

**NILS HARTMANN**

<https://nilshartmann.net>

Building  
**GraphQL APIs**  
with Java

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

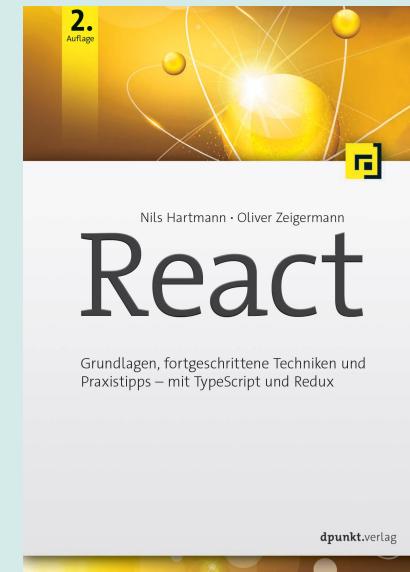
# NILS HARTMANN

nils@nilshartmann.net

**Freelance developer, architect, trainer from Hamburg**

Java  
JavaScript, TypeScript  
React  
GraphQL

Trainings, Workshops,  
Coaching



<https://reactbuch.de>

**HTTPS://NILSHARTMANN.NET**

PART 1

# GraphQL Basics

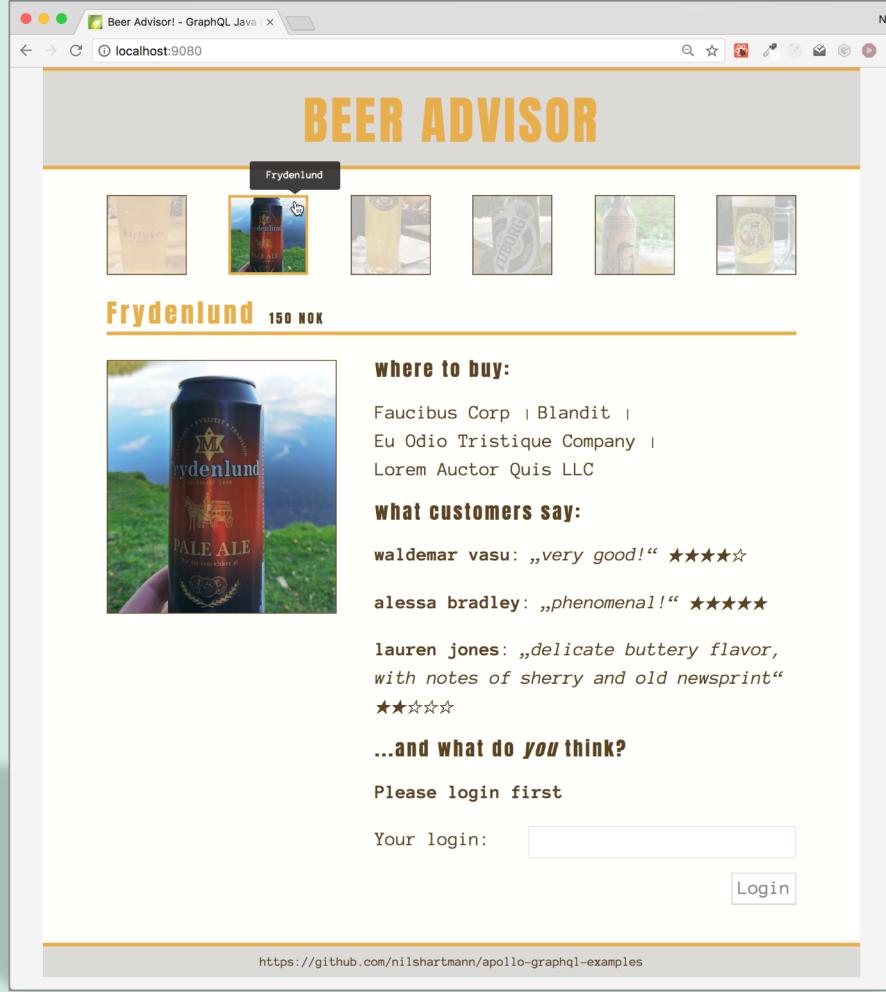
*"GraphQL is a **query language for APIs** and a **runtime for fulfilling those queries** with your existing data"*

- <https://graphql.org>

# GraphQL

*Specifikation: <https://spec.graphql.org/>*

- Published in 2015 by Facebook
- Since 2018, the GraphQL specification is developed by the GraphQL Foundation
- Spec includes:
  - Query language
  - Schema Definition Language
  - No implementation!
    - Server reference implementation: graphql-js



# Example Application

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

The screenshot shows the GraphiQL interface running on a Mac OS X desktop. The title bar says "GraphQL" and the address bar shows "localhost:9000/graphiql?operationName=BeerAppQuery&query=quer...". The left panel contains a code editor with a GraphQL query named "BeerAppQuery". The right panel displays the results of the query, which include a list of beers with their details like id, name, price, and ratings. A sidebar on the right provides a schema browser with definitions for "beers", "beer", "ratings", "ping", and "\_schema".

```
1 v query BeerAppQuery {  
2 v   beers {  
3 v     id  
4 v     name  
5 v     price  
6 v   }  
7 v   ratings {  
8 v     id  
9 v     beerId  
10 v    author  
11 v    comment  
12 v  }  
13 v }  
14 v  
15 v |  
16 v | beers  
17 v | beer  
18 v | ratings  
19 v | ping  
20 v | __schema  
21 v | __type  
22 v Returns all beers in our store
```

```
{  
  "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!"  
          }  
        ]  
      }  
    ]  
  }  
}
```

# Demo: GraphiQL

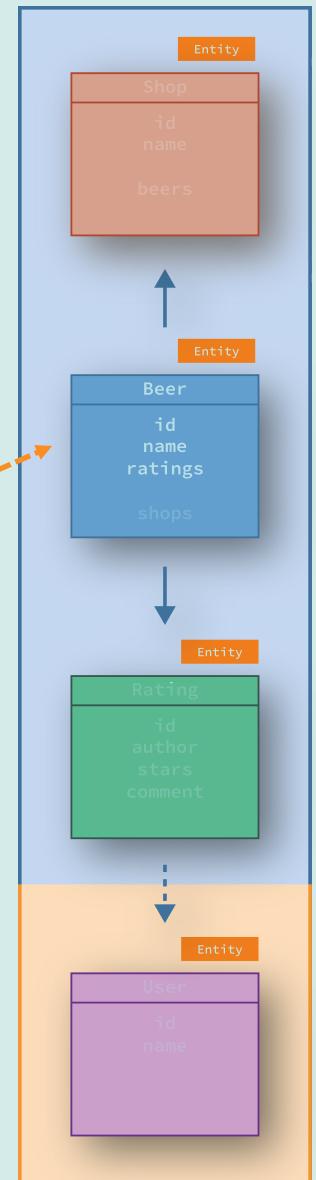
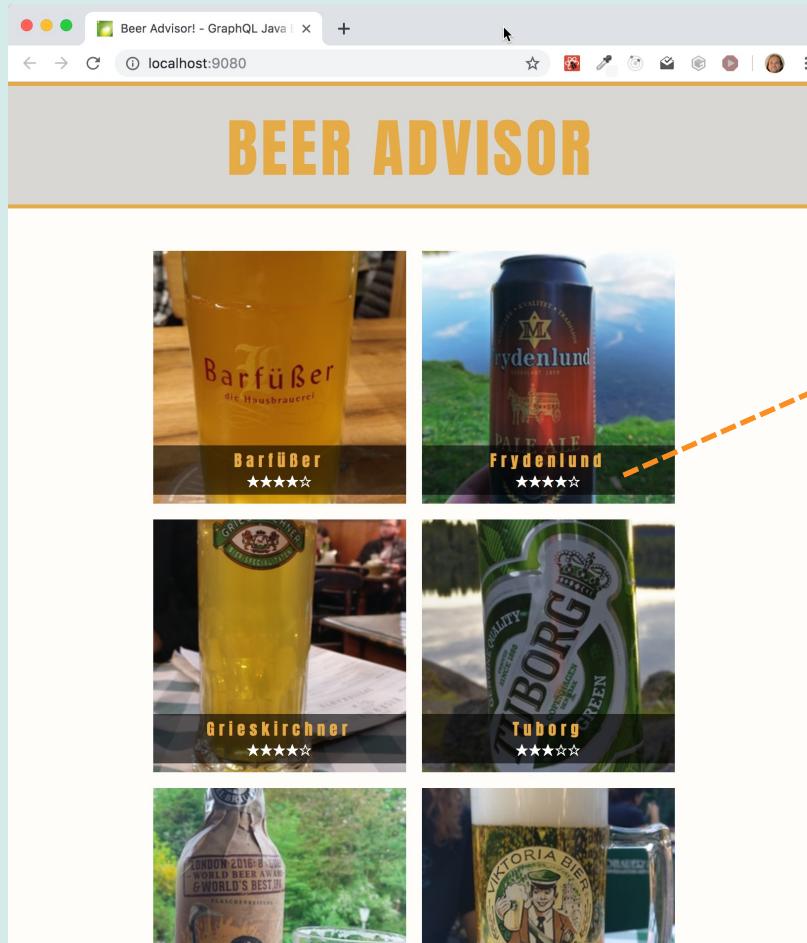
<https://github.com/graphql/graphiql>

<http://localhost:9000>

# GRAPHQL QUERIES

Select specific data needed for each use-case

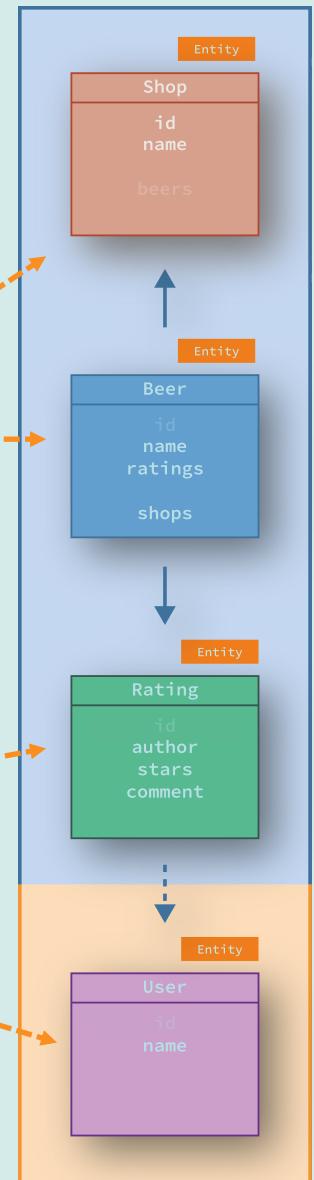
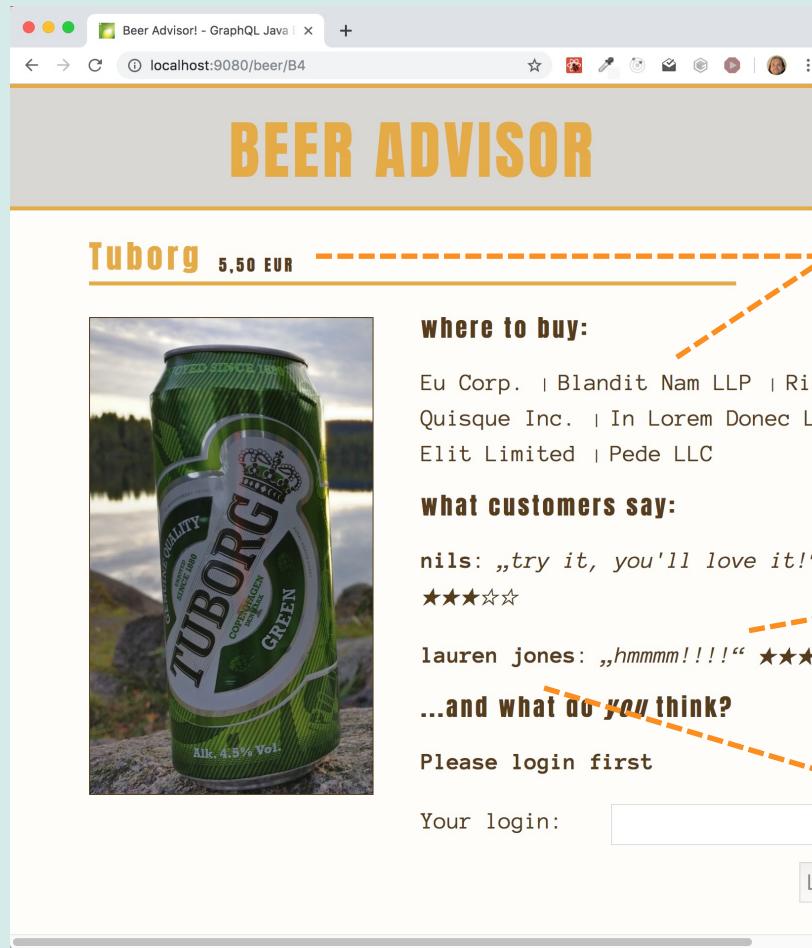
```
{ beer {  
  id  
  name  
  averageStars  
}
```



# GRAPHQL QUERIES

Select specific data needed for each use-case

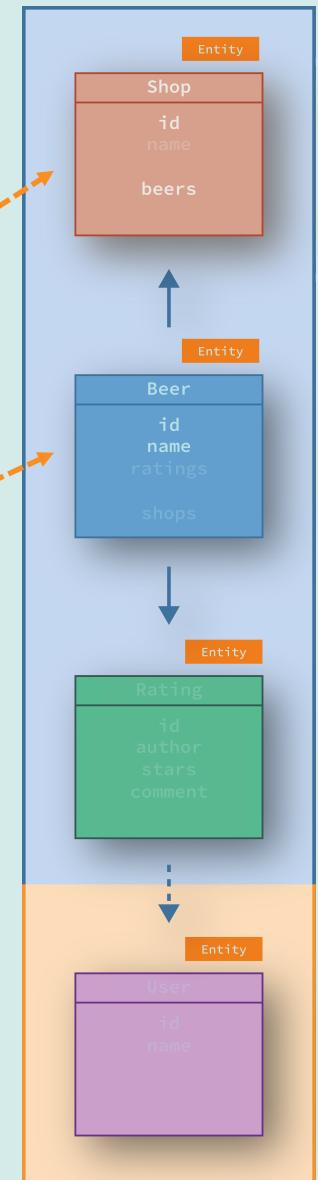
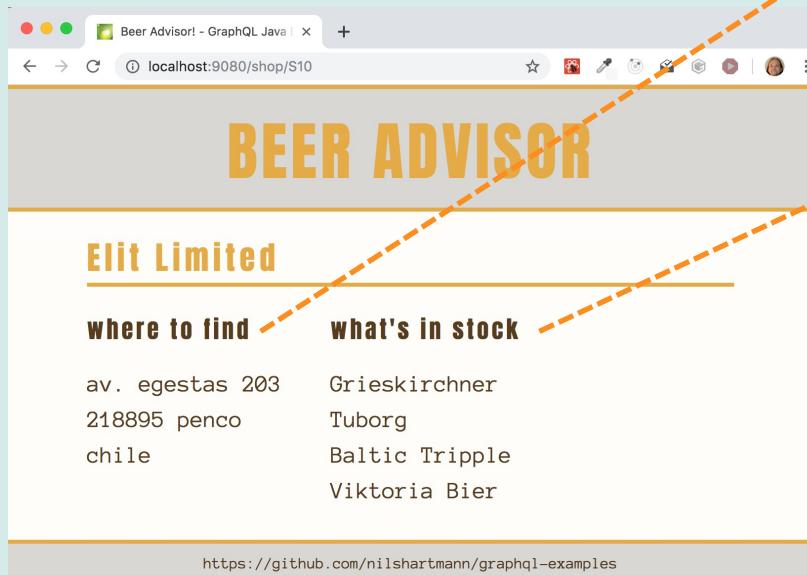
```
{ beer(beerId: "B1" {  
    name  
    price  
    ratings {  
        stars  
        comment  
        author {  
            name  
        }  
    }  
    shops { name }  
}
```



# GRAPHQL QUERIES

Select specific data needed for each use-case

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



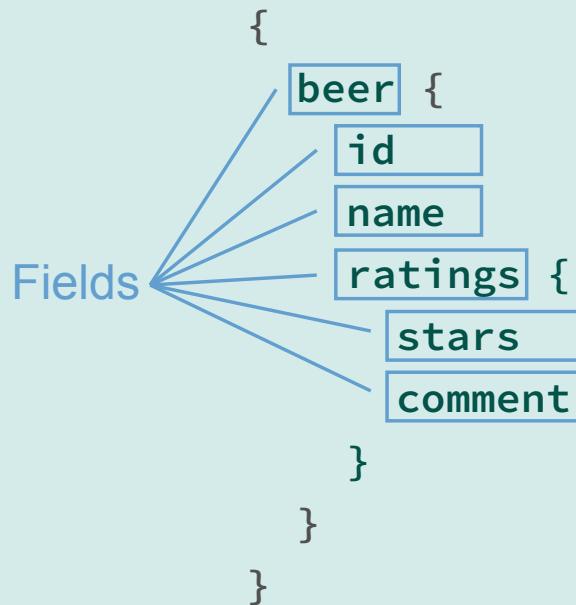
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

# GraphQL

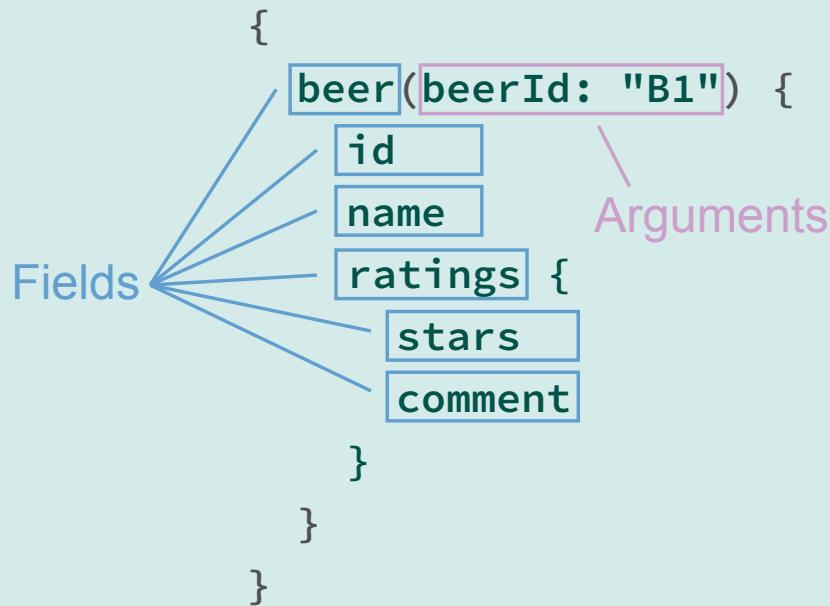
# The GraphQL Query Language

# QUERY LANGUAGE



- Structured Language to query/request data from your API
- With the language, you select **fields** from object graphs

# QUERY LANGUAGE



- Structured Language to query/request data from your API
- With the language, you select **fields** from object graphs
- Fields can have **arguments**

# QUERY LANGUAGE

## Query Result

```
{  
  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!"  
      }  
    ]  
  }  
}
```

- Identical structure as your query

# QUERY LANGUAGE: OPERATIONS

## Operation: describe, what the query should do

- query, mutation, subscription

Operation type

  |   Operation name (optional)

  |  
  query GetMeABeer {

    beer(beerId: "B1") {

      id

      name

      price

    }

}

# QUERY LANGUAGE: MUTATIONS

## Mutations

- Mutations can be used to modify data
- (would be POST, PUT, PATCH, DELETE in REST)

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

- Client of your API can subscribe to Server Events, published by the API

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

## PART II

# GraphQL Server

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

- <https://graphql.org>

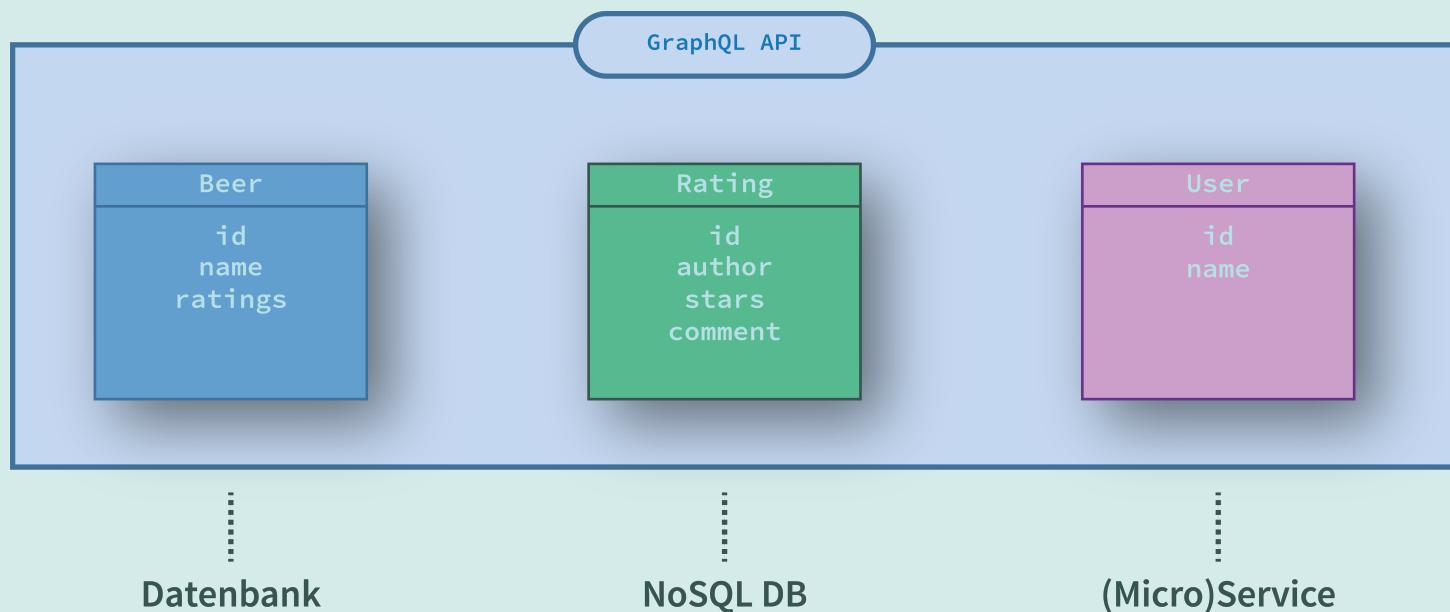
# GraphQL Server

RUNTIME (AKA: YOUR APPLICATION)

# GRAPHQL RUNTIME

**GraphQL doesn't say anything about the source of data**

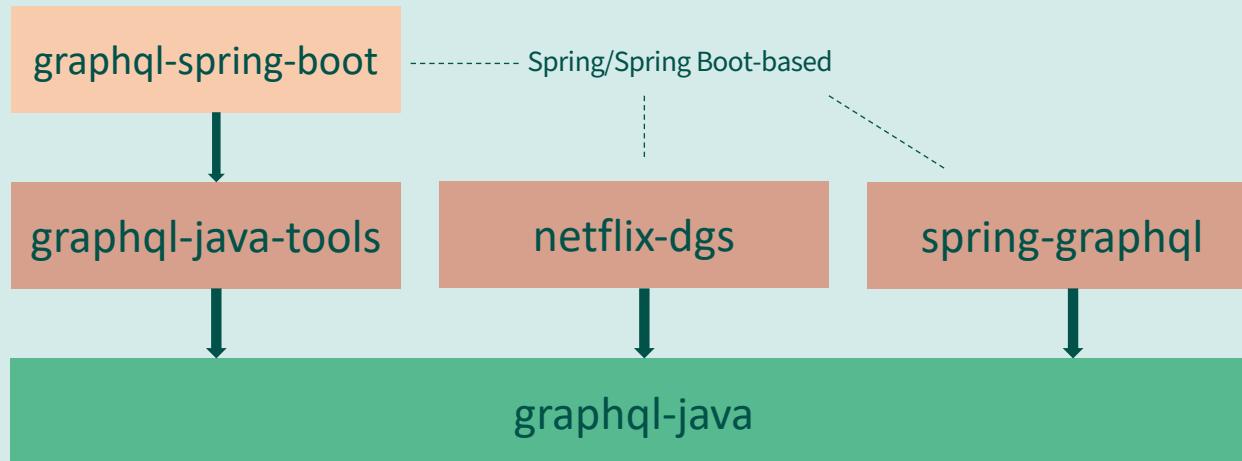
- 👉 Determining the requested data is our task
- 👉 Data might not only come from (one) database



# GRAPHQL FOR JAVA APPLICATIONS

## Two general options

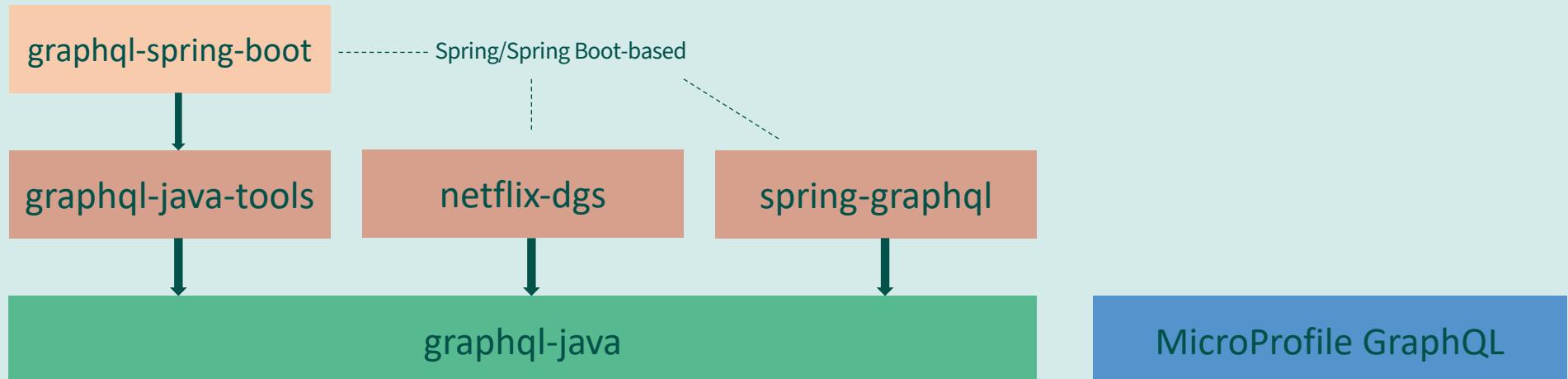
- **graphl-java**: first GraphQL framework for Java
  - Low level approach, no server etc.
  - Multiple abstractions on top including support for Spring Boot



# GRAPHQL FOR JAVA APPLICATIONS

## Two general options

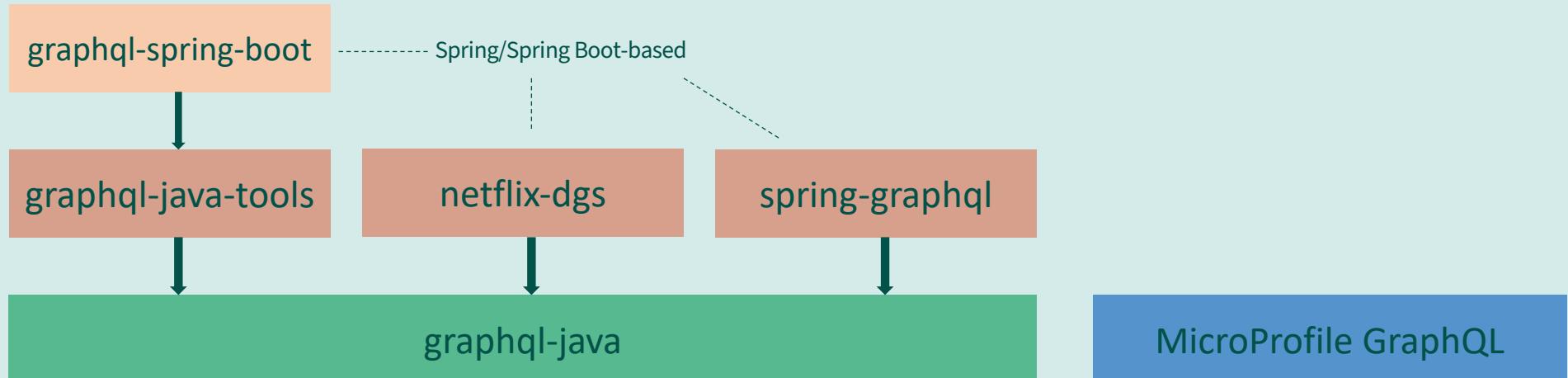
- **graphl-java**: first GraphQL framework for Java
  - Low level approach, no server etc.
  - Multiple abstractions on top including support for Spring Boot
- **MicroProfile**



# GRAPHQL FOR JAVA APPLICATIONS

## Two general options

- **graphql-java**: first GraphQL framework for Java
  - Low level approach, no server etc.
  - Multiple abstractions on top including support for Spring Boot
- **MicroProfile**
- We focus first on graphql-java and its related frameworks



# GRAPHQL FOR JAVA APPLICATIONS

## graphql-java

- <https://www.graphql-java.com/>
- Pure GraphQL implementation, environment independent (no relations to Spring or JEE)
- API is low level

# GRAPHQL SERVER WITH GRAPHQL-JAVA

## Our development tasks

1. Define the Schema of our API
  - graphql-java has "schema-first" approach
2. Implement *DataFetchers* that collect the requested data
  - This part varies in the more high level frameworks

## GRAPHQL SCHEMA

We have to describe our API in a *Schema*

- 👉 What data we provide is our decision
- 👉 How our data is provided is our decision
- 👉 We're still designing our own APIs according to our domain needs and likes

# GRAPHQL SCHEMA

## Schema

- A GraphQL API *have to* be defined with a Schema
- **Schema Definition Language** (SDL)

# GRAPHQL SCHEMA

## Defining a Schema with SDL

Object Type

Fields

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

# GRAPHQL SCHEMA

## Defining a Schema with SDL

```
type Rating {  
    id: ID! ----- Return Type (non-nullable)  
    comment: String!  
    stars: Int ----- Return Type (nullable)  
}  
}
```

# GRAPHQL SCHEMA

## Defining a Schema with SDL

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



# GRAPHQL SCHEMA

## Defining a Schema with SDL

```
type Rating { ←  
  id: ID!  
  comment: String!  
  stars: Int  
  author: User!  
}  
  
type User {  
  id: ID!  
  name: String!  
}  
  
type Beer {  
  name: String!  
  ratings: [Rating!]! ----- List / Array  
}  
}
```

# GRAPHQL SCHEMA

## Defining a Schema with 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: Entry-Points into the 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>	

## RESOLVING YOUR QUERY: DATA FETCHERS

## DATA FETCHERS

### DataFetcher

- A **DataFetcher** determines and returns the *value* for a Field
  - Required for all fields of your Root-Types (Query, Mutation)
  - For all other fields, Reflection is used (getter/setter, Maps, ...) by default

### DataFetcher

- A **DataFetcher** determines and returns the *value* for a Field
  - Required for all fields of your Root-Types (Query, Mutation)
  - For all other fields, Reflection is used (getter/setter, Maps, ...) by default
- A DataFetcher is a functional Java interface

## DataFetcher

- A **DataFetcher** determines and returns the *value* for a Field
  - Required for all fields of your Root-Types (Query, Mutation)
  - For all other fields, Reflection is used (getter/setter, Maps, ...) by default
- A DataFetcher is a functional Java interface

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

# DATAFETCHER

## Implementing DataFetchers

- Example: A simple field

Schema Definition

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

# DATAFETCHER

## Implementing DataFetchers

- Example: A simple field

Schema Definition

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

Query

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

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

# DATAFETCHER

## Implementing DataFetchers

- Example: A simple field

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);  
        }  
    };  
}
```

# DATAFETCHER

## Implementing DataFetchers

- Example: A simple field

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);  
        }  
    };  
}
```

Assume Beer Pojo  
contains "name" and "price" property

# DATAFETCHER

## DataFetcher: Mutations

- technically the same as queries, but you're allowed to modify data here

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

- Same as DataFetchers for Query, but must return Reactive Streams Publisher
- Typically used in Web-Clients with 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;
        }
    };
}
```

### DataFetcher for own Types (not Root Types)

- By default graphql-java uses a "PropertyDataFetcher" for all fields that are not on Root Types
- PropertyDataFetcher uses Reflection to return the requested data from your Pojo
- (Fields not defined in your schema, but part of your Pojo are never returned to the client!)
- Your returned Pojo and GraphQL schema might not match
  - Different/missing fields

# OBJECT GRAPHS

## DataFetcher for own Types (not Root Types)

- Example: There is no field "shops" on our Beer class

```
query {  
  beer(id: 1) {  
    name  
    shops {  
      name  
    }  
  }  
}
```

no 'shops' here 🤔

```
public class Beer {  
  String id;  
  String name;  
  ...  
}
```

# OBJECT GRAPHS

## DataFetcher for own Types (not Root Types)

- You can write DataFetcher for *all* fields in your GraphQL API
- Non-Root Fetcher works the same, as DataFetchers for Root-Fields
- They receive their parent object as "Source"-Property from the DataFetchingEnvironment

```
query {  
  beer(id: 1) {  
    name  
    shops {  
      name  
    }  
  }  
}
```

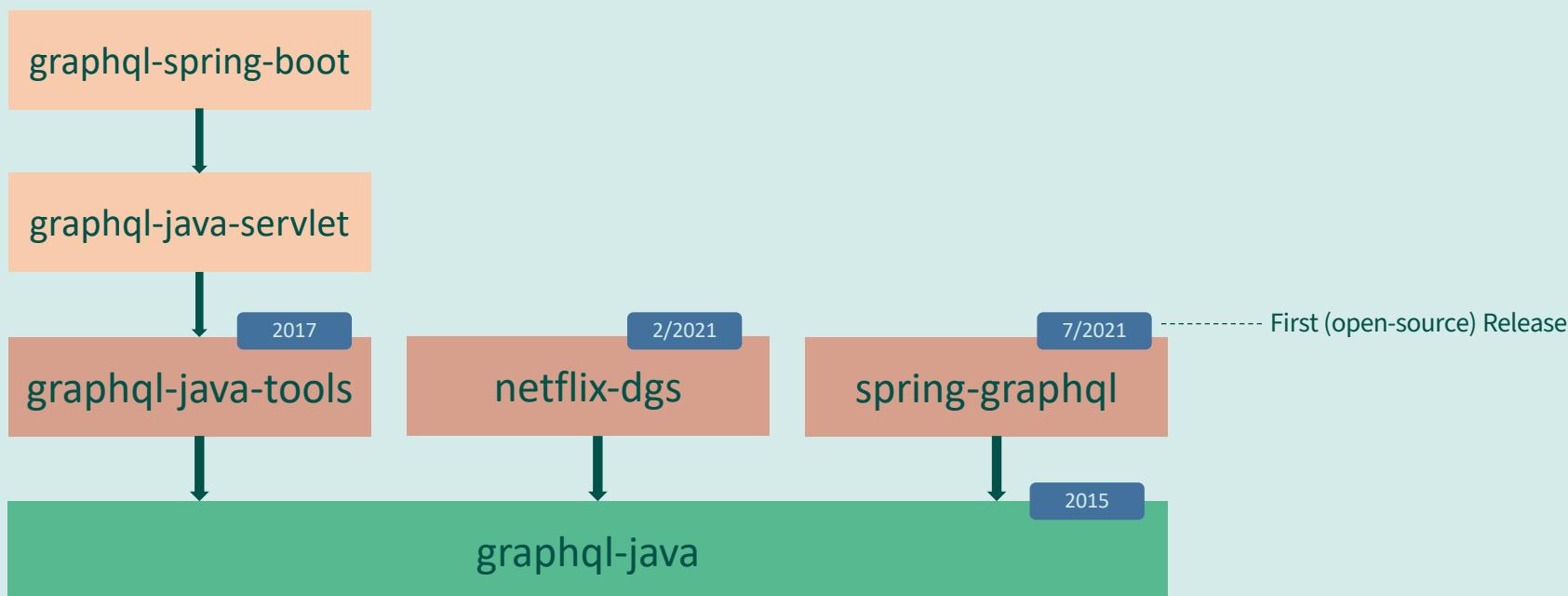
```
public class BeerDataFetchers {  
  DataFetcher<List<Shop>> shops = new DataFetcher<>() {  
    public String get(DataFetchingEnvironment env) {  
      Beer parent = env.getSource();  
      String beerId = parent.getId();  
  
      return shopRepository.findShopsSellingBeer(beerId);  
    }  
  };  
}
```

# HIGHER LEVEL FRAMEWORKS

# HIGHER LEVEL FRAMEWORKS

## All share similiar ideas

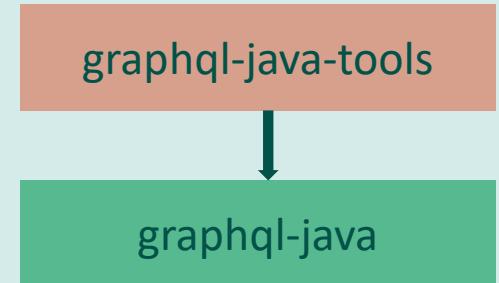
- Do not use DataFetcher directly, but abstraction
- graphql-java-tools not Spring/JEE related, but there are adapters
- Netflix DGS and spring-graphql look very similiar
  - Both target Spring Boot



# GRAPHQL FOR JAVA APPLICATIONS

## graphql-java-tools

- <https://github.com/graphql-java-kickstart/graphql-java-tools>
- Implement your API using POJOs



## Resolver with graphql-java-tools

- Example: Resolver for Root-Field

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

## Resolver with `graphql-java-tools`

- Example: Mutation and Input Types
- Complex arguments (GraphQL input types) are deserialized to POJO instances and passed to your resolver method

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

## Resolver with `graphql-java-tools`

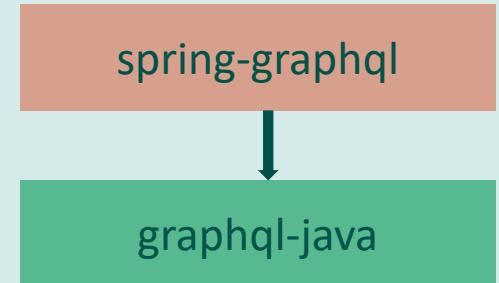
- Examples: Resolvers for fields not on root types
- The parent object (`getSource` in `graphql-java`) will be passed as first method parameter

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

# GRAPHQL FOR JAVA APPLICATIONS

## spring-graphql

- <https://docs.spring.io/spring-graphql/docs/current-SNAPSHOT/reference/html/>
  - "Official" Spring solution for GraphQL
  - Takes graphql-java and combines it with Spring Boot (concepts)
  - Exposes Endpoint using Spring WebMVC, Spring WebFlux
  - Supports subscriptions using WebSockets
- Still in beta status, released only some weeks ago in July



# GRAPHQL FOR JAVA APPLICATIONS

## Annotated Controllers

- Annotation-based programming model, similar to REST controllers

**@Controller**

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

**@QueryMapping**

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

Mapping to Schema via Naming convention

**@MutationMapping**

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

}

# GRAPHQL FOR JAVA APPLICATIONS

## Annotated Controllers

- Annotation-based programming model, similar to REST controllers

### @Controller

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

### @QueryMapping

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

Arguments via Method Parameters

### @MutationMapping

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

}

# GRAPHQL FOR JAVA APPLICATIONS

## Annotated Controllers

- Annotation-based programming model, similar to REST controllers

### @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);  
}
```

Parent-Objects as Method Parameters

### @SchemaMapping

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

# GRAPHQL FOR JAVA APPLICATIONS

## Netflix Domain Graph Service Framework (DGS)

- <https://netflix.github.io/dgs/>
- Published first in February 2021
- Based on Spring Boot and graphql-java

## Netflix Domain Graph Service Framework (DGS)

- <https://netflix.github.io/dgs/>
- Published first in February 2021
- Based on Spring Boot and graphql-java

Features not available in spring-graphql:

- Code-Generator for Gradle and Maven creates from your schema definition Java classes
- Supports Subscriptions
- Support for Apollo Federation (combining multiple APIs into one)

## Netflix Domain Graph Service Framework (DGS)

- <https://netflix.github.io/dgs/>
- Published first in February 2021
- Based on Spring Boot and graphql-java

Features not available in spring-graphql:

- Code-Generator for Gradle and Maven creates from your schema definition Java classes
- Supports Subscriptions
- Support for Apollo Federation (combining multiple APIs into one)
- Difficult to say, how DGS and spring-graphql will differ, what will be the "better" framework

## MicroProfile GraphQL

- Annotation based programming model
- Code-first: Schema is derived from your Code (Entities and Components)
  - (as opposite to Schema-first in graphql-java)

MicroProfile GraphQL

# MICROPROFILE GRAPHQL

**Example:** Use Pojos to define your Types

```
@Type  
 @Name("beer")  
 @Description("Represents a Beer that can be rated")  
 public class Beer {  
  
     @Ignore  
     private String primaryKey;    // not part of the GraphQL API  
  
     @NonNull  
     private String name;  
     private int price;  
     ...  
 }
```

# MICROPROFILE GRAPHQL

**Example:** Use Components to define your Queries and Mutations

```
@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);
    }

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

# GRAPHQL FOR JAVA APPLICATIONS

## Conclusion: what framework should I pick?

- Spring World: spring-graphql or Netflix DGS
  - Future will tell what is better (might be that Netflix DGS will be based on spring-graphql but adding more features)

# GRAPHQL FOR JAVA APPLICATIONS

## Conclusion: what framework should I pick?

- Spring World: spring-graphql or Netflix DGS
  - Future will tell what is better (might be that Netflix DGS will be based on spring-graphql but adding more features)
- JEE
  - graphql-java or graphql-java-tools and graphql-java-servlet

# GRAPHQL FOR JAVA APPLICATIONS

## Conclusion: what framework should I pick?

- Spring World: spring-graphql or Netflix DGS
  - Future will tell what is better (might be that Netflix DGS will be based on spring-graphql but adding more features)
- JEE
  - graphql-java or graphql-java-tools and graphql-java-servlet
- MicroProfile
  - Microprofile GraphQL



# Thank you!

Slides: <https://react.schule/jcon-graphql>

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

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