NILS HARTMANN

https://nilshartmann.net

API DAY | FEB. 16. 2023

GraphQL

what is it all about?

Slides (PDF): **https://graphql.schule/api-day2023**

NILS HARTMANN

nils@nilshartmann.net

Software Developer, Architect and Coach from Hamburg

Java, Spring, GraphQL, TypeScript, React

heise Academy
VIDEOKURS
Nils Hartmann
GraphQL
APIs mit Spring Boot entwickeln
Der Praxiskurs für die Web-API-Entwicklung

https://graphql.schule/video-kurs

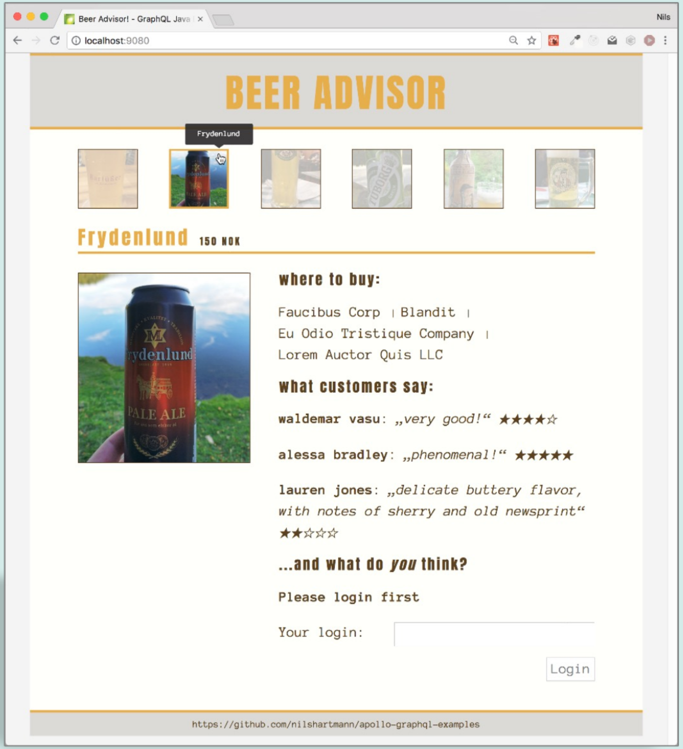
2. Auflage
Nils Hartmann · Oliver Zeigermann
React
Grundlagen, fortgeschrittene Techniken und Praxistipps – mit TypeScript und Redux
dpunkt.verlag

https://reactbuch.de

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



BEER ADVISOR

Frydenlund 150 NOK

where to buy:

Faucibus Corp | Blandit |
Eu Odio Tristique Company |
Lorem Auctor Quis LLC

what customers say:

waldemar vasu: „*very good!*“ ★★★★☆

alessa bradley: „*phenomenal!*“ ★★★★★

lauren jones: „*delicate buttery flavor, with notes of sherry and old newsprint*“ ★★☆☆☆

...and what do *you* think?

Please login first

Your login:

Login

https://github.com/nilshartmann/apollo-graphql-examples

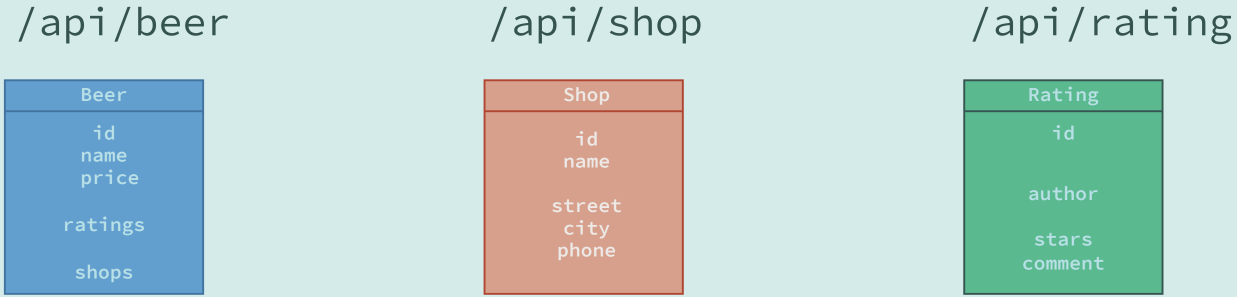
Example application

Source: https://github.com/nilshartmann/spring-graphql-talk

An API for the Beer Advisor

AN API FOR THE BEERADVISOR

Approach 1: Backend defines the API / data



/api/beer

Beer

id
name
price

ratings

shops

/api/shop

Shop

id
name

street
city
phone

/api/rating

Rating

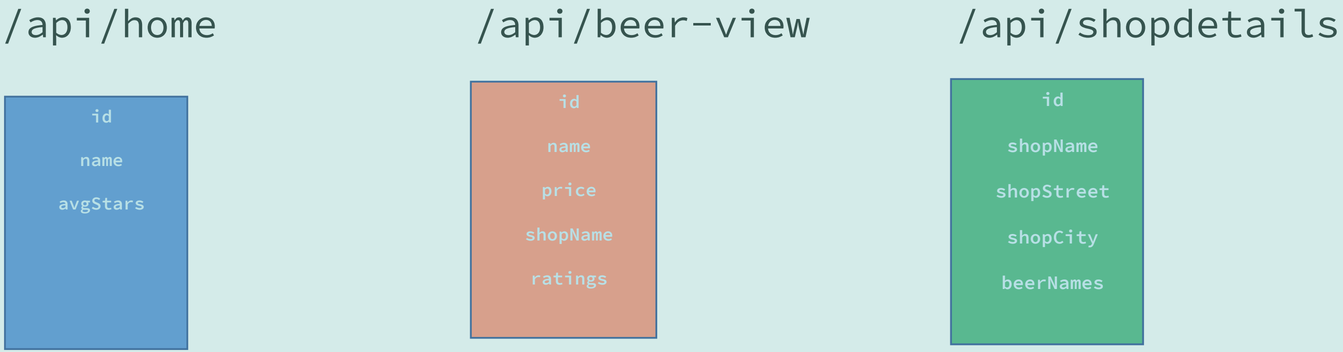
id

author

stars
comment

AN API FOR THE BEERADVISOR

Approach 2: Client defines the API based on its requirements, views, use-cases, ...



/api/home

id

name

avgStars

/api/beer-view

id

name

price

shopName

ratings

/api/shopdetails

id

shopName

shopStreet

shopCity

beerNames

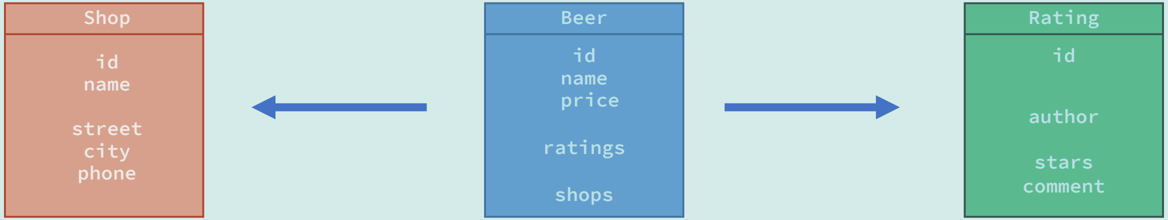
AN API FOR THE BEERADVISOR

Approach 3: GraphQL...

AN API FOR THE BEERADVISOR

Approach 3: GraphQL...

- As approach 1: Server defines the data model

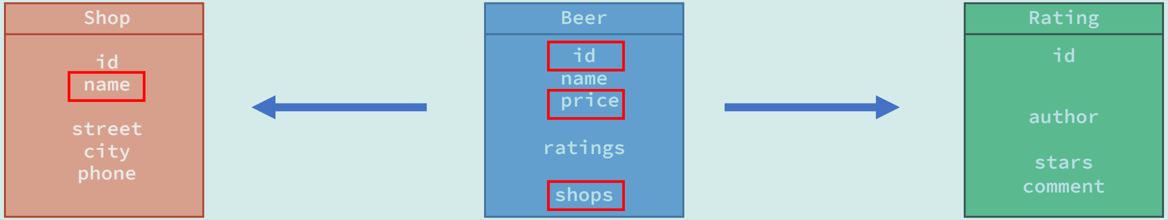


AN API FOR THE BEERADVISOR

Approach 3: GraphQL...

- As approach 1: Server defines the data model
- ...but the client can choose itself in every request the data it wants to read

```
{ beer { id price { shops { name } }
```



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

- Developed by the GraphQL Foundation

- Spec includes:
  - Language
  - Type System
  - General execution behaviour

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

- Developed by the GraphQL Foundation

- Spec includes:
  - Language
  - Type System
  - General execution behaviour

  - **No implementation!**
    - Server reference implementation: graphql-js

GRAPHQL APIS

With GraphQL we publish an api based on our domain model

- What data we expose is up to us
- We define the structure of the data we want to expose

👉 **We explicitly define, how our API looks and behaves**

GRAPHQL APIS

With GraphQL we publish an api based on our domain model

- What data we expose is up to us
- We define the structure of the data we want to expose

👉 **We explicitly define, how our API looks and behaves**

👉 **GraphQL does not create an API "magically" for us**

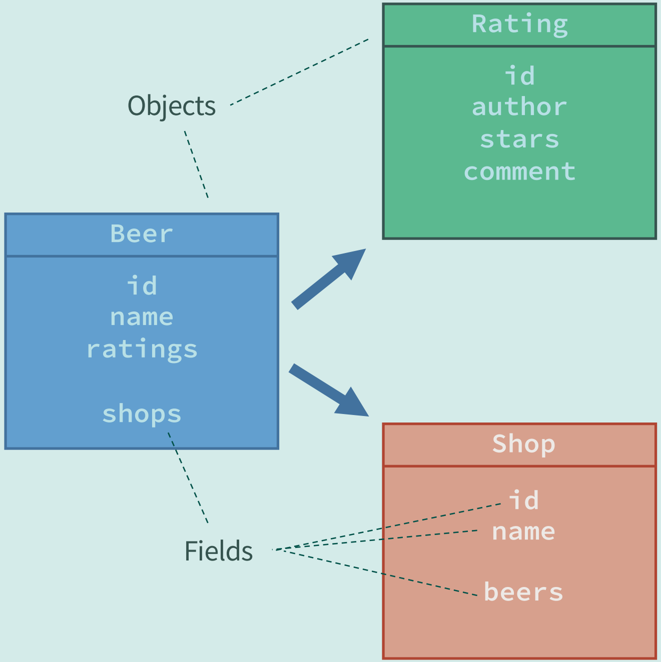
"GraphQL is a ***query language for APIs*** *and a runtime for fulfilling those queries with your existing data"*

- https://graphql.org

GraphQL

QUERY LANGUAGE

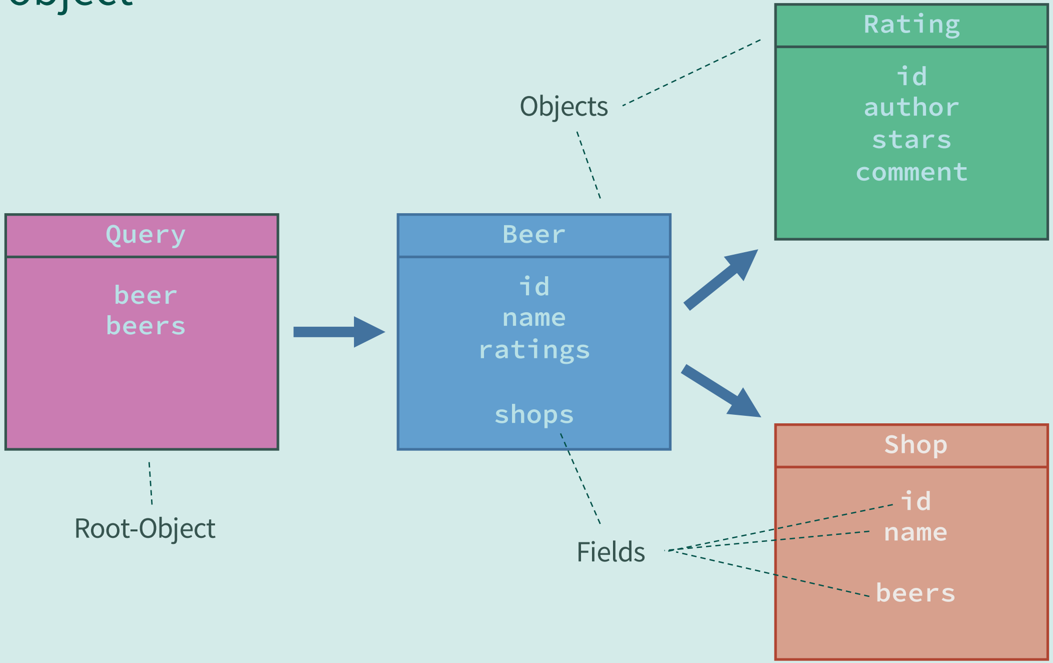
With the query language, you select fields from objects



QUERY LANGUAGE

With the query language, you select fields from objects

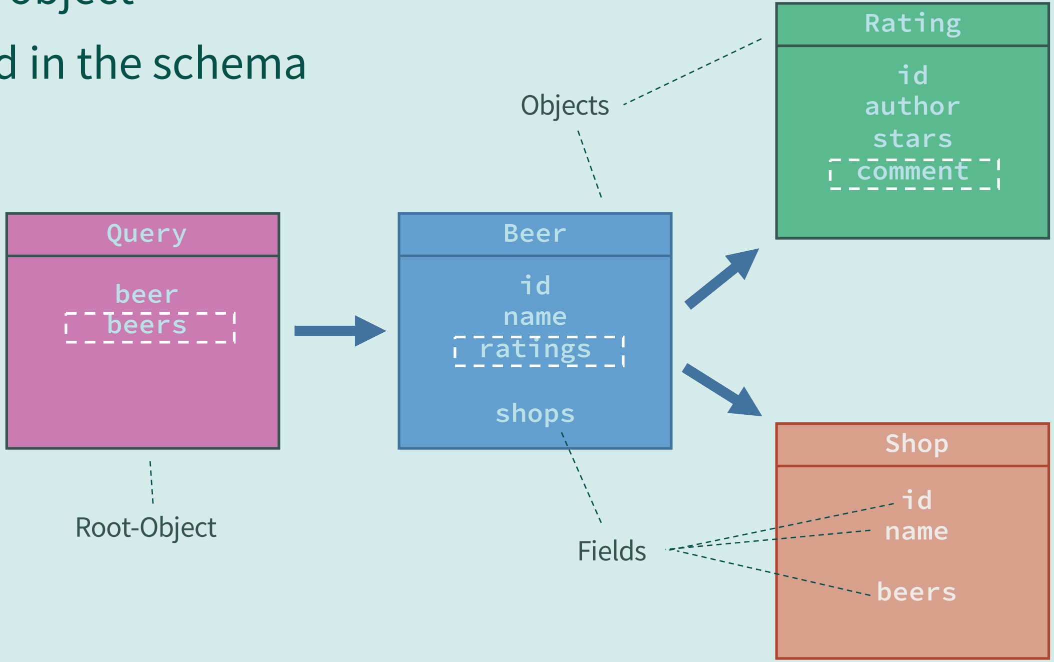
- All queries start on a special root object



QUERY LANGUAGE

With the query language, you select fields from objects

- All queries start on a special root object
- You can only follow paths defined in the schema

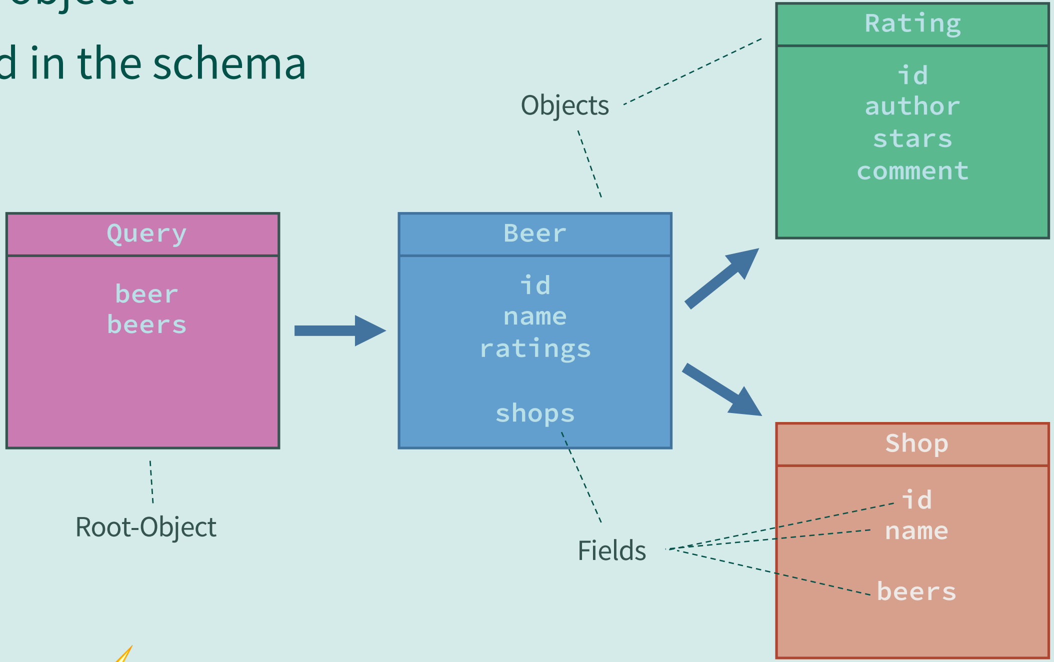


```
query { beers { ratings { comment } } }
```

QUERY LANGUAGE

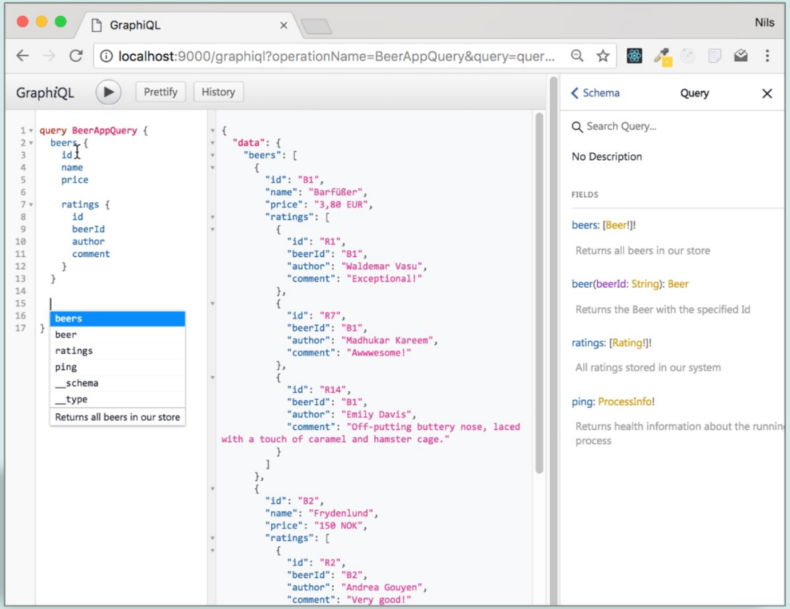
With the query language, you select fields from objects

- All queries start on a special root object
- You can only follow paths defined in the schema
- No other "joins" possible



⚡

```
query { shops { id } }
```


GraphiQL
Nils
localhost:9000/graphiql?operationName=BeerAppQuery&query=quer...
GraphiQL
Prettify
History
beers
beer
ratings
ping
__schema
__type
Returns all beers in our store

```graphql
query BeerAppQuery {
  beers {
    id
    name
    price

    ratings {
      id
      beerId
      author
      comment
    }
  }

  
}
```

```json
{
  "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": "Awwwesome!"
          },
          {
            "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!"
```

< Schema Query ×

Search Query...

No Description

FIELDS

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

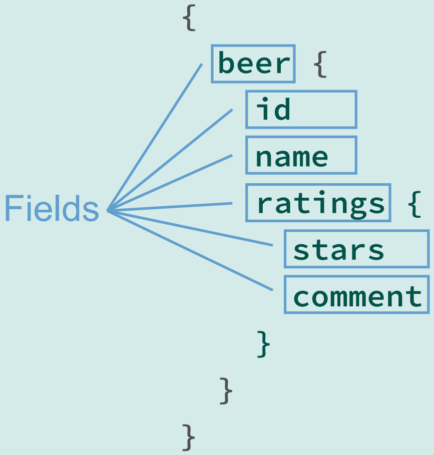
ping: ProcessInfo!

Returns health information about the runnin
process

Demo Query Language

https://github.com/graphql/graphiql

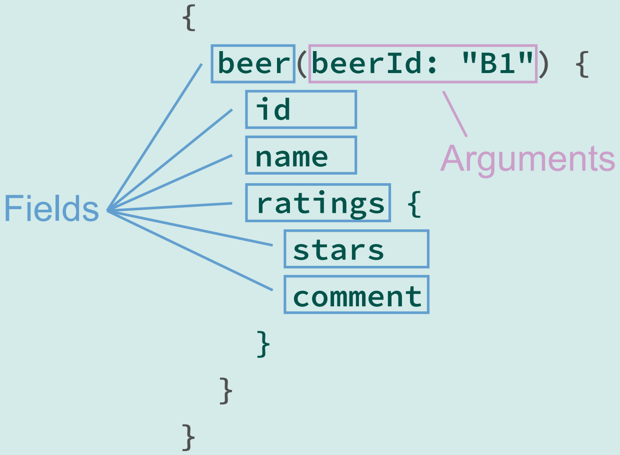
QUERY LANGUAGE



```
{
  beer {
    id
    name
    ratings {
      stars
      comment
    }
  }
}
```

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

QUERY LANGUAGE



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

- 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
    }
  }
}
```

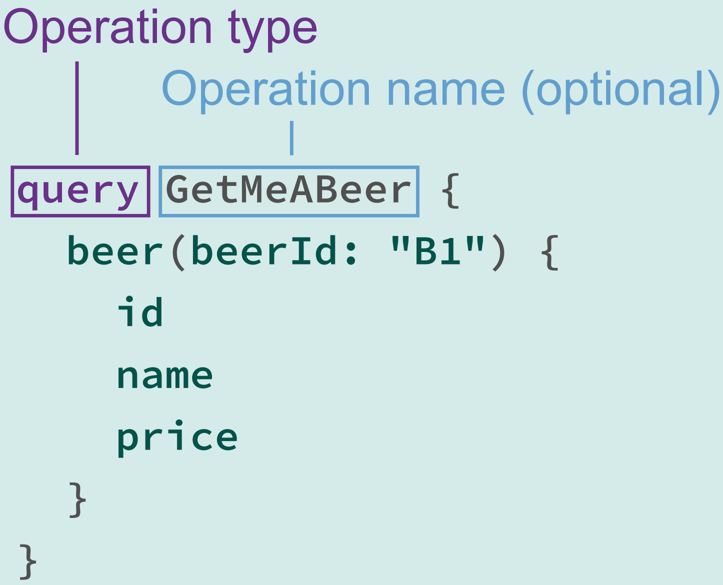
```
"data": {
  "beer": {
    "id": "B1"
    "name": "Barfüßer"
    "ratings": [
      {
        "stars": 3,
        "comment": "grate taste"
      },
      {
        "stars": 5,
        "comment": "best beer ever!"
      }
    ]
  }
}
```

- Identical structure as your query

QUERY LANGUAGE: OPERATIONS

Operation: describe, what the query should do

- query, mutation, subscription



Operation type

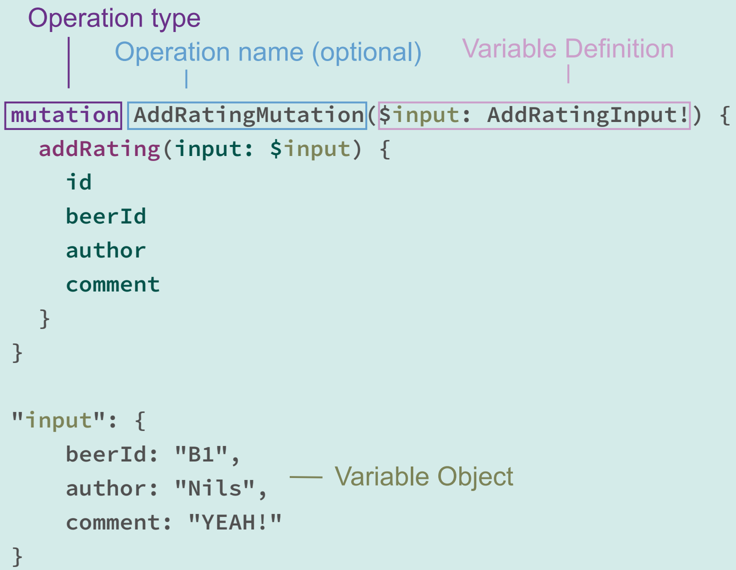
Operation name (optional)

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

QUERY LANGUAGE: MUTATIONS

Mutations

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



Operation type

Operation name (optional)

Variable Definition

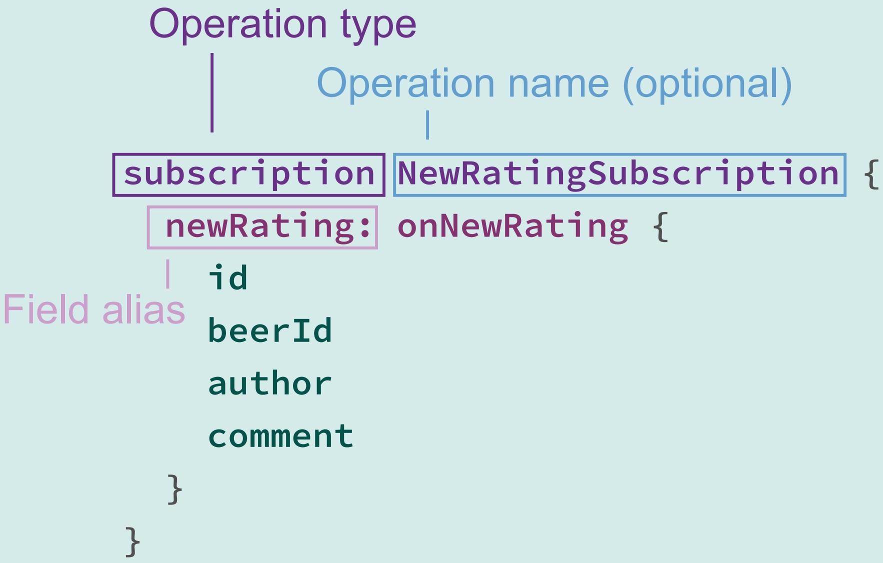
```
mutation AddRatingMutation($input: AddRatingInput!) {
  addRating(input: $input) {
    id
    beerId
    author
    comment
  }
}

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

Variable Object

QUERY LANGUAGE: MUTATIONS

Subscription

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



```
subscription NewRatingSubscription {
  newRating: onNewRating {
    id
    beerId
    author
    comment
  }
}
```

EXECUTING QUERIES

Queries usually are executed via HTTP

- One single HTTP endpoint /graphql
  - queries are sent using POST (or sometimes GET)
  - Other HTTP verbs do not matter

- Implementation depends on your serverside framework
  - There is a specification being developed standardizing the server protocol

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 BACKENDS

Implementing a GraphQL backend

- Specification does not a force a specific implementation
- There are frameworks for a lot of programming languages
- Almost all of them are following the same principles

GRAPHQL BACKENDS

Processing a GraphQL request

- GraphQL request ("document") is received by your backend

Processing a GraphQL request

- GraphQL request ("document") is received by your backend
- GraphQL framework parses and validates the operations
  - Syntax valid? Valid according to schema?
  - If invalid, error is sent to the client

Processing a GraphQL request

- GraphQL request ("document") is received by your backend
- GraphQL framework parses and validates the operations
  - Syntax valid? Valid according to schema?
  - If invalid, error is sent to the client
- Otherwise the request will be processed...

GRAPHQL BACKENDS

Processing a GraphQL request

- For each field, a **resolver function** is invoked by the framework
    - A resolver function determines the value for a field

Processing a GraphQL request

- For each field, a **resolver function** is invoked by the framework
  - A resolver function determines the value for a field
  - It's our task to implement the resolver functions
  - ("Implement a GraphQL API" == "Implement resolver functions")

Processing a GraphQL request

- For each field, a **resolver function** is invoked by the framework
  - A resolver function determines the value for a field
  - It's our task to implement the resolver functions
  - ("Implement a GraphQL API" == "Implement resolver functions")
- Result from resolver functions is validated by the GraphQL framework

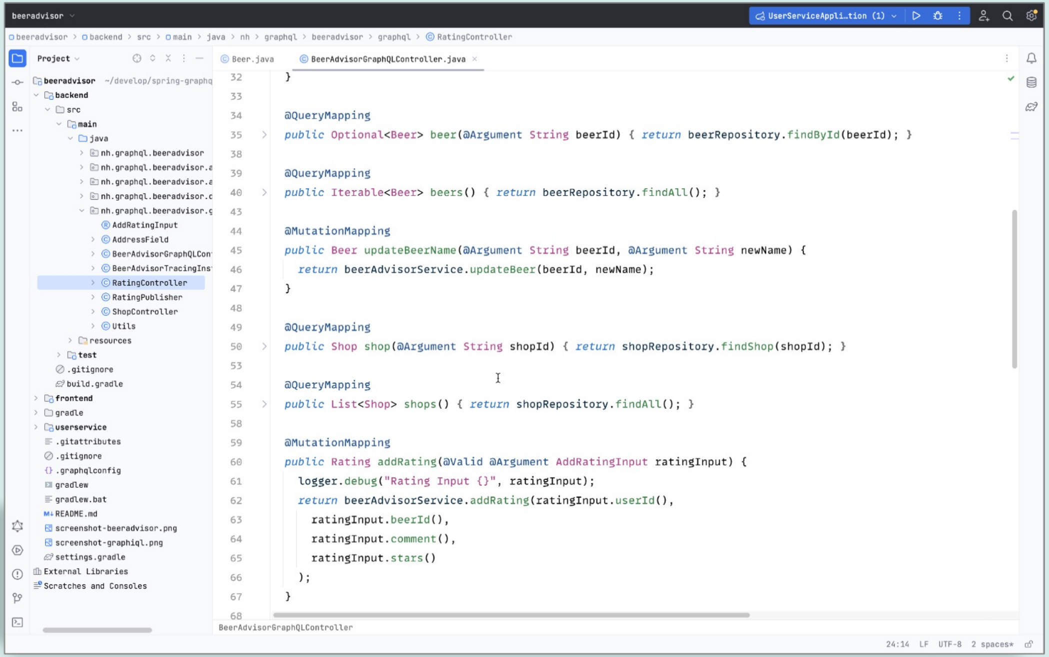
GRAPHQL BACKENDS

Processing a GraphQL request

- For each field, a **resolver function** is invoked by the framework
  - A resolver function determines the value for a field
  - It's our task to implement the resolver functions
  - ("Implement a GraphQL API" == "Implement resolver functions")
- Result from resolver functions is validated by the GraphQL framework
- Result is sent back to client

Implementing a GraphQL API

- Step one: defining a schema that expresses your API
- Step two: implement the logic for determining the data



```
}

@QueryMapping
public Optional<Beer> beer(@Argument String beerId) { return beerRepository.findById(beerId); }

@QueryMapping
public Iterable<Beer> beers() { return beerRepository.findAll(); }

@MutationMapping
public Beer updateBeerName(@Argument String beerId, @Argument String newName) {
  return beerAdvisorService.updateBeer(beerId, newName);
}

@QueryMapping
public Shop shop(@Argument String shopId) { return shopRepository.findShop(shopId); }

@QueryMapping
public List<Shop> shops() { return shopRepository.findAll(); }

@MutationMapping
public Rating addRating(@Valid @Argument AddRatingInput ratingInput) {
  logger.debug("Rating Input {}", ratingInput);
  return beerAdvisorService.addRating(ratingInput.userId(),
    ratingInput.beerId(),
    ratingInput.comment(),
    ratingInput.stars()
  );
}
```

Demo GraphQL with Java

https://spring.io/projects/spring-graphql

GRAPHQL SCHEMA

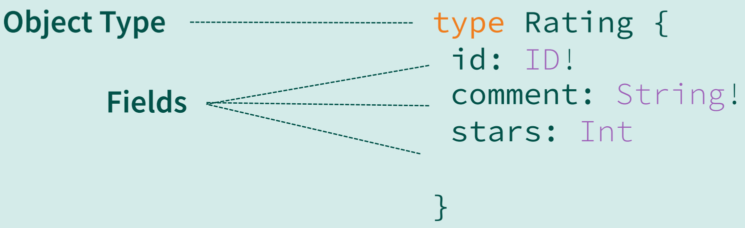
Step 1: GraphQL schema

- Every GraphQL API *must* be defined in a Schema
- The schema defines *Types* and *Fields*
- Only requests and responses that match the schema are processed and returned to the client

- **Schema Definition Language** (SDL)

GRAPHQL SCHEMA

Schema Definition with SDL



Object Type

Fields

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

}
```

GRAPHQL SCHEMA

Schema Definition with SDL

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

}
```

Return Type (non-nullable)

Return Type (nullable)

GRAPHQL SCHEMA

Schema Definition with SDL

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

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

Referenz auf anderen Typ

GRAPHQL SCHEMA

Schema Definition with 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

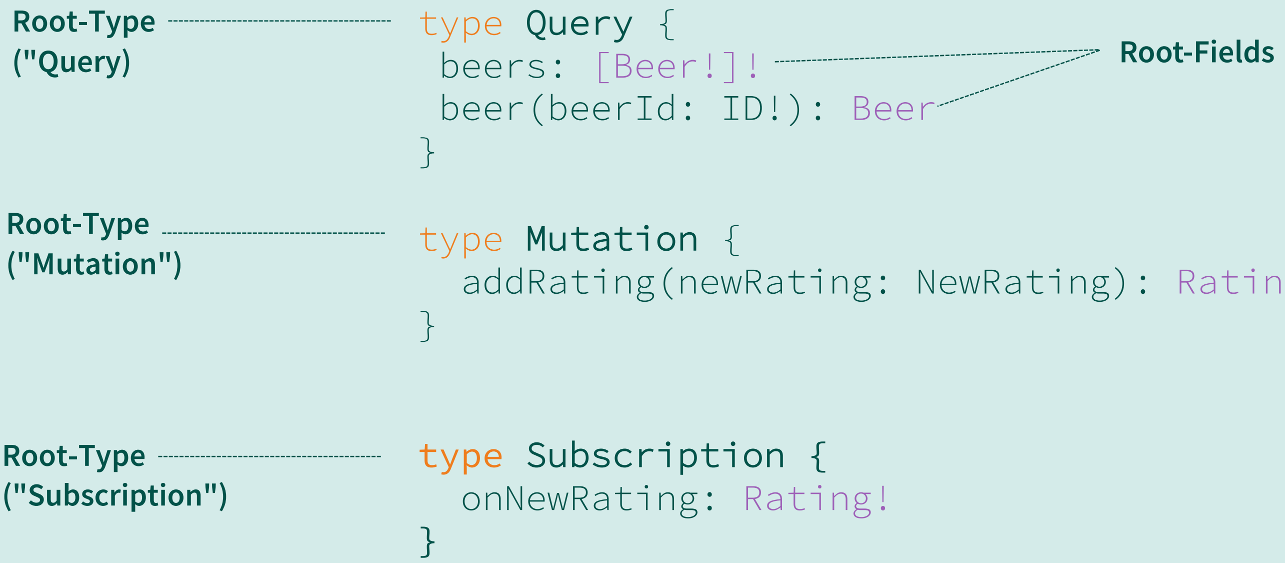
Schema Definition with SDL

```
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)

Root-Fields

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

Root-Type
("Mutation")

```
type Mutation {
  addRating(newRating: NewRating): Rating!
}
```

Root-Type
("Subscription")

```
type Subscription {
  onNewRating: Rating!
}
```

GRAPHQL BACKENDS

Example: graphql-java

- Note that there are other (high level) frameworks for Java (Spring for GraphQL, MicroProfile GraphQL) that you should consider, but all of these are backed by graphql-java

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

- In graphql-java resolver functions are called **DataFetcher**

DATA FETCHERS

DataFetcher

- In graphql-java resolver functions are called **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

- In graphql-java resolver functions are called **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 starts = 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;
    }
  };
}
```

OBJECT GRAPHS

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);
    }
  };
}
```

Thank you!

Slides: https://graphql.schule/api-day2023 (PDF)

Source code: https://github.com/nilshartmann/spring-graphql-talk

Contakt: nils@nilshartmann.net