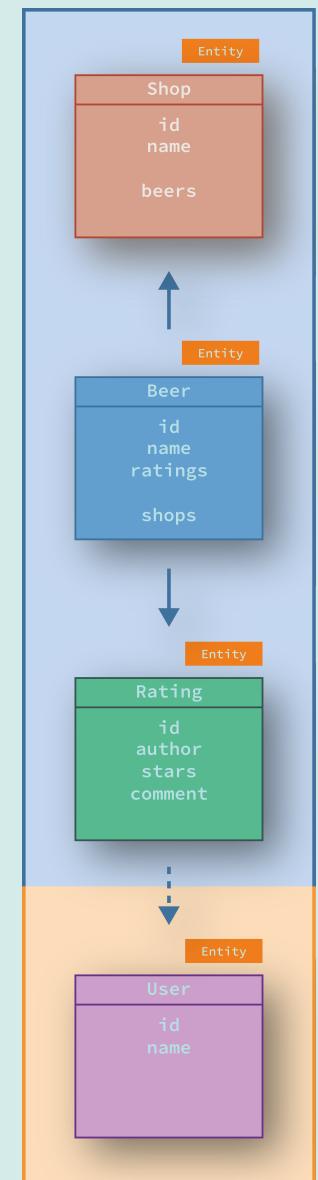
- 👉 Ermittlung der Daten ist unsere Aufgabe
- 👉 Müssen nicht aus einer Datenbank kommen
- 👉 Wir bestimmen, wie und welche Daten zur Verfügung gestellt werden!



# GRAPHQL EINSATZSzenarien

## Zusammenfassung

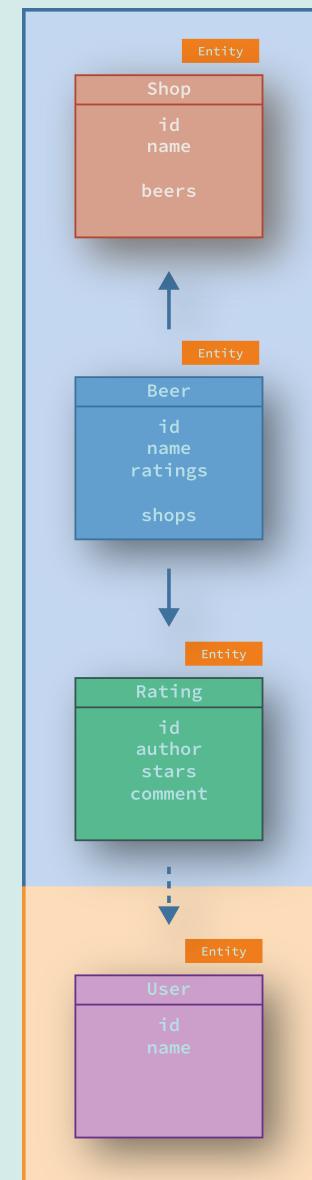
- Queries bieten *explizite* Sicht auf benötigte Daten
  - Queries können nach Geschmack ausgeführt werden  
Pro Seite, pro Komponente, ...



# GRAPHQL EINSATZSzenariEN

## Zusammenfassung

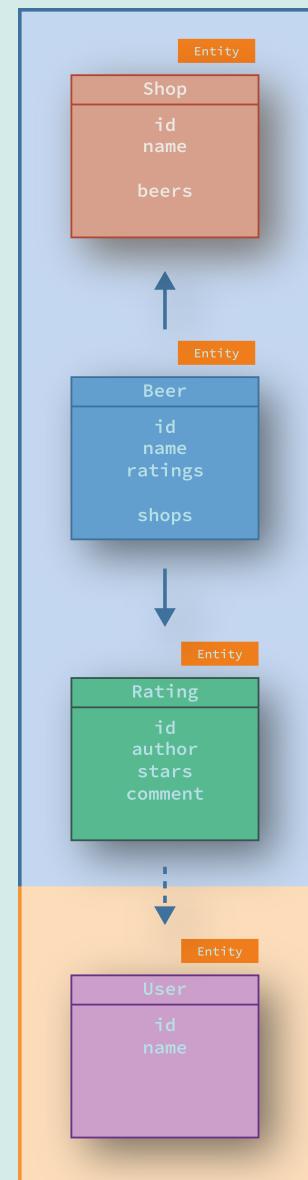
- Queries bieten *explizite* Sicht auf benötigte Daten
  - Queries können nach Geschmack ausgeführt werden  
Pro Seite, pro Komponente, ...
- Abgefragt werden *Daten*, nicht *Endpunkte*



# GRAPHQL EINSATZSzenariEN

## Zusammenfassung

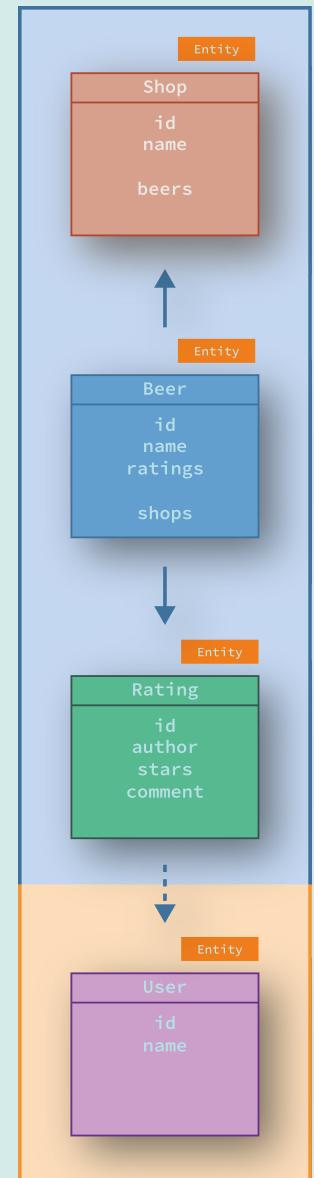
- Queries bieten *explizite* Sicht auf benötigte Daten
  - Queries können nach Geschmack ausgeführt werden  
Pro Seite, pro Komponente, ...
- Abgefragt werden *Daten*, nicht *Endpunkte*
- API kann unabhängig vom Client erweitert werden
  - Server kann neue Daten und Funktionen anbieten
  - Client fragt Daten explizit an und bekommt nie "zuviel"



# GRAPHQL EINSATZSzenariEN

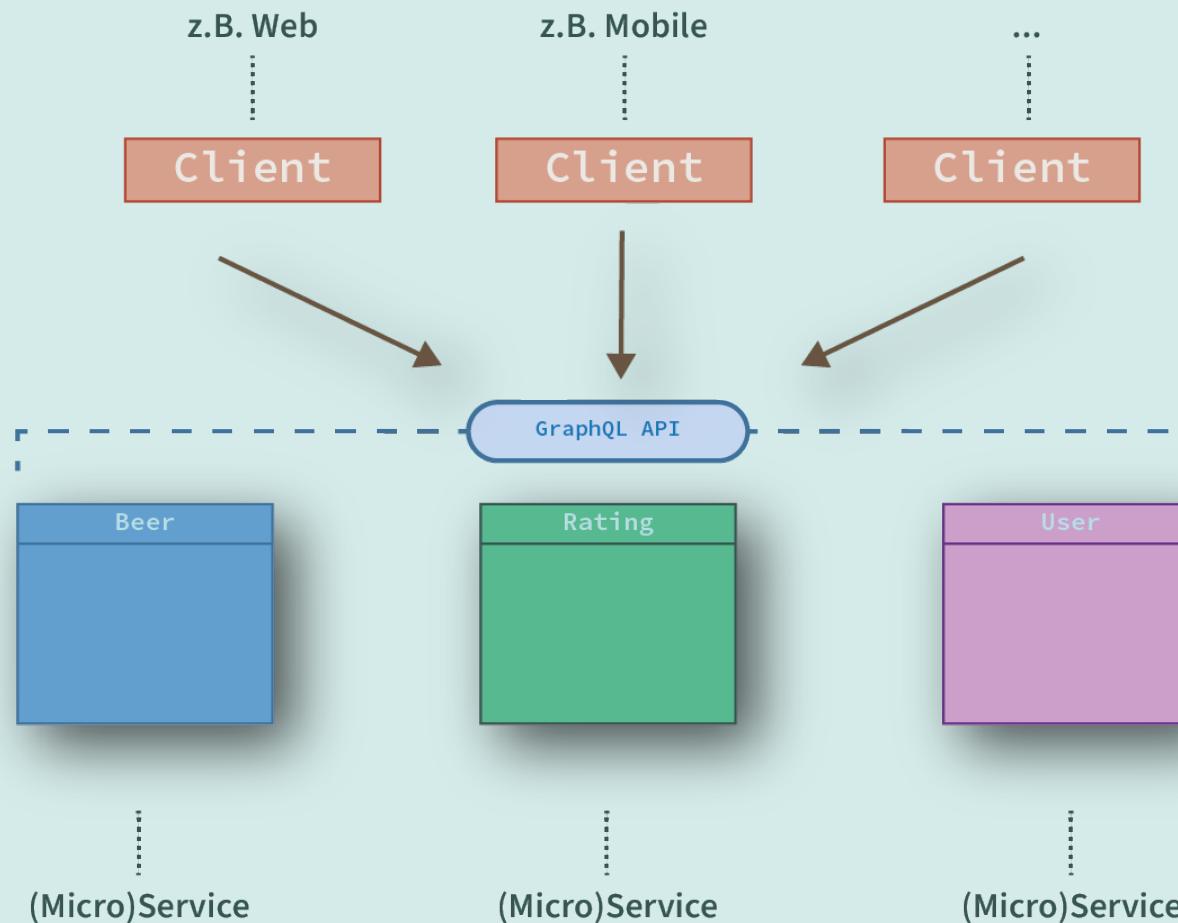
## Zusammenfassung

- Queries bieten *explizite* Sicht auf benötigte Daten
  - Queries können nach Geschmack ausgeführt werden  
Pro Seite, pro Komponente, ...
- Abgefragt werden *Daten*, nicht *Endpunkte*
- API kann unabhängig vom Client erweitert werden
  - Server kann neue Daten und Funktionen anbieten
  - Client fragt Daten explizit an und bekommt nie "zuviel"
- Mehr aus einer Hand als bei REST (Doku, Typisierung, ...)



# DATEN QUELLEN

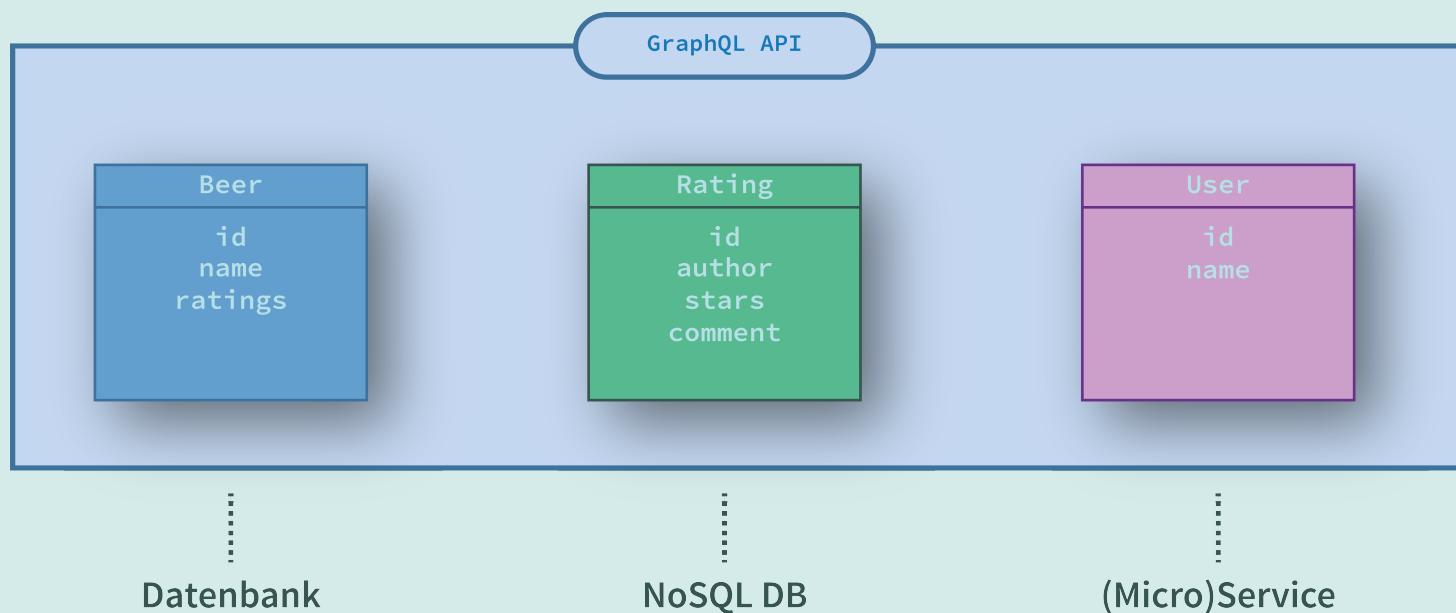
## GraphQL: Architekturbild



# DATEN QUELLEN

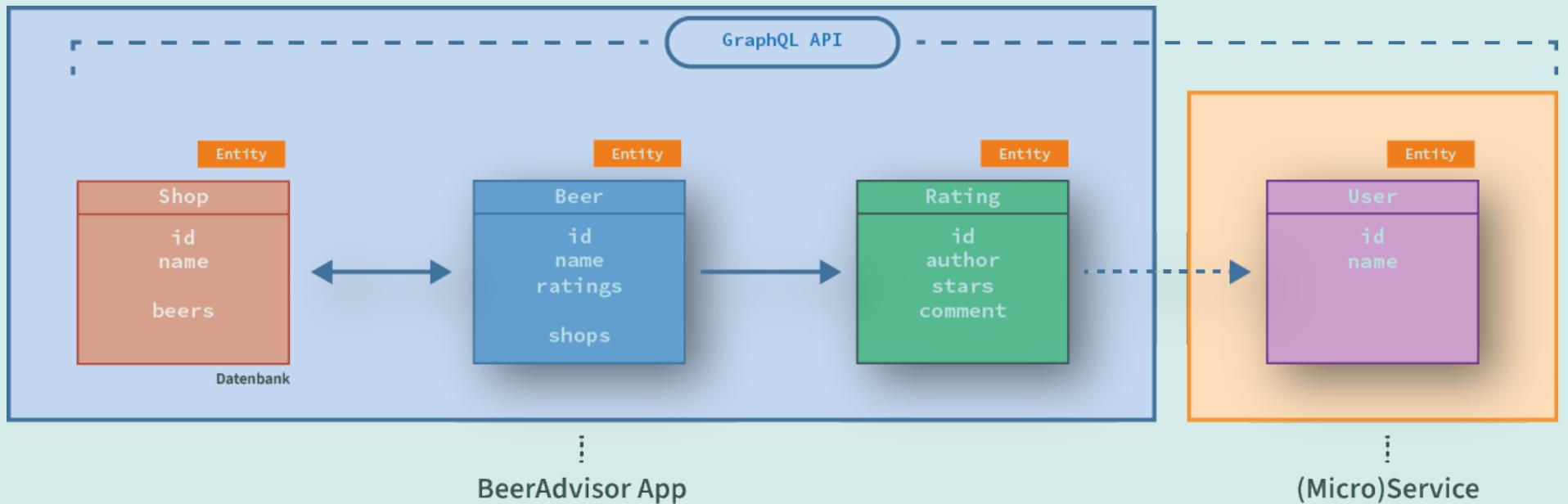
**GraphQL macht keine Aussage, wo die Daten herkommen**

👉 Ermittlung der Daten ist unsere Aufgabe



# HINTERGRUND

## "Architektur" Beer Advisor



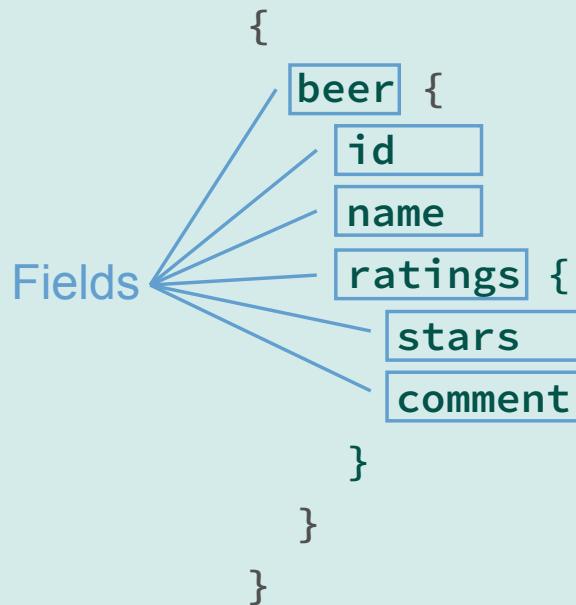
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

# GraphQL

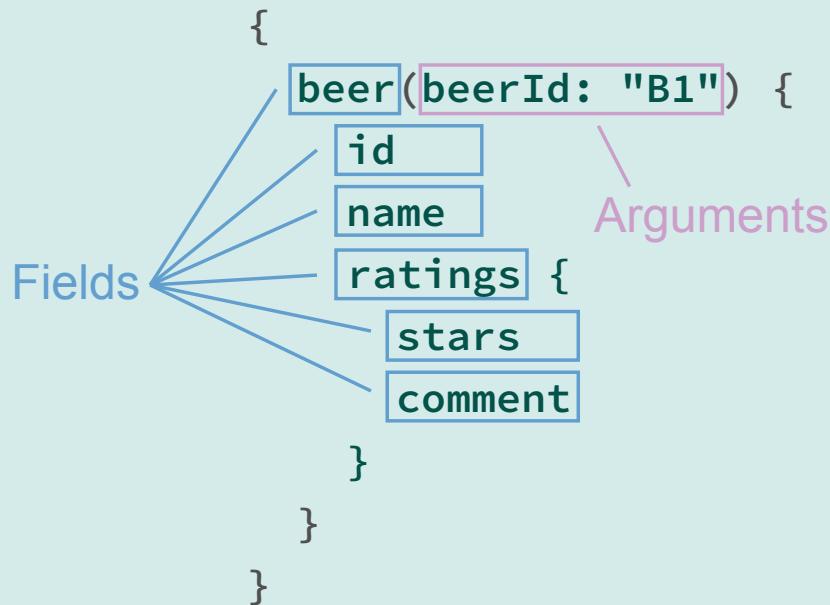
**TEIL 1: ABFRAGEN UND SCHEMA**

# 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

Operation type

```
    |     Operation name (optional)
    |
query GetMeABeer {
    beer(beerId: "B1") {
        id
        name
        price
    }
}
```

# 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"}  
  ]}  
}
```

# CACHING



Gunnar Morling   
@gunnarmorling



Antwort an [@nilshartmann](#) und [@cyberjug](#)

Wie sieht das denn aus mit Caching, das geht mit diesem  
GraphQL doch gar nicht, oder ?

1:23 nachm. · 11. Sep. 2020 · Twitter Web App

---

3 „Gefällt mir“-Angaben

<https://twitter.com/gunnarmorling/status/1304379991044567042>

# CACHING

## REST Requests

(typischerweise HTTP GET)

Browser / Anwendung

- 
- Browser Cache
  - CDN (Cloud Flare, Akami)
  - Web-Server/Proxy Cache (nginx, varnish)
  - Application Cache (z.B. @Cacheable Spring, JCache)
  - Persistenz Cache (z.B. Hibernate)
  - Datenbank Cache (Postgres, MySQL, Oracle, ...)
  - Betriebssystem Cache
  - Hardware Cache (z.B. CPU, SSD)

# CACHING

## REST Requests

(typischerweise HTTP GET)

Browser / Anwendung

- Browser Cache
- CDN (Cloud Flare, Akami)
- Web-Server/Proxy Cache (nginx, varnish)
- Application Cache (z.B. @Cacheable Spring, JCache)
- Persistenz Cache (z.B. Hibernate)
- Datenbank Cache (Postgres, MySQL, Oracle, ...)
- Betriebssystem Cache
- Hardware Cache (z.B. CPU, SSD)

HTTP GET Request



## GraphQL Requests

(typischerweise HTTP POST)

Browser / Anwendung

- Browser Cache
- ~~CDN (Cloud Flare, Akami)~~
- ~~Web-Server/Proxy Cache (nginx, varnish)~~
- Application Cache (z.B. @Cacheable Spring, JCache)
- Persistenz Cache (z.B. Hibernate)
- Datenbank Cache (Postgres, MySQL, Oracle, ...)
- Betriebssystem Cache
- Hardware Cache (z.B. CPU, SSD)

HTTP POST Request



# CACHING

## REST Requests

(typischerweise HTTP GET)

Browser / Anwendung

- Browser Cache
- CDN (Cloud Flare, Akami)
- Web-Server/Proxy Cache (nginx, varnish)
- Application Cache (z.B. @Cacheable Spring, JCache)
- Persistenz Cache (z.B. Hibernate)
- Datenbank Cache (Postgres, MySQL, Oracle, ...)
- Betriebssystem Cache
- Hardware Cache (z.B. CPU, SSD)

HTTP GET Request



- Was wollen wir überhaupt cachen?
  - Wie veränderlich sind unsere Daten? Abfrage meines Kontostandes?
- Statische Assets können weiterhin gecached werden (kein GraphQL)
- Eine Frage der Anforderungen!

## GraphQL Requests

(typischerweise HTTP POST)

Browser / Anwendung

- Browser Cache
- CDN (Cloud Flare, Akami) ~~HTTP POST Request~~
- Web-Server/Proxy Cache (nginx, varnish) ~~HTTP POST Request~~
- Application Cache (z.B. @Cacheable Spring, JCache)
- Persistenz Cache (z.B. Hibernate)
- Datenbank Cache (Postgres, MySQL, Oracle, ...)
- Betriebssystem Cache
- Hardware Cache (z.B. CPU, SSD)

HTTP POST Request



# 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": { . . . }  
}
```

# GRAPHQL SCHEMA

## Schema

- 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

Object Type ----- type Rating {  
Fields                    id: ID!  
                          comment: String!  
                          stars: Int  
                          }

# 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" } }  
      },  
      ...  
    ]  
  }  
}
```

# 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...)

*"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)**

## Variante 1: graphql-java und graphql-kickstart

- Reine GraphQL Implementierung, keine Aussage über Laufzeitumgebung
- Modular aufgebaut; es existieren z.B. GraphQL Servlets, Auto-Konfiguration für Spring Boot etc.

# GRAPHQL-JAVA UND GRAPHQL-JAVA-KICKSTART

## Variante 1: graphql-java und graphql-kickstart

- Reine GraphQL Implementierung, keine Aussage über Laufzeitumgebung
- Modular aufgebaut; es existieren z.B. GraphQL Servlets, Auto-Konfiguration für Spring Boot etc.

## Variante 2: MicroProfile GraphQL

- Erst seit Anfang 2020
- Kein Support für Subscriptions
- Schema wird über Annotations definiert
- Support u.a. in Quarkus und Open Liberty

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Low-Level API: graphql-java

- <https://www.graphql-java.com/>
- *Die gezeigten Konzepte sind in GraphQL-Frameworks für andere Programmier-Sprachen ähnlich!*

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Schritt 1: Schema definieren

- Per API oder per .graphqls-Datei

```
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
}
```

## Schritt 2: DataFetcher

- Ein **DataFetcher** liefert ein Wert für ein angefragtes Feld
  - Zwingend erforderlich für Root-Types (Query, Mutation)
  - Default: per Reflection (getter/setter, Maps, ...)
- (In anderen Implementierungen auch **Resolver** genannt)

## Schritt 2: DataFetcher

- Ein **DataFetcher** liefert ein Wert für ein angefragtes Feld
  - Zwingend erforderlich für Root-Types (Query, Mutation)
  - Default: per Reflection (getter/setter, Maps, ...)
- (In anderen Implementierungen auch **Resolver** genannt)
- DataFetcher ist funktionales Interface (kann als Lambda implementiert werden):

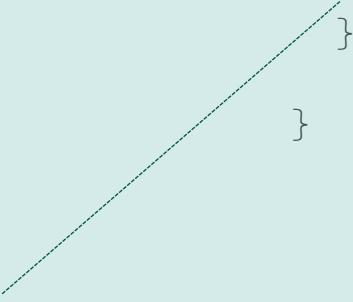
```
interface DataFetcher<T> {  
    T get(DataFetchingEnvironment environment);  
}
```

# DATAFETCHER

## DataFetcher implementieren

- Beispiel: beers-Feld

```
public class BeerAdvisorDataFetchers {  
  
    public DataFetcher<List<Beer>> beersFetcher() {  
        return environment -> beerRepository.findAll();  
    }  
  
}  
  
type Query {  
    beers: [Beer!]!  
}  
}
```



# DATAFETCHER

## DataFetcher implementieren: environment-Parameter

- environment gibt Informationen über den Query (z.B. Argumente)

```
public class BeerAdvisorDataFetchers {

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

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

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

# DATAFETCHER

## DataFetcher implementieren: Mutations

- technisch analog zu Query
- dürfen Daten verändern

```
public DataFetcher<Rating> addRatingMutationFetcher() {  
    return environment -> {  
        final Map<String, Object> ri =  
            environment.getArgument("ratingInput");  
  
        type Mutation {  
            addRating  
            (ratingInput: AddRatingInput):  
                Rating!  
        }  
  
        Rating r = new Rating();  
        r.setBeerId((String)ratingInput.get("beerId"));  
        r.setComment((String)ratingInput.get("comment"));  
        r.setStars((Integer)ratingInput.get("stars"));  
        r.setUserId((String)ratingInput.get("userId"));  
  
        return ratingService.addRating(r);  
    };  
}
```

# DATAFETCHER

## DataFetcher implementieren: Subscriptions

- Müssen Reactive Streams Publisher zurückliefern
- Beim Lesen über HTTP üblicherweise über Websockets

```
import org.reactivestreams.Publisher;

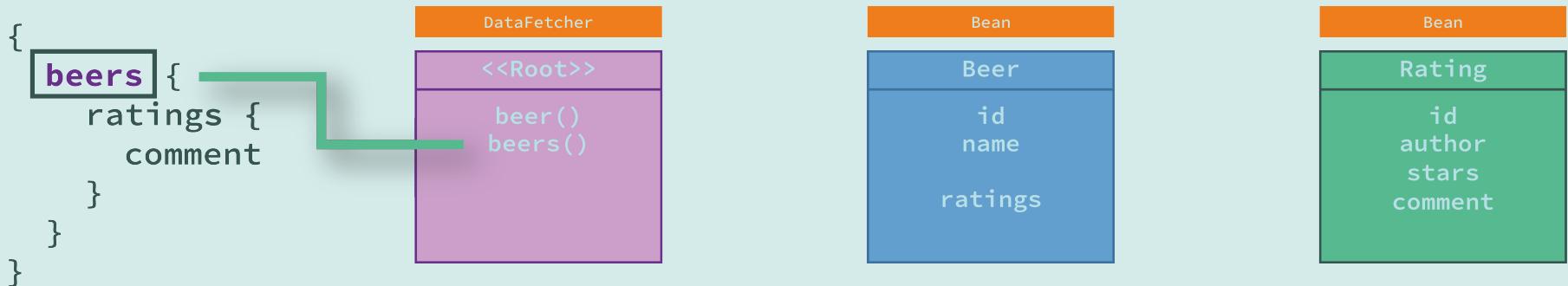
public DataFetcher<Publisher<Rating>> onNewRatingFetcher() {

    type Subscription {
        onNewRating: Rating!
    }

    return environment -> {
        Publisher<Rating> publisher = getRatingPublisher();
        return publisher;
    };
}
```

# DATEN ERMITTLEMENT ZUR LAUFZEIT

- 1. DataFetcher (wie eben implementiert)



# DATEN ERMITTLEMENT ZUR LAUFZEIT

- 2. Zugriff auf Bean (PropertyDataFetcher)

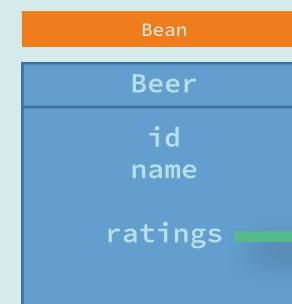
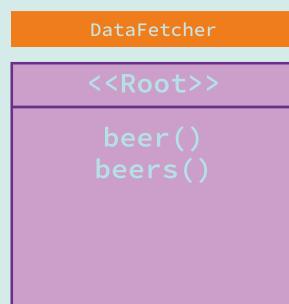
```
{  
  beers {  
    ratings {  
      comment  
    }  
  }  
}
```



# DATEN ERMITTLEMENT ZUR LAUFZEIT

- 3. Zugriff auf Bean (PropertyDataFetcher)

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

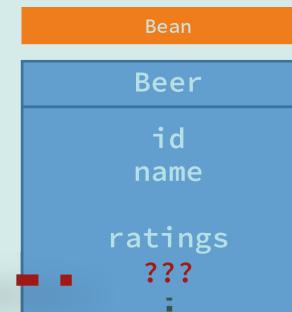
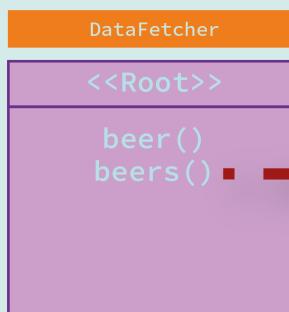


PropertyDataFetcher (Reflection per Getter oder Field-Access)

# DATEN ERMITTLEMENT ZUR LAUFZEIT

**Problem:** Mismatch zwischen Java-Klassen und Schema

```
{  
  beers {  
    ratingsWithStars  
    (stars: 3) {  
      comment  
    }  
  }  
}
```



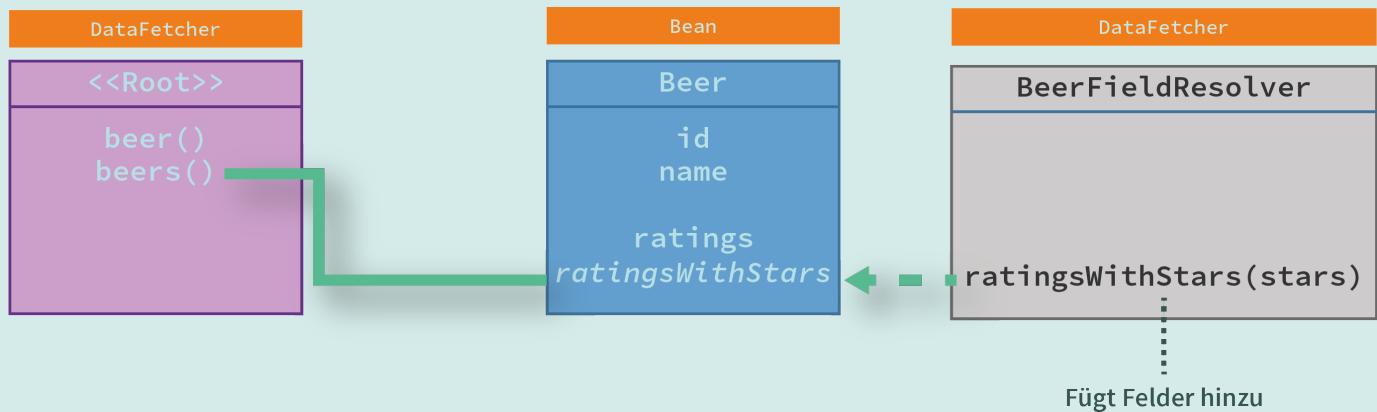
Feld/Methode „ratingWithStars“ nicht in Beer-Klasse vorhanden

# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

- PropertyDataFetcher ist nur default, Fetcher können *pro Feld* festgelegt werden
- Z.B. auch für Felder, deren Signatur zwischen API und Java-Klasse abweicht
  - (Rückgabe-Wert oder Parameter)
- Oder die aus anderer Datenbank, Daten-Quelle kommen oder berechnet werden
- *DataFetcher wird nur ausgeführt, wenn Feld auch im Query abgefragt wird*

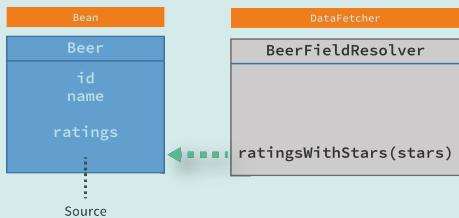
```
{  
  beers {  
    ratingsWithStars  
    (stars: 3) {  
      comment  
    }  
  }  
}
```



# DATA FETCHER FÜR NICHT-ROOT-FELDER

## DataFetcher implementieren

- `getSource()` liefert das Parent-Objekt zurück, auf dem das Feld abgefragt wird



```
public class BeerDataFetchers {  
  
    public DataFetcher<List<Rating>> ratingsWithStarsFetcher() {  
        return environment -> {  
            Beer beer = environment.getSource();  
  
            return beer.ratingsWithStars(environment.getArgument("stars"));  
        };  
    };  
};
```

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

# DATA FETCHER FÜR NICHT-ROOT-FELDER

## DataFetcher implementieren

- `getSource()` liefert das Parent-Objekt zurück, auf dem das Feld abgefragt wird



```
public class BeerDataFetchers {  
  
    public DataFetcher<List<Rating>> ratingsWithStarsFetcher() {  
        return environment -> {  
            Beer beer = environment.getSource();  
            int starsInput = environment.getArgument("stars");  
  
            return beer.getRatings().stream()  
                .filter(r -> r.getStars() == starsInput)  
                .collect(Collectors.toList());  
        };  
    }  
}
```

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

## ALTERNATIVE: GRAPHQL-JAVA-TOOLS

### Resolver mit graphql-java-tools

- <https://github.com/graphql-java-kickstart/graphql-java-tools>
- Abstraktion, basierend auf graphql-java, arbeitet mit POJOs

## ALTERNATIVE: GRAPHQL-JAVA-TOOLS

### Resolver mit graphql-java-tools

- <https://github.com/graphql-java-kickstart/graphql-java-tools>
- Abstraktion, basierend auf graphql-java, arbeitet mit POJOs

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



## ALTERNATIVE: GRAPHQL-JAVA-TOOLS

### Resolver mit graphql-java-tools

- <https://github.com/graphql-java-kickstart/graphql-java-tools>
- Abstraktion, basierend auf graphql-java, arbeitet mit POJOs

```
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 implementieren

- Beispiel: Root-Resolver (Mutation)
- Ähnlich wie Query-Resolver

```
public class RatingMutationResolver implements
GraphQLMutationResolver {

    // z.B via DI
    private RatingRepository ratingRepository;

    public Rating addRating(AddRatingInput newRating) {
        Rating rating = Rating.from(newRating);
        ratingRepository.save(rating);
        return rating;
    }
}

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

# RESOLVER

# Resolver implementieren

- Beispiel: Root-Resolver (Mutation)
  - Input-Typ kann normales POJO sein

```
input AddRatingInput {  
    beerId: ID!  
    userId: ID!  
    comment: String!  
    stars: Int!  
}  
  
public class AddRatingInput {  
    private String beerId;  
    private String userId;  
    private String comment;  
    private int stars;  
  
    // ... getter und setter ...  
}
```

## Validierung zur Laufzeit

- Alle Resolver müssen vorhanden sein
  - Return-Types und Methoden-Parameter der Resolver-Funktionen müssen zum Schema passen
- Resolver werden immer mit korrekten Parametern aufgerufen
  - Argumente haben korrekten Typ
  - Argumente sind ggf. nicht null
- Rückgabe-Wert eines Resolvers wird überprüft
  - Client erhält nie ungültige Werte
- Es werden nur Felder herausgegeben, die auch im Schema definiert sind
  - Alle anderen Felder einer Java-Klasse sind "unsichtbar"

### Weitere GraphQL Projekte im Java-Umfeld

- **HTTP Endpunkt:** graphql-java-servlet (<https://github.com/graphql-java-kickstart/graphql-java-servlet>)
- **Spring Boot Starter:** <https://github.com/graphql-java-kickstart/graphql-spring-boot>
- **GraphQL Schema mit Java Annotations beschreiben:** <https://github.com/Enigmatis/graphql-java-annotations>

### GraphQL MicroProfile

- **Spezifikation:** <https://github.com/eclipse/microprofile-graphql>
- **Quarkus:** <https://quarkus.io/guides/microprofile-graphql>
- **Open Liberty:** <https://openliberty.io/blog/2020/06/05/graphql-open-liberty-20006.html#GQL>

### GraphQL Code Generator

- **Generator für zahlreiche Sprachen und Bibliotheken:**  
<https://graphql-code-generator.com/>
- **Generator für Queries und Antworten (Java):**  
<https://github.com/adobe/graphql-java-generator>
- **Spring Boot Starter:** <https://github.com/graphql-java-kickstart/graphql-spring-boot>

# 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 a "message", "locations", "path", and "extensions" field.

```
{
  "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

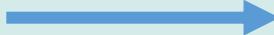
- Fachliche Fehler können auch im fachlichen Error-Objekt untergebracht werden

# 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!  
}
```

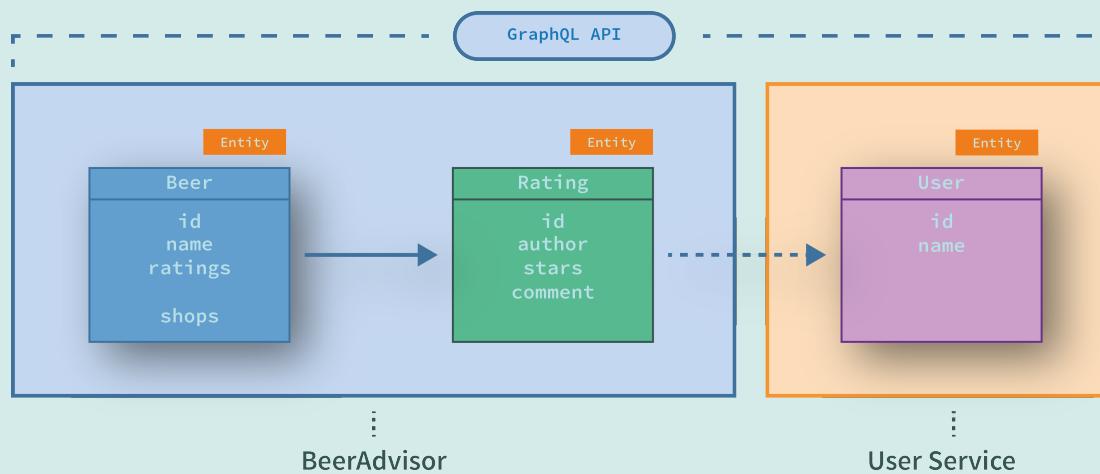


```
type Mutation {  
  addRatingInput  
  (ratingInput:  
   RatingInput!):  
   AddRatingResult!  
}  
  
type AddRatingResult {  
  newRating: Rating  
  validationErrors: [ValidationError!]  
}  
  
type ValidationError {  
  field: String!  
  msg: String!  
}
```

# 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

## 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);  
    });  
}
```

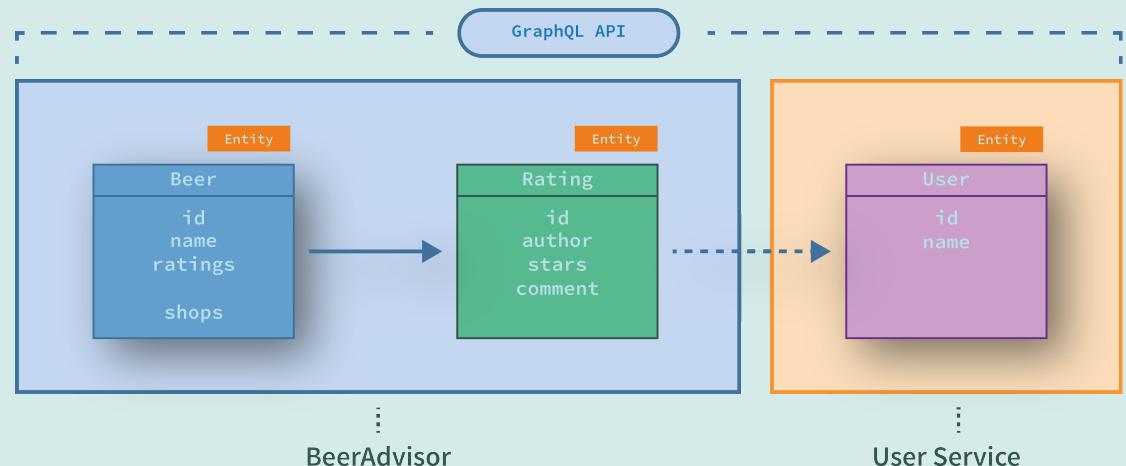
👉 @async

# IMPLEMENTIERUNG



Was gibt es bei der Ausführung dieses Querys 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)

```
{ 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);  
    };  
}
```

Remote-Call!

# 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);  
    };  
}
```

=> 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

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

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*

```
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!)

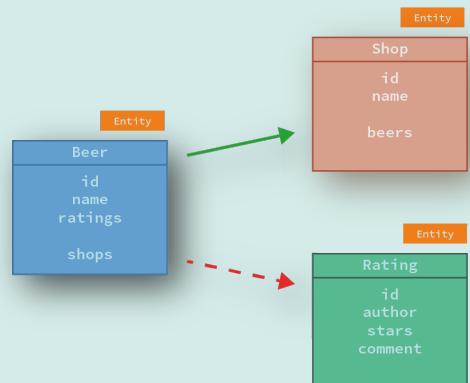
```
query {  
  beers {  
    ratings {  
      author @useDataLoader {  
        id  
        name  
      }  
    }  
  }  
}
```

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

# EXKURS: OPTIMIERUNGEN

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

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

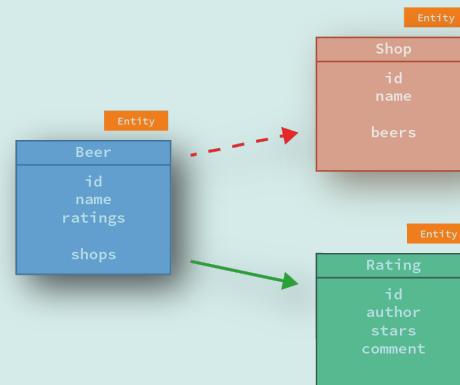
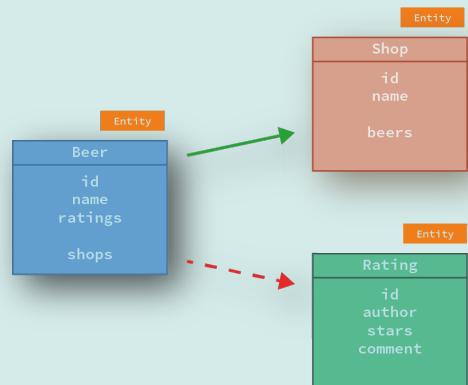


# EXKURS: OPTIMIERUNGEN

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

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

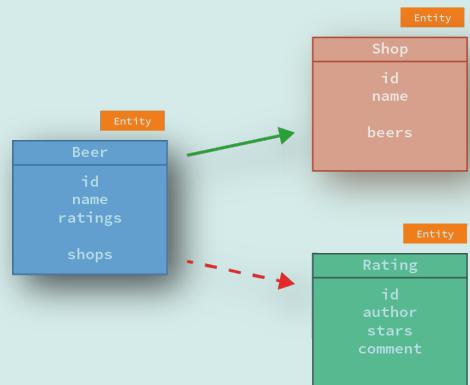
```
beers {  
    name  
    ratings {  
        comment  
    }  
}
```



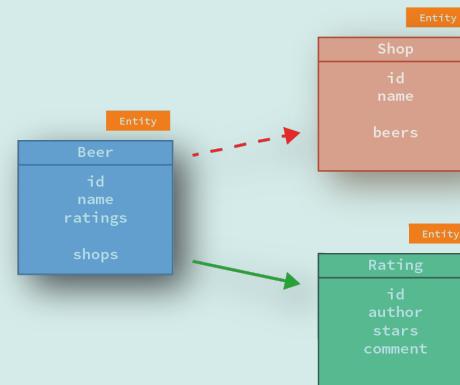
# EXKURS: OPTIMIERUNGEN

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

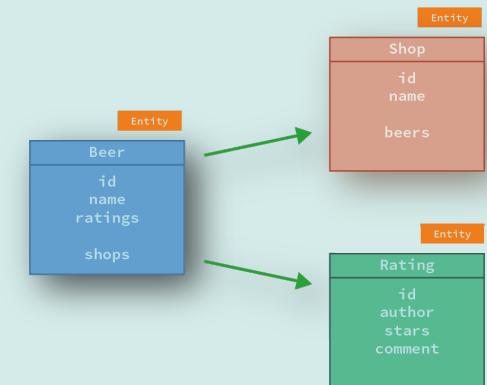
```
beers {  
    name  
    shops {  
        name  
    }  
}
```



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



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

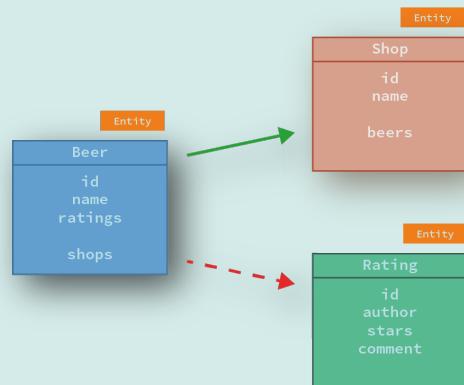


# EXKURS: OPTIMIERUNGEN

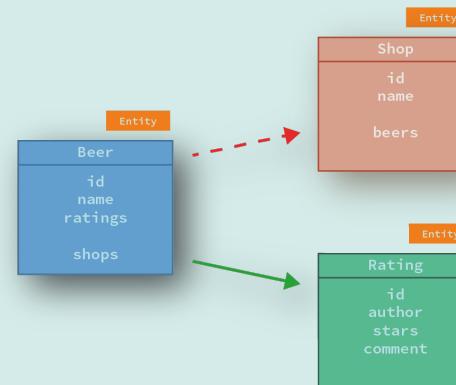
## Problem: optimaler Datenbankzugriff (Beispiel: JPA/JOINs)

- nur zur Laufzeit ermittelbar
- möglichst auf oberstem DataFetcher entscheiden

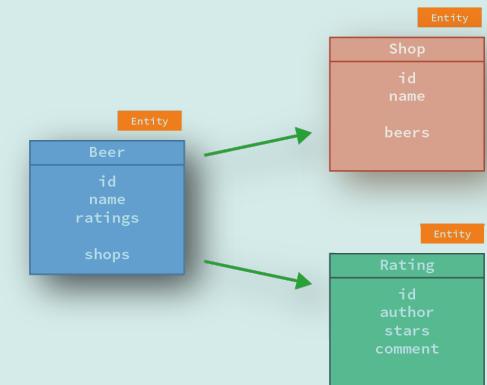
```
beers {  
    name  
    shops {  
        name  
    }  
}
```



```
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);  
    };  
}
```

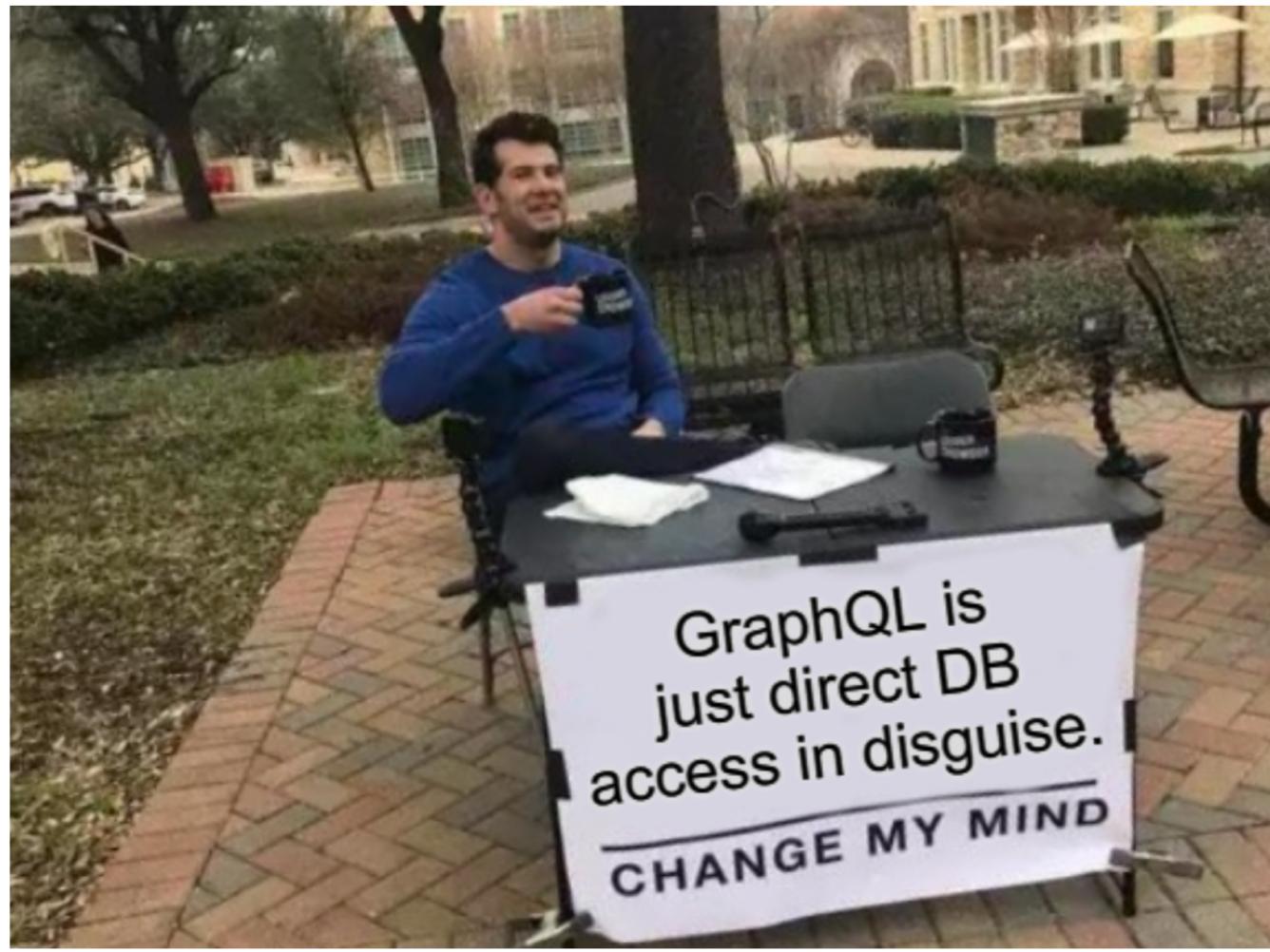
# EXKURS: OPTIMIERUNGEN

## 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);  
    };  
}
```



## 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**

**GraphQL...**

**Heilsbringer**

**oder**

**Teufelszeug**

**...müsst ihr selbst entscheiden**



<https://reactbuch.de>

# Vielen Dank!

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

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

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

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