NILS HARTMANN

GraphQL

für Java-Entwickler

Slides: https://bit.ly/javaland-graphql

JAVALAND, BRÜHL | MÄRZ 2019 | @NILSHARTMANN

NILS HARTMANN

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

Java
JavaScript, TypeScript
React

nils@nilshartmann.net | @nilshartmann | https://nilshartmann.net

GraphQL-Einführung bei heise developer:

Teil 1: http://bit.ly/heise-graphql-1
Teil 2: http://bit.ly/heise-graphql-2

*"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!

GraphQL != Mainstream

- Implementierungen und Einsatz noch "bleeding edge"
- Wenig erprobte Best-Practices

GraphQL != Mainstream

- Implementierungen und Einsatz noch "bleeding edge"
- Wenig erprobte Best-Practices

- ...dennoch wird es von einigen verwendet!

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

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

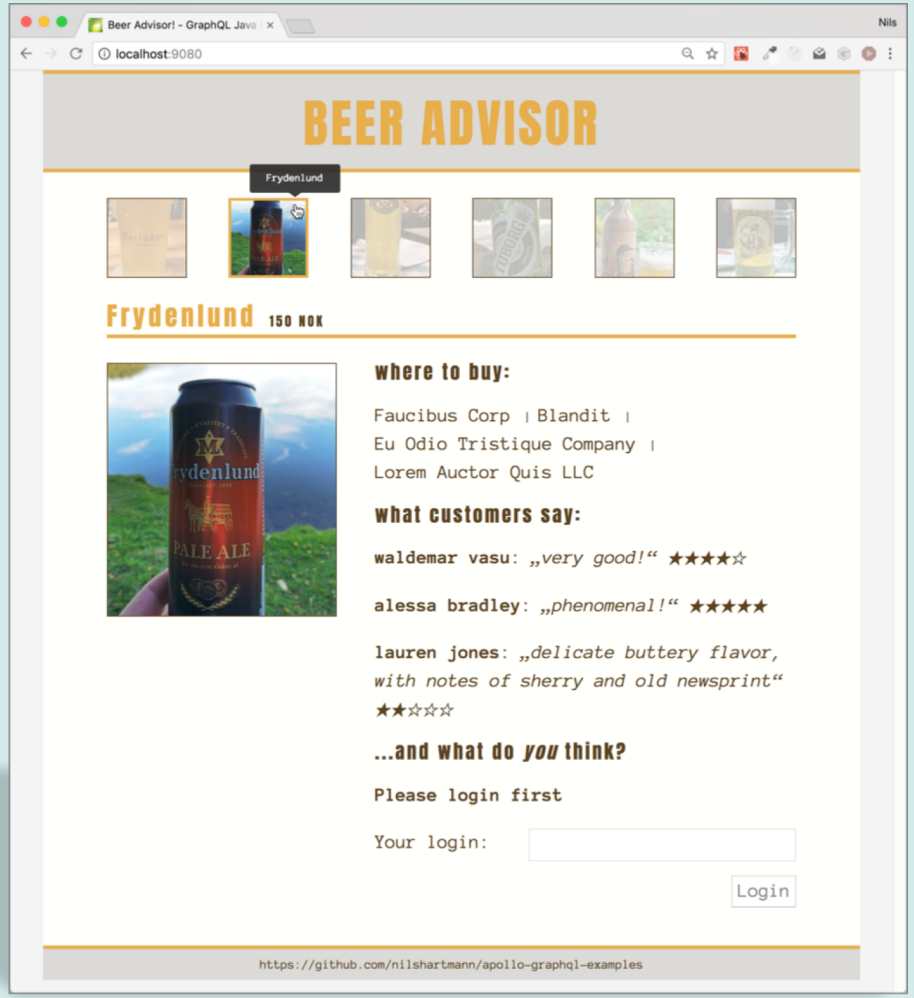
React, Relay and GraphQL: Under the Hood of the Times Website Redesign

A look under the hood.

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

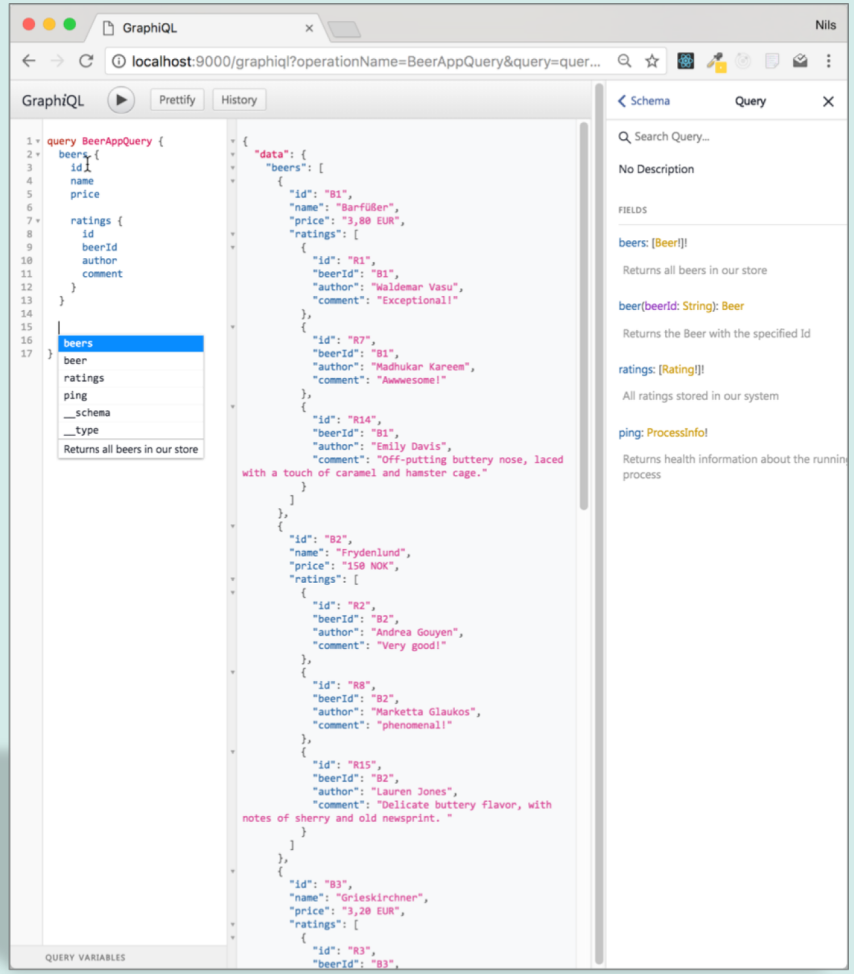
https://open.nytimes.com/react-relay-and-graphql-under-the-hood-of-the-times-website-redesign-22fb62ea9764

NEW YORK TIMES



GraphQL praktisch

Source-Code: https://bit.ly/javaland-graphql-example



Demo: GraphiQL

http://localhost:9000/graphiql.html

Demo: IDE Support

Beispiel: Intellij IDEA

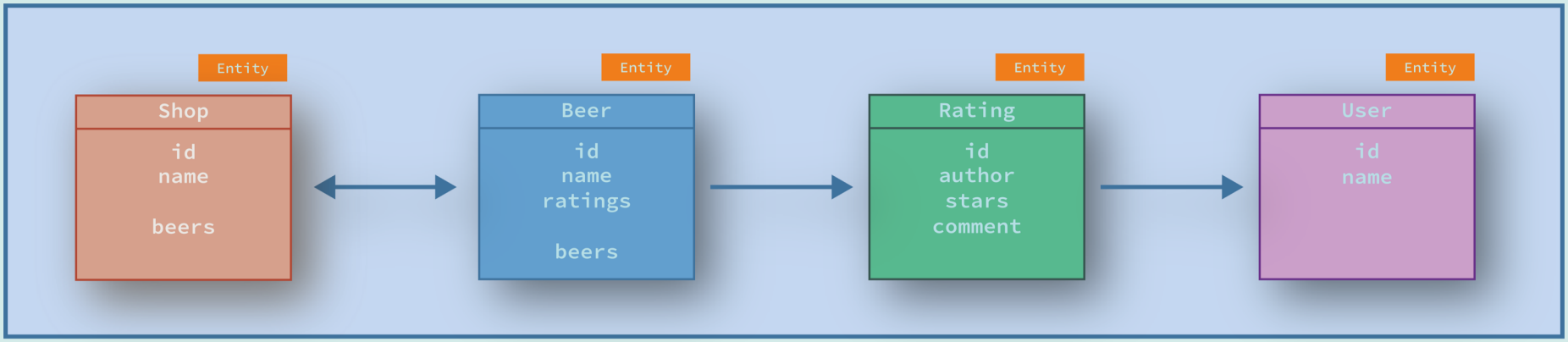
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- https://graphql.org

GraphQL

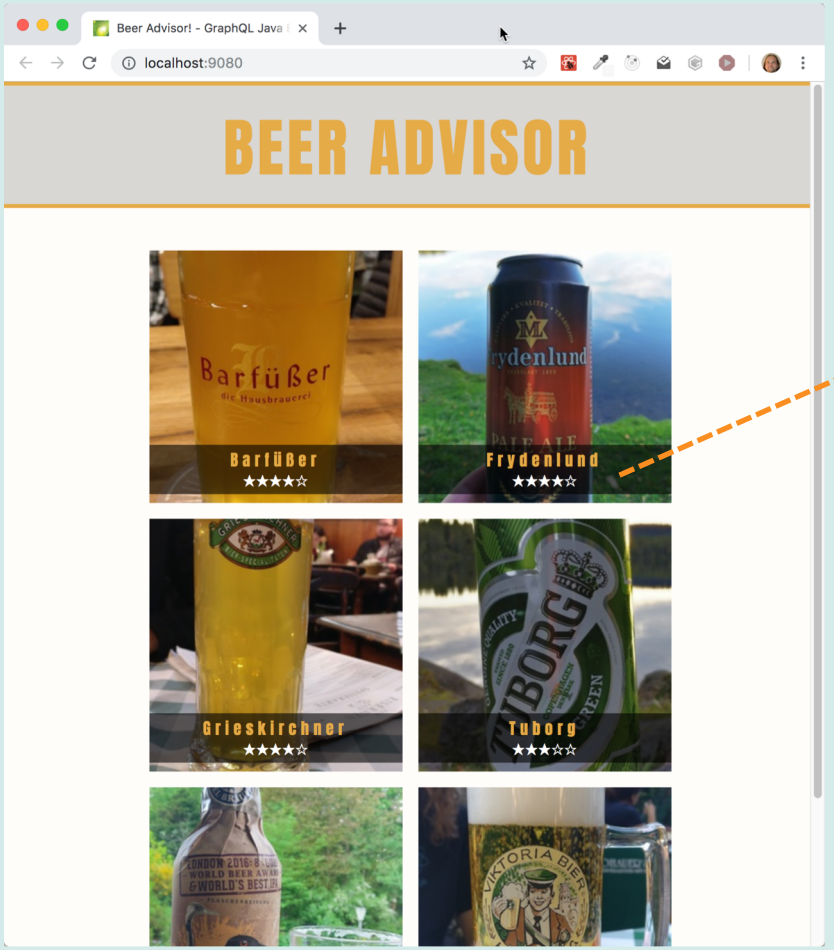
TEIL 1: ABFRAGEN UND SCHEMA

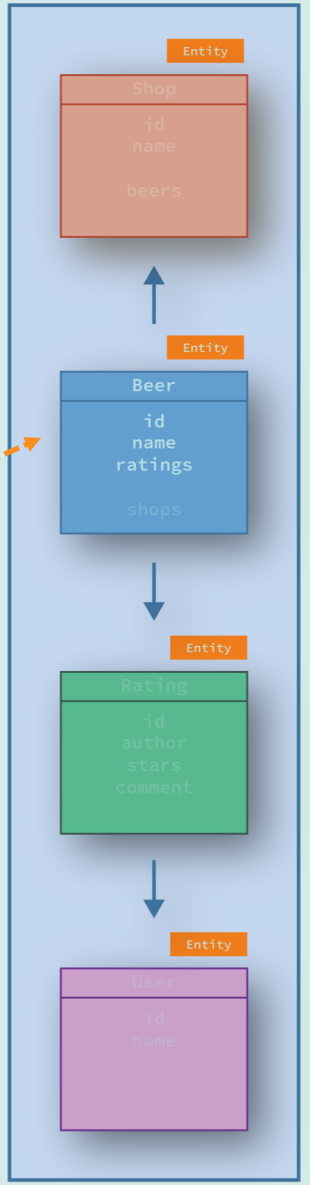
Domain-Model "Beer Advisor"



Use-Case spezifische Abfragen – 1

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

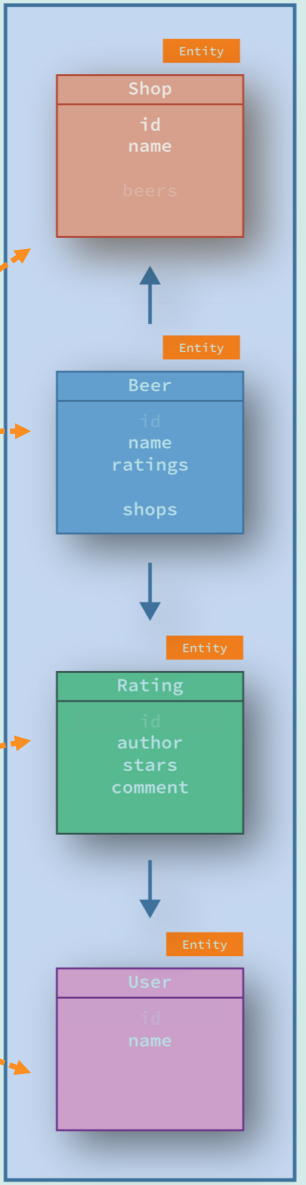




GRAPHQL EINSATZSZENARIEN

Use-Case spezifische Abfragen – 2

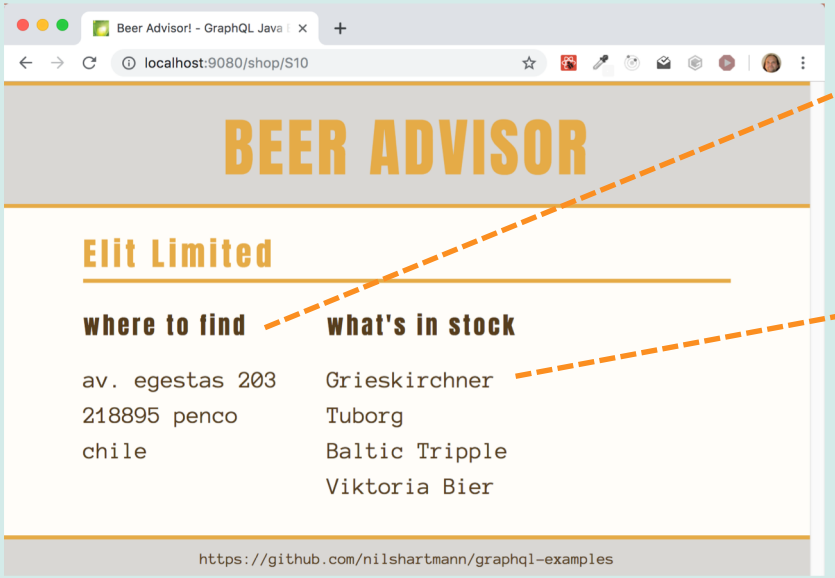
```
{ beer(beerId: "B1" {
    name
    price
    ratings {
       stars
       comment
       author {
         name
       }
    }
    shops { name }
  }
}
```

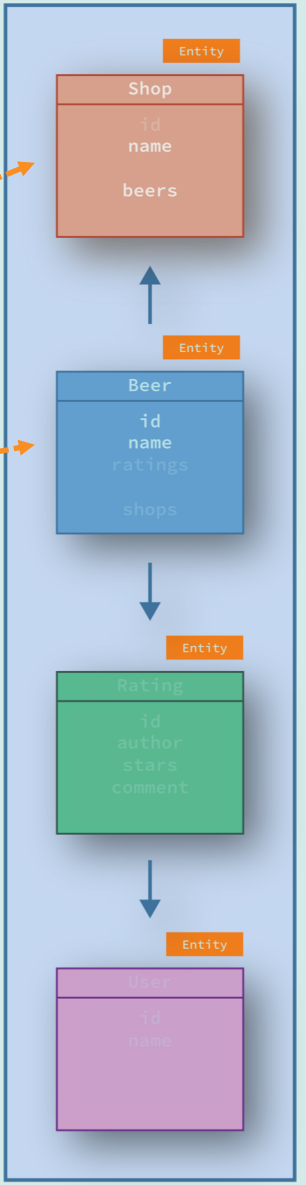


GRAPHQL EINSATZSZENARIEN

Use-Case spezifische Abfragen – 3

```
{ shop(shopId: "S3" {
    name
    beers { id name }
  }
}
```

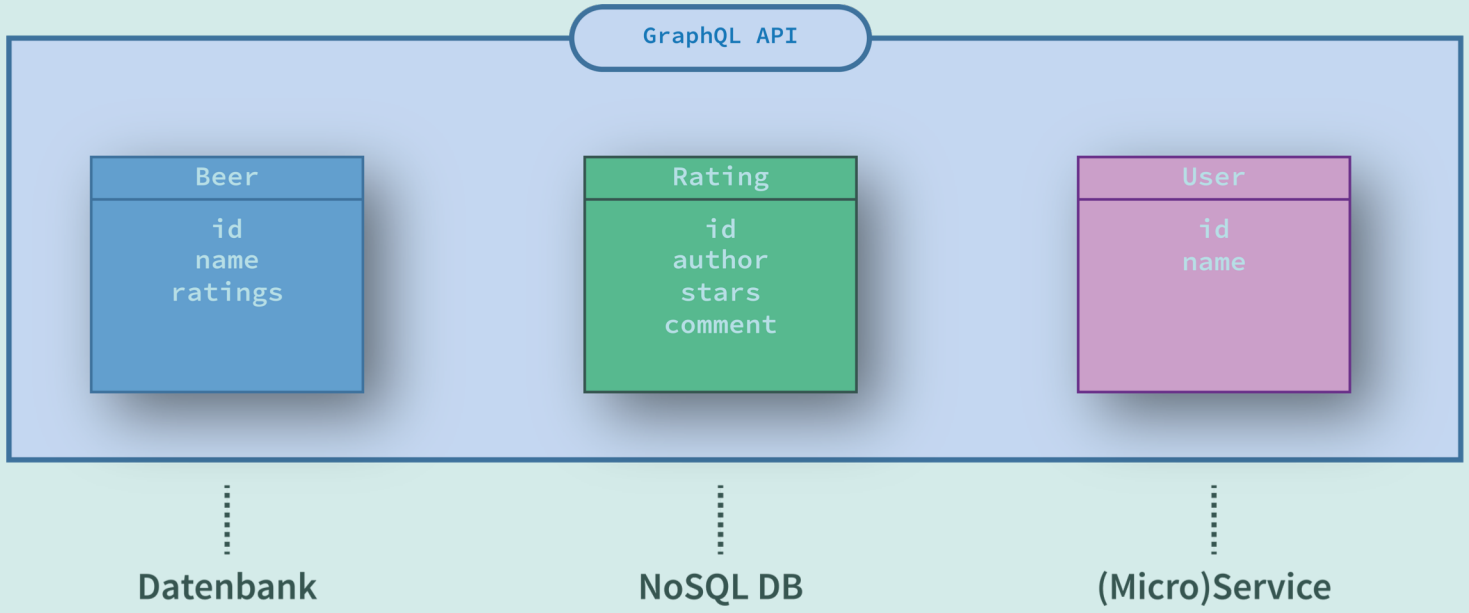




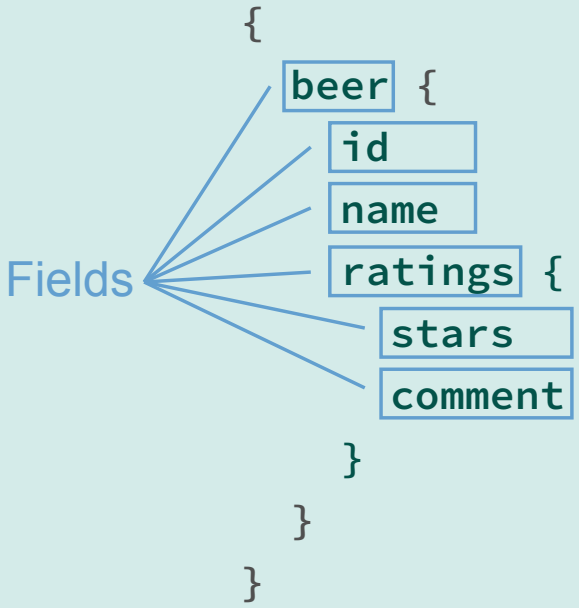
DATEN QUELLEN

GraphQL macht keine Aussage, wo die Daten herkommen

- Ermittlung der Daten ist unsere Aufgabe



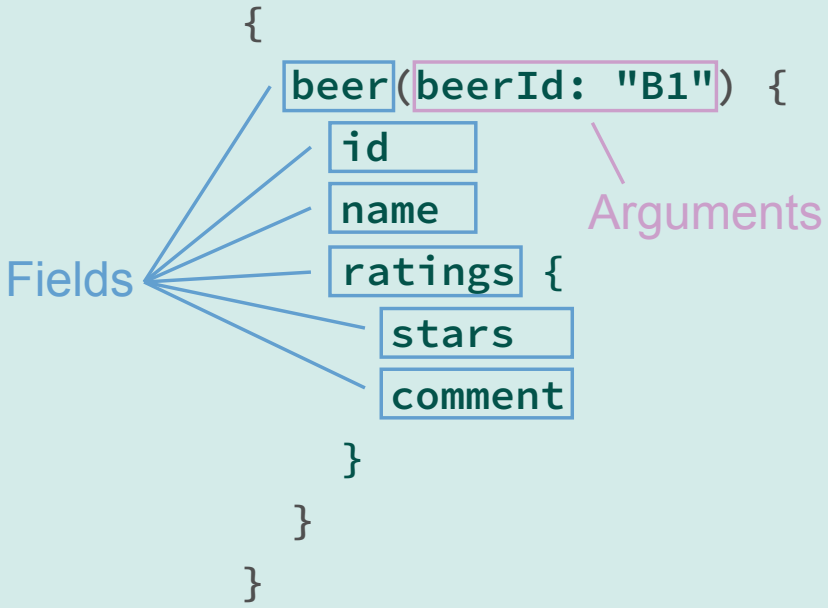
QUERY LANGUAGE



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

Credits: https://dev-blog.apollodata.com/the-anatomy-of-a-graphql-query-6dffa9e9e747

QUERY LANGUAGE



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

Credits: https://dev-blog.apollodata.com/the-anatomy-of-a-graphql-query-6dffa9e9e747

Ergebnis

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

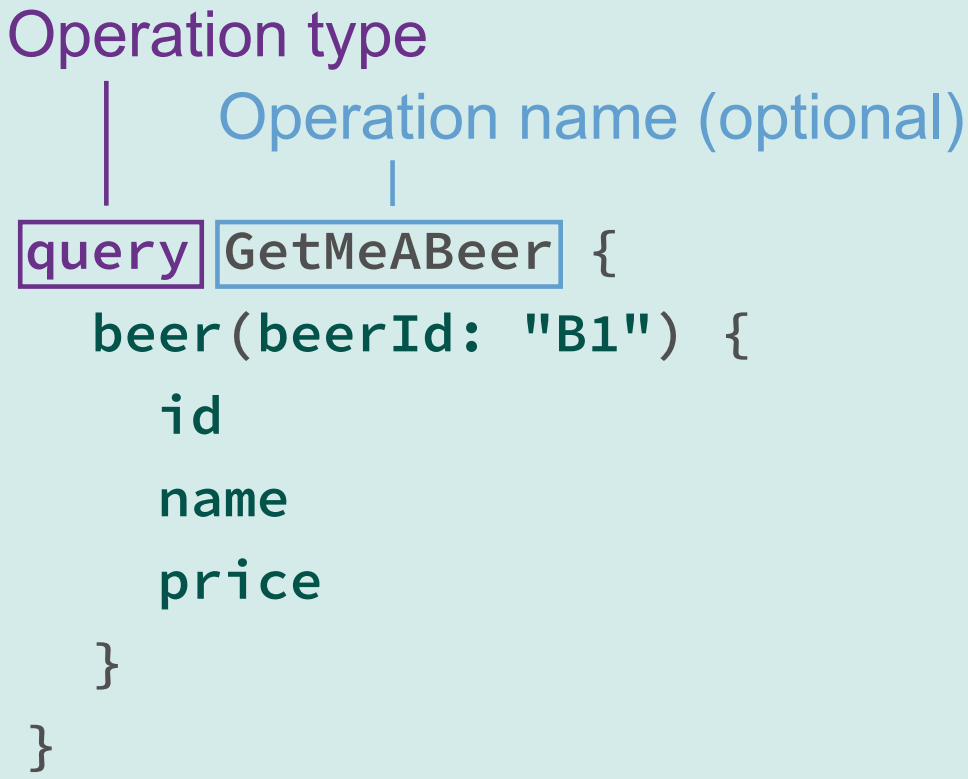
```
"data": {
  "beer": {
    "id": "B1"
    "name": "Barfüßer"
    "ratings": [
      {
        "stars": 3,
        "comment": "grate taste"
      },
      {
        "stars": 5,
        "comment": "best beer ever!"
      }
    ]
  }
}
```

- 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

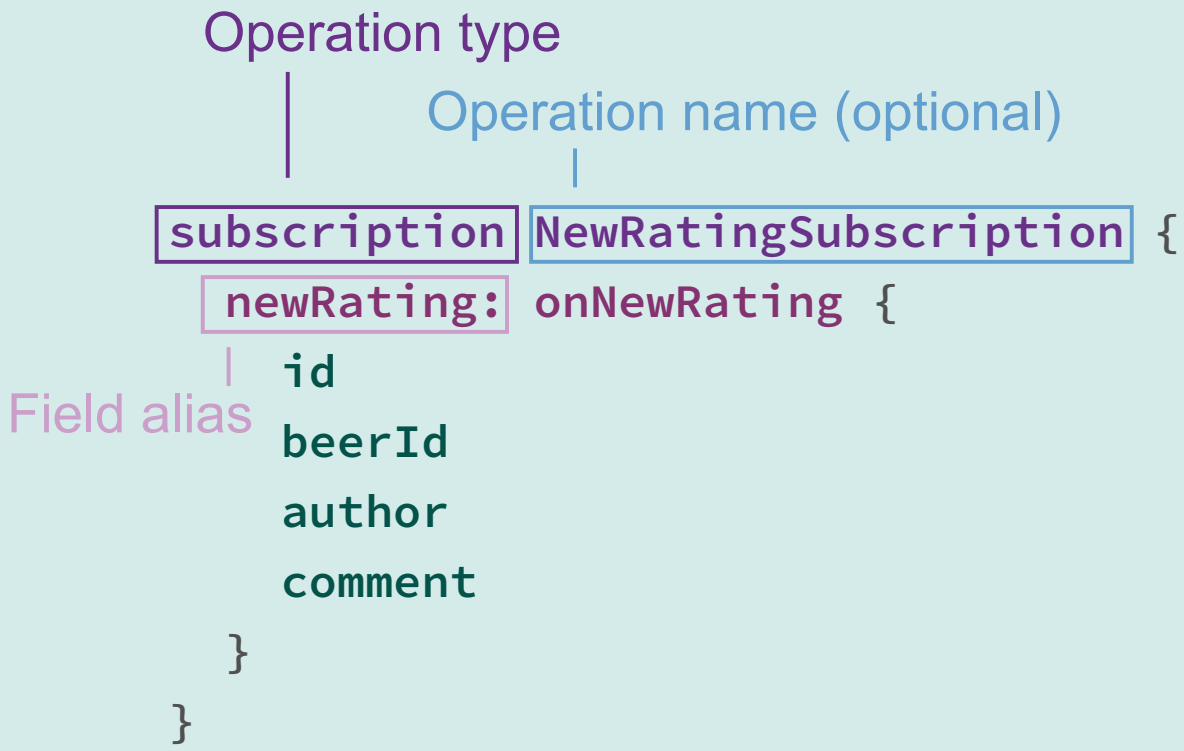
```
mutation AddRatingMutation($input: AddRatingInput!) {
  addRating(input: $input) {
    id
    beerId
    author
    comment
  }
}

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

— Variable Object

QUERY LANGUAGE: MUTATIONS

Subscription

- Automatische Benachrichtigung bei neuen Daten



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 Tripple"},
    {"name": "Viktoria Bier"}
   ]}
}
```

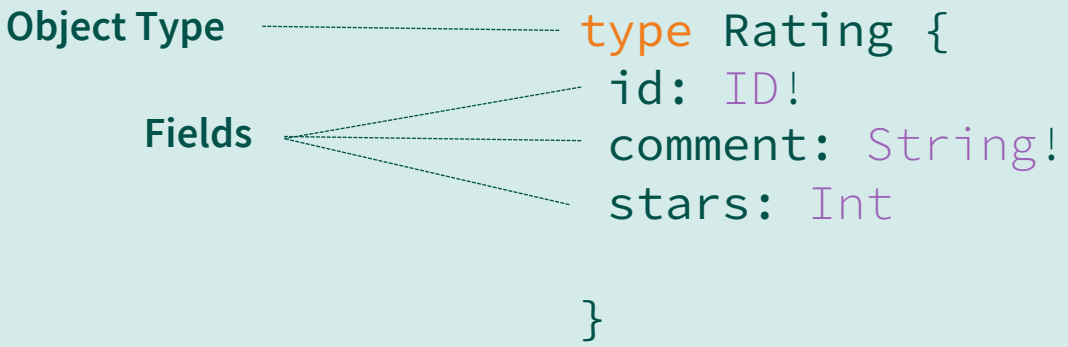
GRAPHQL SCHEMA

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)

Schema Definition per SDL



Schema Definition per SDL

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

}
```

Return Type (non-nullable)

Return Type (nullable)

Schema Definition per SDL

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

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

Referenz auf anderen Typ

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

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

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!
}
```

*"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-java: https://www.graphql-java.com/

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:

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

DataFetcher implementieren

- Beispiel: beers-Feld

```
public class BeerAdvisorDataFetchers {

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

}
```

```
type Query {
  beers: [Beer!]!

}
```

DataFetcher implementieren: environment-Parameter

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

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

}
```

```
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);
    };
  }

}
```

DataFetcher implementieren: environment-Parameter

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

```
public DataFetcher<Beer> beerFetcher() {
    return environment -> {
      DataFetchingFieldSelectionSet selection = environment.getSelectionSet();

      String beerId = environment.getArgument("beerId");
      return beerRepository.getBeer(beerId);
    };
  }
}
```

DataFetcher implementieren: environment-Parameter

- SelectionSet enthält alle abgefragten Felder
- Kann genutzt werden, um Zugriffe zu optimieren (z.B. JPA EntityGraph)

```
public DataFetcher<Beer> beerFetcher() {
    return environment -> {
      DataFetchingFieldSelectionSet selection = environment.getSelectionSet();

      EntityGraph entityGraph = entityManager.createEntityGraph(Beer.class);

      if (selection.contains("ratings/user") {
        entityGraph.addSubgraph("ratings").addSubgraph("user");
      }
      if (selection.contains("shops") {
        entityGraph.addSubgraph("shops");
      }

      String beerId = environment.getArgument("beerId");
      return beerRepository.getBeer(beerId, entityGraph);
    };
  }
}
```

DataFetcher implementieren: Mutations

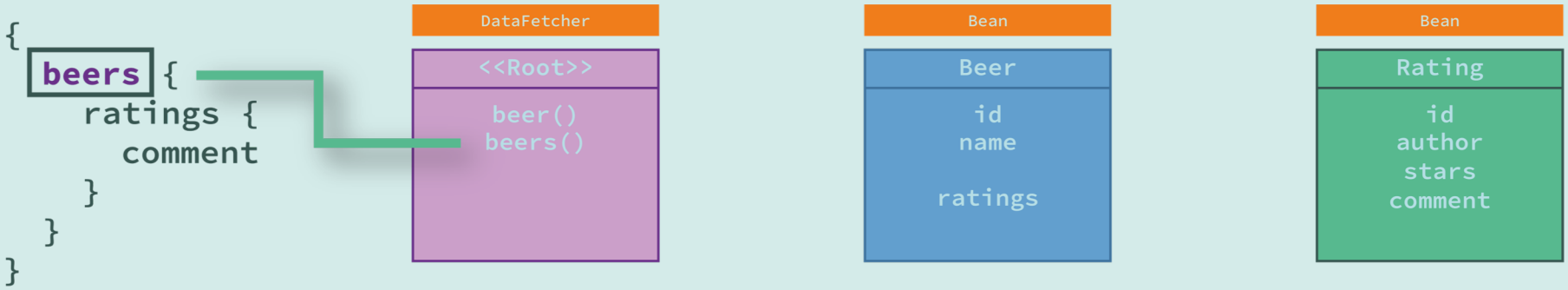
- technisch analog zu Query
- dürfen Daten verändern

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

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

      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 ERMITTLUNG ZUR LAUFZEIT

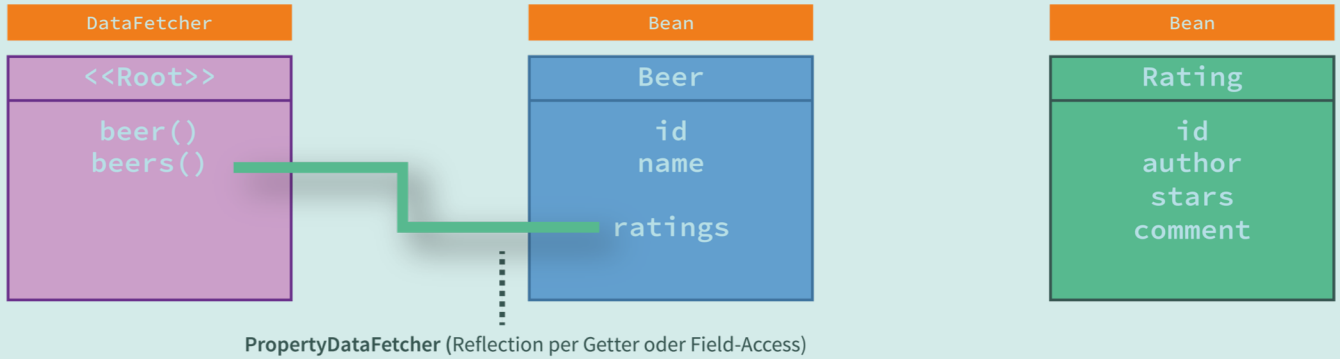
- 1. DataFetcher (wie eben implementiert)



DATEN ERMITTLUNG ZUR LAUFZEIT

- **2. Zugriff auf Bean (PropertyDataFetcher)**

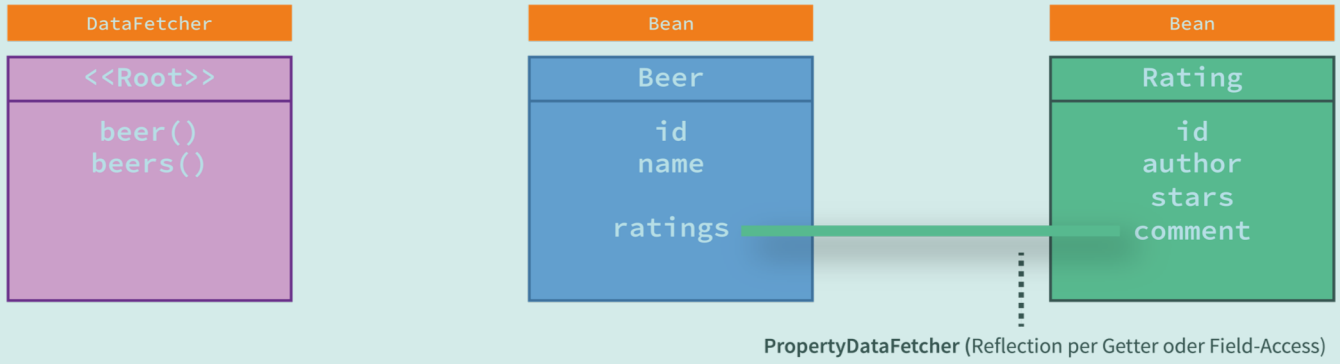
```
{
  beers {
    ratings {
      comment
    }
  }
}
```



DATEN ERMITTLUNG ZUR LAUFZEIT

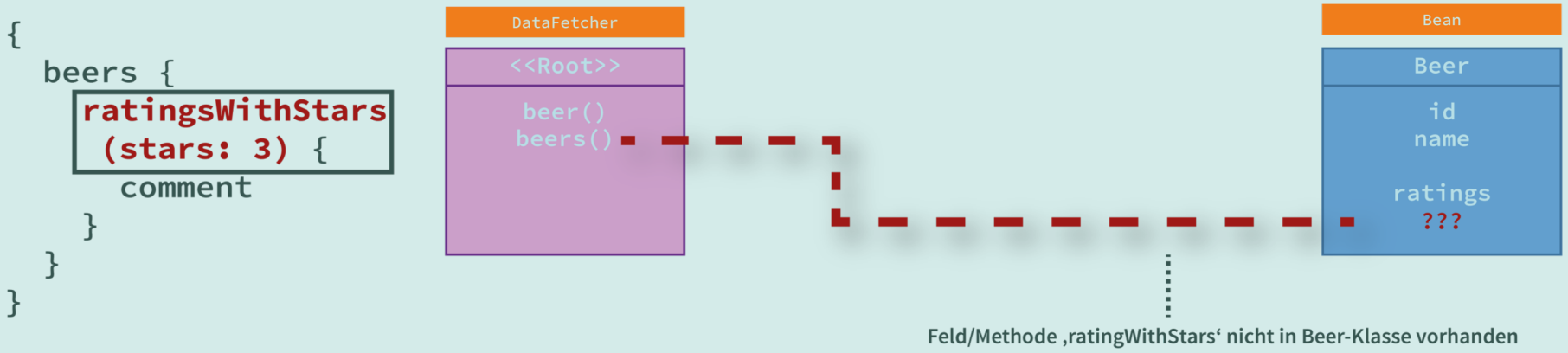
- **3. Zugriff auf Bean (PropertyDataFetcher)**

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



DATEN ERMITTLUNG ZUR LAUFZEIT

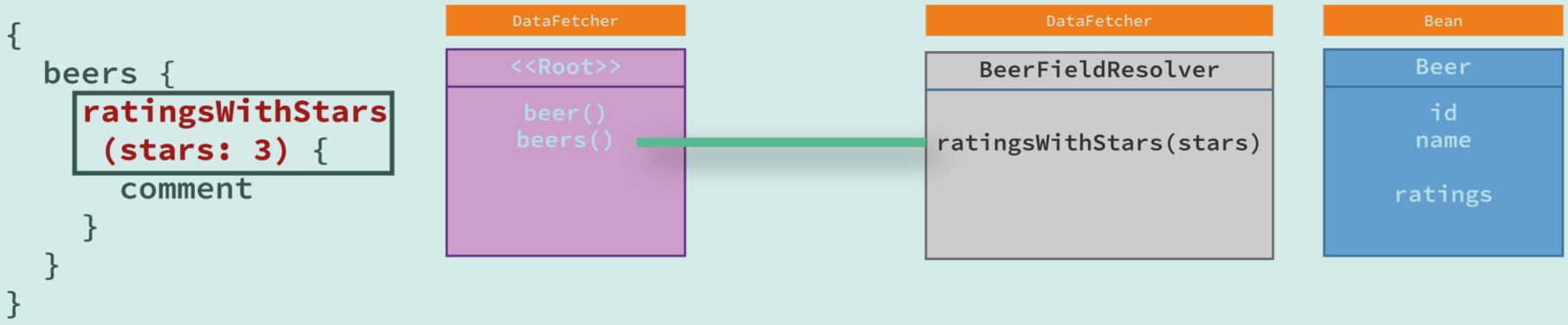
Problem: Mismatch zwischen Java-Klassen und Schema



DATEN ERMITTLUNG 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



DATA FETCHER FÜR NICHT-ROOT-FELDER

DataFetcher implementieren

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

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

```
public class BeerDataFetchers {

  public DataFetcher<List<Rating>> ratingsWithStarsFetcher() {
    return environment -> {
      Beer beer = environment.getSource();


    };
  }
}
```

DATA FETCHER FÜR NICHT-ROOT-FELDER

DataFetcher implementieren

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

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

```
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());
    };
  }
}
```

Schritt 3: 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 3: Verbinden von Schema und DataFetcher

- Im RuntimeWiring werden Schema und DataFetcher verknüpft

```
type Query {

}
```

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

    return RuntimeWiring.newRuntimeWiring()
      .type(newTypeWiring("Query")


      .build();
  }
}
```

Schritt 3: Verbinden von Schema und DataFetcher

- Im RuntimeWiring werden Schema und DataFetcher verknüpft

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

```
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();
  }
}
```

Schritt 3: Verbinden von Schema und DataFetcher

- Im RuntimeWiring werden Schema und DataFetcher verknüpft

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

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

```
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();
  }
}
```

Schritt 4: 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
    );
  }
}
```

Schritt 5: 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("query { beers { name ratings { stars } } }").build();


Map<String, Object> result = graphQL.execute(executionInput).toSpecification();
```

Schritt 5: 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);
  }

}
```

Mutations mit Resolver

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

```
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://bit.ly/javaland-graphql-example

Slides: https://bit.ly/javaland-graphql

GraphQL heise: https://bit.ly/heise-graphql-1 | https://bit.ly/heise-graphql-2

HTTPS://NILSHARTMANN.NET | @NILSHARTMANN