NILS HARTMANN
https://nilshartmann.net

Building GraphQL APIs with Java

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

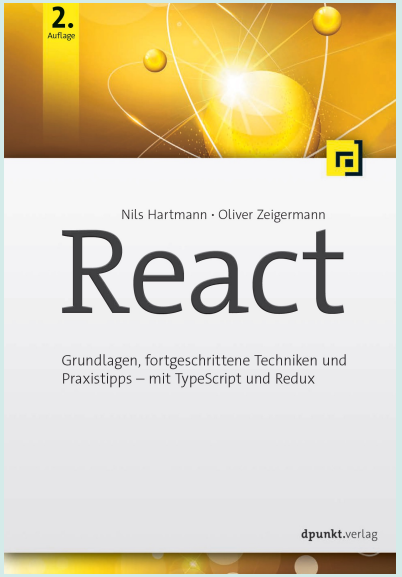
NILS HARTMANN

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"*

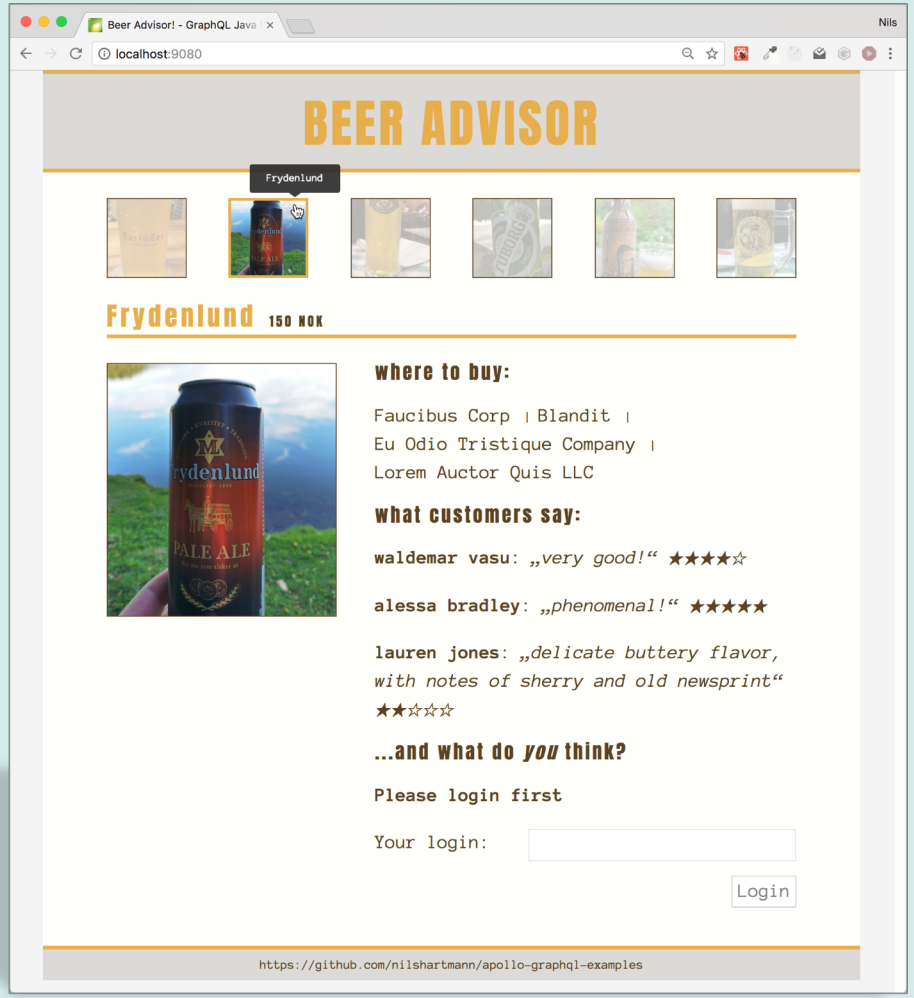
- https://graphql.org

GraphQL

Specifikation: https://spec.graphql.org/

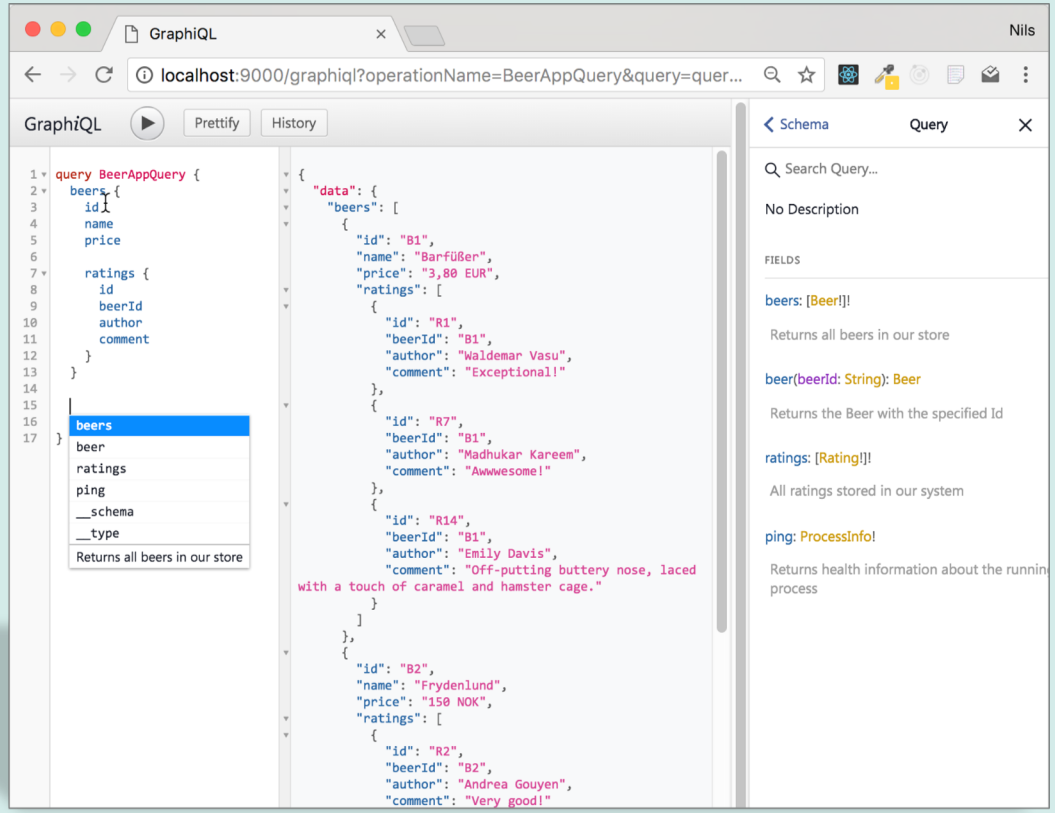
- Published in 2015 by Facebook
- Since 2018, the GraphQL specification is developed by the GraphQL Foundation

- 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



Demo: GraphiQL

https://github.com/graphql/graphiql

http://localhost:4000

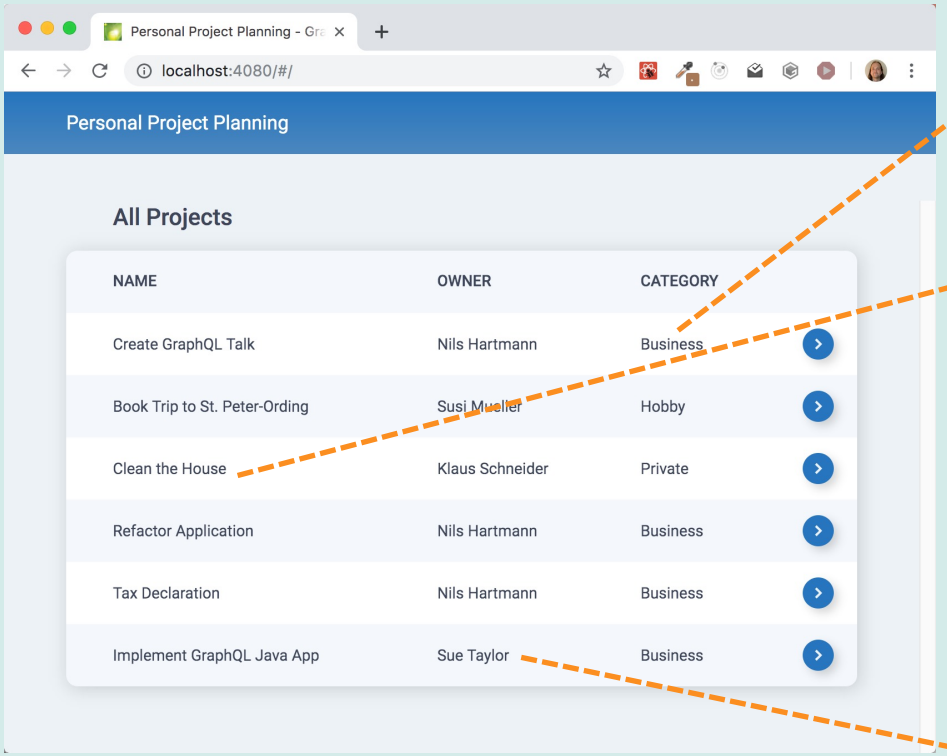
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

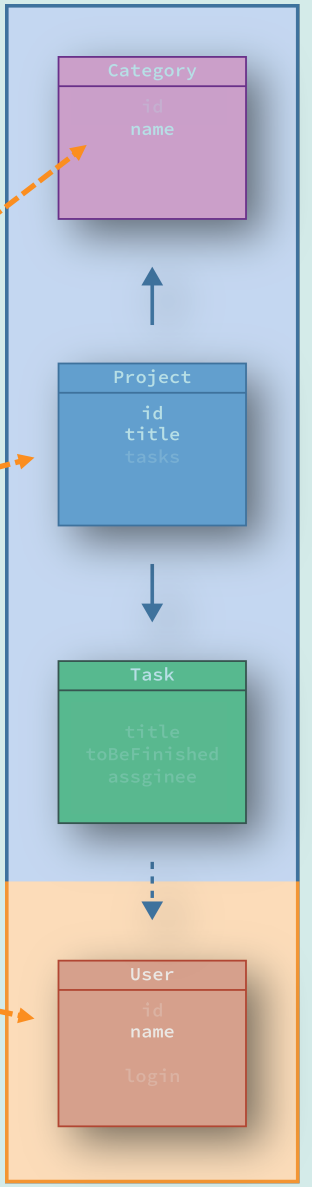
- https://graphql.org

GraphQL

Use-case specific queries

```
{ projects {
    id
    title
    owner { name }
    category { name }
  }
}
```

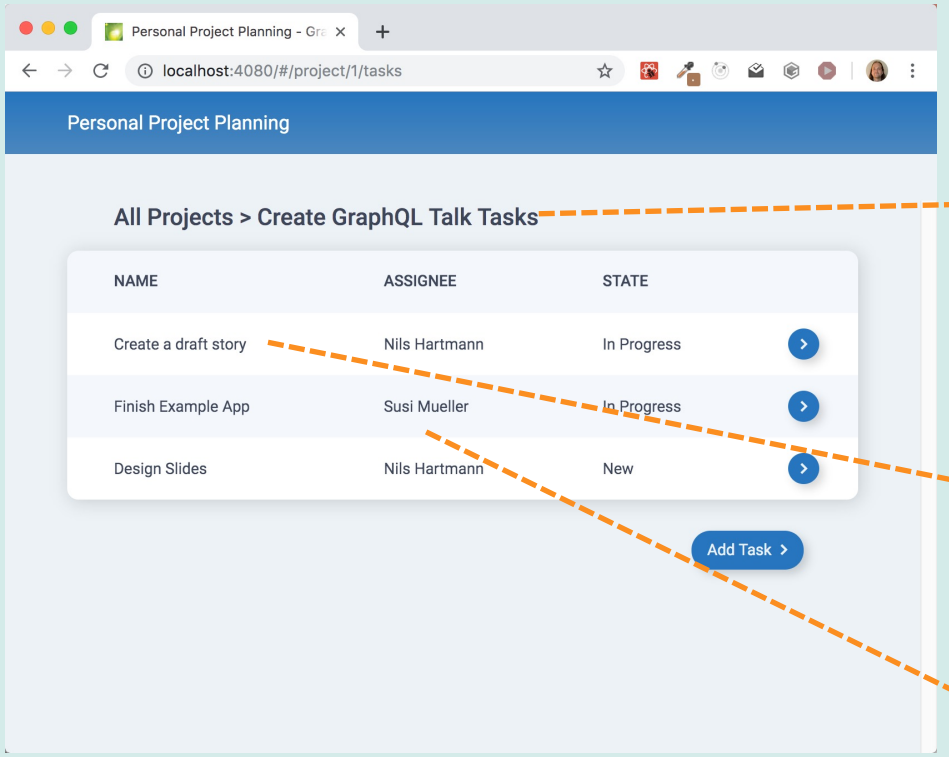


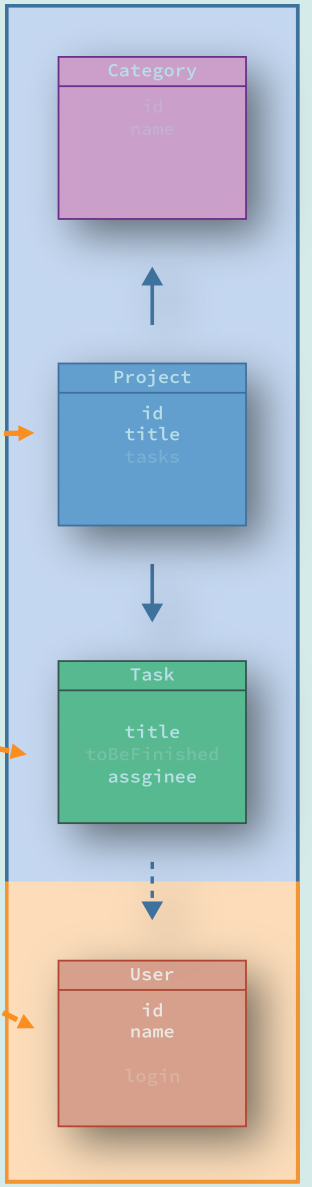


GRAPHQL EINSATZSZENARIEN

Use-case specific queries 2

```
{ project(...) {
    title
    tasks {
      name
      assignee { name }
      state
    }
  }
}
```

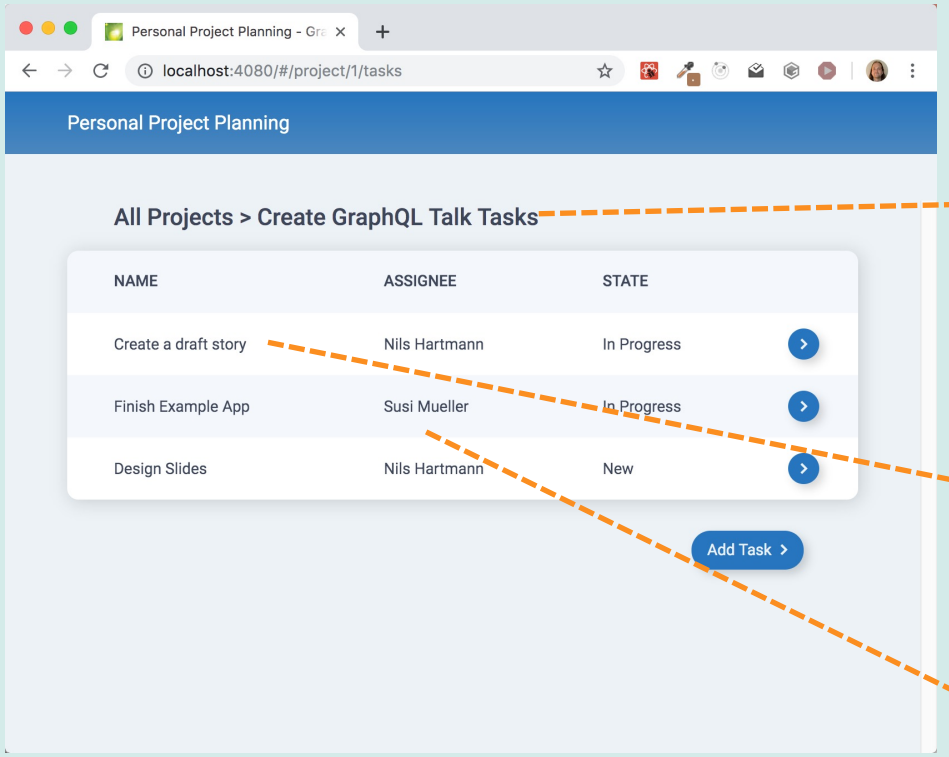


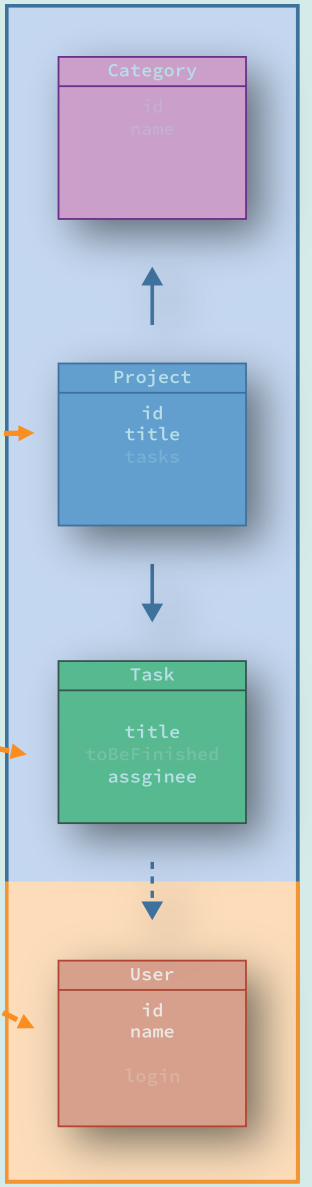


GRAPHQL EINSATZSZENARIEN

Use-case specific queries 2

```
{ project(...) {
    title
    tasks {
      name
      assignee { name }
      state
    }
  }
}
```





Data is queried and requested, not endpoints

GRAPHQL APIS

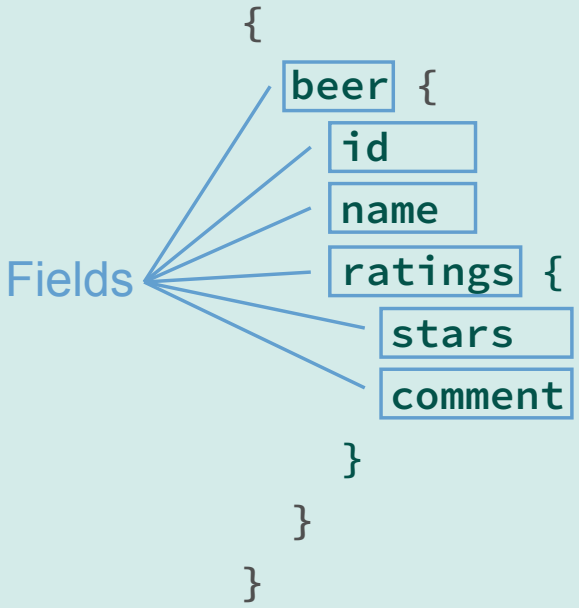
GraphQL makes no statement about how our API looks like

- 👉 *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 needs and likes***

GraphQL is "only" technology

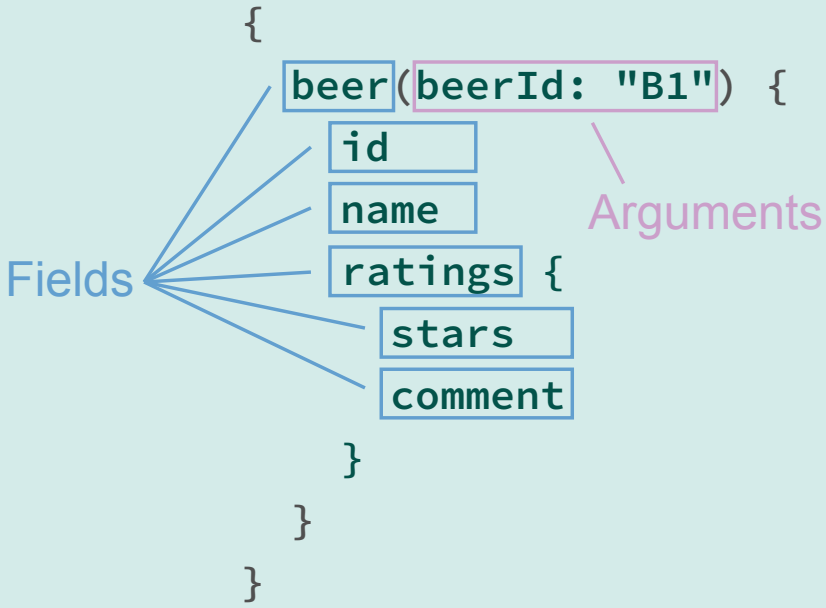
The GraphQL Query Language

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 Result

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

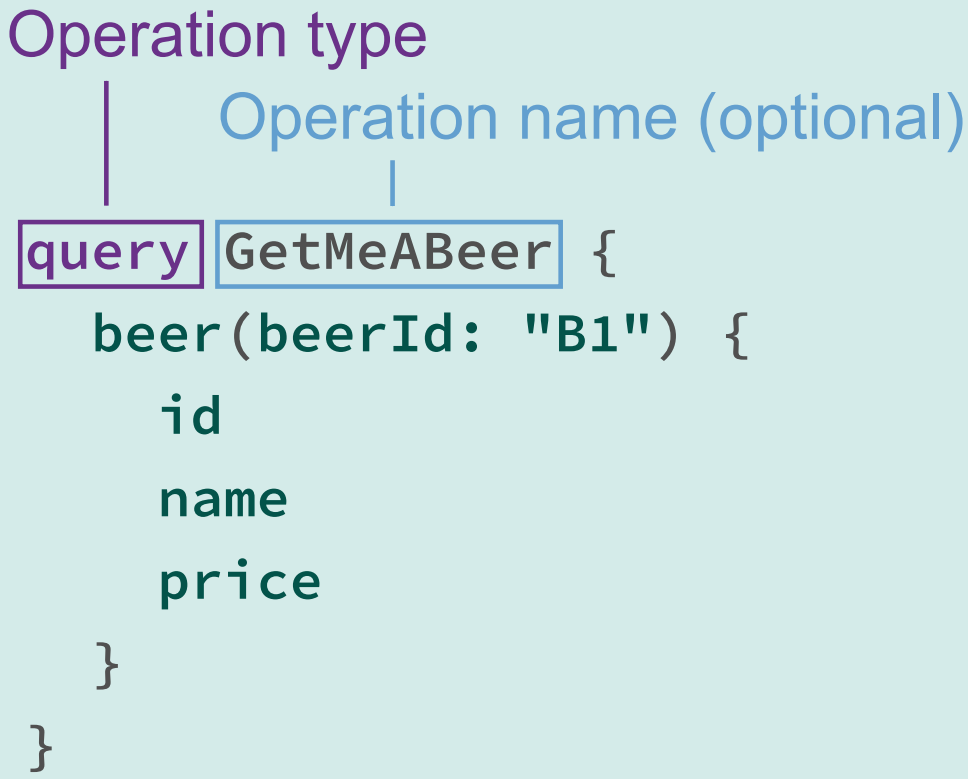
```
"data": {
  "beer": {
    "id": "B1"
    "name": "Barfüßer"
    "ratings": [
      {
        "stars": 3,
        "comment": "grate taste"
      },
      {
        "stars": 5,
        "comment": "best beer ever!"
      }
    ]
  }
}
```

- Identical structure as your query

QUERY LANGUAGE: OPERATIONS

Operation: describe, what the query should do

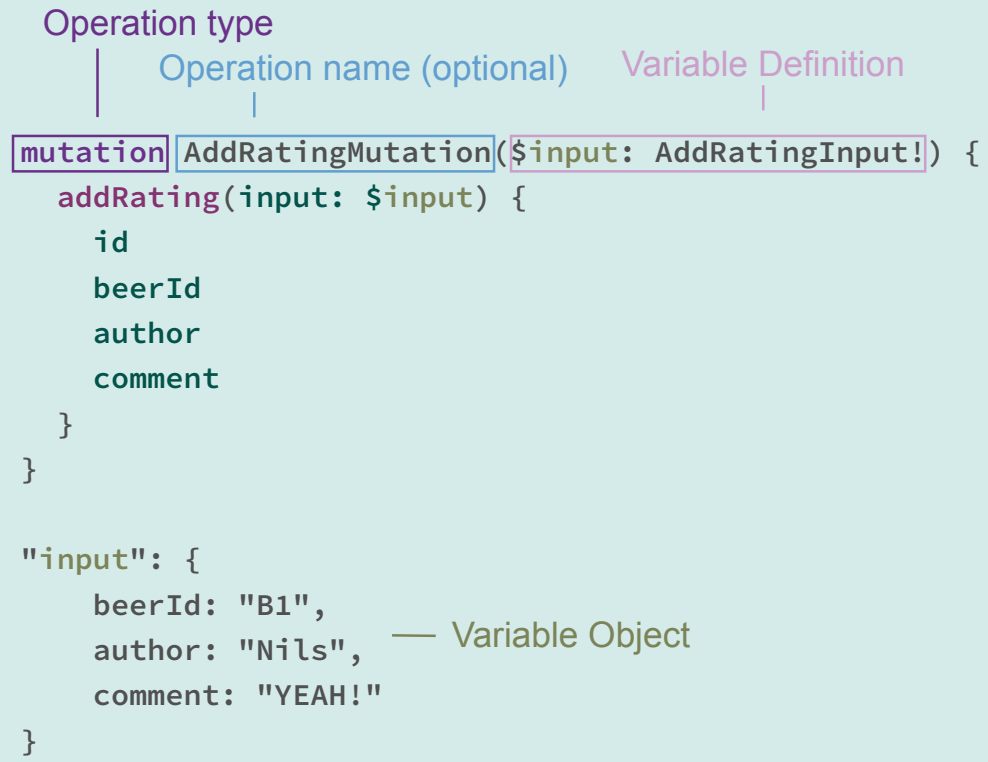
- query, mutation, subscription



QUERY LANGUAGE: MUTATIONS

Mutations

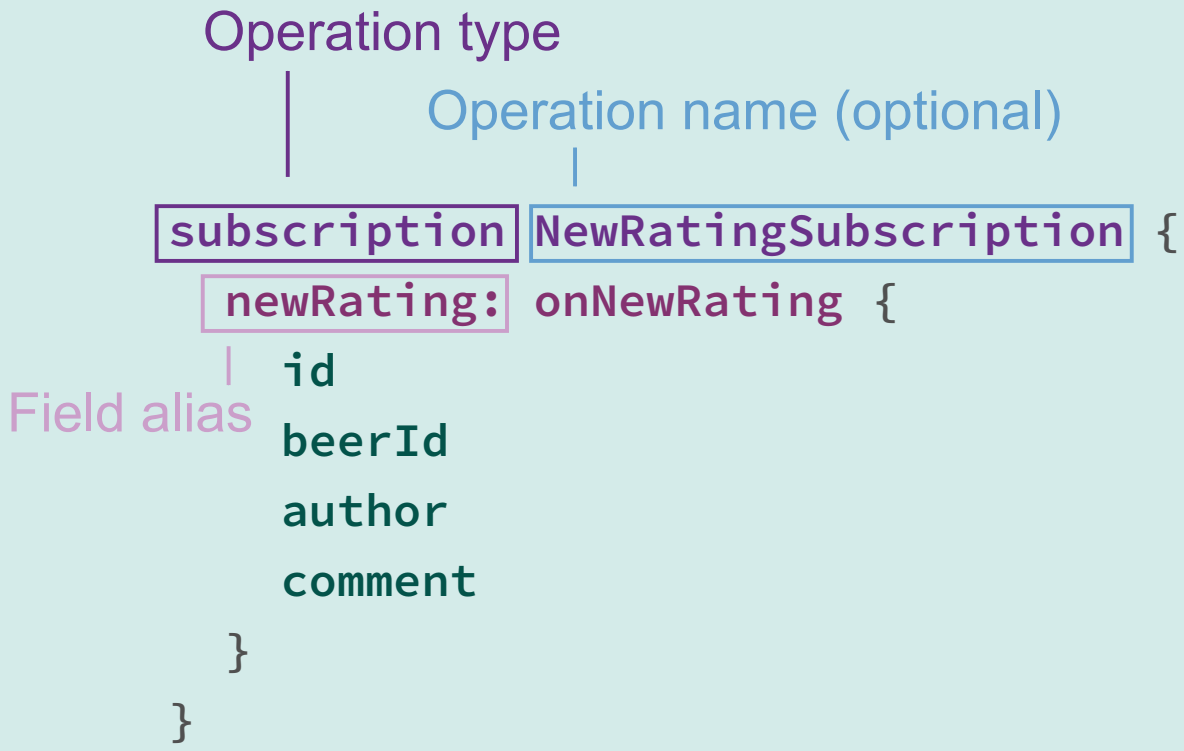
- Mutations can be used to modifiy data
- (would be POST, PUT, PATCH, DELETE in REST)



QUERY LANGUAGE: MUTATIONS

Subscription

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



EXECUTING QUERIES

Queries can be executed by HTTP requests

- Normaly using HTTP POST
- Only one endpoint for all queries
- Response usualy HTTP 200 (besides technical ones)

👉 Differes enormous from REST...

```
$ curl -X POST -H "Content-Type: application/json" \
    -d '{"query":"{ beers { name } }"}' \
    http://localhost:9000/graphql
```

```
{"data":
  {"beers": [
    {"name": "Barfüßer"},
    {"name": "Frydenlund"},
    {"name": "Grieskirchner"},
  ]}
}
```

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

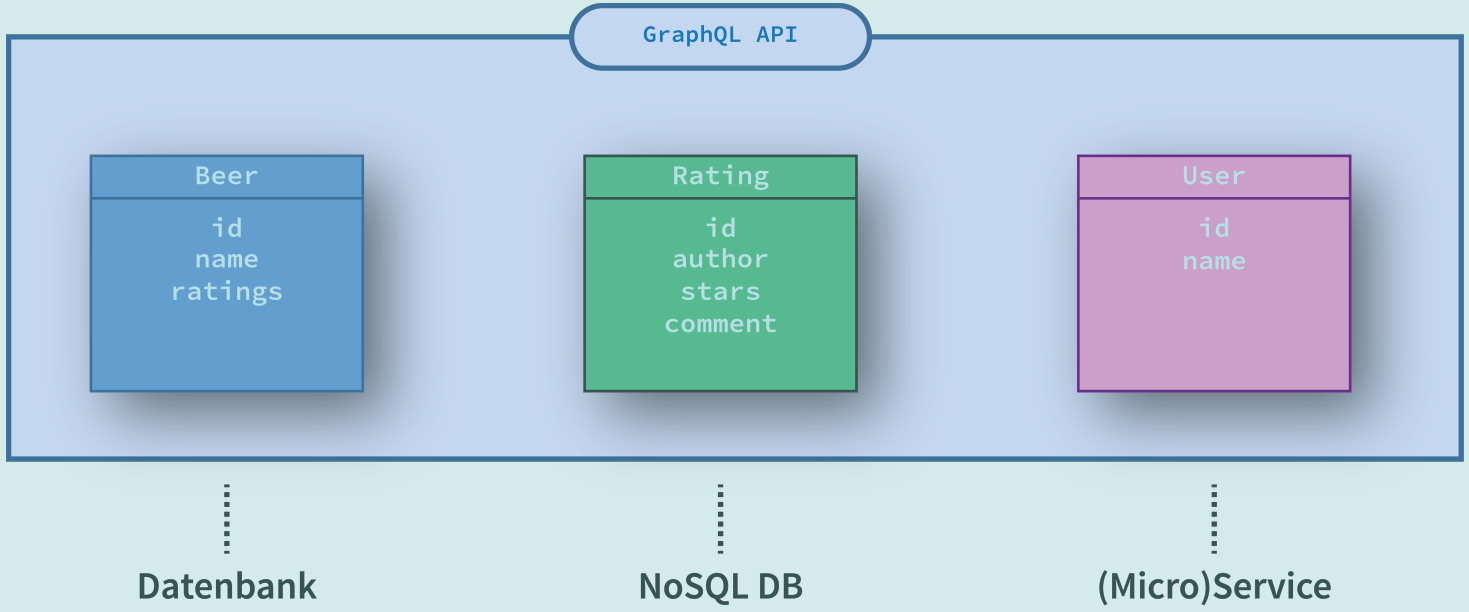
RUNTIME (AKA: YOUR APPLICATION)

GRAPHQL RUNTIME

GraphQL doesn't say anthing about the source of data

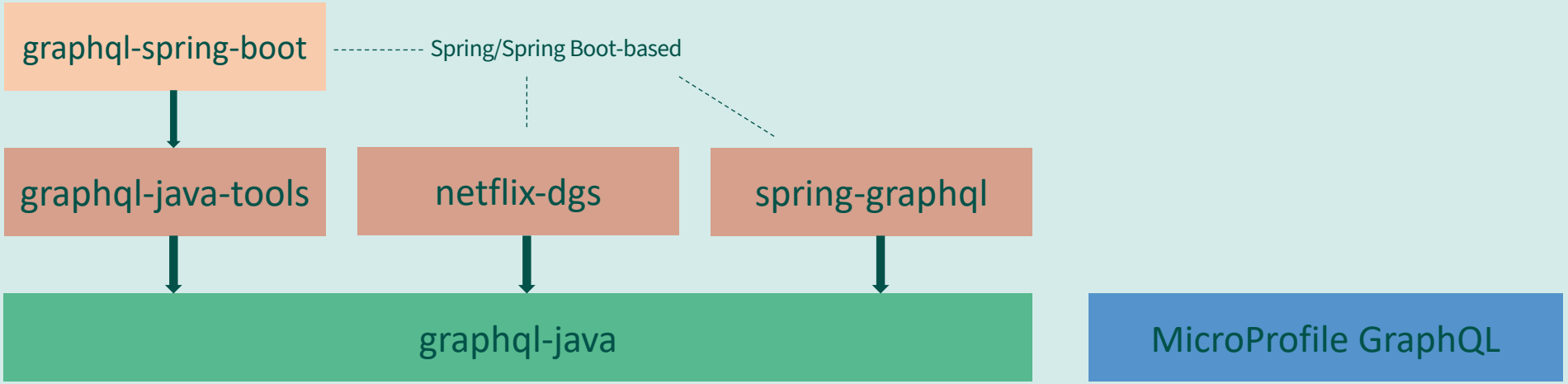
👉 *Determining the requested data is our task*

👉 *Data might not only come from (one) database*



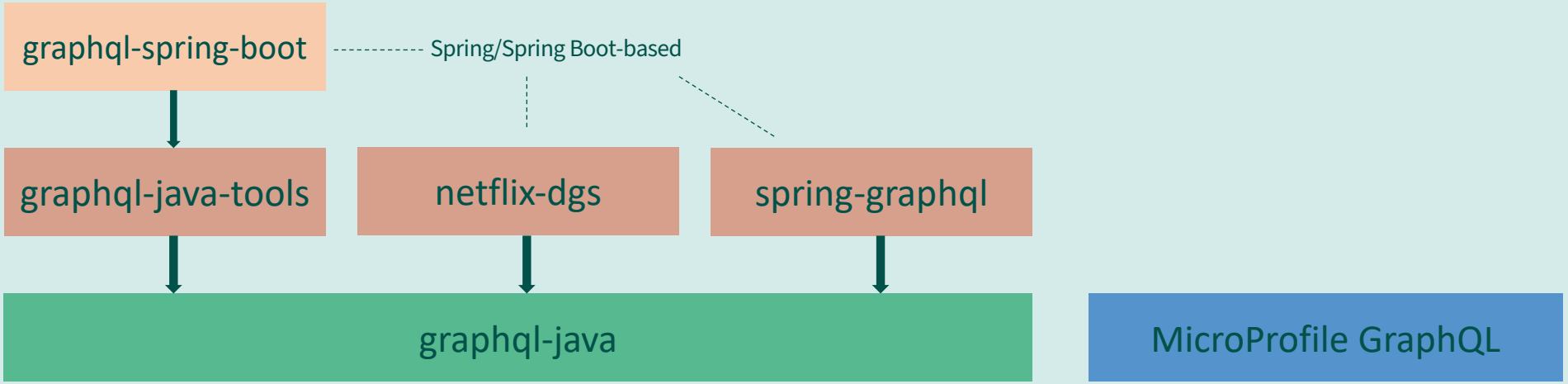
Two general options

- graphl-java: first GraphQL framework for Java
  - Low level approach, no server etc.
  - Multiple abstractions on top including support for Spring Boot
- MicroProfile first released 2020



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 first released 2020
- 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

Our development tasks

1. Define the Schema of our API
   - graphql-java has "schema-first" approach
   - Works the same also in the more high level frameworks

2. Implement *DataFetchers* that collect the requested data
   - This part varies in the more high level frameworks

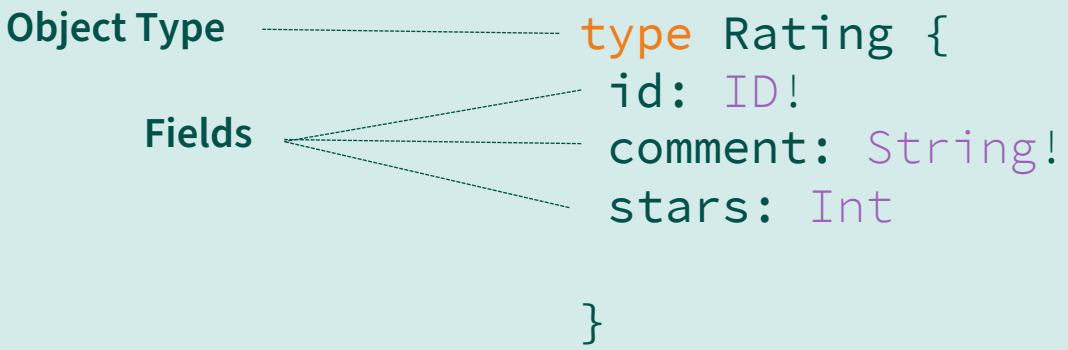
GRAPHQL SCHEMA

Schema

- A GraphQL API *have to* be defined with a Schema
- Schema describes *Types* and *Fields* of your API
- Only Queries (and Responses) that match your schema are executed (and returned)

- **Schema Definition Language** (SDL)

Defining a Schema with SDL



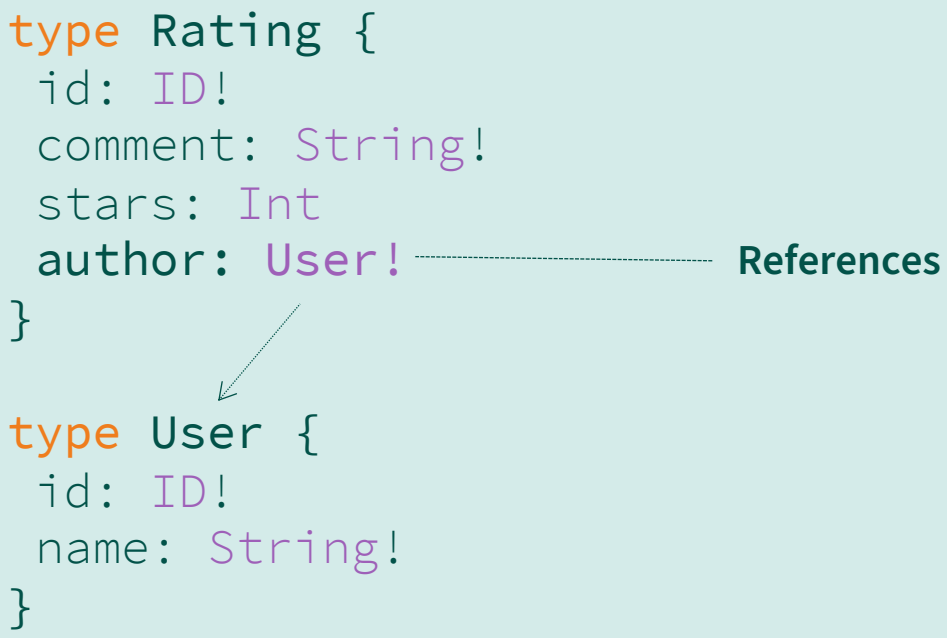
Defining a Schema with SDL

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

}
```

Return Type (non-nullable)

Return Type (nullable)

Defining a Schema with SDL



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

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

Root-Types: Entry-Points into the 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!
}
```

Define Schema using the Schema Definition Language

- Documentation

```
"""
A **Rating** as submitted by a User
"""
type Rating {
  # The unique ID of this Rating
  id: ID!
  ...
}

type Query {
  """Get a Rating by its ID or null if not found"""
  rating(ratingId: ID!): Rating
}
```

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

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

- A `DataFetcher` is a functional Java interface

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

- A `DataFetcher` is a functional Java interface

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

Implementing DataFetchers

- Example: A simple field

Schema Definition

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

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 }
}
```

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);
    }
  };
}
```

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: 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 starts = input.get("stars");

      return ratingService.newRating(beerId, stars);
    }
  };
}
```

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 DataFetcher<Publisher<Rating>> onNewRating() {

  return environment -> {
    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!)

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

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;
  ...
}
```

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);
    }
  };
}
```

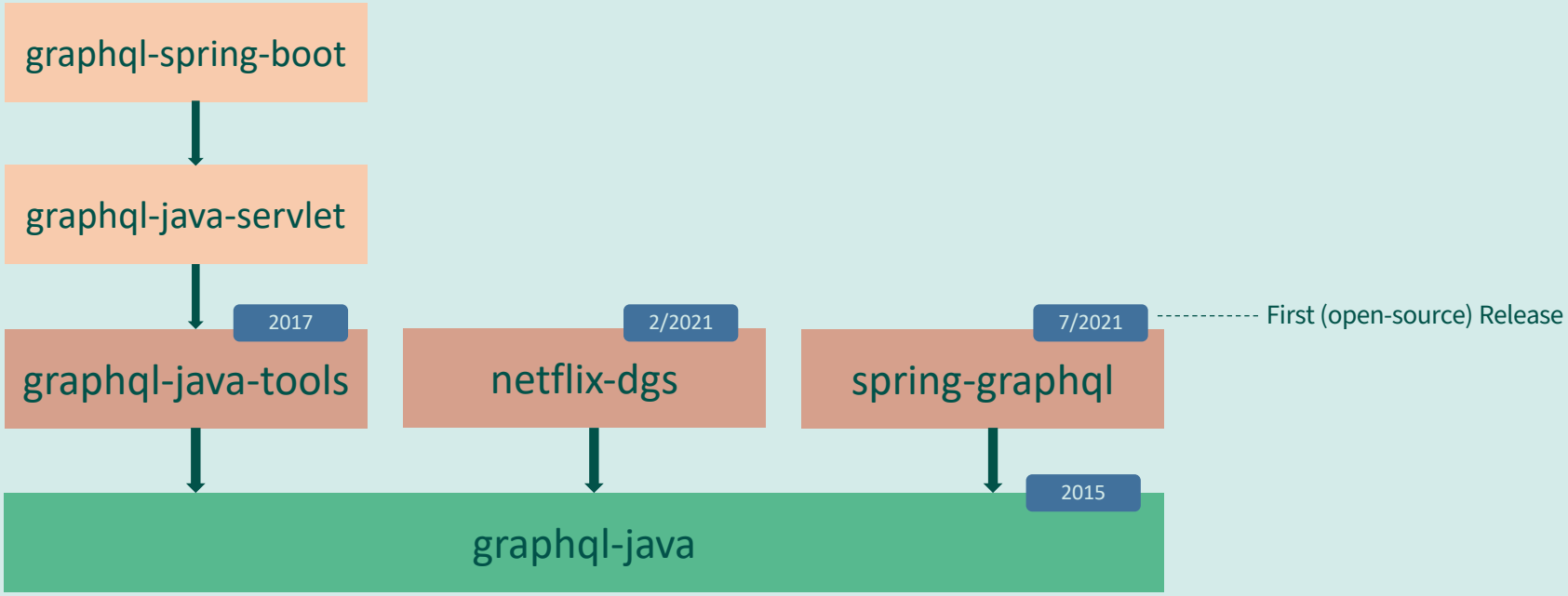
HIGHER LEVEL FRAMEWORKS

HIGHER LEVEL FRAMEWORKS

All share similiar ideas

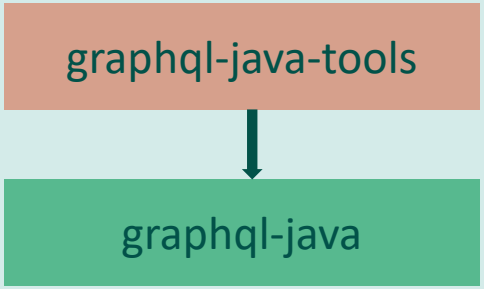
- Do not use DataFetcher directly, but abstraction

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



graphql-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 {

  // Methods on resolver classes must match field names in the schema
  // GraphQL field arguments are passed as method arguments
  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

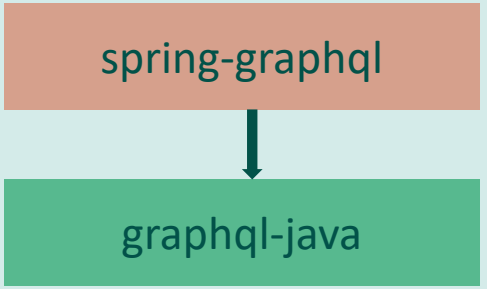
- Examples: Resolvers for fields not on root types
- The parent object (getSource in graphql-java) will be passed as first method parameter

```
public class BeerResolver implements GraphQLResolver<Beer> {

  public List<Shop> getShops(Beer parent) {
    return shopRepository.findShopsSellingBeer(parent.getId());
  }

}
```

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



Annotated Controllers

- Annotation-based programming model, similiar to REST controllers in Spring

```
@Controller
public class BeerQueryController {

  @QueryMapping
  public Beer beer(String beerId) {
    return beerRepository.getBeerById(beerId);
  }

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

  @SchemaMapping(typeName="Beer")
  public List<Shop> shops(Beer beer) {
    return shopRepository.findShopsSellingBeer(beer.getId());
  }
}
```

Nextflix Domain Graph Service Framework (DGS)

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

Optiona 4: Nextflix 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)

Optiona 4: Nextflix 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-frist in graphql-java)

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;
  ...
}
```

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());
  }
}
```

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)

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

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

Advanced Topics

- Security
  - GraphQL makes no statement how to implement security
  - Typically security from your server infrastructure is used, for example Spring Security

- Performance
  - DataFetchers can be implemented asynchronous
  - DataLoader can be used for batching (n+1 problem)

Thank you!

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

Contact: nils@nilshartmann.net