

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

# Einführung in GraphQL

am Beispiel Java

Slides: <https://nils.buzz/hc-graphql>

# NILS HARTMANN

nils@nilshartmann.net

**Freiberuflischer Entwickler, Architekt, Trainer aus Hamburg**

Java  
JavaScript, TypeScript  
React  
GraphQL

**Trainings, Workshops und Beratung**

<https://nilshartmann.net/react-workshops>

**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://facebook.github.io/graphql/>*

- 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 != SQL*

- kein SQL, keine "vollständige" Query-Sprache
    - z.B. keine Sortierung, keine (beliebigen) Joins etc
  - keine Datenbank!
  - kein Framework!
- 
- *Ersetzt weder Backend noch Datenbank*

# *GraphQL != JavaScript*

- Populär in JS, aber auch außerhalb

## *GraphQL != Mainstream*

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



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



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

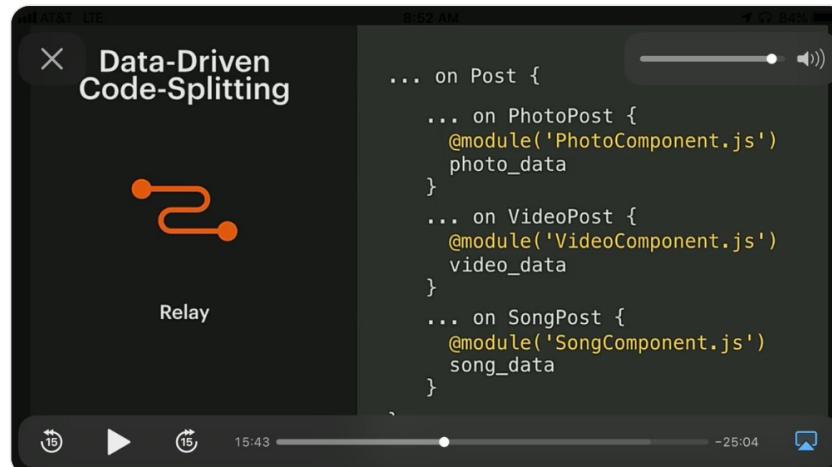


**Nick Schrock**  
@schrockn

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.

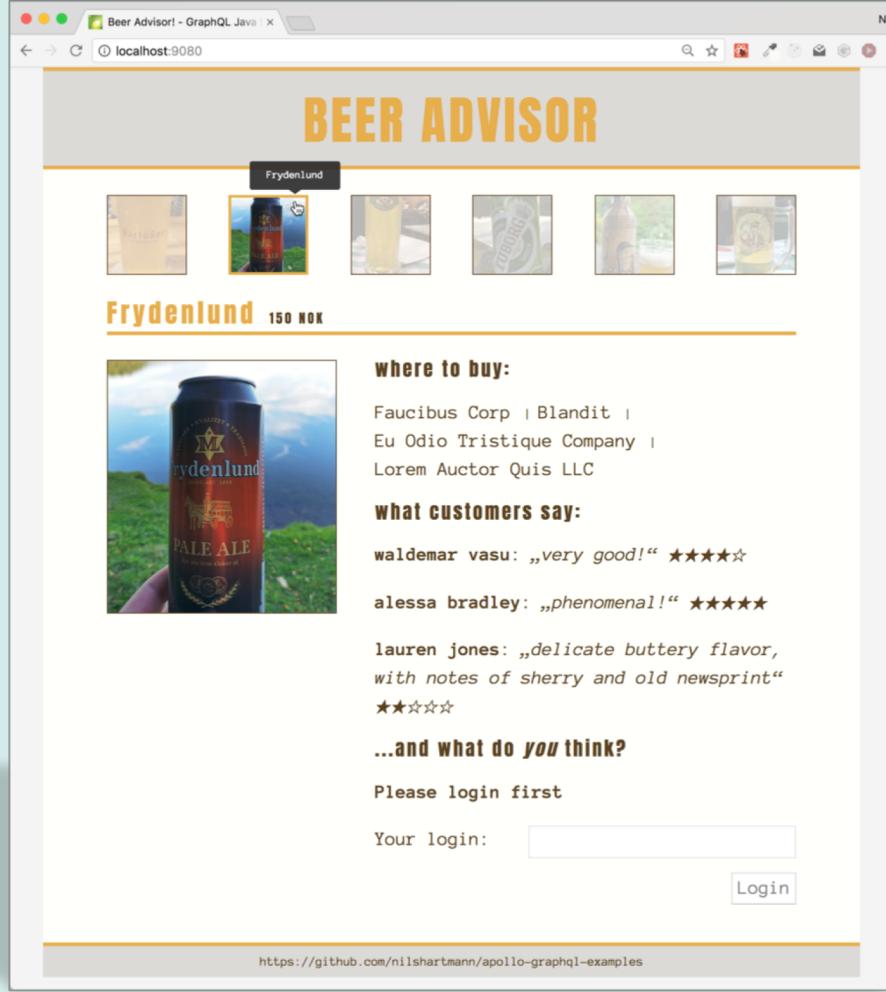
Tweet übersetzen



18:06 - 1. Mai 2019

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

## NEXT GEN GRAPHQL?



# GraphQL praktisch

Source-Code: <https://nils.buzz/hc-graphql-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' that retrieves beers, their ratings, and a ping. The right panel shows the resulting JSON data and a detailed schema pane.

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

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

QUERY VARIABLES

```
[{"id": "B1", "name": "Barfüßer", "price": "3,88 EUR", "ratings": [{"id": "R1", "beerId": "B1", "author": "Waldemar Vasu", "comment": "Exceptional!"}, {"id": "R7", "beerId": "B1", "author": "Madhukar Kareem", "comment": "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": "158 NOK", "ratings": [{"id": "R2", "beerId": "B2", "author": "Andrea Gouyen", "comment": "Very good!"}, {"id": "R8", "beerId": "B2", "author": "Marketta Glaukos", "comment": "phenomenal!"}, {"id": "R15", "beerId": "B2", "author": "Lauren Jones", "comment": "Delicate buttery flavor, with notes of sherry and old newsprint."}], {"id": "B3", "name": "Grieskirchner", "price": "3,28 EUR", "ratings": [{"id": "R3", "beerId": "B3", "author": "Nils", "comment": "Great beer, great price!"}]}
```

Schema pane details:

- 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/graphiql.html>

A screenshot of the IntelliJ IDEA code editor showing a GraphQL query. The code is as follows:

```
const BEER_RATING_APP_QUERY = gql`query BeerRatingAppQuery {
  backendStatus: ping {
    name
    nodeJsVersion
    uptime
  }
}

${/* Intellisense suggestion box */}
```

The cursor is positioned at the end of the first line of the query. An intellisense suggestion box is open, listing the following options for the current context:

- 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 suggestions, a note states: "Dot, space and some other keys will also close this lookup and be inserted into editor".

At the bottom right of the code editor window, the file name "BeerPage.tsx" is visible.

# Demo: IDE Support

Beispiel: IntelliJ IDEA

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

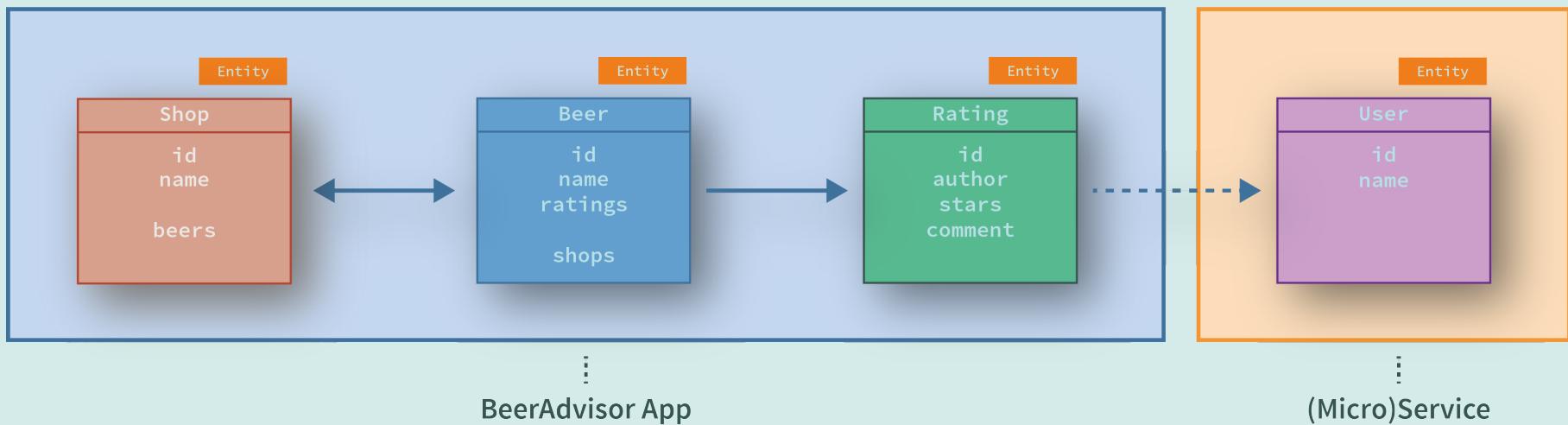
- <https://graphql.org>

# GraphQL

**TEIL 1: ABFRAGEN UND SCHEMA**

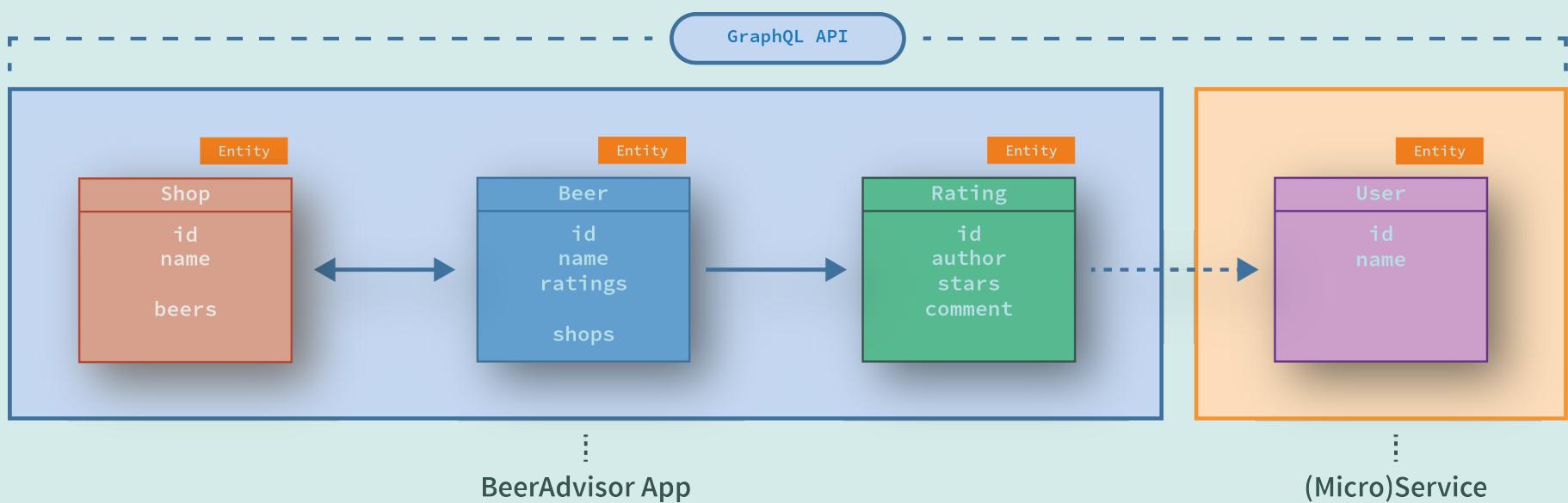
# GRAPHQL EINSATZSzenariEN

## "Architektur" Beer Advisor



# GRAPHQL EINSATZSzenariEN

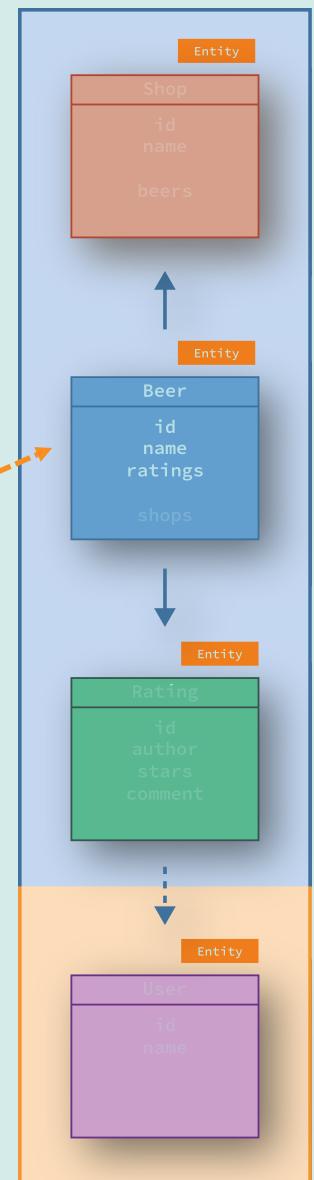
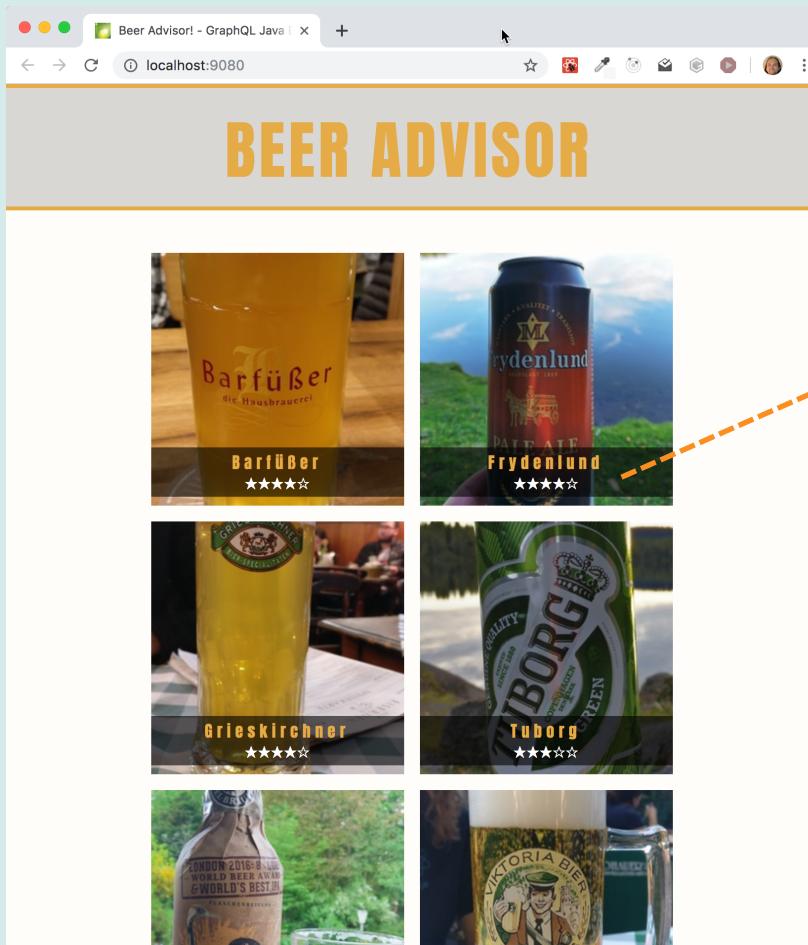
## "Architektur" Beer Advisor



# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 1

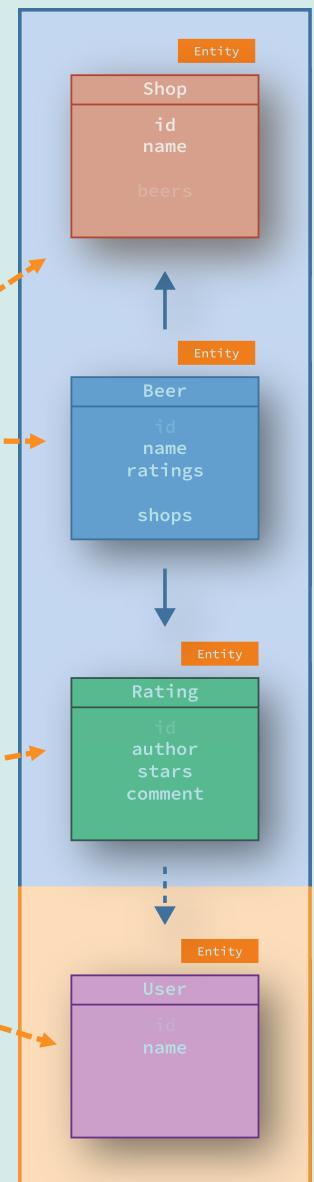
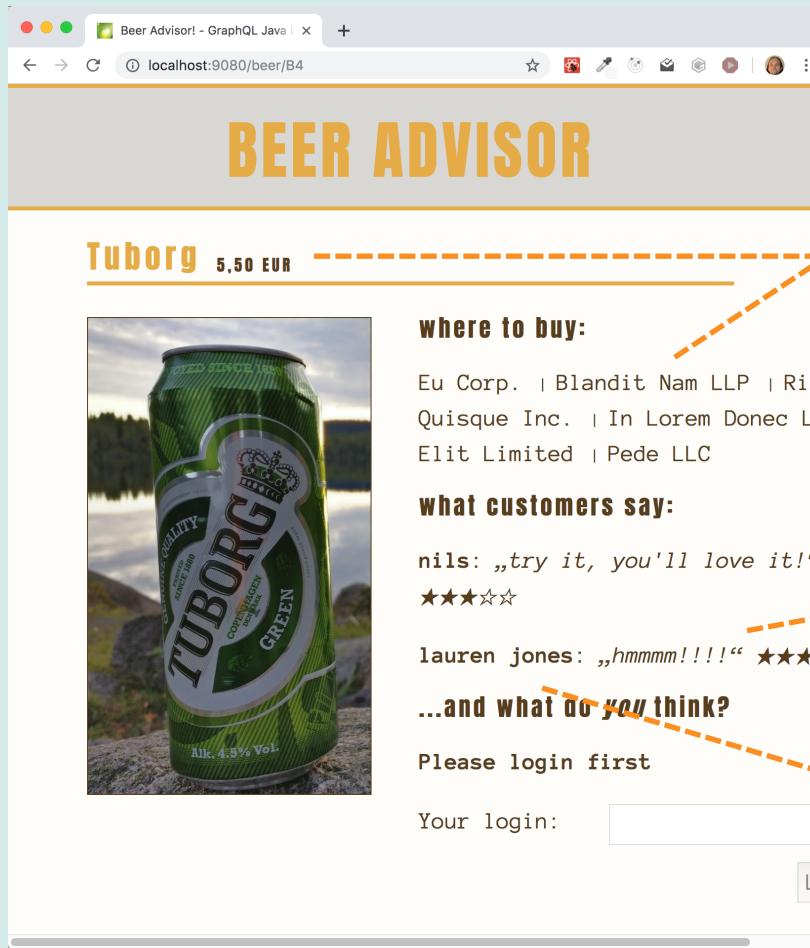
```
{ beer {  
    id  
    name  
    averageStars  
}
```



# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 2

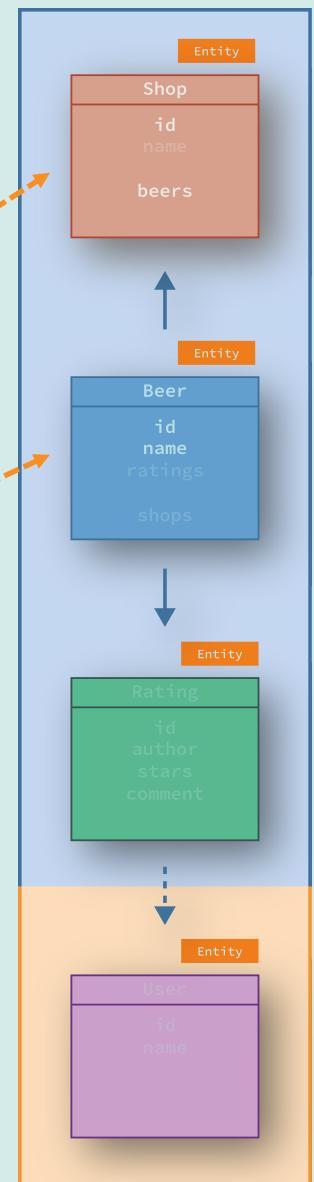
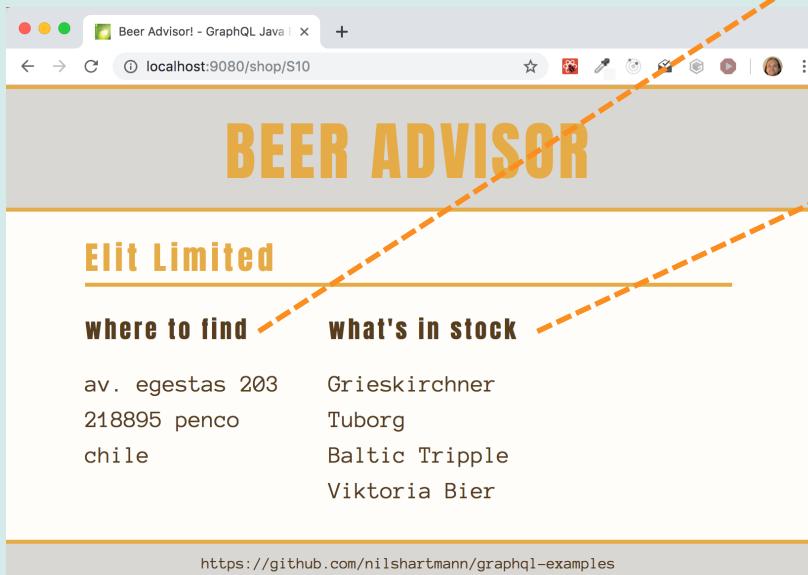
```
{ beer(beerId: "B1" {  
    name  
    price  
    ratings {  
        stars  
        comment  
        author {  
            name  
        }  
    }  
    shops { name }  
}
```



# GRAPHQL EINSATZSzenarien

## Use-Case spezifische Abfragen – 3

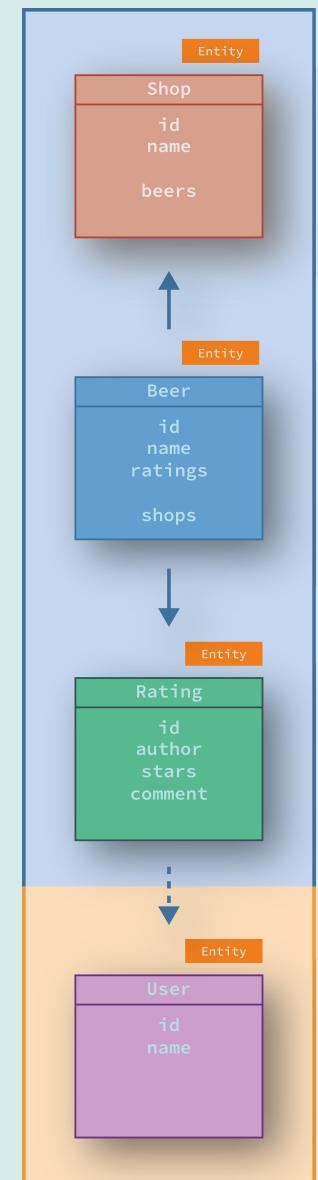
```
{ shop(shopId: "S3") {  
    name  
    address { street city }  
    beers { id name }  
}
```



# GRAPHQL EINSATZSzenariEN

## Zusammenfassung

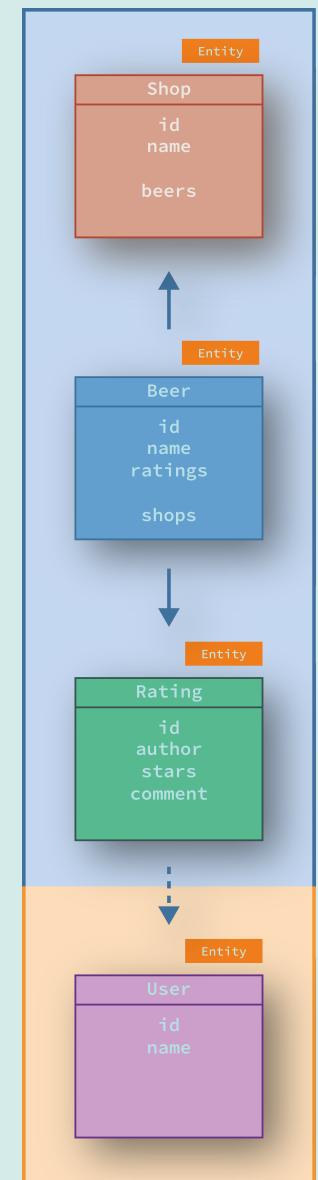
- Queries bieten *explizite* Sicht auf benötigte Daten
  - Queries können nach Geschmack ausgeführt werden  
Pro Seite, pro Komponente, ...



# GRAPHQL EINSATZSzenariEN

## Zusammenfassung

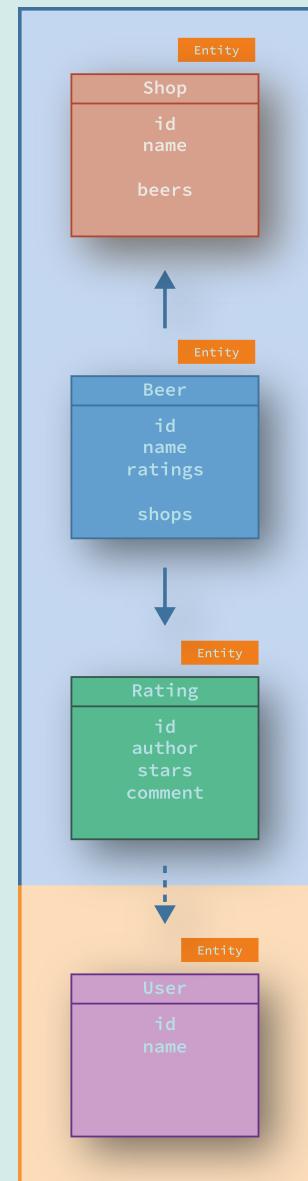
- Queries bieten *explizite* Sicht auf benötigte Daten
  - Queries können nach Geschmack ausgeführt werden  
Pro Seite, pro Komponente, ...
- Abgefragt werden *Daten*, nicht *Endpunkte*



# GRAPHQL EINSATZSzenariEN

## Zusammenfassung

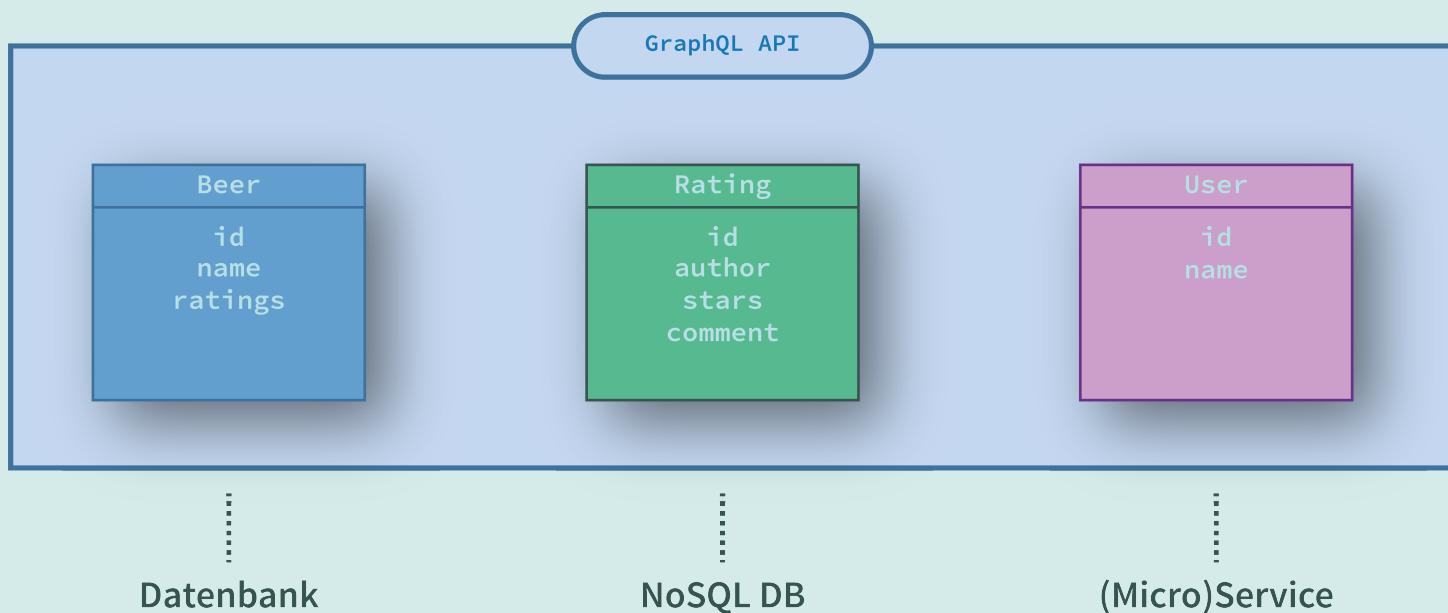
- Queries bieten *explizite* Sicht auf benötigte Daten
  - Queries können nach Geschmack ausgeführt werden  
Pro Seite, pro Komponente, ...
- Abgefragt werden *Daten*, nicht *Endpunkte*
- Durch Typisierung
  - ist sichergestellt, dass  
Abfrage und Antwort gültig sind
  - guter Tool-Support während der Entwicklung
  - API ist "explorierbar"



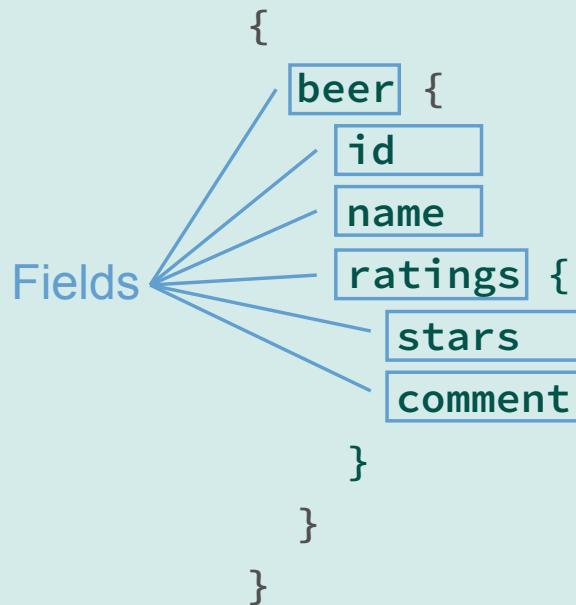
# DATEN QUELLEN

## GraphQL macht keine Aussage, wo die Daten herkommen

- Versteckt unterschiedliche APIs/Services
- Gesamt-Sicht auf die Domain/Anwendung
  - Fachliche Abfragen möglich
- *Ermittlung der Daten ist unsere Aufgabe*

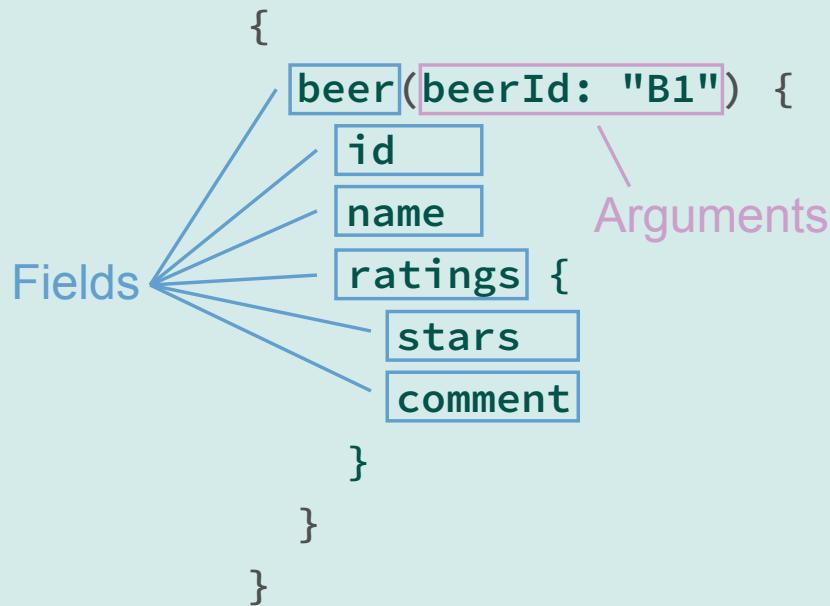


# QUERY LANGUAGE



- 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  
    }  
  }  
}
```



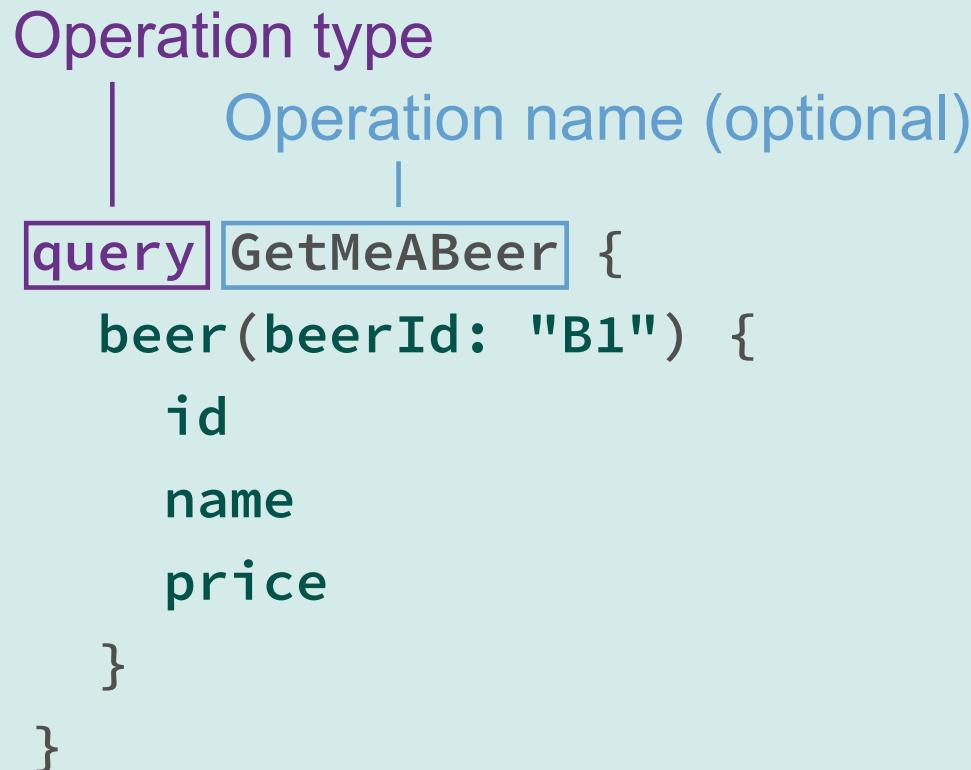
```
"data": {  
  "beer": {  
    "id": "B1"  
    "name": "Barfüßer"  
    "ratings": [  
      {  
        "stars": 3,  
        "comment": "grate taste"  
      },  
      {  
        "stars": 5,  
        "comment": "best beer ever!"  
      }  
    ]  
  }  
}
```

- Identische Struktur wie bei der Abfrage

# QUERY 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

```
Operation type
  |
  |     Operation name (optional)
  |
  |     subscription NewRatingSubscription {
  |       newRating: onNewRating {
  |         id
  |         beerId
  |         author
  |         comment
  |       }
  |     }
  |   }
```

Field alias

# 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"}  
  ]}  
}
```

## 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")	<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
- Alte Felder können 'deprecated' werden
- Verwendung der Felder kann einzeln getrackt werden

Neues Feld -----

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

*"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)**

# GRAPHQL FÜR JAVA-ANWENDUNGEN

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

*Die gezeigten Konzepte sind in GraphQL-Frameworks für andere 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

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

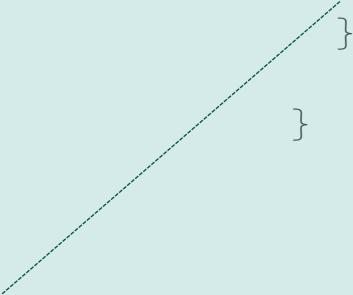
```
interface DataFetcher<T> {  
    T get(DataFetchingEnvironment environment);  
}
```

# 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;

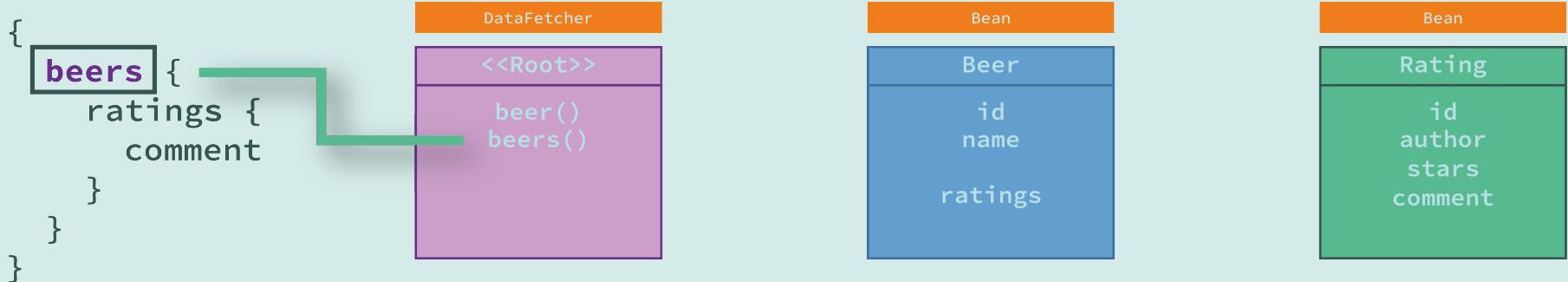
public DataFetcher<Publisher<Rating>> onNewRatingFetcher() {

    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)

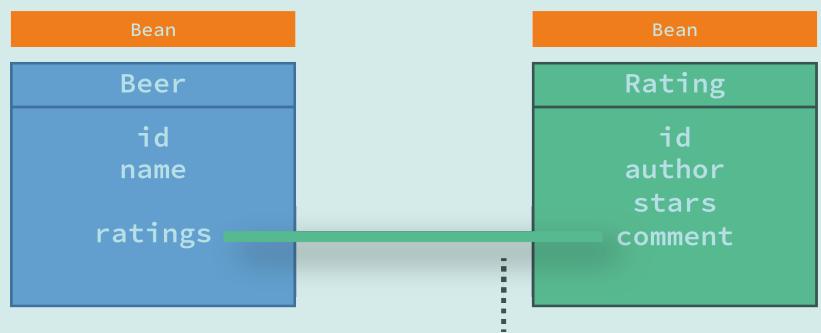
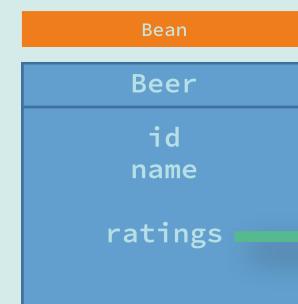
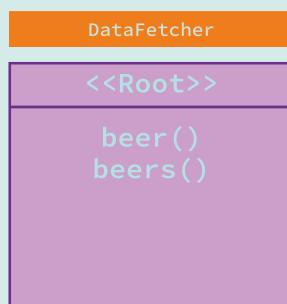
```
{  
  beers {  
    ratings {  
      comment  
    }  
  }  
}
```



# DATEN ERMITTLEMENT ZUR LAUFZEIT

- 3. Zugriff auf Bean (PropertyDataFetcher)

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

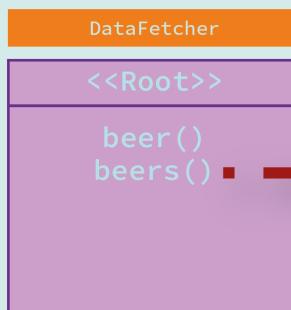


PropertyDataFetcher (Reflection per Getter oder Field-Access)

# DATEN ERMITTLEMENT ZUR LAUFZEIT

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

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



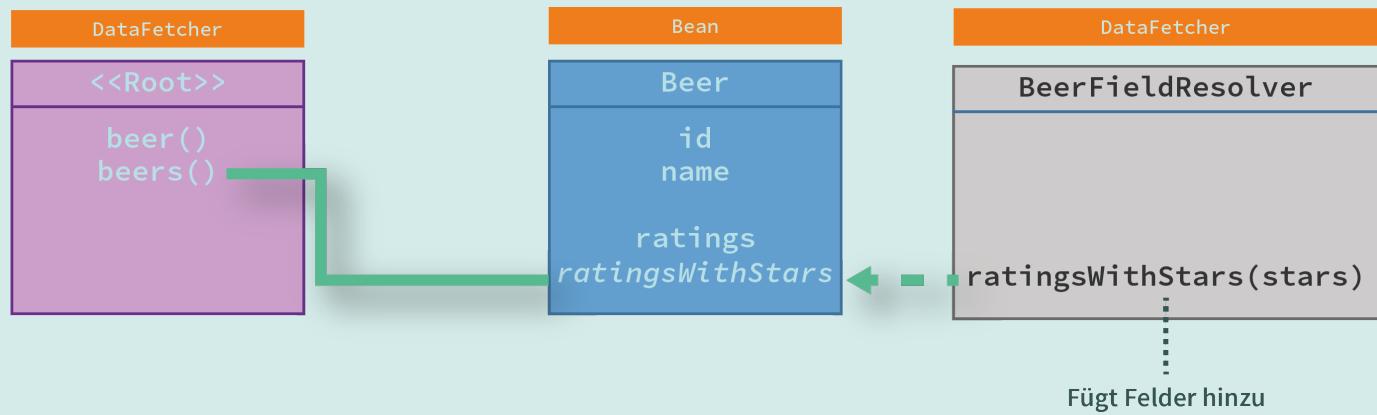
Feld/Methode ,ratingWithStars' nicht in Beer-Klasse vorhanden

# 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*

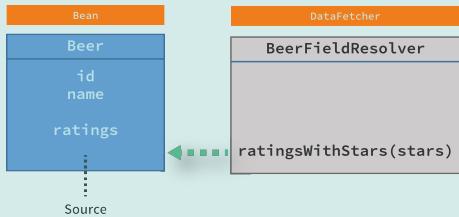
```
{  
  beers {  
    ratingsWithStars  
    (stars: 3) {  
      comment  
    }  
  }  
}
```



# DATA FETCHER FÜR NICHT-ROOT-FELDER

## DataFetcher implementieren

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



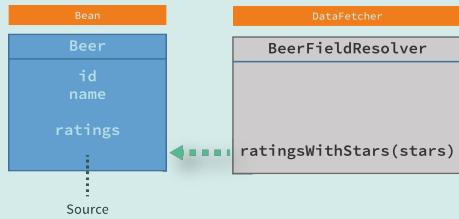
```
public class BeerDataFetchers {  
  
    public DataFetcher<List<Rating>> ratingsWithStarsFetcher() {  
        return environment -> {  
            Beer beer = environment.getSource();  
  
            return beer.ratingsWithStars(environment.getArgument("stars"));  
        };  
    }  
}
```

```
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!]!  
}
```

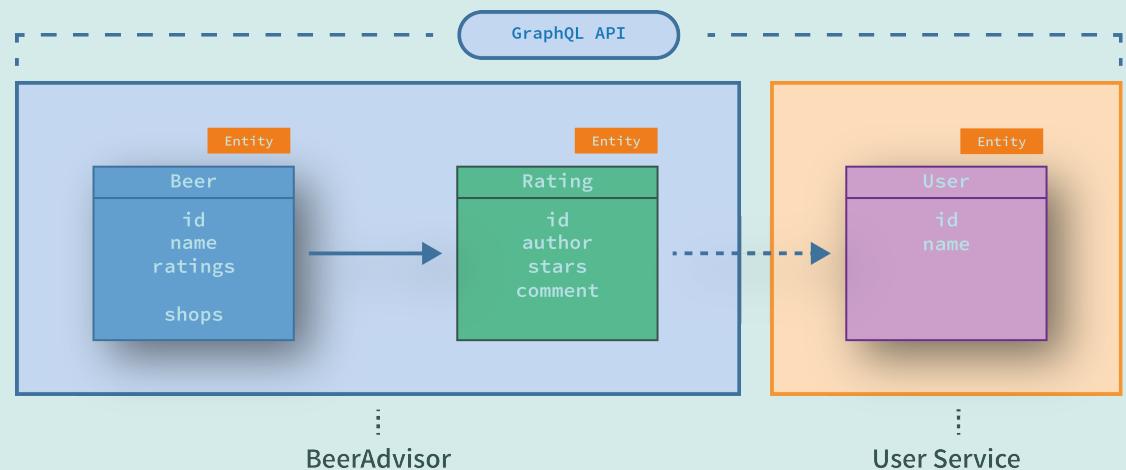
## EXKURS: OPTIMIERUNGEN

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

# 1+N-PROBLEM

## Beispiel: Zugriff auf (Remote-)Services

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



# 1+N-PROBLEM

## 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);  
    };  
}
```

# 1+N-PROBLEM

## 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. 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!

# 1+N-PROBLEM

## 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. 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:

- Fasst Aufrufe zusammen (Batching)
- Cached die Ergebnisse
- Wird asynchron ausgeführt

```
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

## Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück  
(unverändert)

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

# 1+N-PROBLEM

## Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück  
(unverändert)
2. author-DataFetcher delegiert Ermitteln der Daten  
an den DataLoader.  
GraphQL verzögert das eigentliche Laden der Daten  
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+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 (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);  
    };  
}
```

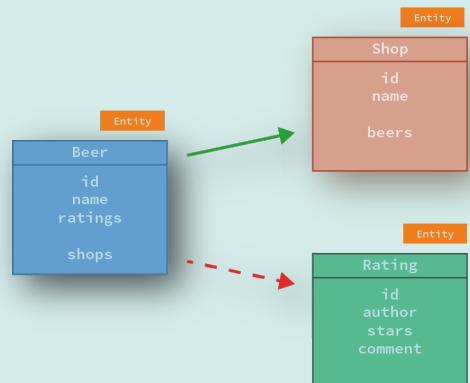
 Sammelt alle load-Aufrufe ein und führt erst dann den DataLoader aus

=> 1 (Beer) + 1 (Remote)-Call 😊

# EXKURS: OPTIMIERUNGEN

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

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

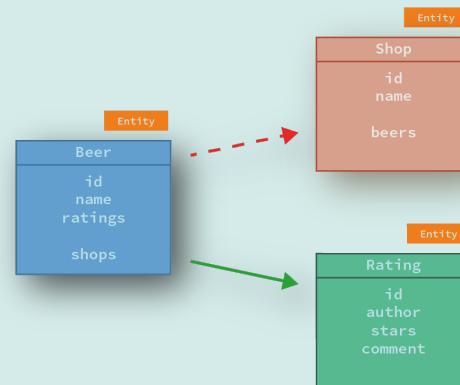
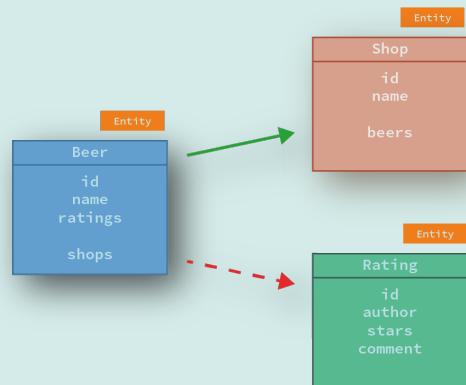


# EXKURS: OPTIMIERUNGEN

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

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

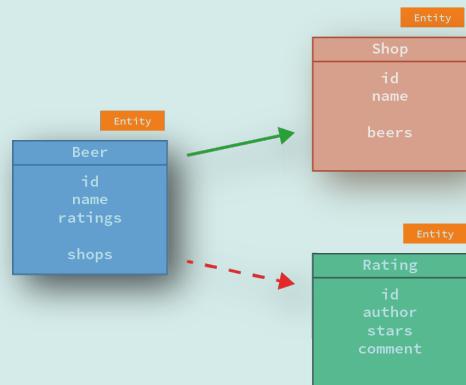
```
beers {  
    name  
    ratings {  
        comment  
    }  
}
```



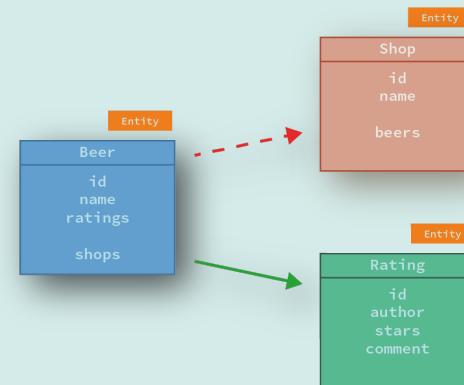
# EXKURS: OPTIMIERUNGEN

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

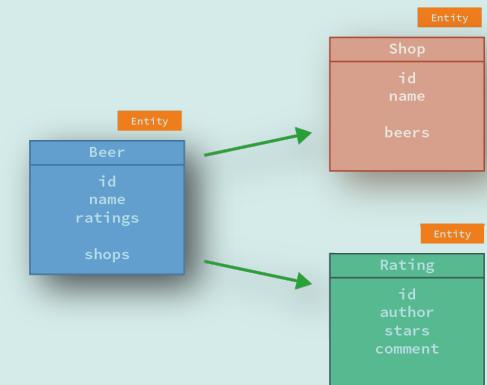
```
beers {  
    name  
    shops {  
        name  
    }  
}
```



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



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

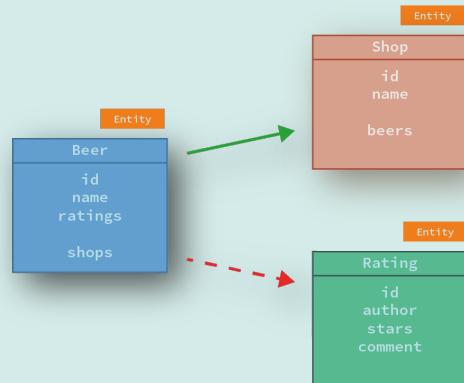


# EXKURS: OPTIMIERUNGEN

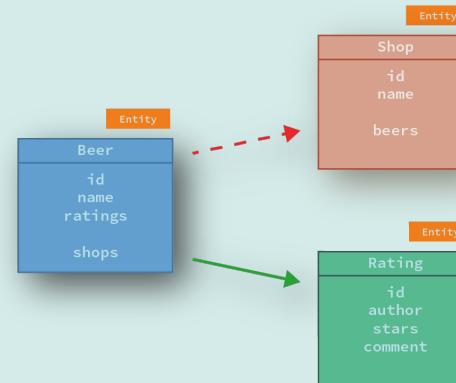
## Problem: optimaler Datenbankzugriff (Beispiel: JPA/JOINS)

- nur zur Laufzeit ermittelbar
- möglichst auf oberstem DataFetcher entscheiden

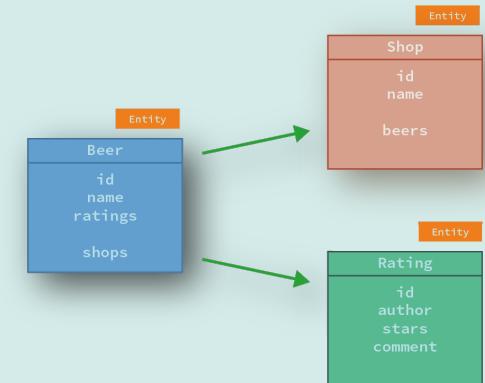
```
beers {  
    name  
    shops {  
        name  
    }  
}
```



```
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);  
    };  
}
```

**Queries ausführen**

&

**API bereitstellen**

**GRAPHQL ZUR LAUFZEIT**

# RUNTIME WIRING

## Schritt 1: Verbinden von Schema und DataFetcher

- Im RuntimeWiring werden Schema und DataFetcher verknüpft

```
class BeerAdvisorGraphQLSetup {  
    public RuntimeWiring setupWiring() {  
        BeerAdvisorDataFetchers fetchers = ...; // z.B. Spring DI  
  
        return RuntimeWiring.newRuntimeWiring()  
  
            .build();  
    }  
}
```

## Schritt 1: Verbinden von Schema und DataFetcher

- Im RuntimeWiring werden Schema und DataFetcher verknüpft

```
class BeerAdvisorGraphQLSetup {  
    public RuntimeWiring setupWiring() {  
        BeerAdvisorDataFetchers fetchers = ...; // z.B. Spring DI  
  
        return RuntimeWiring.newRuntimeWiring()  
            .type(Query.class, typeBuilder ->  
                typeBuilder.name("Query")  
                    .build();  
            );  
    }  
}
```

# RUNTIME WIRING

## Schritt 1: Verbinden von Schema und DataFetcher

- Im RuntimeWiring werden Schema und DataFetcher verknüpft

```
class BeerAdvisorGraphQLSetup {  
    public RuntimeWiring setupWiring() {  
        BeerAdvisorDataFetchers fetchers = ...; // z.B. Spring DI  
  
        return RuntimeWiring.newRuntimeWiring()  
            .type(newTypeWiring("Query")  
                .dataFetcher("beers", fetchers.beersFetcher())  
                .dataFetcher("beer", fetchers.beerFetcher()))  
  
            .build();  
    }  
}
```

# RUNTIME WIRING

## Schritt 1: Verbinden von Schema und DataFetcher

- Im RuntimeWiring werden Schema und DataFetcher verknüpft

```
class BeerAdvisorGraphQLSetup {  
    public RuntimeWiring setupWiring() {  
        BeerAdvisorDataFetchers fetchers = ...; // z.B. Spring DI  
  
        return RuntimeWiring.newRuntimeWiring()  
            .type(newTypeWiring("Query")  
                .dataFetcher("beers", fetchers.beersFetcher())  
                .dataFetcher("beer", fetchers.beerFetcher()))  
            .type(newTypeWiring("Beer").  
                .dataFetcher("ratingsWithStars", fetchers.beersFetcher()))  
            .build();  
    }  
}  
  
type Query {  
    beers: [Beer!]!  
    beer(beerId: ID!): Beer  
}  
  
type Beer {  
    ratingsWithStars(stars:Int!)  
    [Rating!]!  
}
```

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Schritt 2: Ausführbares Schema erzeugen

- Statisches Schema und DataFetcher (Wirings) werden verknüpft
- Einstiegspunkt zum Ausführen von Queries

```
class BeerAdvisorGraphQLSetup {  
  
    public GraphQLSchema setupGraphQLSchema() {  
  
        // Schritt 1: Schema-Beschreibung  
        File schemaFile = new File("beeradvisor.graphqls");  
  
        // Schritt 2+3: DataFetcher & RuntimeWiring (wie zuvor gesehen)  
        RuntimeWiring runtimeWiring = setupWiring();  
  
        SchemaGenerator schemaGenerator = new SchemaGenerator();  
  
        return schemaGenerator.makeExecutableSchema(  
            new SchemaParser().parse(schemaFile),  
            runtimeWiring  
        );  
    }  
}
```

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Schritt 3a: Queries ausführen (per API)

- Ergebnis wird in verschachtelter Map zurückgeliefert

```
GraphQLSchema schema = new BeerAdvisorGraphQLSetup().setupGraphQLSchema();  
  
GraphQL graphQL = GraphQL.newGraphQL(schema).build();  
  
ExecutionInput executionInput =  
    ExecutionInput.newExecutionInput  
        .dataLoaderRegistry(...)  
        .query("query { beers { name ratings { stars } } }").build();  
  
Map<String, Object> result = graphQL.execute(executionInput).toSpecification();
```

## Schritt 3b: Queries ausführen (per HTTP)

- Voraussetzung: GraphQL Schema ist erzeugt
- Variante 1: <https://github.com/graphql-java/graphql-java-spring>
  - REST Controller für Spring (Boot)
  - Stammt aus graphql-java Projektfamilie
  - Kein Support für Subscriptions zurzeit
- Variante 2: <https://github.com/graphql-java-kickstart/graphql-java-servlet>
  - HTTP Servlet (für Spring bzw Servlet Container)
  - Auch als Starter für Spring Boot verfügbar

## ALTERNATIVE: GRAPHQL-JAVA-TOOLS

### Resolver mit graphql-java-tools

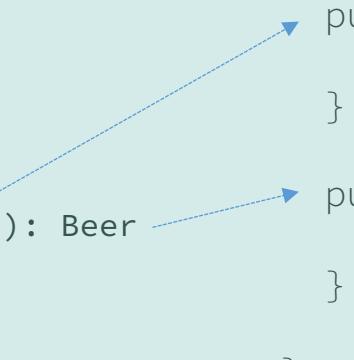
- <https://github.com/graphql-java-kickstart/graphql-java-tools>
- Abstraktion, basierend auf graphql-java, arbeitet mit POJOs

## 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!]!  
  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);  
  }  
}
```



# ALTERNATIVE: GRAPHQL-JAVA-TOOLS

## Mutations mit Resolver

```
type Mutation {  
    addRating  
    (ratingInput: AddRatingInput): Rating!  
}  
  
class AddRatingInput {  
    private String beerId;  
    private int stars;  
    private String comment;  
    ...  
}  
  
public class BeerAdvisorMutationResolver implements  
    GraphQLMutationResolver {  
  
    public Rating addRating(AddRatingInput ratingInput) {  
        Rating rating = Rating.from(ratingInput);  
        ratingRepository.save(rating);  
        return rating;  
    }  
}
```

## Spring Boot Starter

- <https://github.com/graphql-java-kickstart/graphql-spring-boot>
- Basiert auf Resolvern (aus graphql-java-tools)
- Mergt alle Schema-Dateien im Klassenpfad zusammen (\*.graphqls)
- Resolver werden als Beans annotiert (zB. @Component) und automatisch dem Schema hinzugefügt
- Servlet-Konfiguration erfolgt per application.properties
- GraphiQL (API Explorer) kann ebenfalls per Konfiguration aktiviert werden



# Vielen Dank!

Beispiel-Code: <https://nils.buzz/hc-graphql-example>

Slides: <https://nils.buzz/hc-graphql>

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