NILS HARTMANN
https://nilshartmann.net

Introduction to GraphQL

Slides: https://nils.buzz/jds-graphql

NILS HARTMANN

nils@nilshartmann.net

Freelance Developer, Coach and Trainer from Hamburg

Java
JavaScript, TypeScript
React
GraphQL

https://reactbuch.de

HTTPS://NILSHARTMANN.NET

NILS HARTMANN

nils@nilshartmann.net

Freelance Developer, Coach and Trainer from Hamburg

Java
JavaScript, TypeScript
React
GraphQL

Trainings, Workshops and Consulting

https://reactbuch.de

NILS HARTMANN

nils@nilshartmann.net

Freelance Developer, Coach and Trainer from Hamburg

Java
JavaScript, TypeScript
React
GraphQL

Trainings, Workshops and Consulting

Beer drinking

https://reactbuch.de
(german only)

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

Specification: https://graphql.org/

- 2015 published by Facebook
- Development since 2018 inside GraphQL Foundation

- Spec contains:
  - Query language and execution
  - Schema definition language
  - But not: Implementation
    - Reference implementation: graphql-js

GraphQL != SQL

- it‘s not SQL
  - it‘s not as powerful and flexible as SQL
  - no ordering, no paginating, no joins out-of-the-box
- no database!
- no framework!

GraphQL != Implementation

- GraphQL itself only is a specified *language*
- there are libs for a couple of languages that helps you *integrating and implementing* GraphQL into your application

GraphQL != Mainstream

- Implenentation and usage is still „bleeding edge“
- Not many best practices and experiences

- ...but it is already in use, also by big players!

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

https://twitter.com/tgvashworth/status/862049341472522240

TWITTER

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.

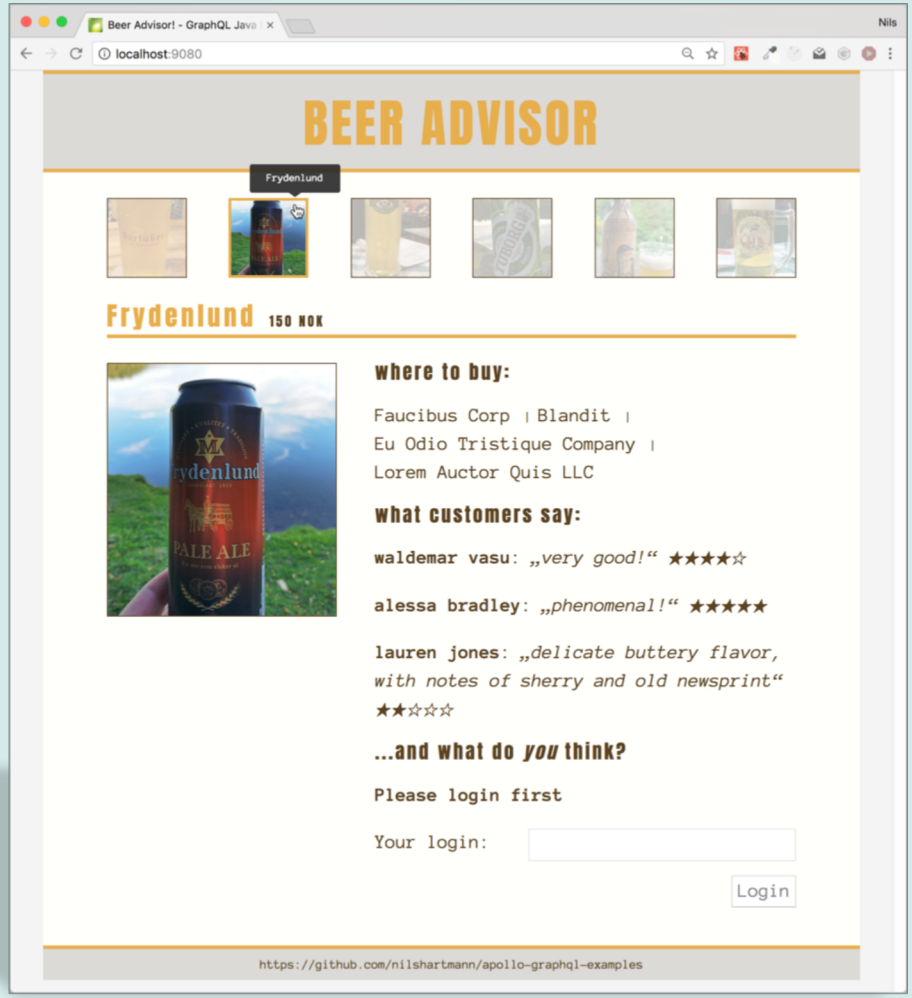
The New York Times website is changing, and the technology we use to run it is changing too.

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

NEW YORK TIMES

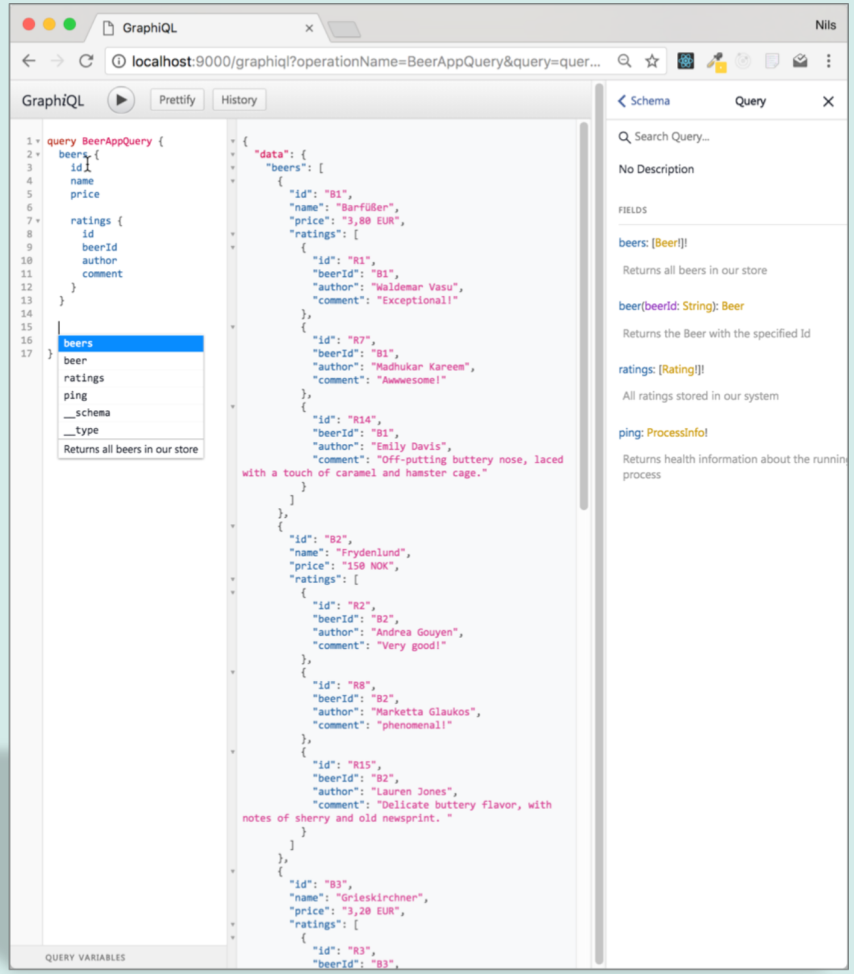
https://twitter.com/leeb/status/1123326647552266241

FACEBOOK 5



GraphQL in action

Source-Code: https://nils.buzz/graphql-java-example



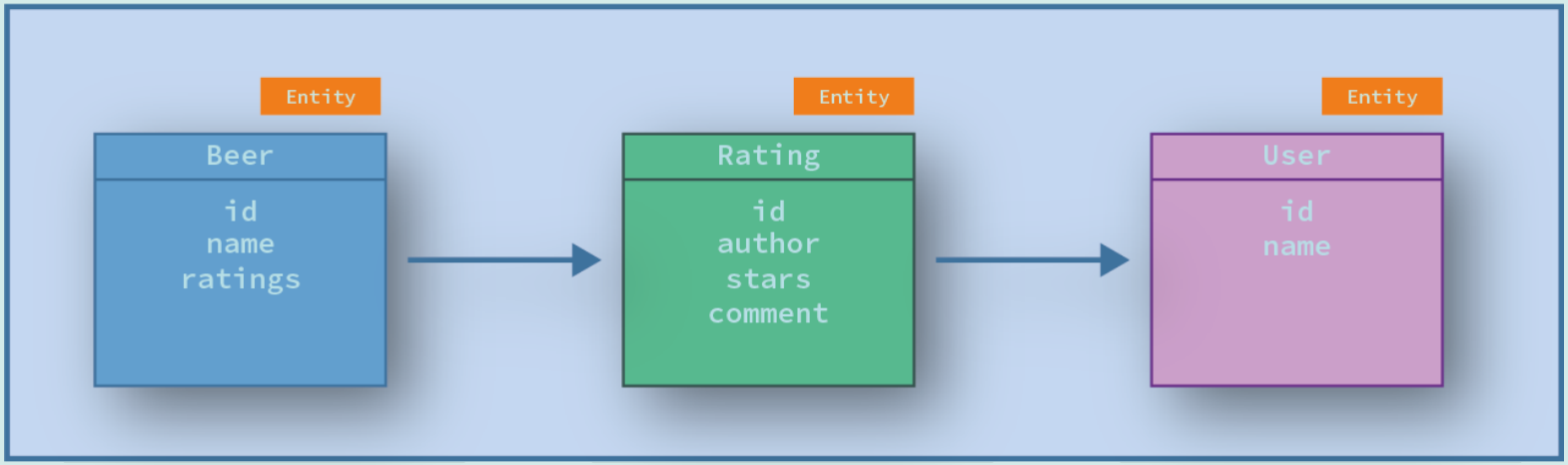
Demo: GraphiQL

http://localhost:9000/

Compare with REST 😱

BEERADVISOR DOMAIN

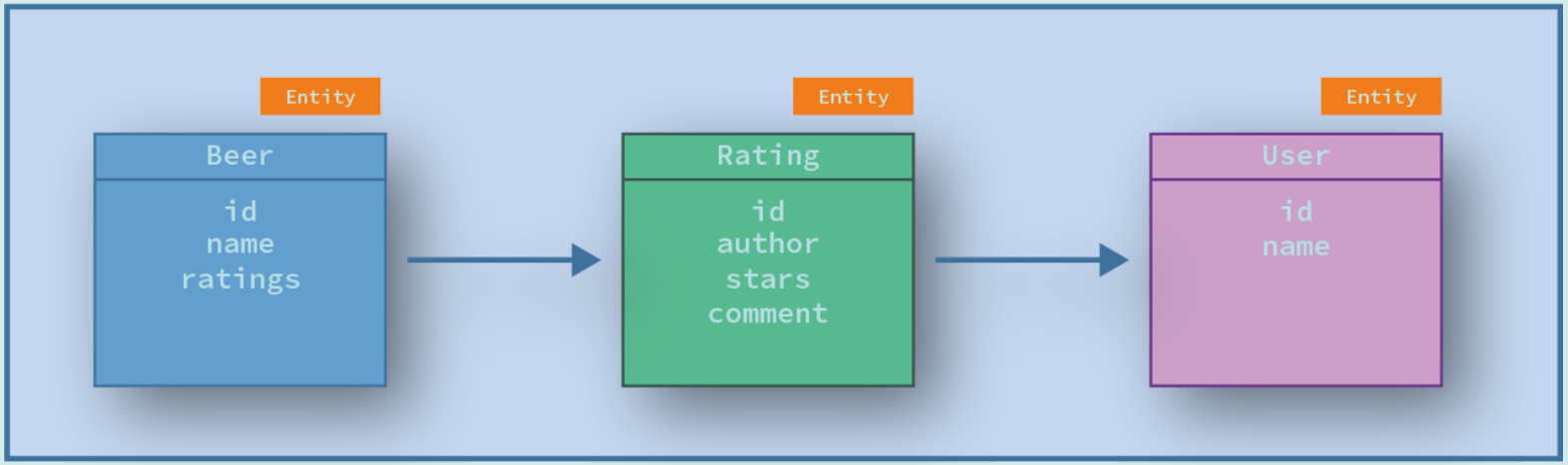
"Domain-Model"



A potential REST API

- Exemplary and simplified

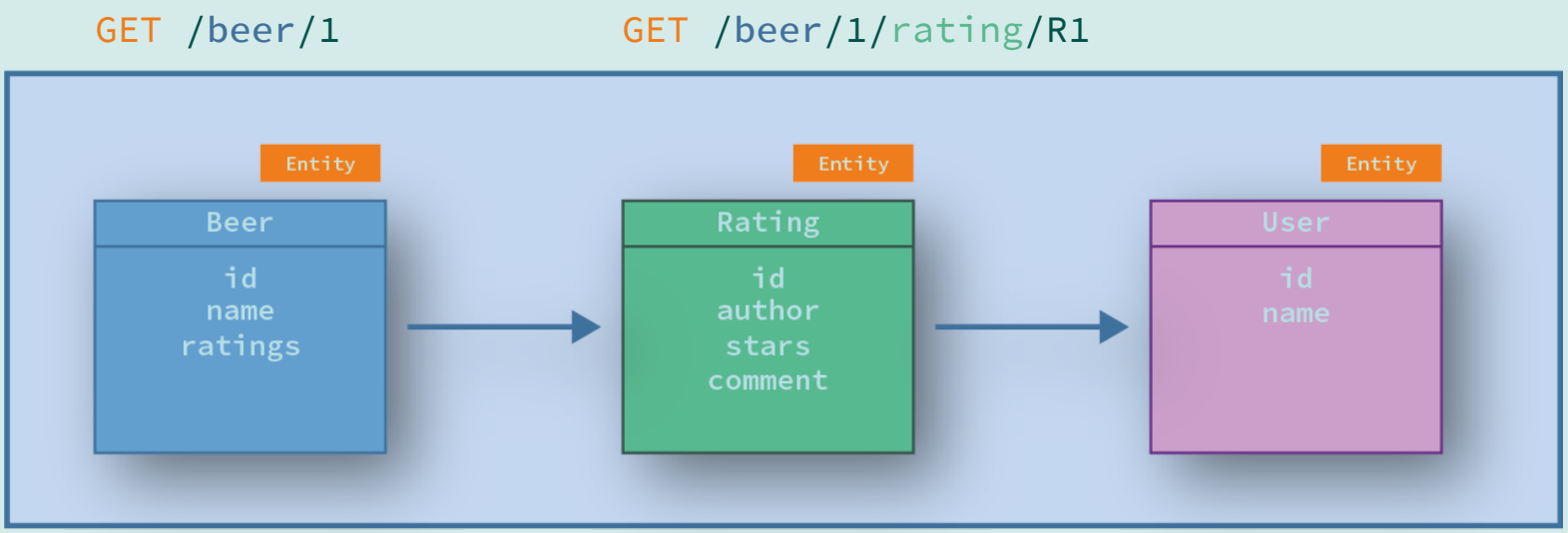
```
GET /beer/1
```




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

A potential REST API

- Exemplary and simplified

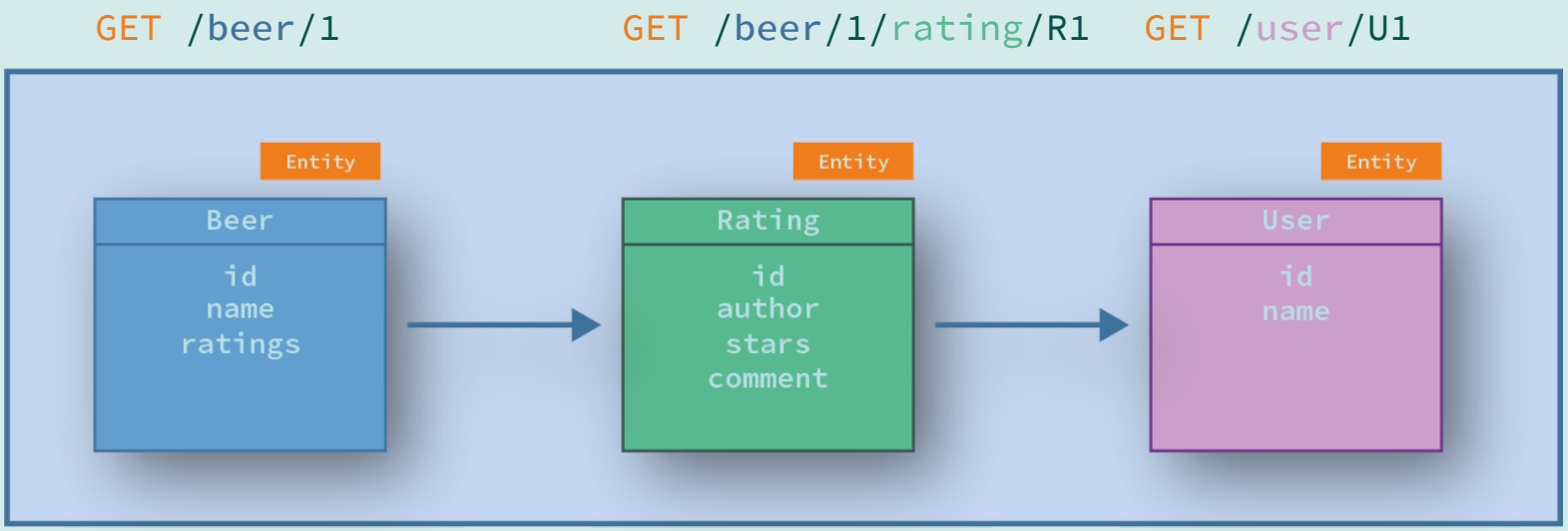



```
{
 "id": "1",
 "name": "Barfüßer"
}
```

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

REST API

- For each entity (Resource) one Request
- We always get the whole Resource (no standard way to get only partial resources)
- There is no „overall view“ on our domain




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

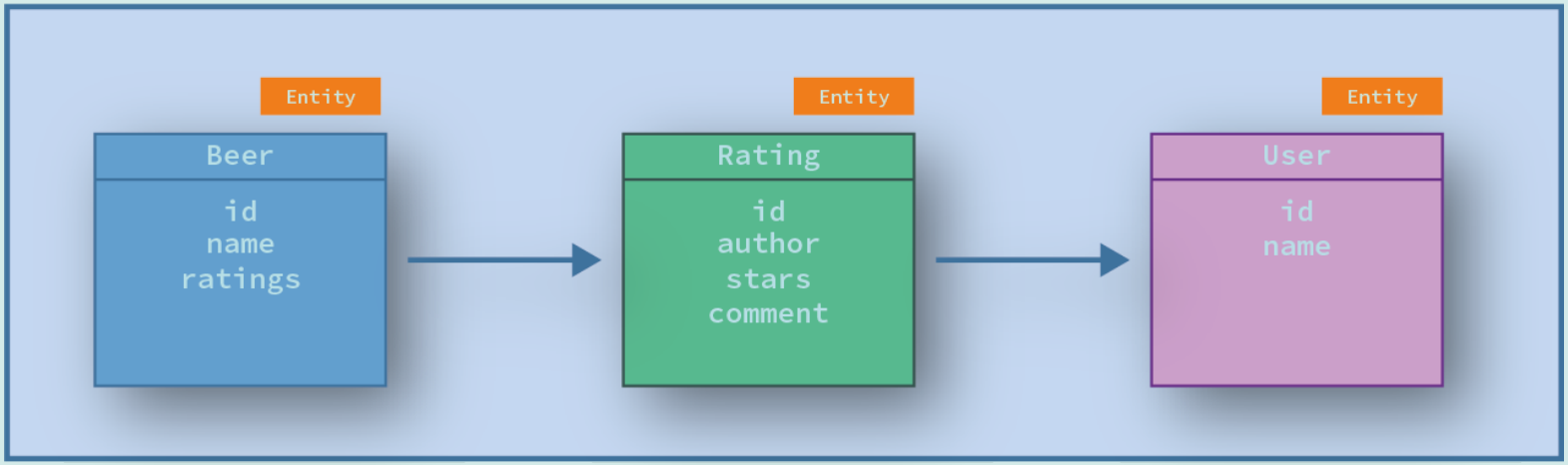
```
{
 "id": "R1",
 "author": "U1",
 "stars": 3,
 "comment": "good!"
}
```

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

REQUESTS WITH GRAPHQL

GraphQL

```
query { beer
  { name ratings(rid: "R1")
    { stars author { name }
  }
}
```

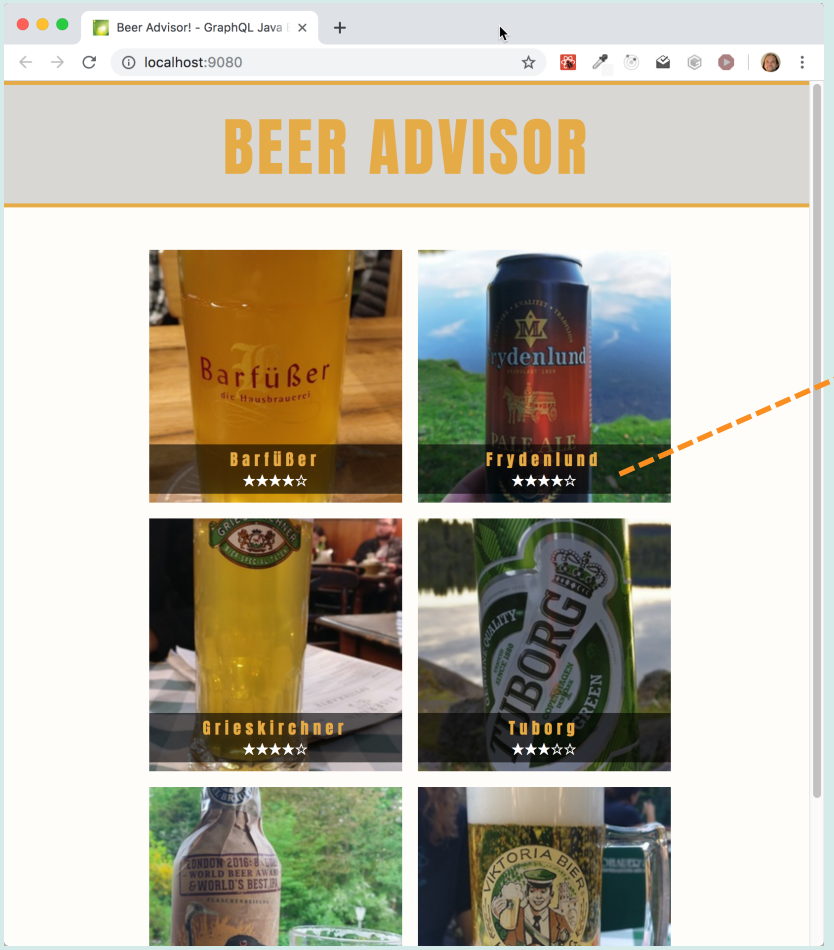


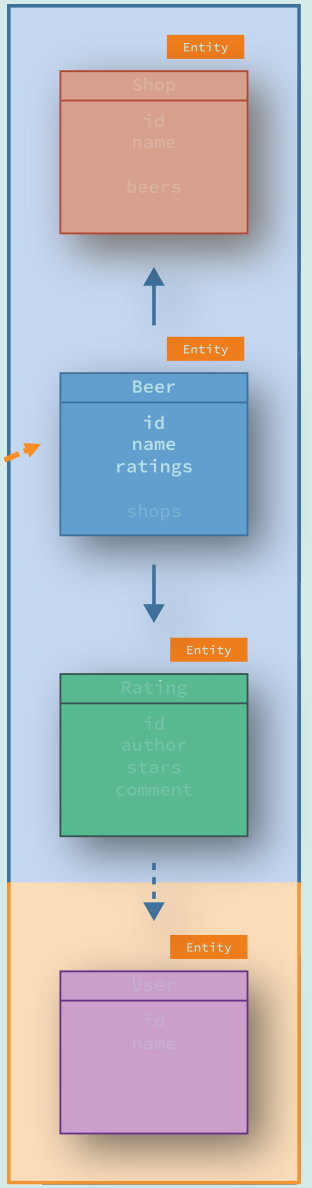

```
{
  "name": "Barfüßer",
  "ratings": {
    "stars": 3,
    "comment: "good",
    "author": { "name": "Klaus" }
  }
}
```

GRAPHQL USE-CASES

Use-Case specific requests – 1

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

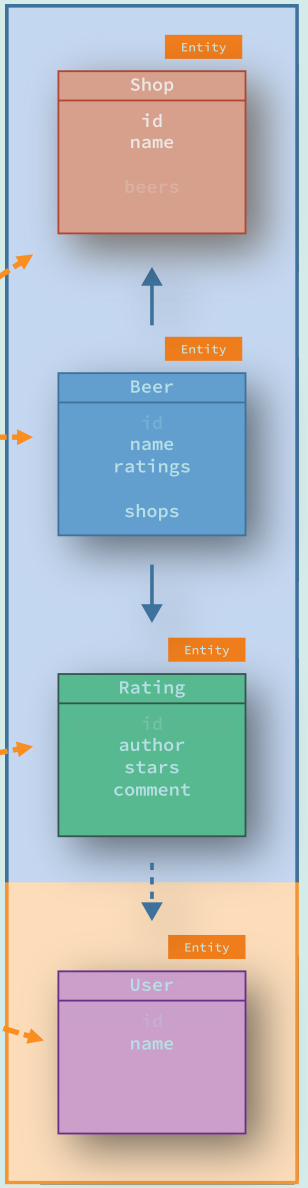




GRAPHQL USE-CASES

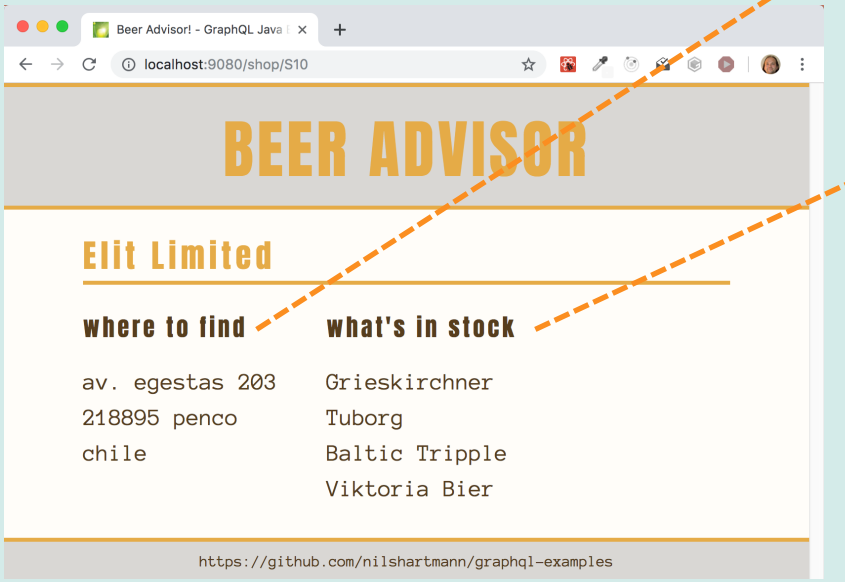
Use-Case specific requests – 2

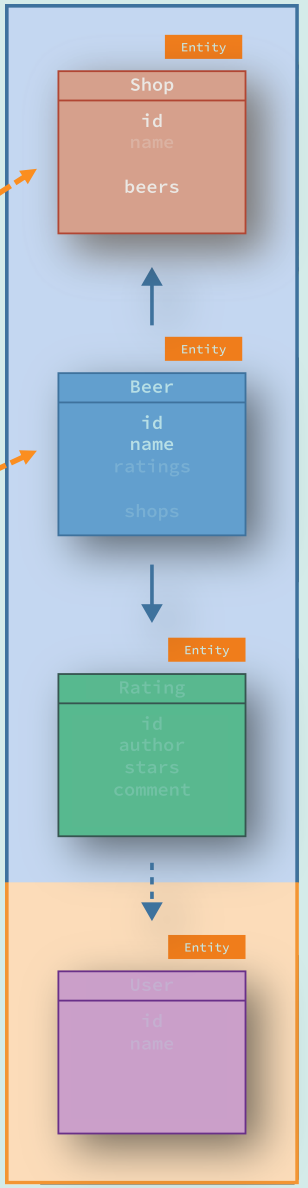
```
{ beer(beerId: "B1" {
    name
    price
    ratings {
      stars
      comment
      author {
        name
      }
    }
    shops { name }
  }
}
```



Use-Case specific requests – 3

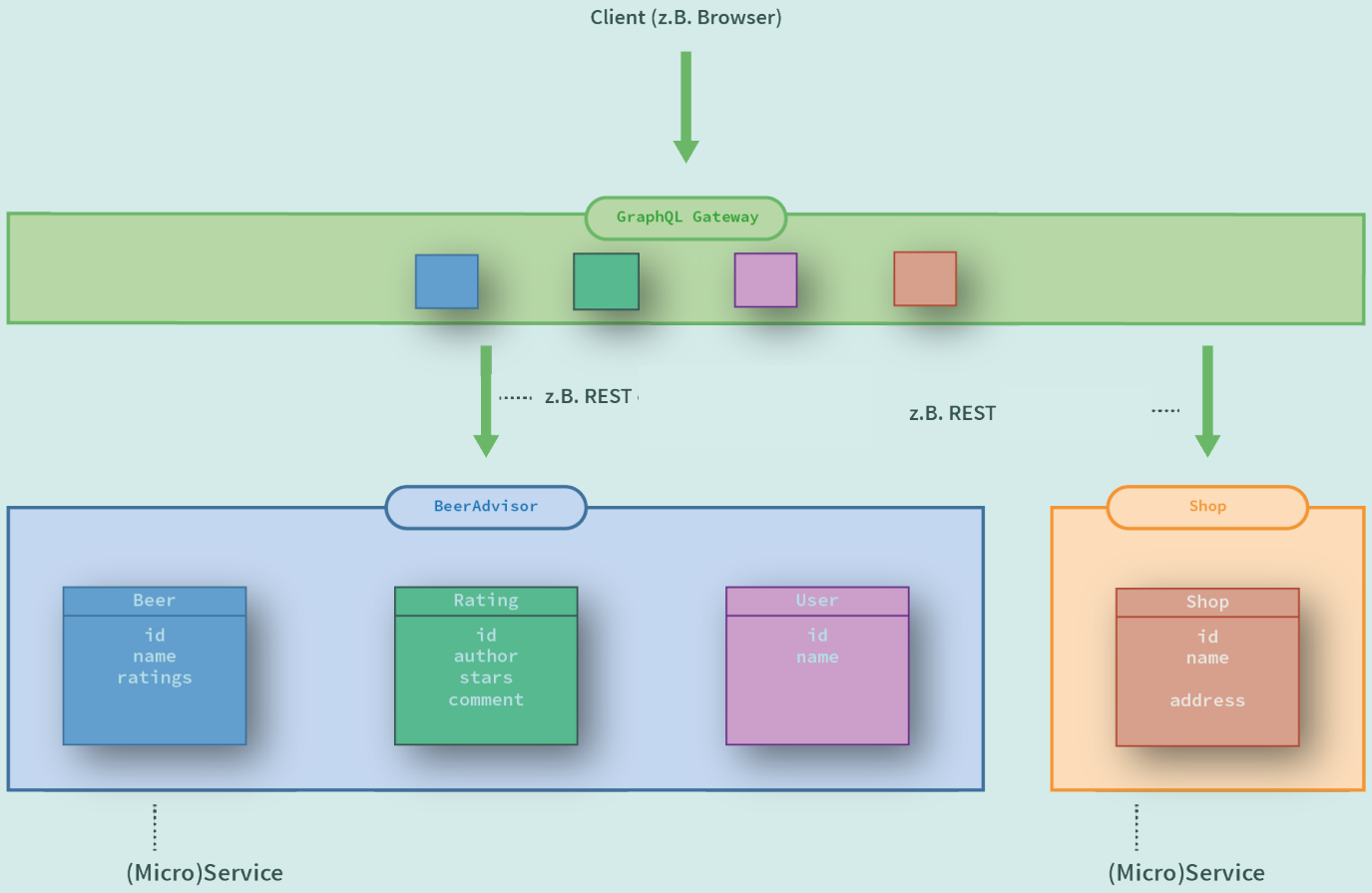
```
{ shop(shopId: "S3") {
    name
    address { street city }
    beers { id name }
  }
}
```





EINSATZSZENARIEN

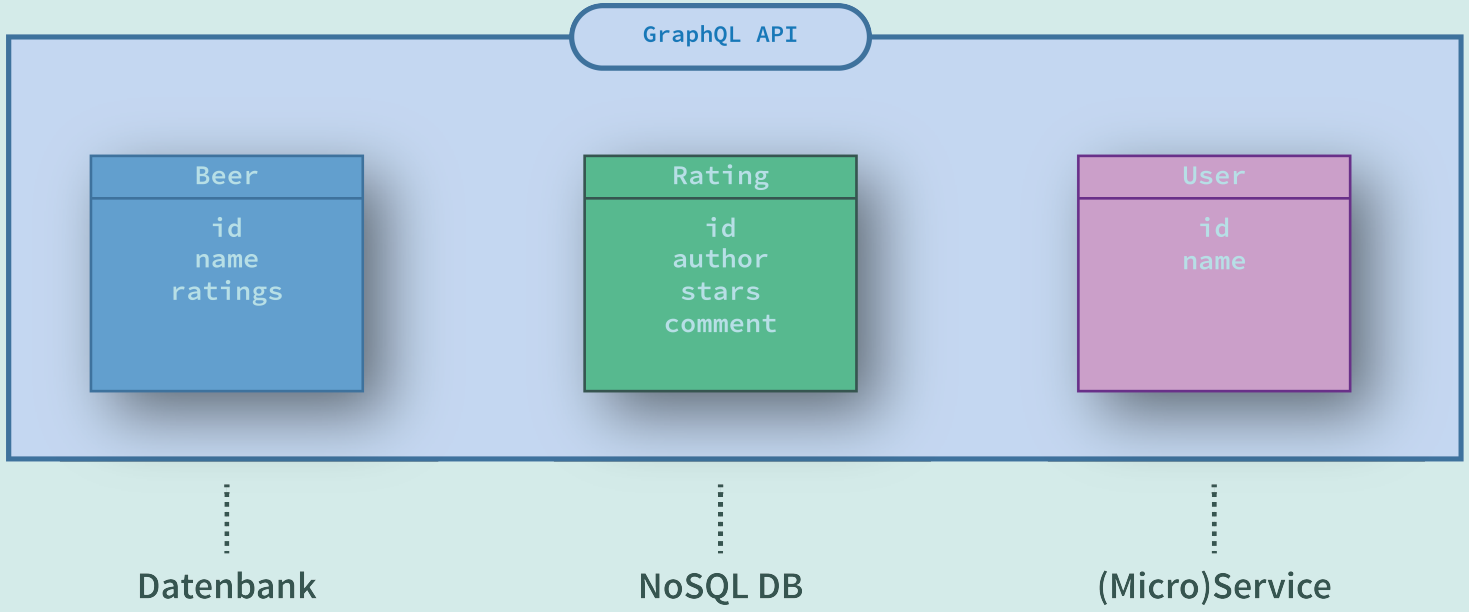
- Single gateway for frontends/clients to couple of services
  - Probably not: service-to-service communication



DATASOURCES

GraphQL makes no statement, where data comes from

👉 *Providing the data is up to us*



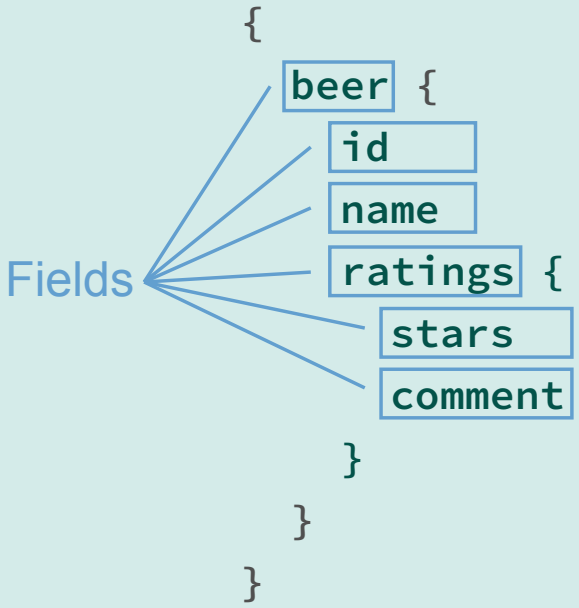
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- https://graphql.org

GraphQL

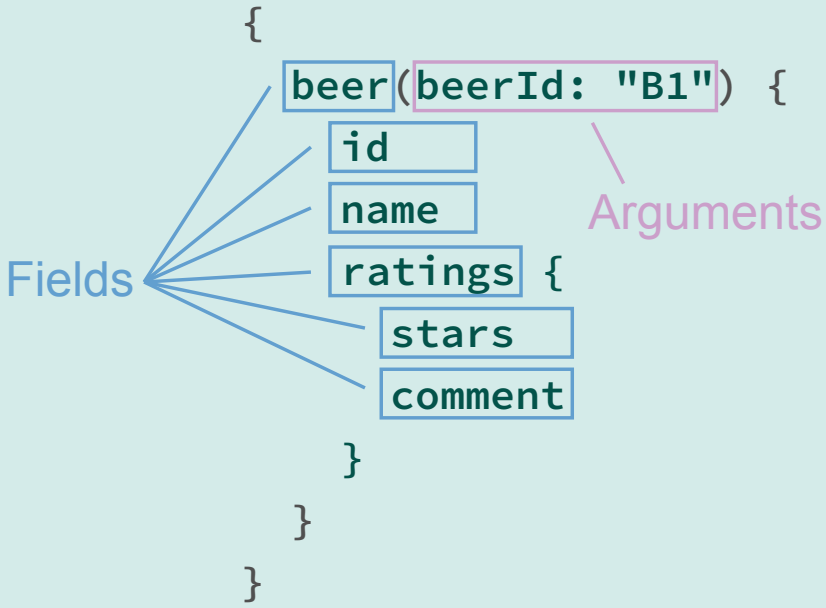
TEIL 1: ABFRAGEN UND SCHEMA

QUERY LANGUAGE



- Structured language to retrieve data from the API
- **Fields** of (nested) objects are queried

QUERY LANGUAGE



- Structured language to retrieve data from the API
- **Fields** of (nested) objects are queried
- Fields can have **arguments**

Response

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

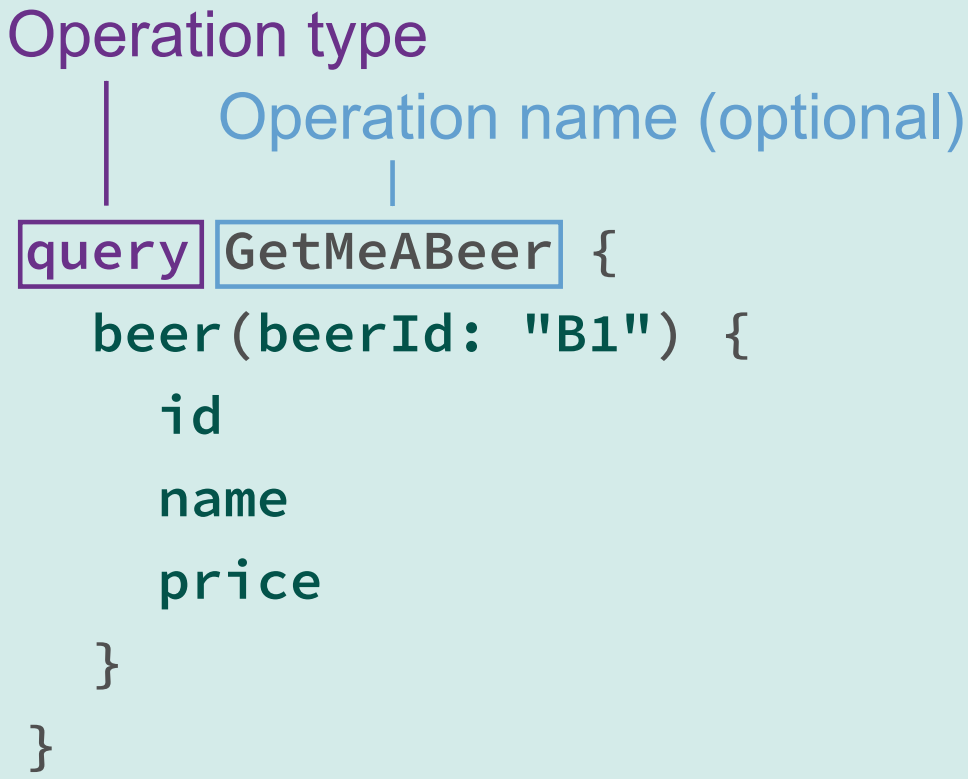
```
"data": {
  "beer": {
    "id": "B1"
    "name": "Barfüßer"
    "ratings": [
      {
        "stars": 3,
        "comment": "grate taste"
      },
      {
        "stars": 5,
        "comment": "best beer ever!"
      }
    ]
  }
}
```

- Identical structure as for the query

QUERY LANGUAGE: OPERATIONS

Operation: describes what is to be done

- query, mutation, subscription



QUERY LANGUAGE: MUTATIONS

Mutations

- Mutations are used to *modify* data
- Comparable to POST, PUT, PATCH, DELETE in REST

Operation type

Operation name (optional)

Variable Definition

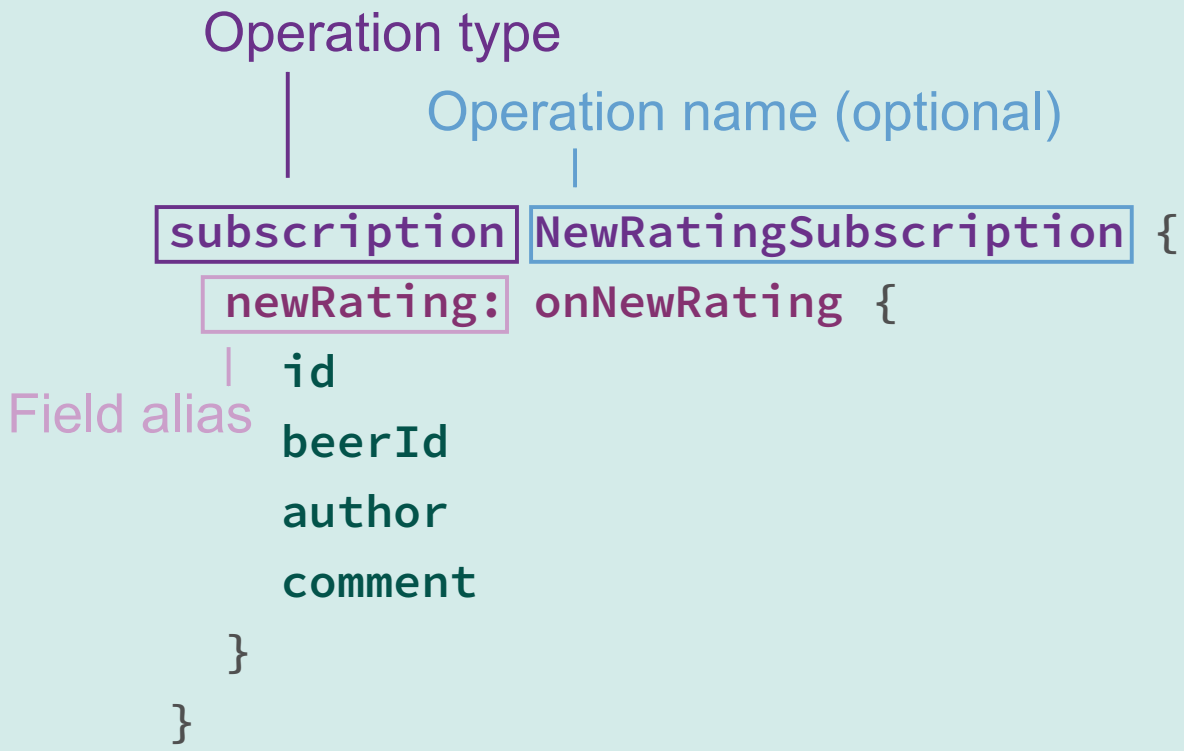
```
mutation AddRatingMutation($input: AddRatingInput!) {
  addRating(input: $input) {
    id
    beerId
    author
    comment
  }
}

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

— Variable Object

QUERY LANGUAGE: MUTATIONS

Subscription

- Receiving events from the server



EXECUTION OF QUERIES

Queries are executed normally using HTTP requests

```
$ 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"}
   ]}
}
```

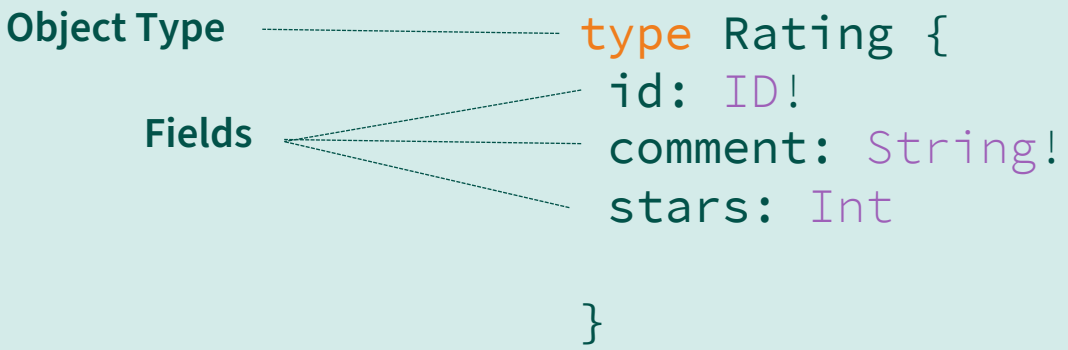
GRAPHQL SCHEMA

Schema

- A GraphQL API *must* have a schema describing it
- The schema determines the available *Types* and *Fields*
- Only requests and responses matching the query are executed and returned

- **Schema Definition Language** (SDL)

Schema Definition using SDL



Schema Definition using SDL

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

}
```

Return Type (non-nullable)

Return Type (nullable)

Schema Definition using SDL

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

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

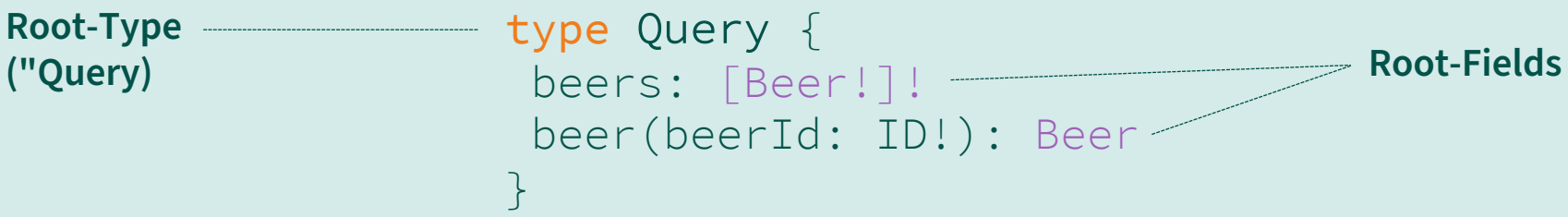
Reference to other type

Schema Definition using 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

Root-Types: Entry points into the API (Query, Mutation, Subscription)



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-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!
}
```

Evolution of the Schema: Only one version of the schema

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


}
```

Evolution of the Schema: Only one version of the schema

- Fields are *always* requested explicitly by name (no „select all“)
- *New* fields can be added without harming existing clients

New field

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

}
```

Evolution of the Schema: Only one version of the schema

- Fields are *always* requested explicitly by name (no „select all“)
- *New* fields can be added without harming existing clients
- *Old* fields can be 'deprecated‘

- Usage of fields can be tracked individually

New Feld

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

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

- https://graphql.org

GraphQL in your App

(Example: Java, but concepts similiar with other languages)

PART 2: RUNTIME-ENVIRONMENT (AKA: YOUR APPLICATION)

EXAMPLE: GRAPHQL FOR JAVA

Step 1: Define your Schema

```
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!
}
```

EXAMPLE: GRAPHQL FOR JAVA

Step 2: DataFetcher

- (In other libraries often called **Resolver**)

- *A **DataFetcher** (or Resolver) returns a value for a request field*
  - Mandatory only for Root-Types (Query, Mutation)

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

Step 2: DataFetcher

- Example: DataFetcher for beers field

```
public class BeerAdvisorDataFetchers {

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

}
```

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


}
```

Step 2: DataFetcher

- Example: accessing field arguments

```
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);
    };
  }

}
```

OPTIMIZATION

There are several ways to optimize for perfomance

- Depending on the library you use
- One common technique: **DataLoader** to avoid n+1-problems
- Others address
  - caching (yes, you can cache!)
  - reducing calls to databases

GraphQL Code Generator

- **Generator for various languages and libraries:** https://graphql-code-generator.com/

- **Generator for Queries and Response Types (Java):** https://github.com/adobe/graphql-java-generator

GraphQL for databases

- **Instant GraphQL Schema for PostgresDB (Node.JS):**
  https://www.graphile.org/postgraphile/

- **Instant GraphQL Schema for PostgresDB:**
  https://hasura.io/

- **GraphQL as ORM (JavaScript, Go):**
  https://prisma.io/

Summary

- **Interesting, but still young technology**
  Breaks with habits from REST
  Requires re-thinking concepts and
  REST and GraphQL can be used together in one app

- **Does neither replace your Backend nor your Database**

- **Libs and Frameworks for many languages (Java, JS, C#, ...)**
  Prototypes for exploration and evaluation is setup usually quickly

- **My recommendation: (at least) give it a try and follow further development**

https://reactbuch.de

Thank you very much!

Example code: https://nils.buzz/graphql-java-example

Slides: https://nils.buzz/jds-graphql

Contact: nils@nilshartmann.net

Twitter: @nilshartmann

HTTPS://NILSHARTMANN.NET