

**NILS HARTMANN**

<https://nilshartmann.net>

# GraphQL

für Java-Anwendungen

Slides (PDF): <https://react.schule/cyberjug-graphql>

# 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**

*"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



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

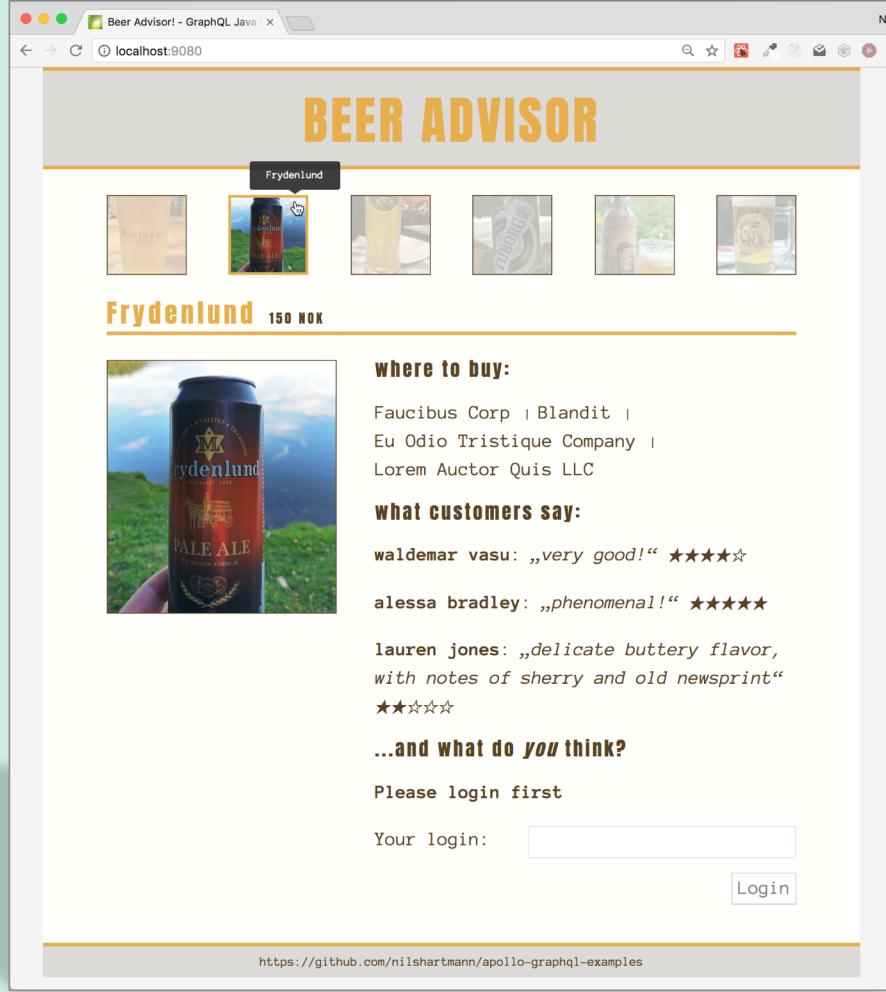


A look under the hood.

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



# GraphQL praktisch

Source-Code: <https://react.schule/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

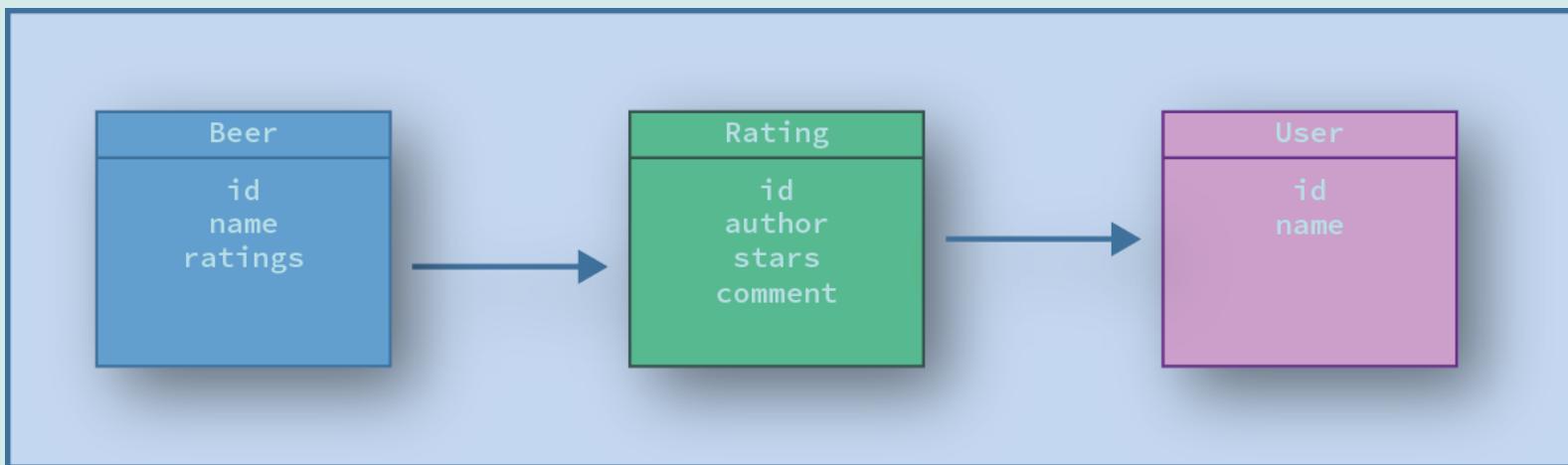
# Demo: GraphiQL

<http://localhost:9000/>

# **Vergleich mit REST**

# BEERADVISOR DOMAINE

## "Domain-Model"

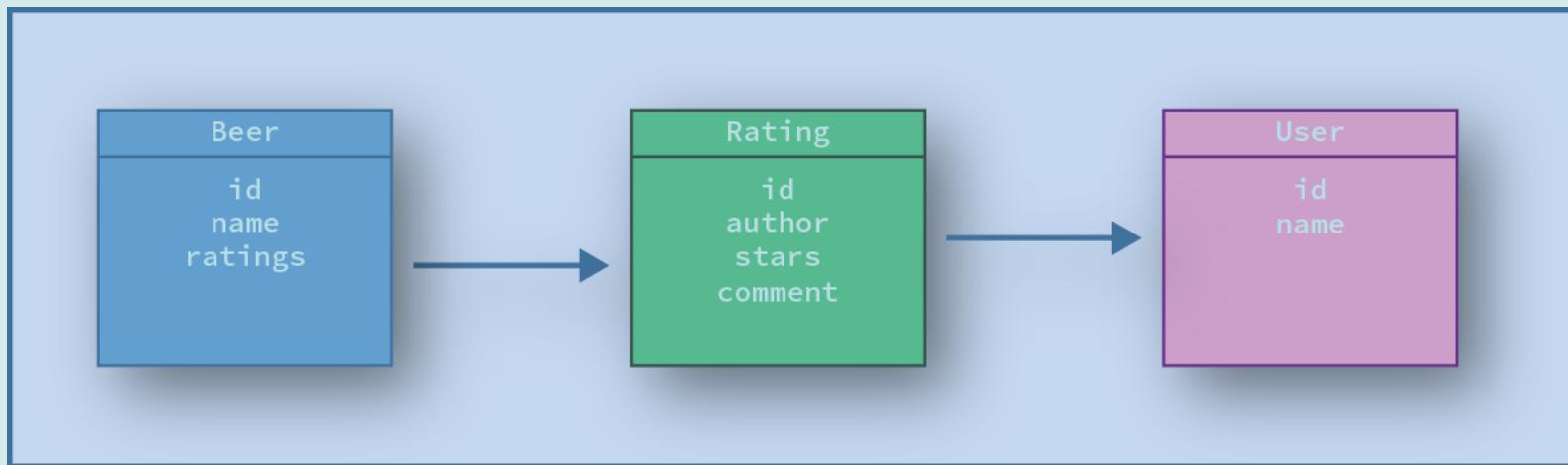


# ABFRAGEN MIT REST

## REST-Zugriff

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

GET /beer/1



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

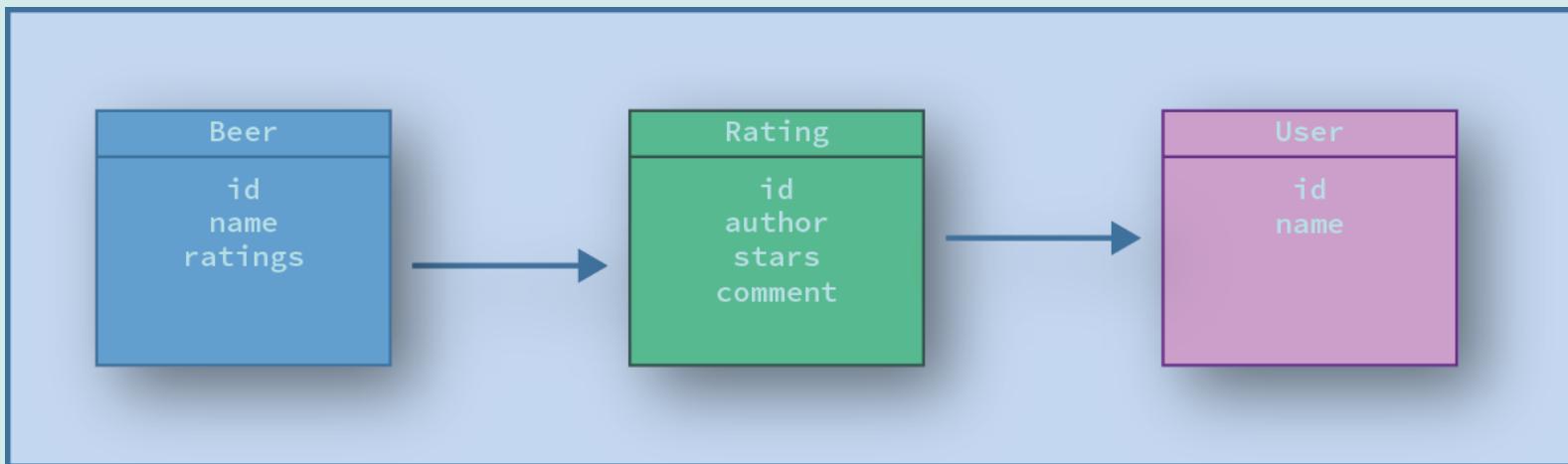
# ABFRAGEN MIT REST

## REST-Zugriff

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

GET /beer/1

GET /beer/1/rating/R1



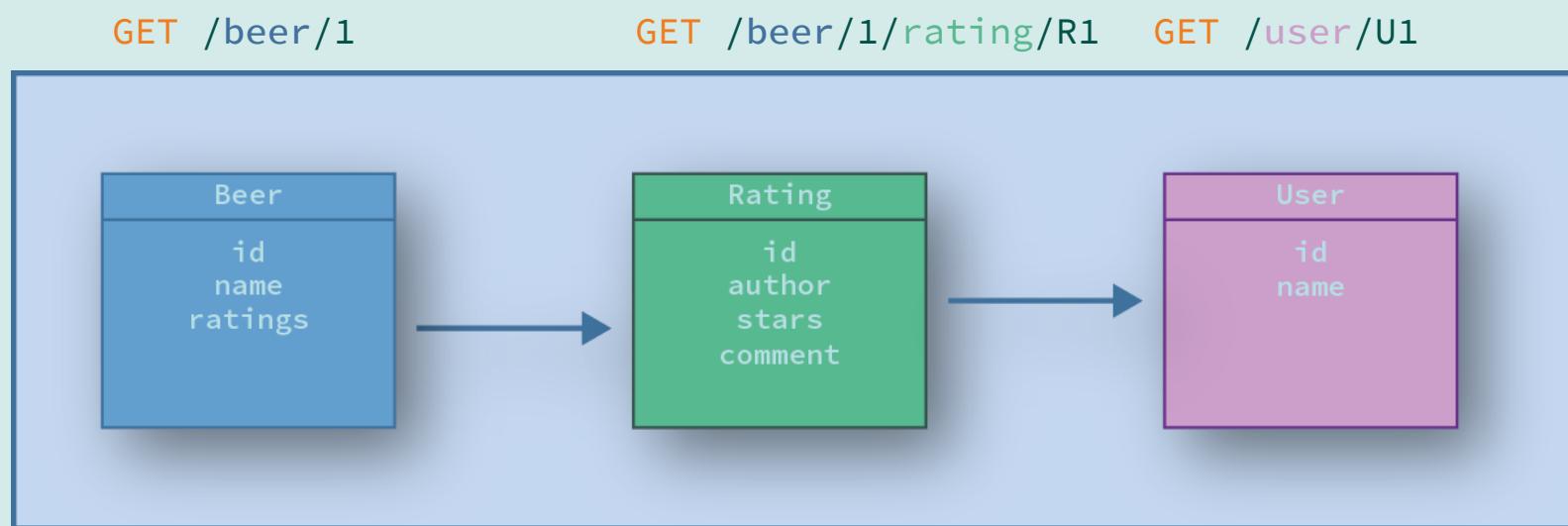
```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

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

# ABFRAGEN MIT REST

## REST-Zugriff

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



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

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

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

## ABFRAGEN MIT REST

### REST-Zugriff

- Pro Entität (Resource) eine Abfrage
- Zurückgeliefert wird immer komplette Resource

## REST-Zugriff

- Pro Entität (Resource) eine Abfrage
- Zurückgeliefert wird immer komplette Resource
- Server kennt seine Client nicht
- Es können neue Clients implementiert werden, aber:
  - Eventuell viele Round-trips mit zu vielen oder zu wenig Daten
  - Keine Gesamt-Sicht auf Domaine (diverse Endpunkte)

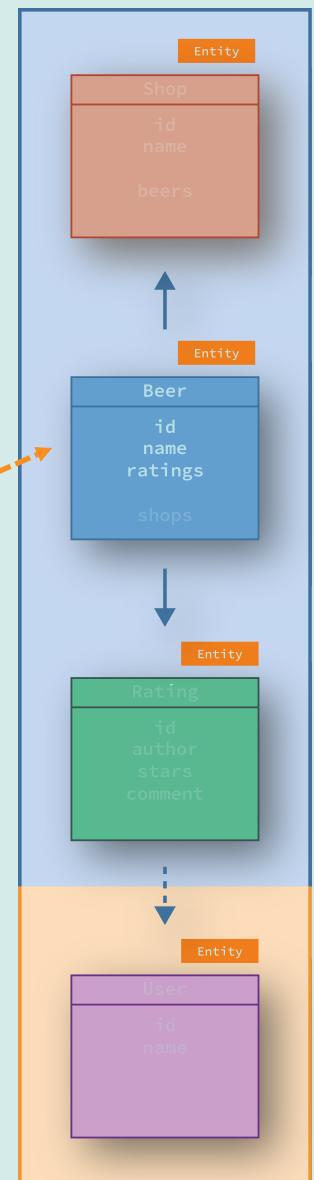
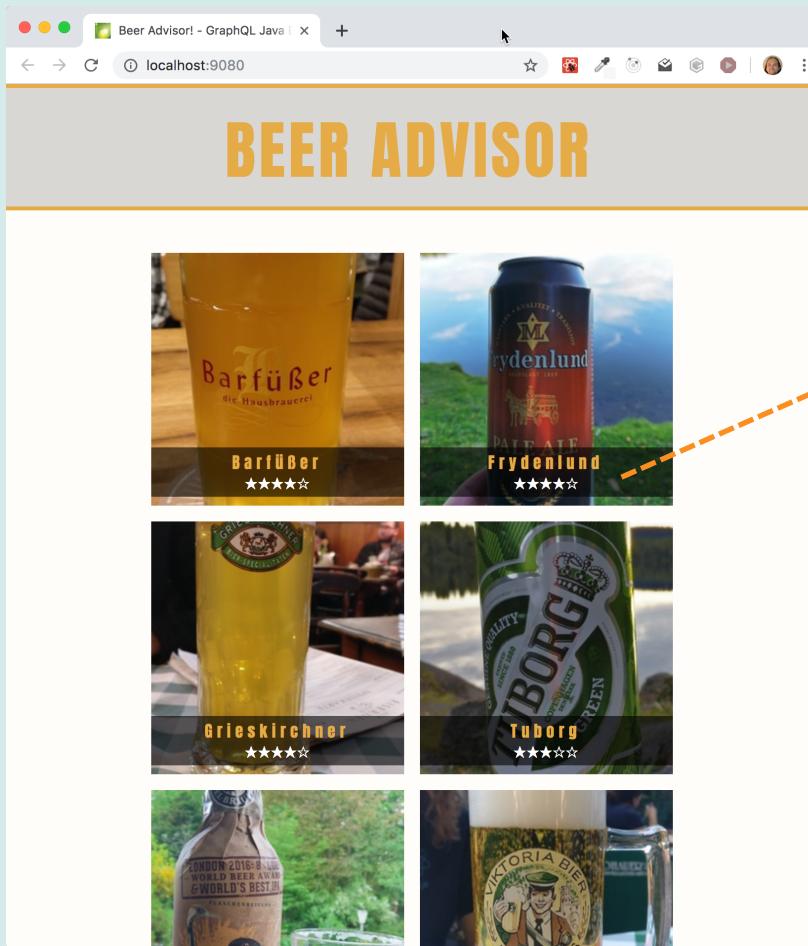
## Alternative: Client-getrieben?

- Ein Endpunkt pro Ansicht
- Client enthält genau die Daten, der braucht, aber:
  - Für jede Änderung (neues Feature, neues UI Design z.B.) muss der Server angepasst werden
  - ebenso für neue Clients

# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 1

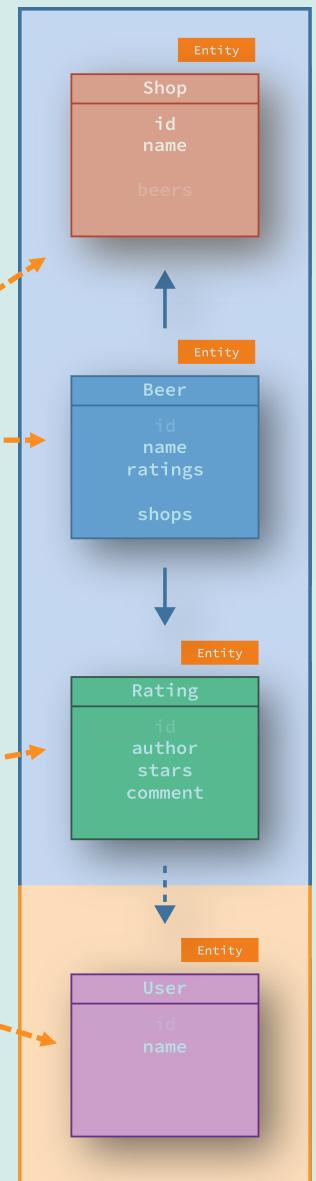
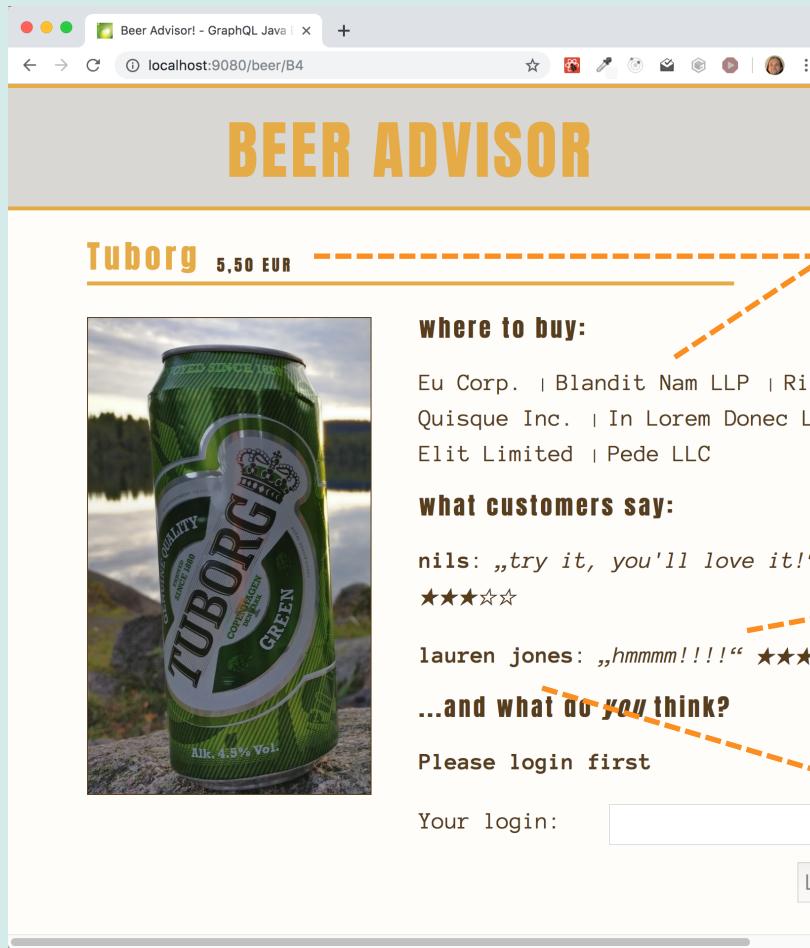
```
{ beer {  
    id  
    name  
    averageStars  
}
```



# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 2

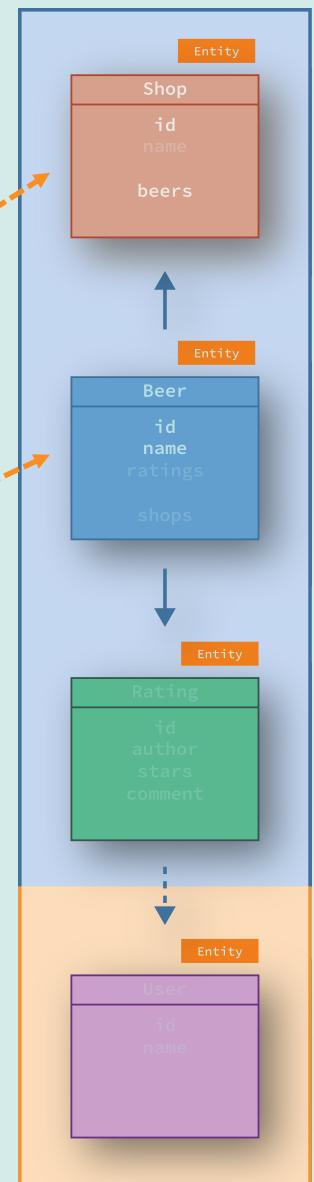
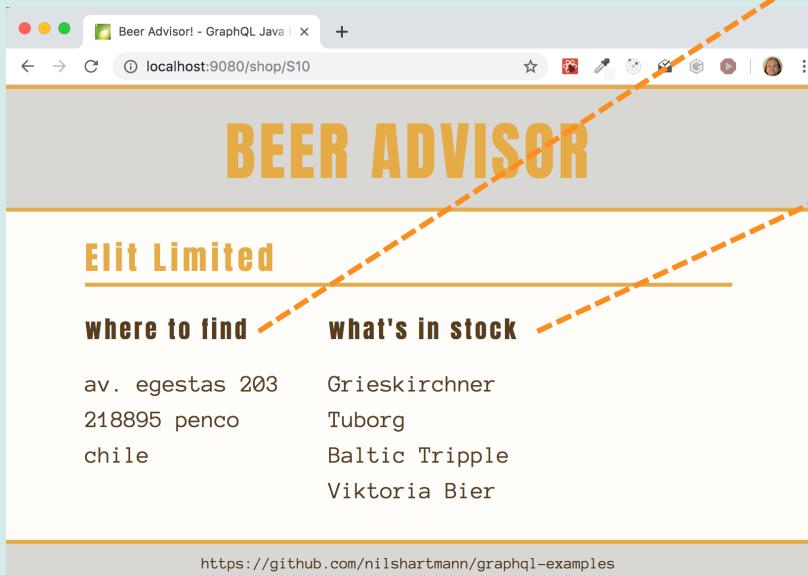
```
{ beer(beerId: "B1" {  
    name  
    price  
    ratings {  
        stars  
        comment  
        author {  
            name  
        }  
    }  
    shops { name }  
}
```



# GRAPHQL EINSATZSzenarien

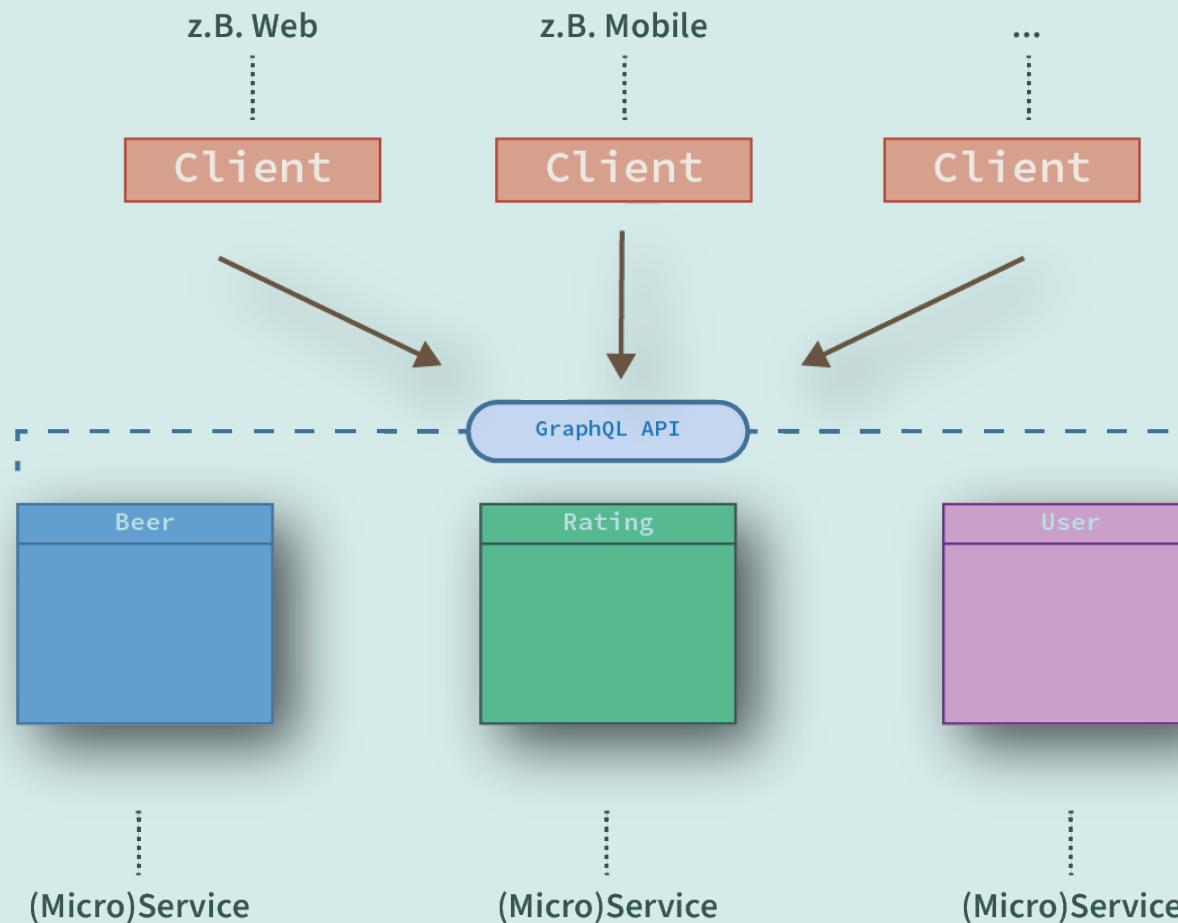
## Use-Case spezifische Abfragen – 3

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



# DATEN QUELLEN

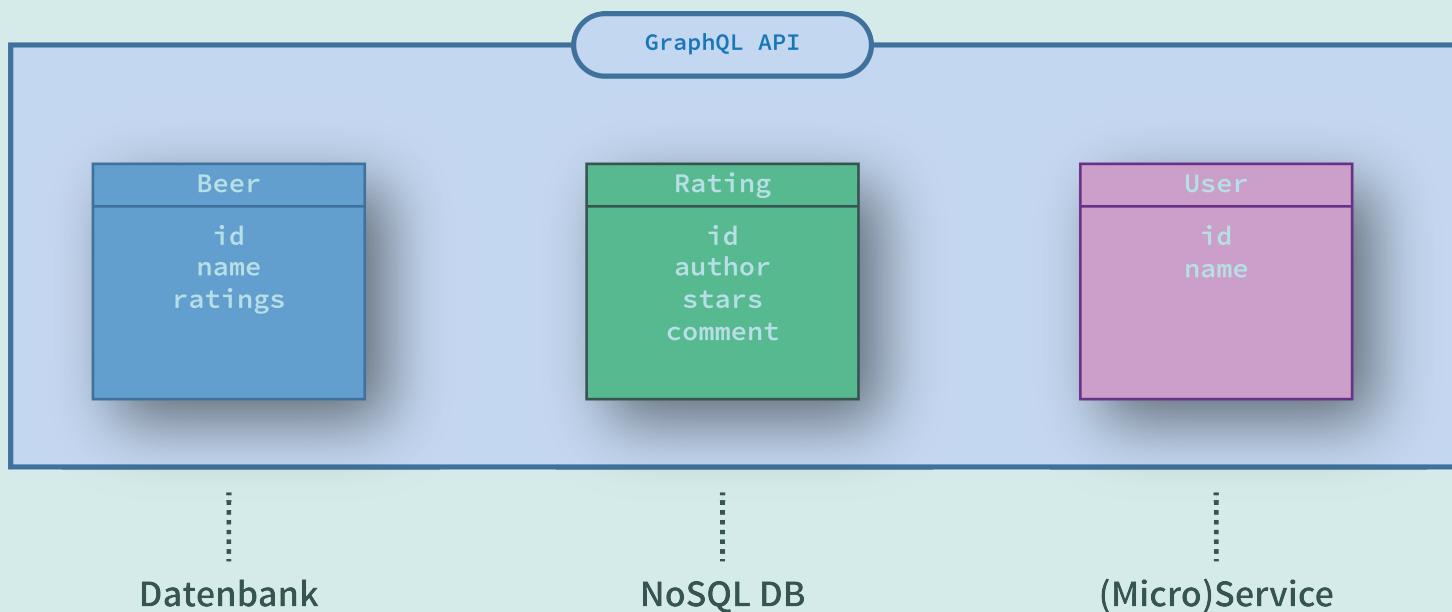
## GraphQL: Architekturbild



# 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!



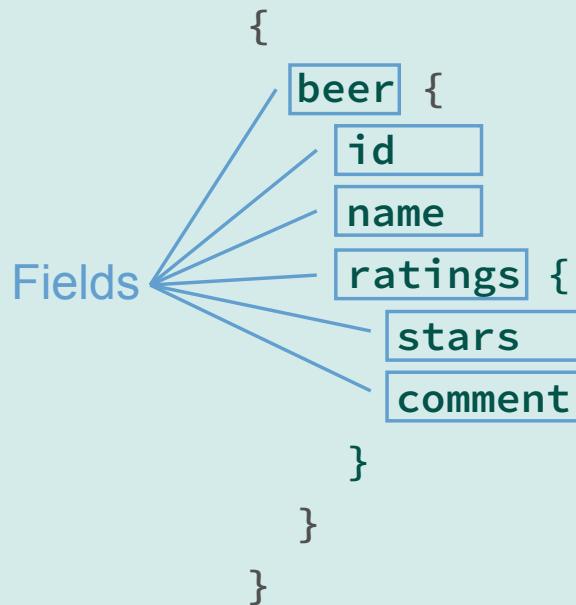
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

# GraphQL

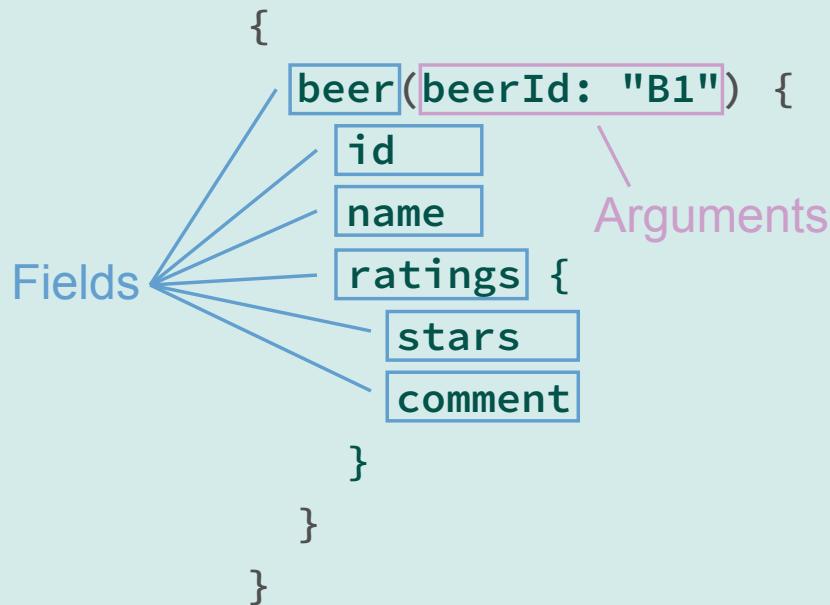
**TEIL 1: ABFRAGEN UND SCHEMA**

# 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  
    }  
  }  
}
```



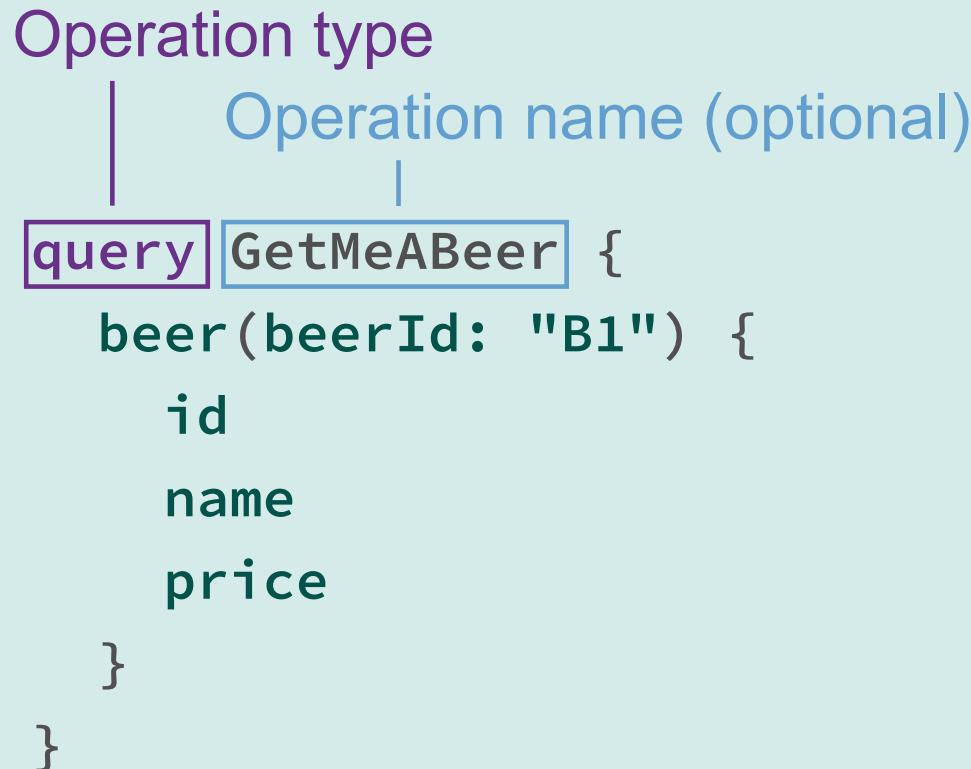
```
"data": {  
  "beer": {  
    "id": "B1"  
    "name": "Barfüßer"  
    "ratings": [  
      {  
        "stars": 3,  
        "comment": "grate taste"  
      },  
      {  
        "stars": 5,  
        "comment": "best beer ever!"  
      }  
    ]  
  }  
}
```

- 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: 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

      }

    }

# 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



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>

## GRAPHQL SCHEMA

### Caching "schwierig"

- Requests werden per POST ausgeführt
- Sehr viele unterschiedliche Query(-Strings) denkbar

### Caching "schwierig"

- Requests werden per POST ausgeführt
- Sehr viele unterschiedliche Query(-Strings) denkbar
- Aber...
  - Wer braucht überhaupt Caching? 😊
  - Cache auf DB-Ebene, -App-Ebene, natürlich möglich
  - Framework-spezifische Lösungen existieren

# 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, Mutation, Subscription)

Root-Type  
("Query")

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

Root-Fields

Root-Type  
("Mutation")

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

Root-Type  
("Subscription")

```
type Subscription {  
    onNewRating: Rating!  
}
```

## 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  
}
```

```
frontend — ShopPage.tsx — https://nilshartmann.net
ShopPage.tsx
BeerPage.tsx
Shop.tsx
ShopPageQuery.ts
OverviewPageQuery.ts
...
FRONTEND
> querytypes
> BeerPage
> querytypes
ts AddRating.tsx
ts AddRatingForm.tsx
ts Beer.tsx
Beer.module.scss
ts BeerPage.tsx
Form.module.scss
ts index.tsx
ts LoginForm.tsx
ts Rating.tsx
ts UpdateBeer.tsx
BeerRatingApp
ts BeerRatingApp.tsx
BeerRatingApp.module.scss
ts Footer.tsx
Footer.module.scss
ts Header.tsx
Header.module.scss
ts index.tsx
components
> OverviewPage
> querytypes
ts OverviewPageQuery.ts
ts index.tsx
ts OverviewPage.tsx
OverviewPage.module.scss
> querytypes
> ShopPage
> querytypes
ts ShopPageQuery.ts
ts index.tsx
ts Shop.tsx
ts ShopPage.tsx
ShopPage.module.scss
AuthContext.tsx
ts createApolloClient.tsx
...
OUTLINE
TIMELINE
NPM SCRIPTS
SSH FILE SYSTEMS
MAVEN PROJECTS
SPRING-BOOT DASHBOARD
master* 0 0 0 Live Share
Ln 49, Col 14  Spaces: 2  UTF-8  LF  TypeScript React  http://localhost:59023/  3.9.4  Go Live  Apollo 3.7  Prettier: ✓  Formatting: ✓  ⌂  ⌂
```

```
1 import { QueryResult } from "@apollo/react-common";
2 import { useQuery } from "@apollo/react-hooks";
3 import gql from "graphql-tag";
4 import * as React from "react";
5 import { RouteComponentProps } from "react-router";
6 import { ShopPageQuery, ShopPageQueryVariables } from "./querytypes/ShopPageQuery";
7 import Shop from "./Shop";
8
9 const SHOP_PAGE_QUERY = gql`query ShopPageQuery($shopId: ID!) {
10   shop(shopId: $shopId) {
11     id
12     name
13
14     address {
15       street
16       postalCode
17       city
18       country
19     }
20     beers {
21       id
22       name
23     }
24   }
25 }
26 `;
27
28 type ShopPageProps = RouteComponentProps<{ shopId: string }>;
29 type ShopPageQueryResult = QueryResult<ShopPageQuery, ShopPageQueryVariables>;
30
31 export default function ShopPage({ match, history }: ShopPageProps) {
32   const { data, loading, error } = useQuery(SHOP_PAGE_QUERY, {
33     variables: { shopId: match.params.shopId },
34   });
35
36   if (loading) {
37     return <h1>Please wait... Shop is loading ...</h1>;
38   }
39 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Local: http://localhost:9080  
On Your Network: http://192.168.178.54:9080

Note that the development build is not optimized.  
To create a production build, use `yarn build`.

ShopPage.tsx  
openingHours: String!

# Ende-zu-Ende Typsicherheit

Beispiel: VS Code

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

- <https://graphql.org>

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

*"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!  
}
```

## 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):

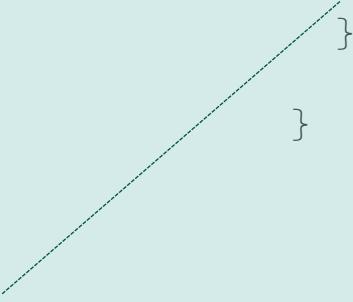
```
interface DataFetcher<T> {  
    T get(DataFetchingEnvironment environment);  
}
```

# 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

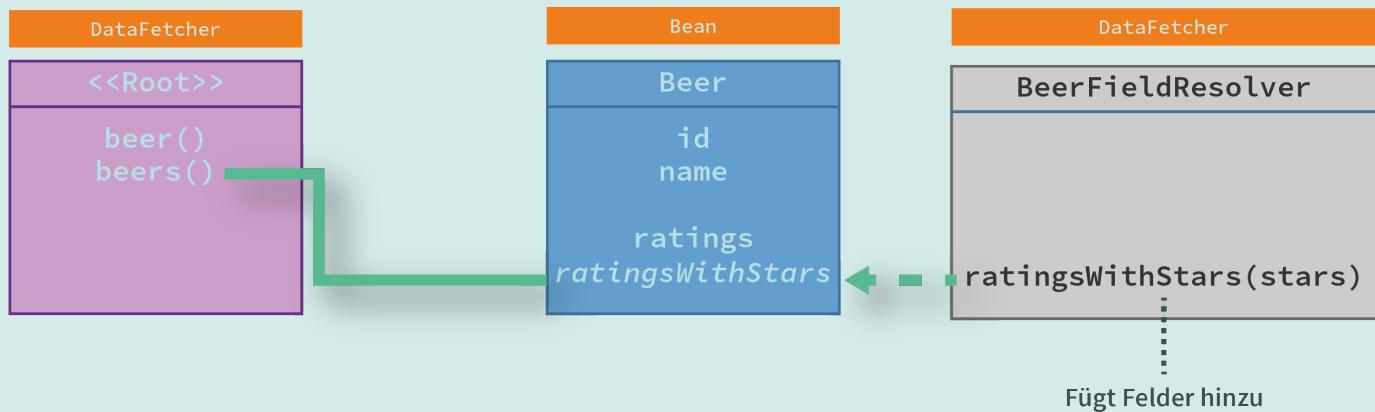
```
public DataFetcher<Rating> addRatingMutationFetcher() {  
    return environment -> {  
        final Map<String, Object> ri =  
            environment.getArgument("ratingInput");  
  
        type Mutation {  
            addRating  
            (ratingInput: AddRatingInput):  
                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);  
    };  
}
```

# 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*

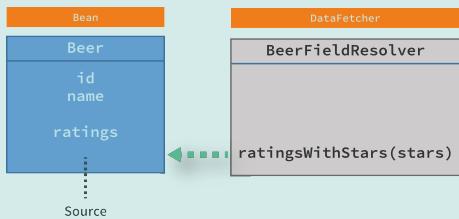
```
{  
  beers {  
    ratingsWithStars  
    (stars: 3) {  
      comment  
    }  
  }  
}
```



# DATA FETCHER FÜR NICHT-ROOT-FELDER

## DataFetcher implementieren

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



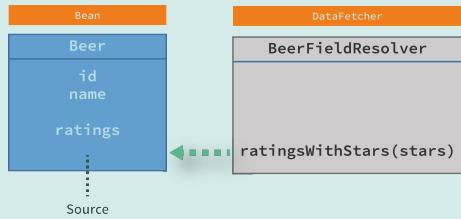
```
public class BeerDataFetchers {  
  
    public DataFetcher<List<Rating>> ratingsWithStarsFetcher() {  
        return environment -> {  
            Beer beer = environment.getSource();  
  
            return beer.ratingsWithStars(environment.getArgument("stars"));  
        };  
    };  
};
```

```
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;

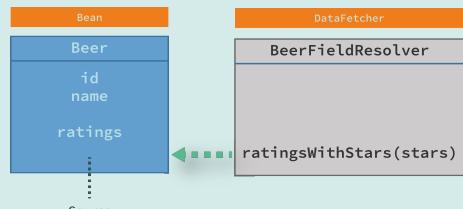
    public Rating addRating(AddRatingInput newRating) {
        Rating rating = Rating.from(newRating);
        ratingRepository.save(rating);
        return rating;
    }
}

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

# DATA FETCHER FÜR NICHT-ROOT-FELDER

## DataFetcher implementieren

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



```
public class BeerFieldResolver implements  
GraphQLResolver<Beer> {  
  
    public List<Rating> ratingsWithStars(Beer beer, int stars) {  
        return beer.getRatings().stream()  
            .filter(r -> r.getStars() == starsInput)  
            .collect(Collectors.toList());  
    }  
}
```

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

## Validierungen

- Beim Start: Alle Resolver müssen vorhanden sein
  - Return-Types und Methoden-Parameter der Resolver-Funktionen müssen zum Schema passen
- Zur Laufzeit: Resolver werden immer mit korrekten Parametern aufgerufen
  - Argumente haben korrekten Typ
  - Argumente sind ggf. nicht null
- 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>

## **IMPLEMENTIERUNG**

### **Typische Probleme bei der Implementierung**

# 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 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;  
    }  
}
```

## EXKURS: OPTIMIERUNGEN

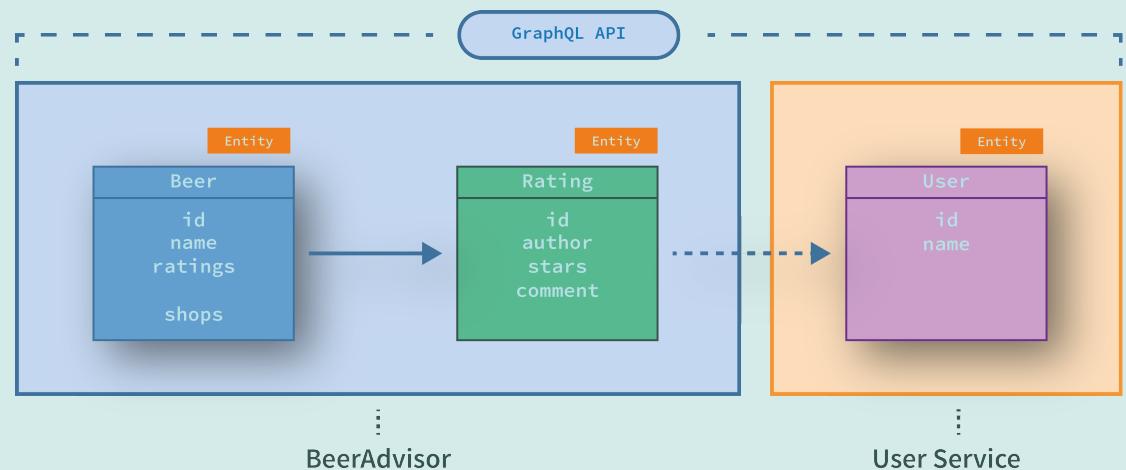
**Achtung! *Optimierungen immer Use-Case-spezifisch***

# DATA LOADER



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

```
{ beer (id: 3) {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}
```



# DATA LOADER

## Beispiel: Zugriff auf (Remote-)Services

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

```
{ beer (id: 3) {  
    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 (id: 3) {  
    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 (id: 3) {  
    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)

```
{ beer (id: 3) {  
    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

# DATA LOADER

## Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück  
(unverändert)

```
{ beer (id: 3) {  
    ratings {  
        comment  
        author {  
            name  
        }  
    }  
}
```

# DATA LOADER

## 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 (id: 3) {  
    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

# DATA LOADER

## 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 (id: 3) {  
    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

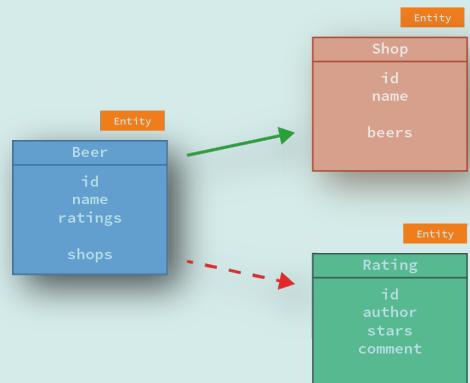
```
public BatchLoader userBatchLoader = new BatchLoader<String, User>() {  
  
    public CompletableFuture<List<User>> load(List<String> userIds) {  
        return CompletableFuture.supplyAsync(() -> userService.findUsersWithId(userIds));  
    }  
  
};
```

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

# EXKURS: OPTIMIERUNGEN

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

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

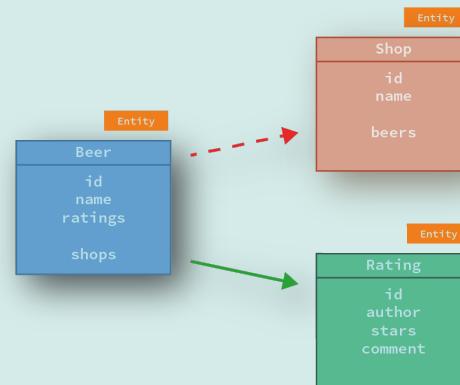
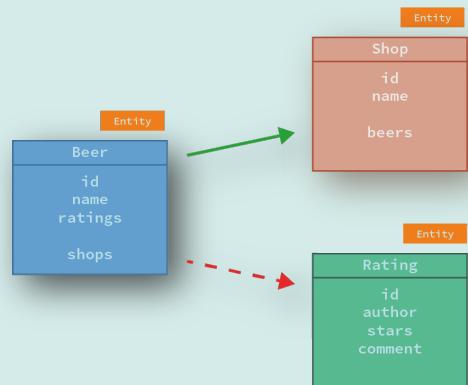


# EXKURS: OPTIMIERUNGEN

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

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

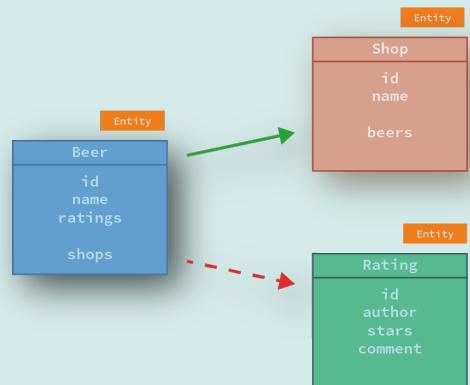
```
beers {  
    name  
    ratings {  
        comment  
    }  
}
```



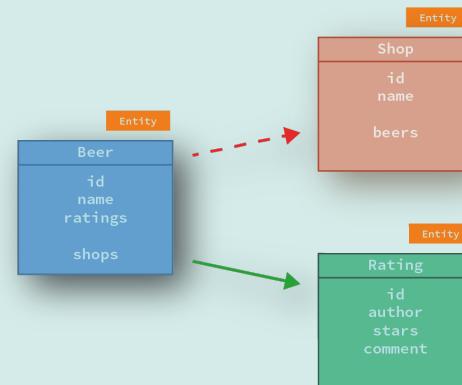
# EXKURS: OPTIMIERUNGEN

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

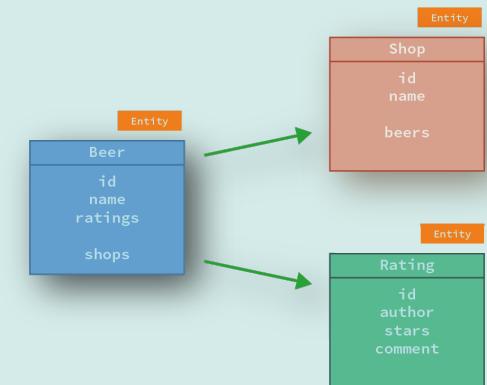
```
beers {  
    name  
    shops {  
        name  
    }  
}
```



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



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

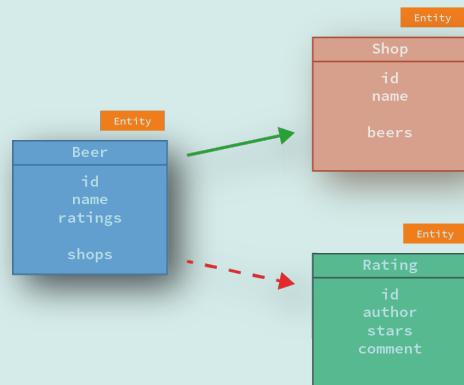


# EXKURS: OPTIMIERUNGEN

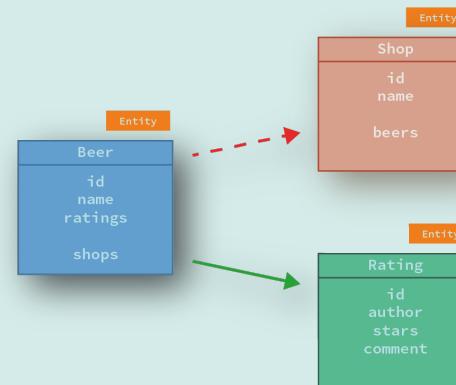
## Problem: optimaler Datenbankzugriff (Beispiel: JPA/JOINs)

- nur zur Laufzeit ermittelbar
- möglichst auf oberstem DataFetcher entscheiden

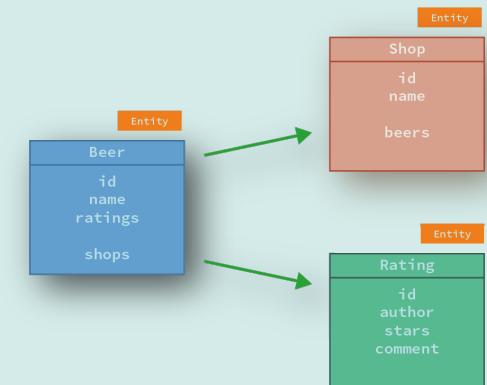
```
beers {  
    name  
    shops {  
        name  
    }  
}
```



```
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);  
    };  
}
```

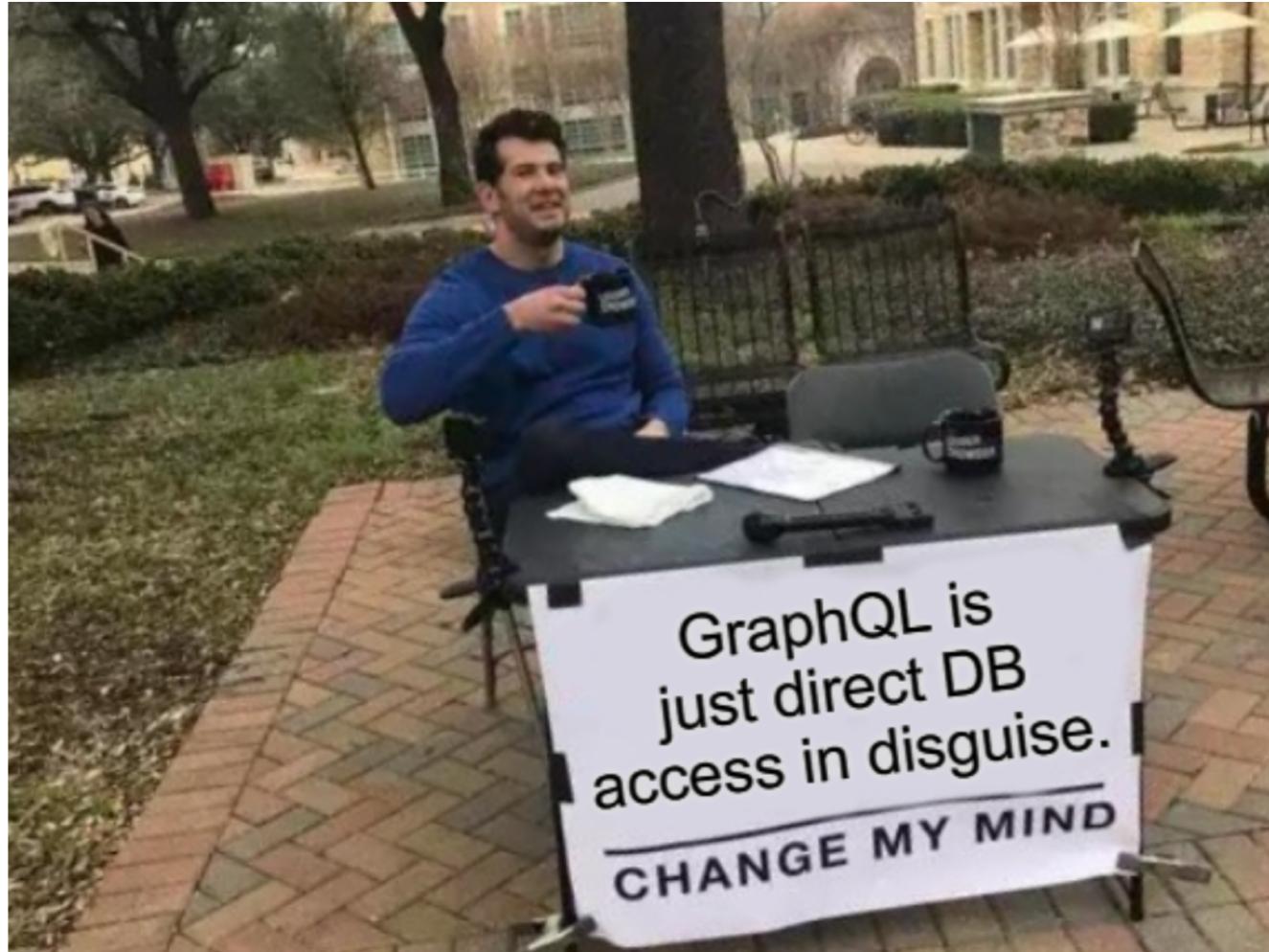
## GraphQL - Zusammenfassung



feststelltaste commented 10 days ago

Collaborator Author ...

Damit es hier nicht so leer ist: Hier mal ein Vorurteil, dass ich mal irgendwo wahrgenommen habe :-D



## 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
- **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://nils.buzz/tk-dev-talk-graphql>

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

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