

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

# GraphQL für Java

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

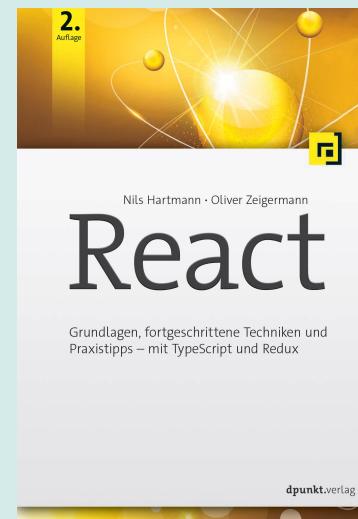
# NILS HARTMANN

nils@nilshartmann.net

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

Java  
JavaScript, TypeScript  
React  
GraphQL

Trainings & Workshops



<https://reactbuch.de>

**HTTPS://NILSHARTMANN.NET**

TEIL 1

# GraphQL

# Grundlagen

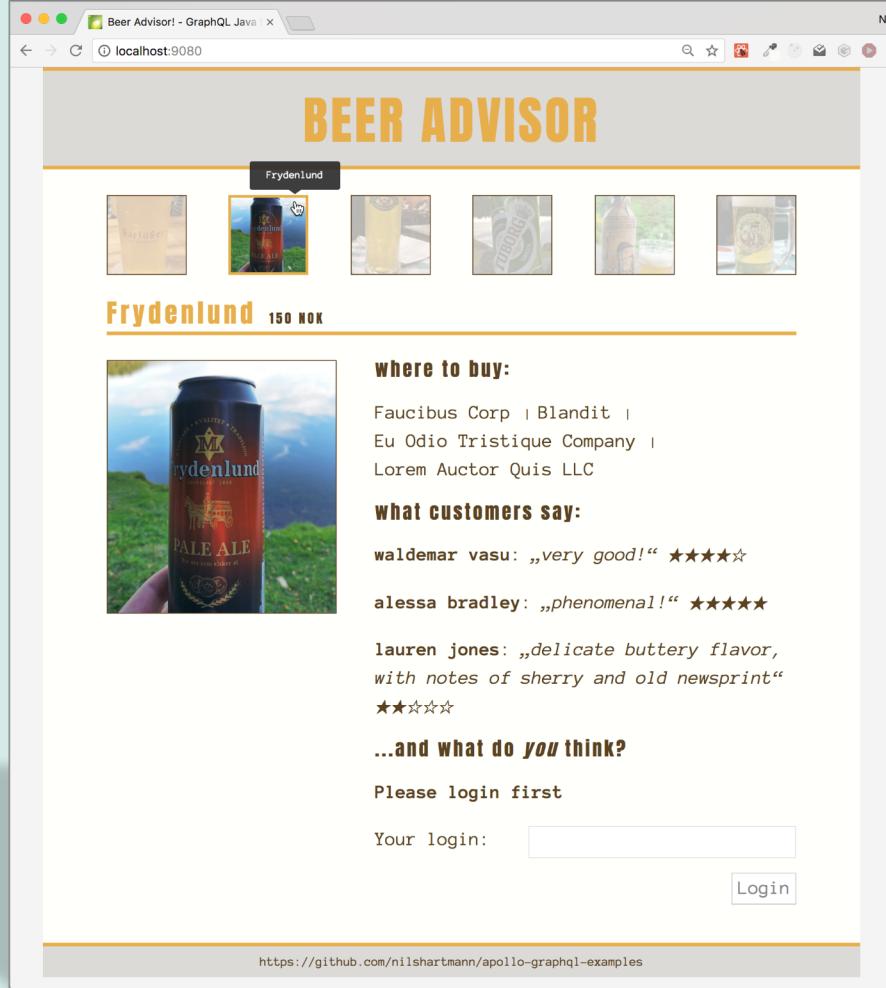
*"GraphQL is a **query language for APIs** and a **runtime for fulfilling those queries** with your existing data"*

- <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
- Kein fertiges Produkt



# Beispiel Anwendung

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

The screenshot shows the GraphiQL interface running on localhost:9000. On the left, the query editor contains a GraphQL query named 'BeerAppQuery':

```
query BeerAppQuery {  
  beers {  
    id  
    name  
    price  
    ratings {  
      id  
      beerId  
      author  
      comment  
    }  
  }  
}  
beers  
beer  
ratings  
ping  
__schema  
__type
```

A tooltip below the 'beers' field says: 'Returns all beers in our store'. The right panel displays the JSON response from the server:

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

The right panel also includes a schema browser with definitions for 'beers', 'beer', 'ratings', 'ping', and 'ProcessInfo'.

# Demo: GraphiQL

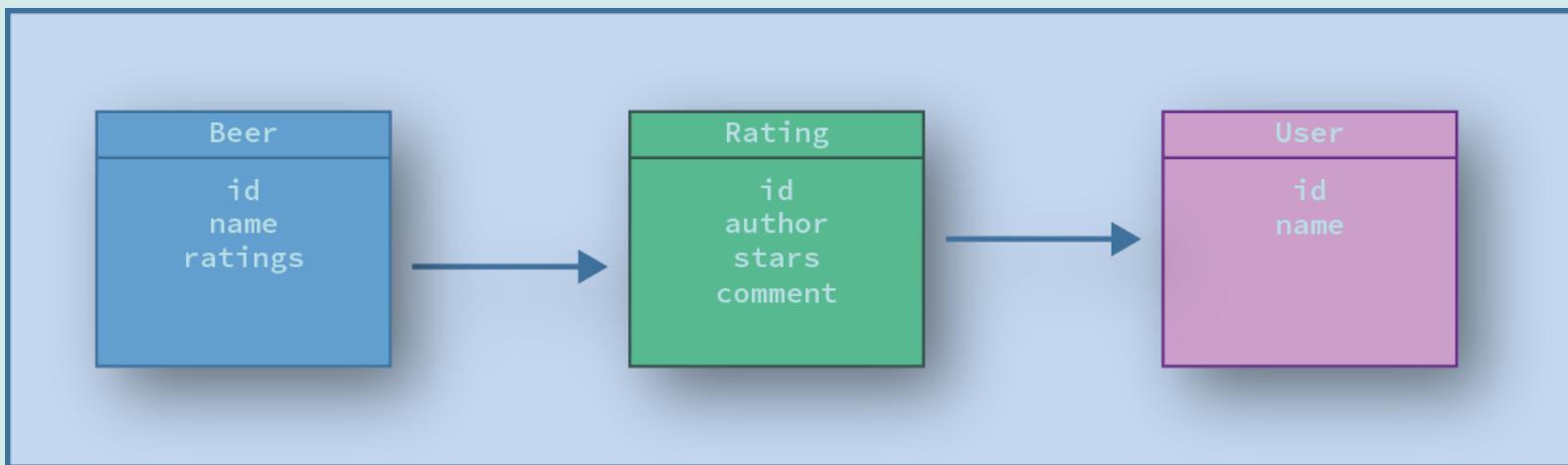
<https://github.com/graphql/graphiql>

<http://localhost:9000>

# **Vergleich mit REST**

# BEERADVISOR DOMAINE

## "Domain-Model"

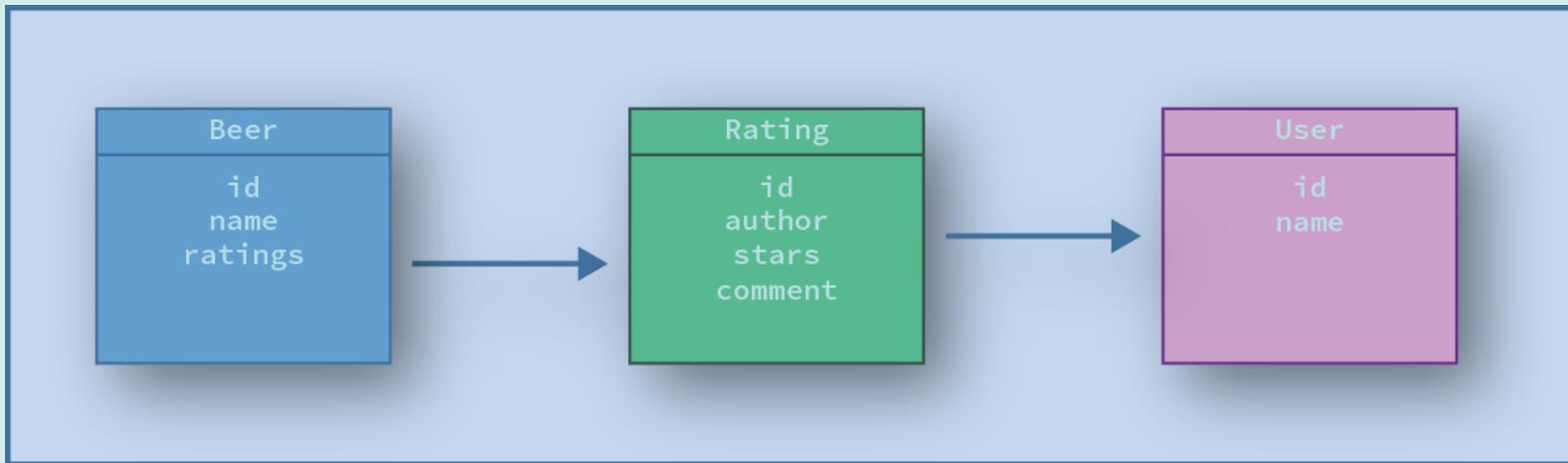


# ABFRAGEN MIT REST

## REST-Zugriff

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

GET /beer/1

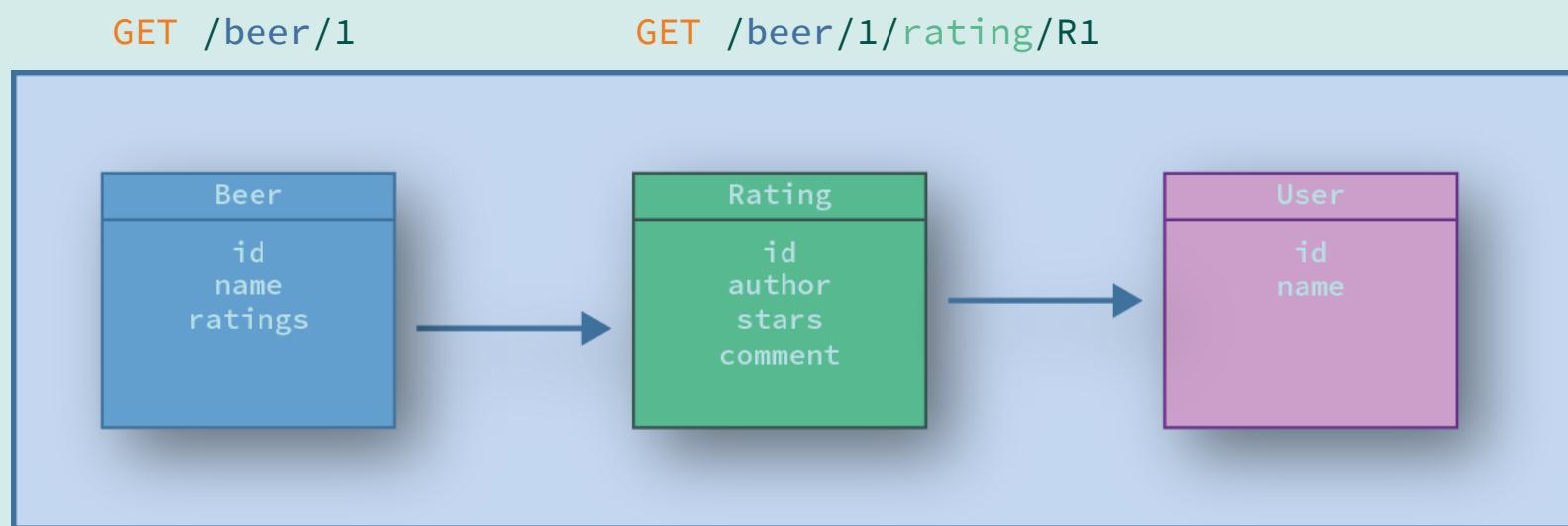


```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

# ABFRAGEN MIT REST

## REST-Zugriff

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



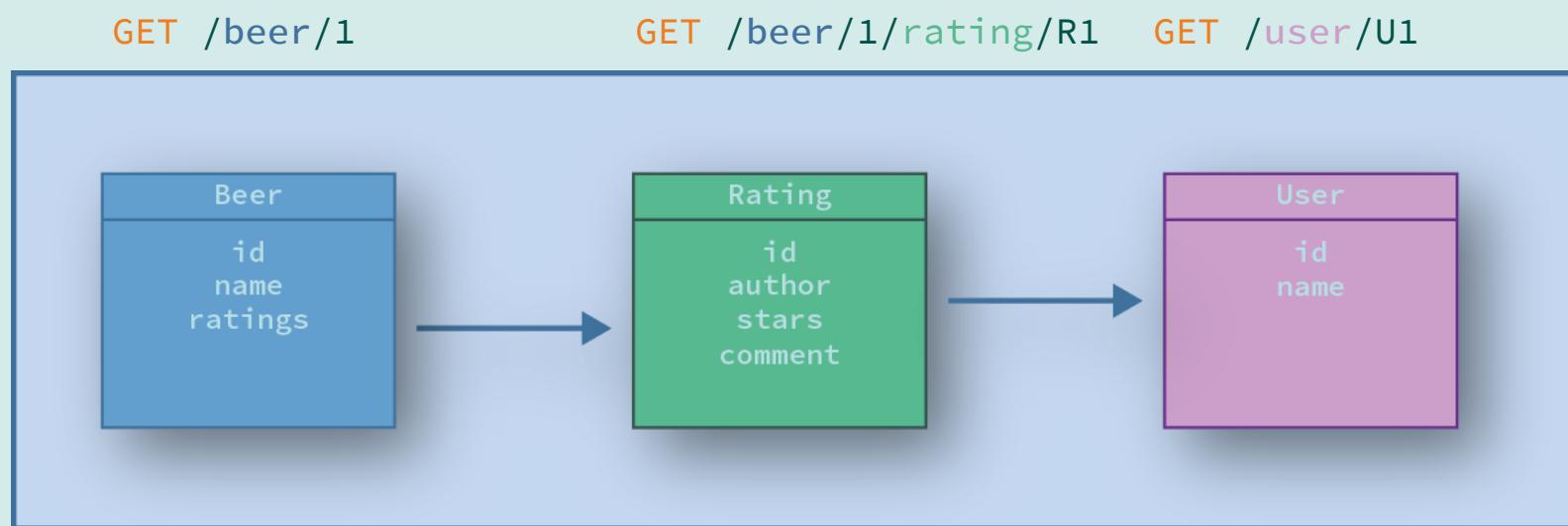
```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

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

# ABFRAGEN MIT REST

## REST-Zugriff

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



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

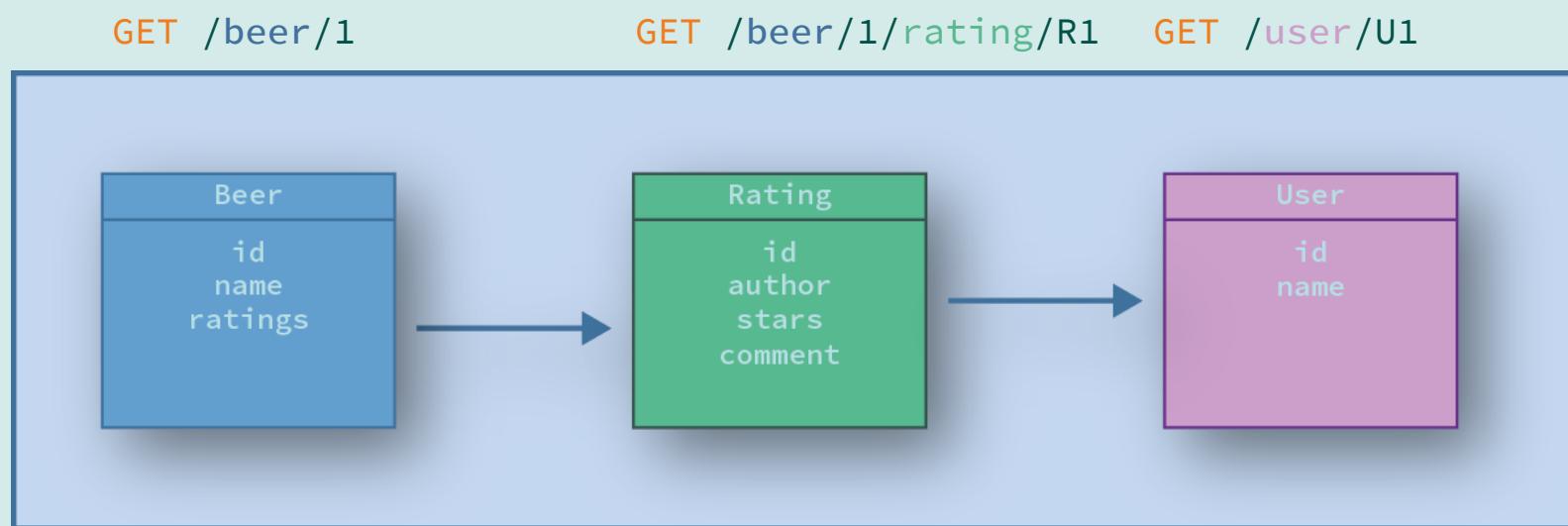
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

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

# ABFRAGEN MIT REST

## REST-Zugriff

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



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

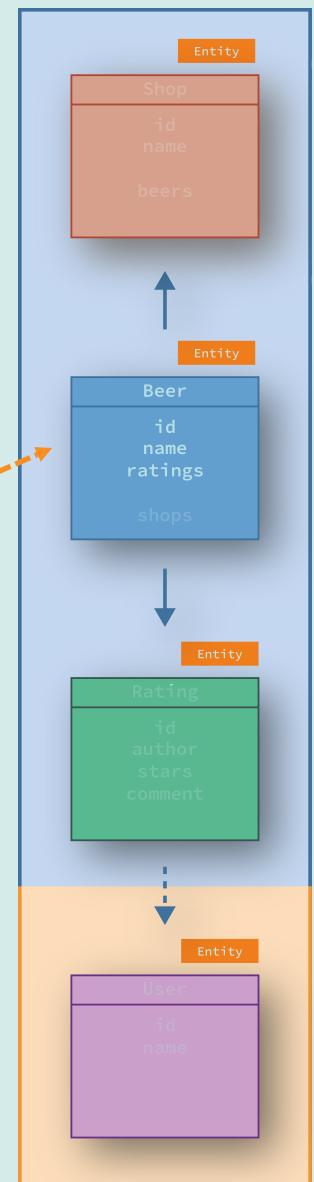
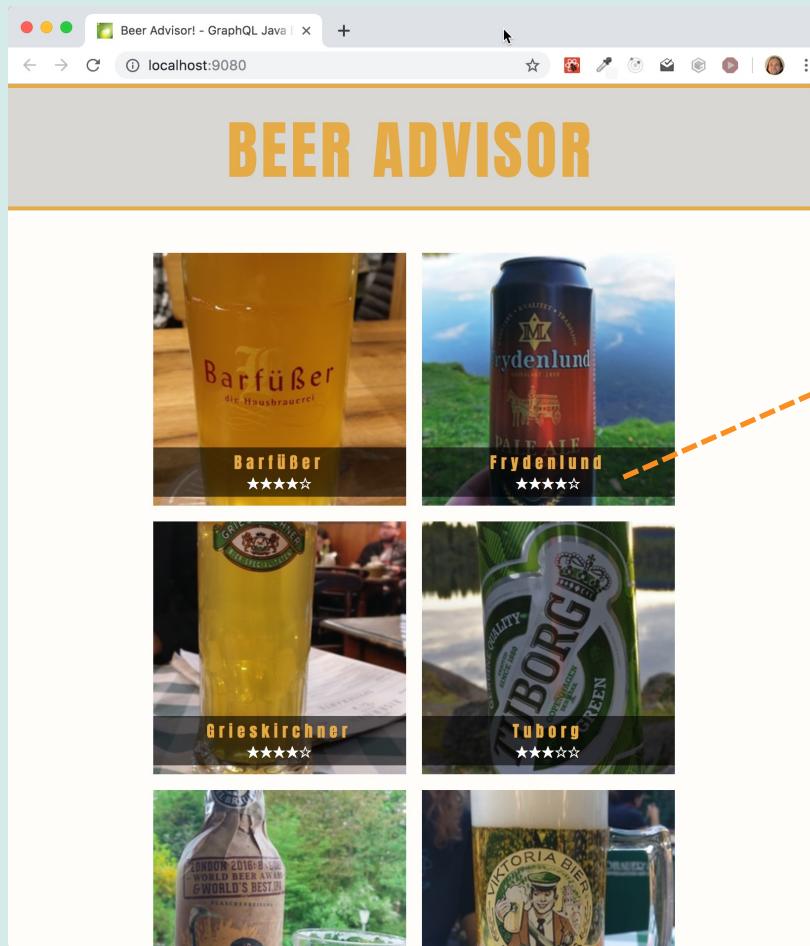
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

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

# GRAPHQL QUERIES

## Use-Case spezifische Abfragen – 1

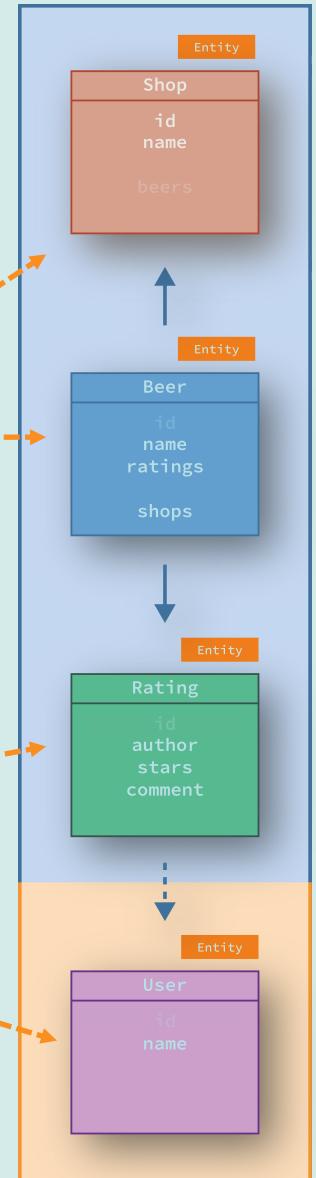
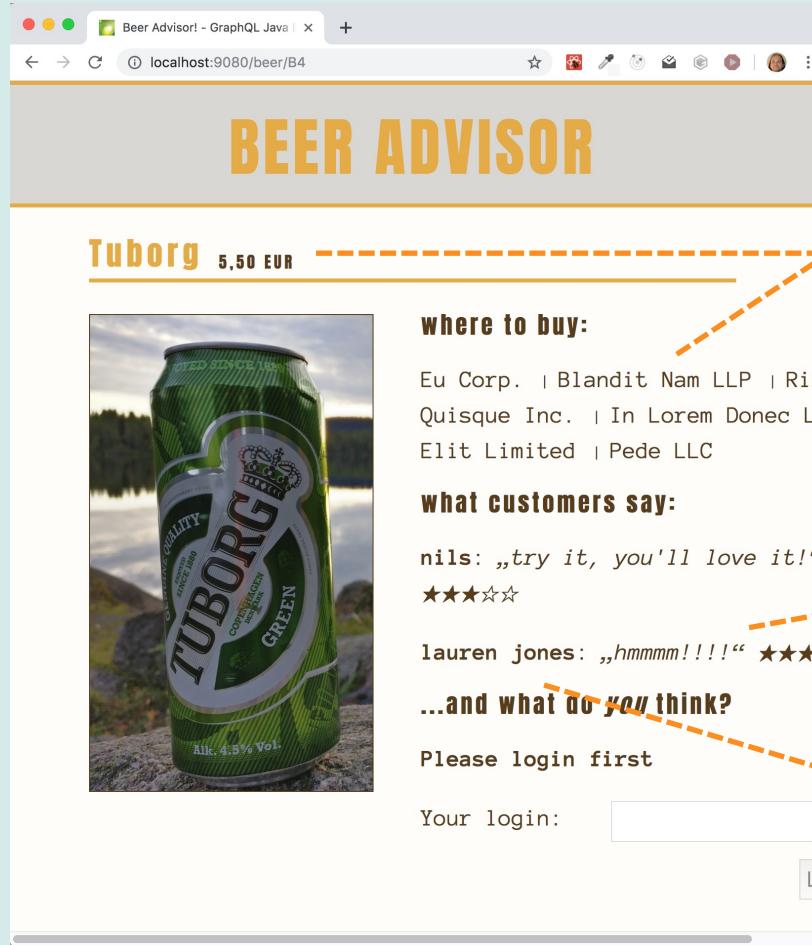
```
{ beer {  
    id  
    name  
    averageStars  
}
```



# GRAPHQL QUERIES

## Use-Case spezifische Abfragen – 2

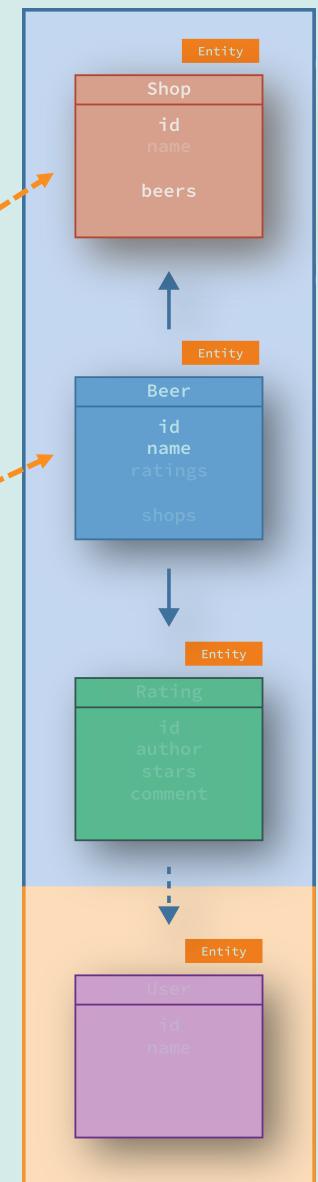
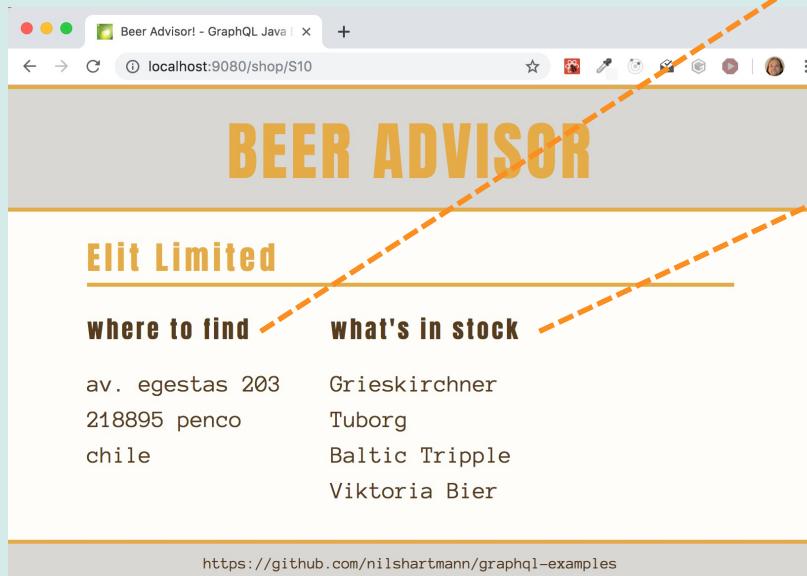
```
{ beer(beerId: "B1" {  
    name  
    price  
    ratings {  
        stars  
        comment  
        author {  
            name  
        }  
    }  
    shops { name }  
}
```



# GRAPHQL QUERIES

## Use-Case spezifische Abfragen – 3

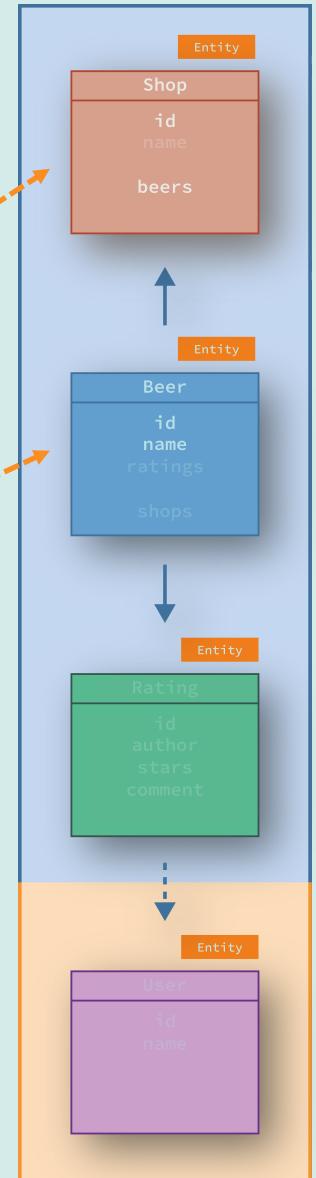
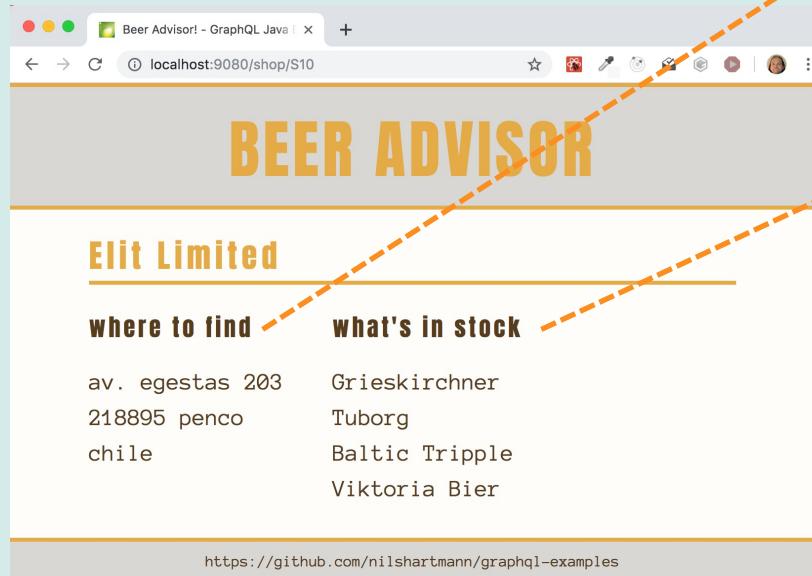
```
{ shop(shopId: "S3") {  
    name  
    address { street city }  
    beers { id name }  
}
```



# GRAPHQL QUERIES

## Use-Case spezifische Abfragen – 3

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



*Abgefragt werden Daten, nicht Endpunkte 😈*

### Wir veröffentlichen mit REST eine fachliche API

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

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

### Wir veröffentlichen mit GraphQL eine fachliche API

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

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

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

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

### Wir veröffentlichen mit GraphQL eine fachliche API

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

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

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

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

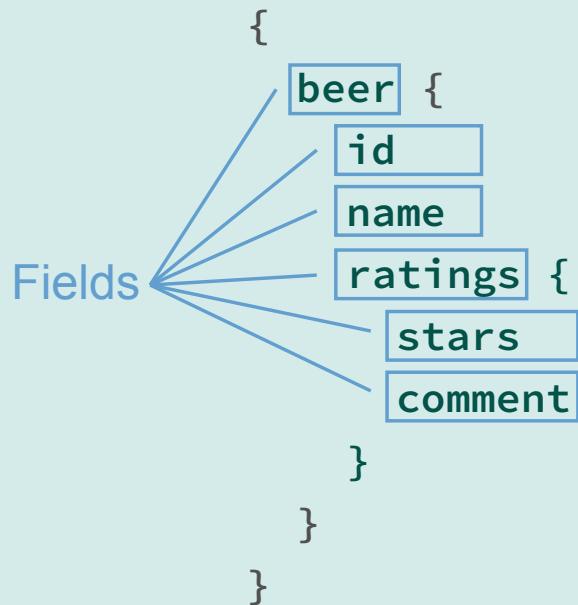
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

# GraphQL

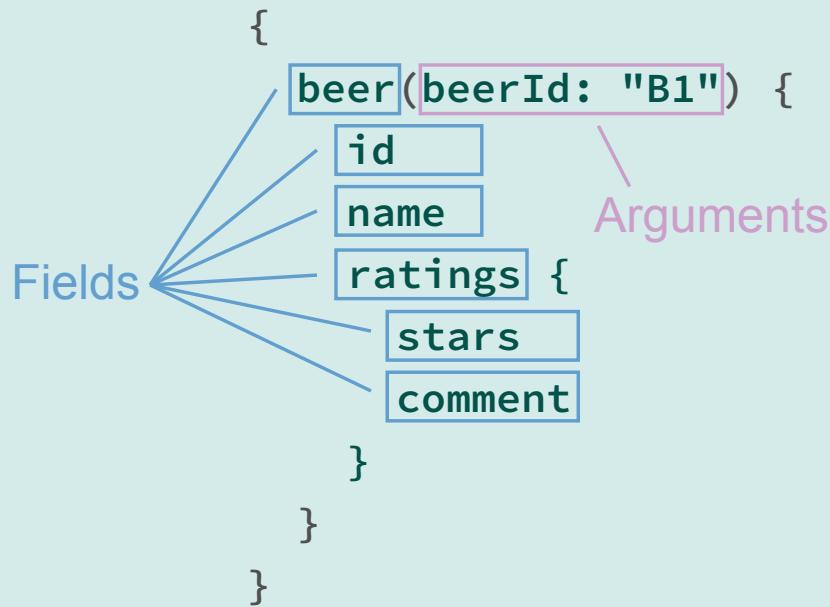
# Die GraphQL Query Sprache

# 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: 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

- „Normaler“ HTTP Endpunkt
  - Queries üblicherweise per POST
  - Ein *einzelner* Endpunkt, z.B. /graphql
  - HTTP Verben spielen keine Rolle

# QUERIES AUSFÜHREN

## Queries werden über HTTP ausgeführt

- „Normaler“ HTTP Endpunkt
  - Queries üblicherweise per POST
  - Ein *einzelner* Endpunkt, z.B. /graphql
  - HTTP Verben spielen keine Rolle
- Der GraphQL-Endpunkt kann parallel zu anderen Endpunkten bestehen
  - REST und GraphQL kann problemlos gemischt werden
- Wie die Anbindung aussieht hängt vom Framework und Umgebung (Spring / JEE) ab

TEIL II

# GraphQL Server

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

- <https://graphql.org>

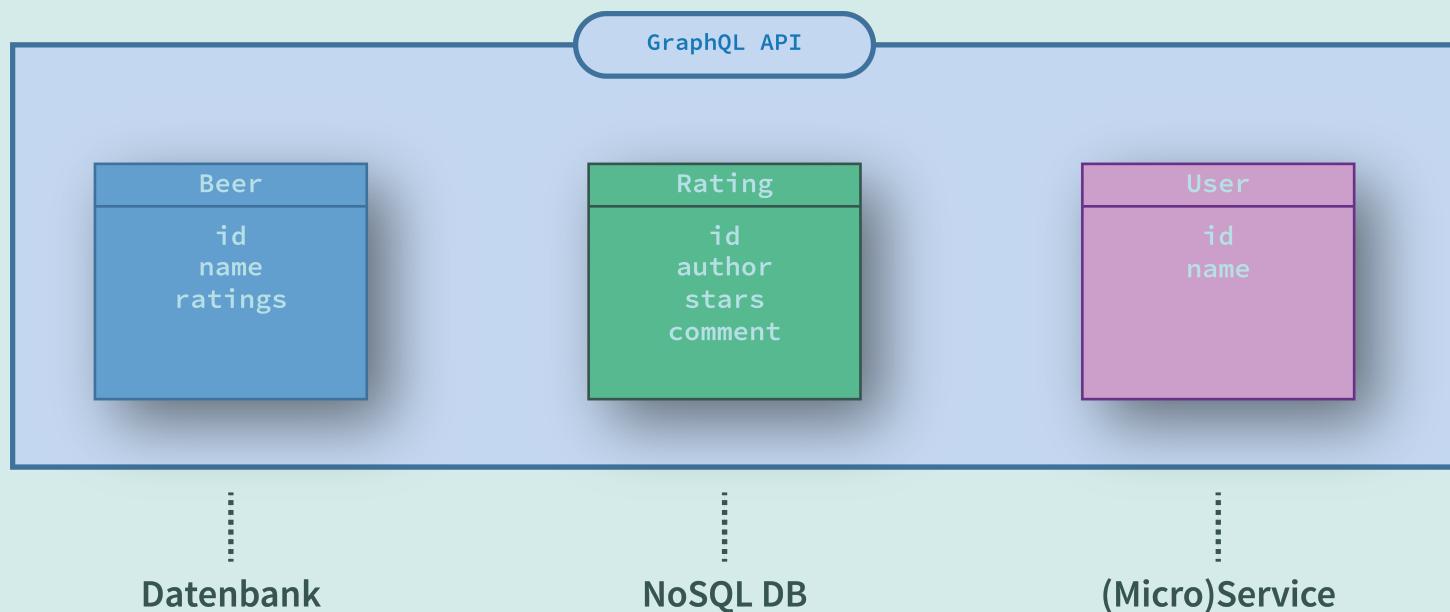
# GraphQL Server

**RUNTIME (AKA: YOUR APPLICATION)**

# GRAPHQL APIs

## GraphQL macht keine Aussage, wo die Daten herkommen

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



# GRAPHQL FÜR JAVA ANWENDUNGEN

## Grundsätzliche Optionen

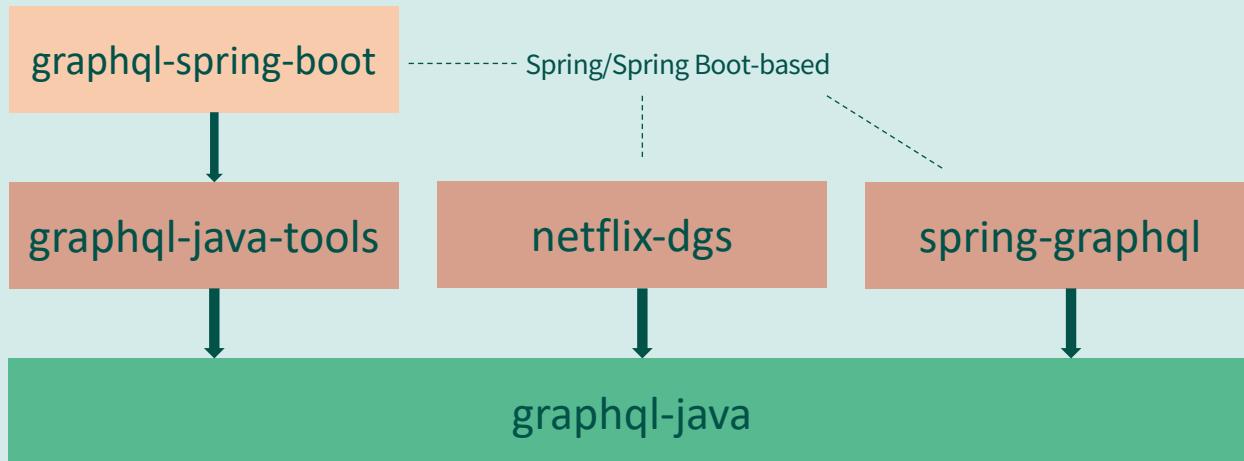
- **graphl-java**: das erste GraphQL Framework für Java (2015!)
  - Low level API, kein Support für Server etc.

graphql-java

# GRAPHQL FÜR JAVA ANWENDUNGEN

## Grundsätzliche Optionen

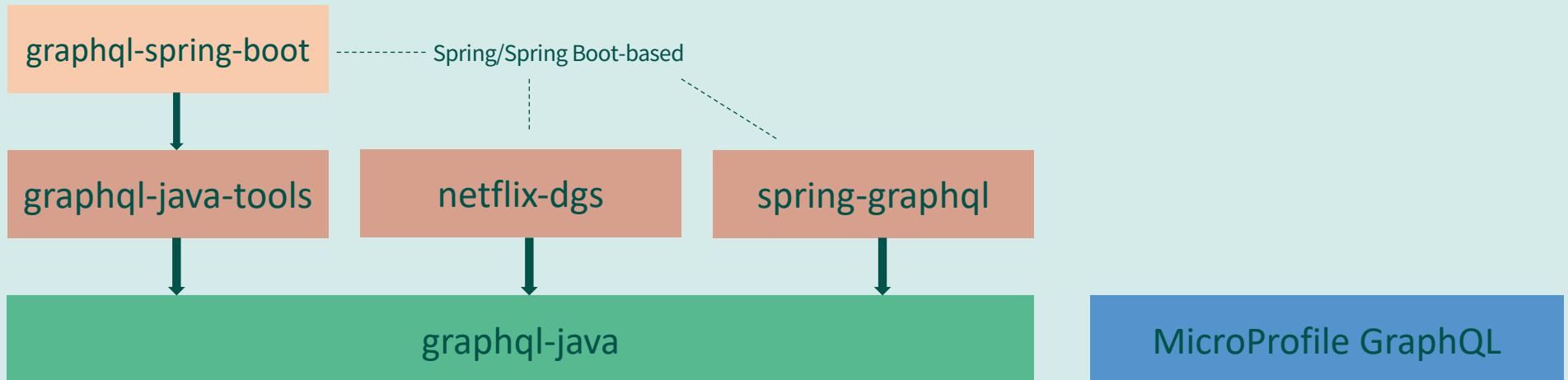
- **graphl-java**: das erste GraphQL Framework für Java (2015!)
  - Low level API, kein Support für Server etc.
  - Verschiede Abstraktionen existieren dafür, insb. für Spring Boot



# GRAPHQL FÜR JAVA ANWENDUNGEN

## Grundsätzliche Optionen

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



# GRAPHQL FÜR JAVA ANWENDUNGEN

**graphql-java**

# GRAPHQL FÜR JAVA ANWENDUNGEN

graphql-java



jbellenger 20 days ago (18.10.2021)

...

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

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

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

# GRAPHQL FÜR JAVA ANWENDUNGEN

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

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

# GRAPHQL SERVER MIT GRAPHQL-JAVA

## Unsere Aufgaben in der Entwicklung

### 1. Das Schema der API festlegen

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

# GRAPHQL SERVER MIT GRAPHQL-JAVA

## Unsere Aufgaben in der Entwicklung

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

## GRAPHQL SCHEMA

Die GraphQL API muss in einem *Schema* beschrieben werden

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

# GRAPHQL SCHEMA

## Schema Definition per SDL

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")	<pre>type Query {     beers: [Beer!]!     beer(beerId: ID!): Beer }</pre>	Root-Fields
Root-Type ("Mutation")	<pre>type Mutation {     addRating(newRating: NewRating): Rating! }</pre>	
Root-Type ("Subscription")	<pre>type Subscription {     onNewRating: Rating! }</pre>	

## 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  
}
```

## QUERY BEANTWORTEN: DATA FETCHER

### DataFetcher

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

## QUERY BEANTWORTEN: DATA FETCHER

### DataFetcher

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

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher

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

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

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher implementieren

- Beispiel: Ein einfaches Feld

Schema Definition

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

Query

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

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

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher implementieren

- Beispiel: Ein einfaches Feld

Schema Definition

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

Query

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

Data Fetcher

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

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher implementieren

- Beispiel: Ein einfaches Feld

Schema Definition

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

Query

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

Data Fetcher

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

# QUERY BEANTWORTEN: DATA FETCHER

## DataFetcher implementieren

- Beispiel: PropertyDataFetcher

Schema Definition

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

Query

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

Data Fetcher

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

```
class Beer {  
    private String name;  
    private double price;  
    ...  
  
    String getName() { ... }  
    double getPrice() { ... }  
}
```

# DATAFETCHER

## DataFetcher: Mutations

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

Schema Definition

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

Data Fetcher

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

# DATAFETCHER

## DataFetcher: Subscriptions

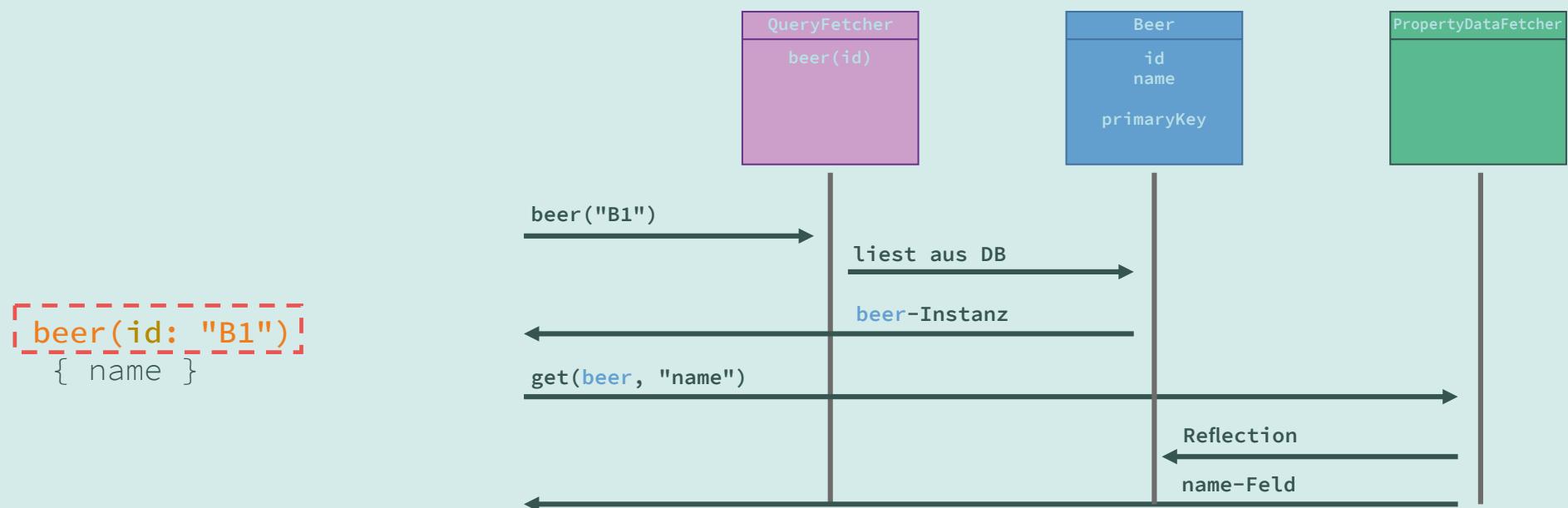
- Wie Queries, müssen aber Reactive Streams Publisher zurückliefern
- Kommunikation zum Web-Client in der Regel mit WebSockets

```
type Subscription {  
    onNewRating: Rating!  
}  
  
import org.reactivestreams.Publisher;  
  
public class SubscriptionDataFetchers {  
    DataFetcher<Publisher<Rating>> onNewRating = new DataFetcher<>() {  
        public Publisher<Rating> get(DataFetchingEnvironment env) {  
            Publisher<Rating> publisher = getRatingPublisher();  
            return publisher;  
        }  
    };  
}
```

# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

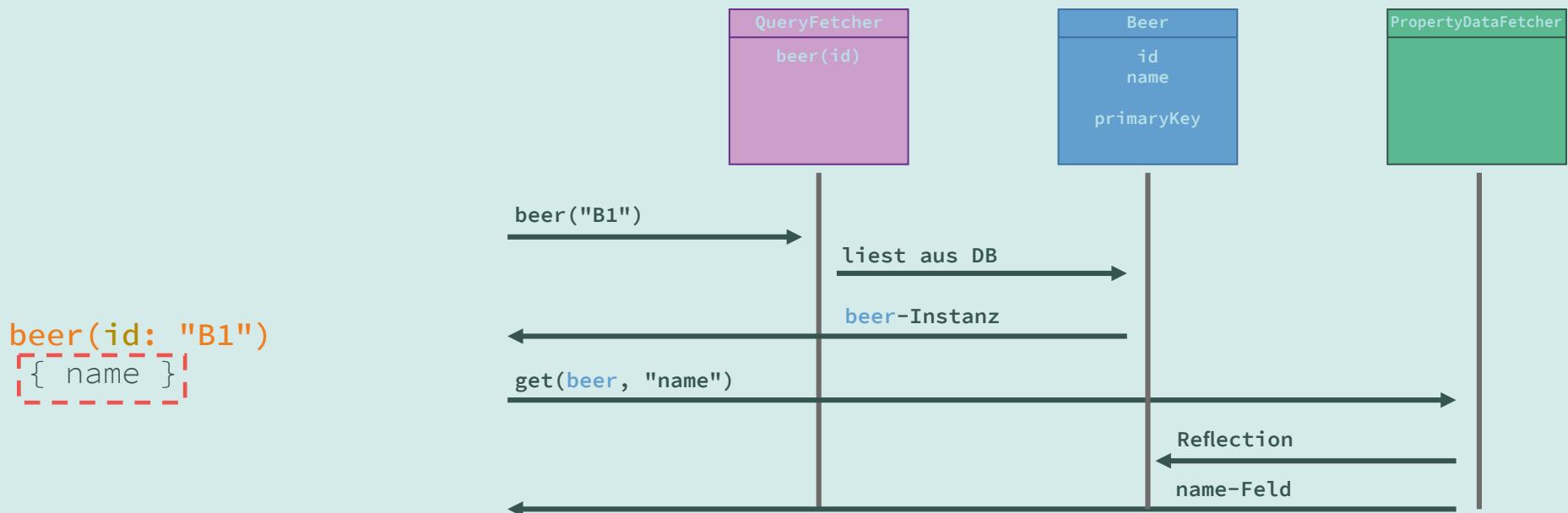
- Für Root-Felder müssen DataFetcher implementiert werden



# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

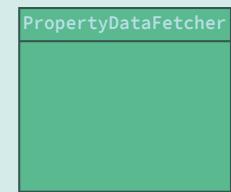
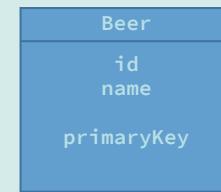
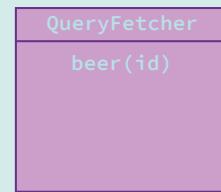
- Für Root-Felder müssen DataFetcher implementiert werden
- Für alle anderen Felder wird ein **PropertyDataFetcher** per Default verwendet
- Der PropertyDataFetcher verwendet Reflection



# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

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



```
beer(id: "B1")
{ name primaryKey }
```

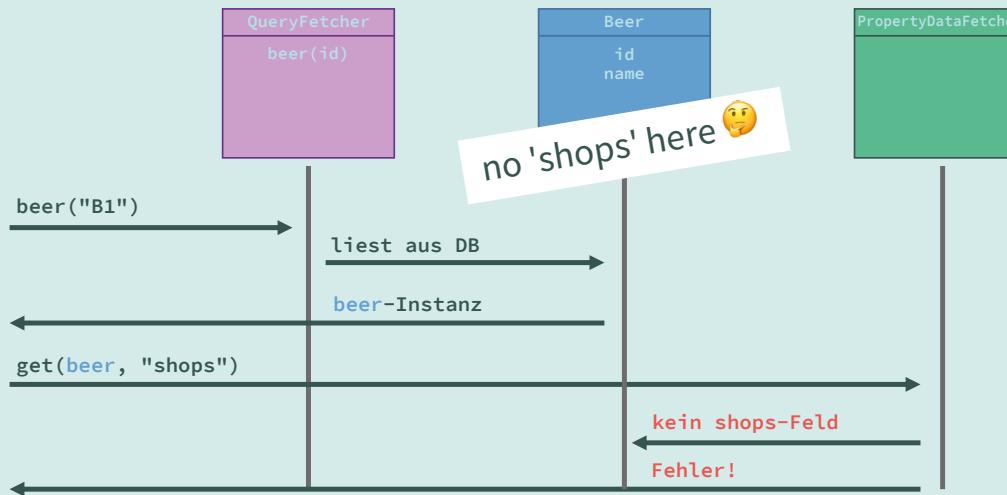
Ungültiger Query!

# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

- Problem: Feld existiert gar nicht am Pojo

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

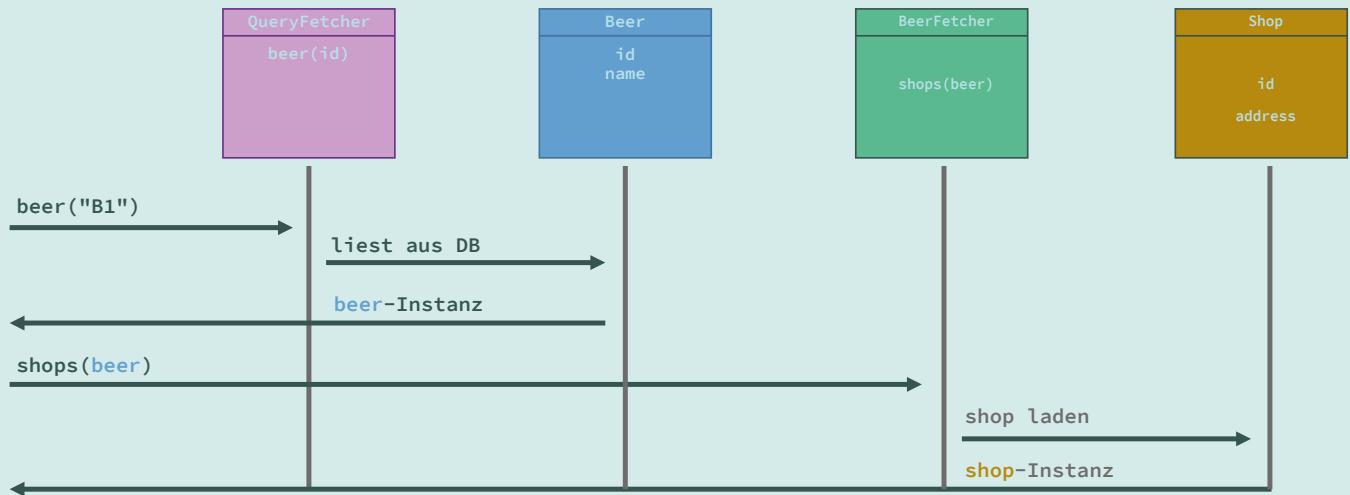


# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

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

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

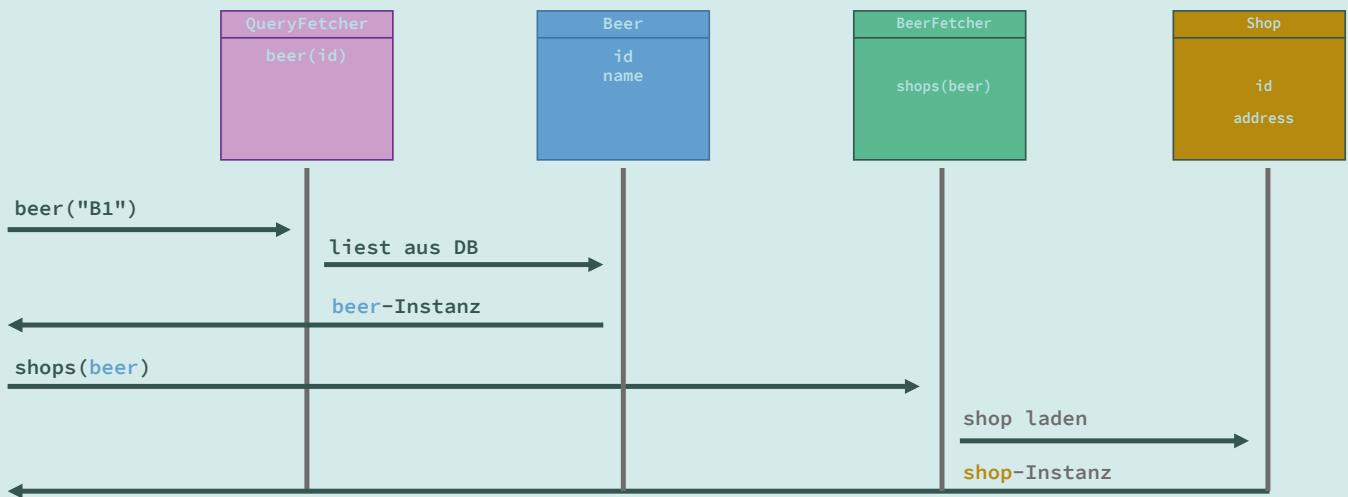


# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

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

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



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

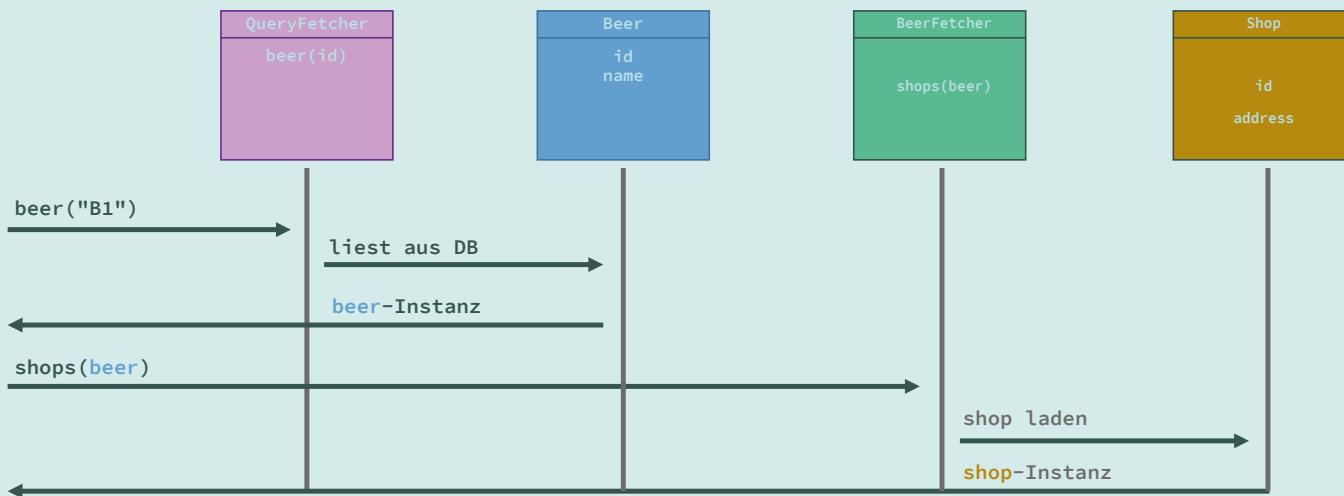
```
    }
};
```

# DATEN ERMITTLEMENT ZUR LAUFZEIT

## DataFetcher für beliebige Felder

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

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



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

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

## Ausblick

- graphql-java verfügt über eine Reihe weiterer Features für den Produktionseinsatz:
  - Performance-Optimierungen (async DataFetchers)
  - DataLoader für Caching und Optimierungen
  - Metriken

# AUSFÜHRUNG VON QUERIES

## Ausführen von Queries

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

```
GraphQLSchema graphQLSchema = ...;

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

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

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

# AUSFÜHRUNG VON QUERIES

## Ausführen von Queries

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

# HIGHER LEVEL FRAMEWORKS

## Auf graphql-java aufbauend

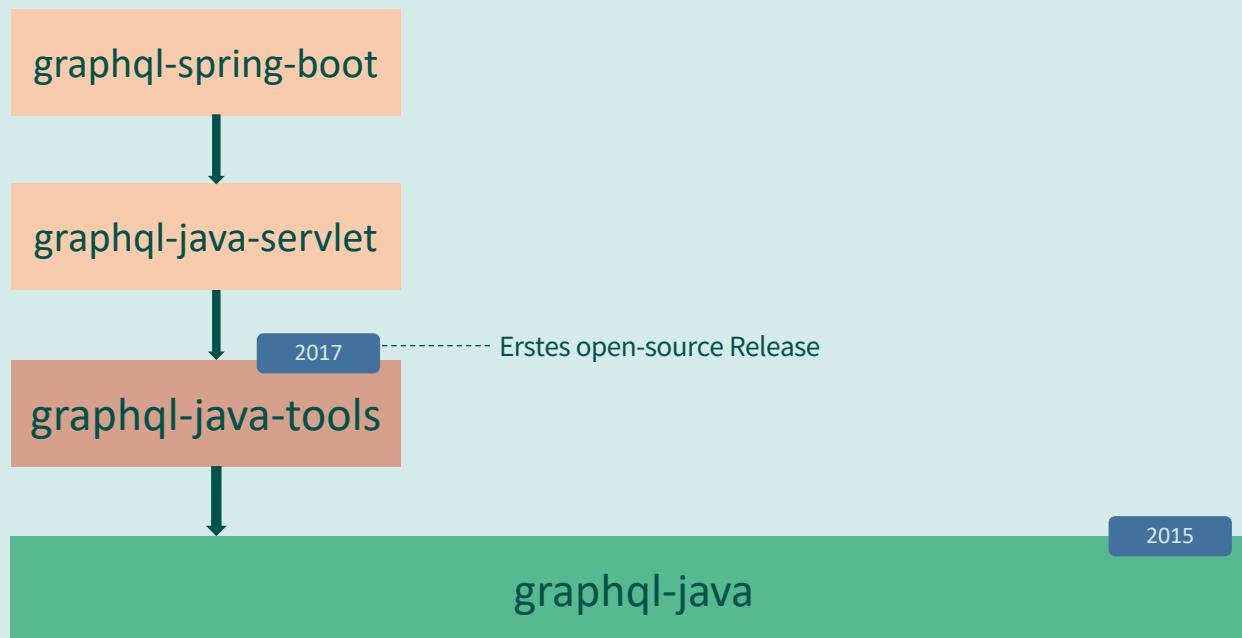
Alle verfolgen dieselbe Idee:

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

# HIGHER LEVEL FRAMEWORKS

## Auf graphql-java aufbauend

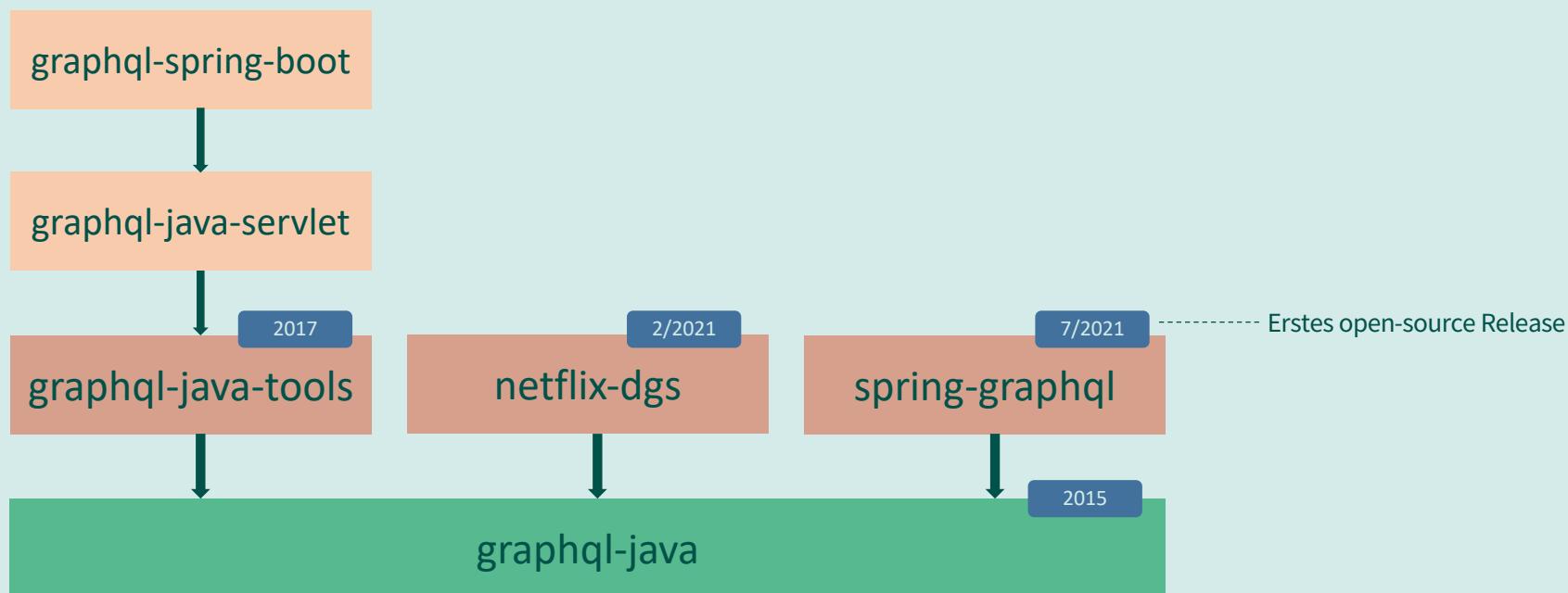
- *graphql-java-tools* ist nicht Spring/JEE-abhängig, aber es gibt Adapter dafür



# HIGHER LEVEL FRAMEWORKS

## Auf graphql-java aufbauend

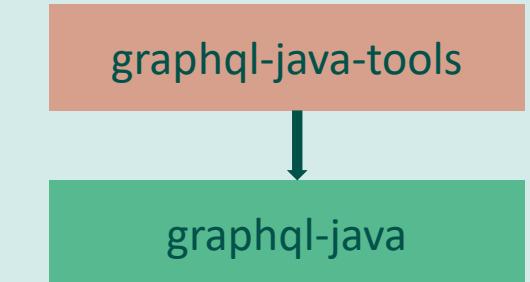
- graphql-java-tools ist nicht Spring/JEE-Abhängig, aber es gibt Adapter dafür
- **Netflix DGS** und **spring-graphql** sehen sehr ähnlich aus
  - Beide sind für Spring Boot



# GRAPHQL-JAVA-TOOLS

## graphql-java-tools

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



## Resolver mit graphql-java-tools

- Beispiel: Resolver für ein Root-Field

```
public class BeerQueryResolver implements GraphQLQueryResolver {  
  
    public Beer beer(String beerId) {  
        return beerRepository.getBeerById(beerId);  
    }  
  
}
```

## Resolver mit graphql-java-tools

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

```
public class AddRatingInput {  
    private String beerId;  
    private int stars;  
    ...  
}  
  
public class BeerAdvisorMutationResolver implements GraphQLMutationResolver {  
  
    public Rating addRating(AddRatingInput input) {  
        return ratingService.createRating(input);  
    }  
}
```

## Resolver mit graphql-java-tools

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

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

## Bonus: Validierung beim Start

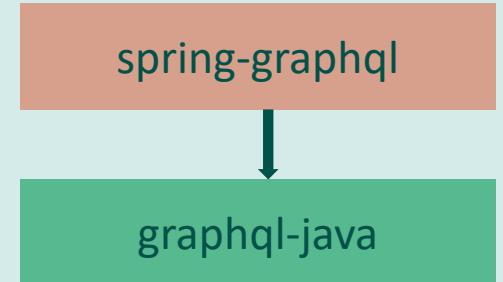
- graphql-java-tools überprüft die Resolver beim Start
- Wenn Resolver fehlen oder fehlerhaft implementiert sind, wird ein Fehler geworfen
- Es gibt also erhöhte Sicherheit, dass die Anwendung auch funktioniert

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

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

## spring-graphql

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



## Annotated Controllers

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

**@Controller**

```
public class BeerQueryController {  
  
    BeerQueryController(BeerRepository beerRepository) { ... }  
  
    @QueryMapping                                         Mapping auf das Schema mit Namenskonventionen  
    public List<Beer> beers() {  
        return beerRepository.findAll();  
    }  
  
    @MutationMapping  
    public Rating addRating(@Argument AddRatingInput input) {  
        return ratingService.createRating(input);  
    }  
}
```

}

## Annotated Controllers

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

### @Controller

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

### @QueryMapping

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

Argumente via Methoden Parameter

### @MutationMapping

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

## Annotated Controllers

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

### @Controller

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

### @QueryMapping

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

### @MutationMapping

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

Eltern-Element als Methoden Parameter

### @SchemaMapping

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

## Netflix Domain Graph Service Framework (DGS)

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

## Netflix Domain Graph Service Framework (DGS)

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

Features, die es nicht in spring-graphql gibt:

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

## Netflix Domain Graph Service Framework (DGS)

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

Features, die es nicht in spring-graphql gibt:

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

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

## MicroProfile GraphQL

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

MicroProfile GraphQL

# MICROPROFILE GRAPHQL

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

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

# MICROPROFILE GRAPHQL

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

```
@GraphQLApi
```

```
public class BeerApi {
```

```
    @Inject
```

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

```
@Query
```

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

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

```
@Query
```

```
public List<Shop> shops(@Source beer) {
```

```
    return shopRepository.findShopsSellingBeer(beer.getId());
```

```
}
```

```
}
```

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

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

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

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

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

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



# Vielen Dank!

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

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