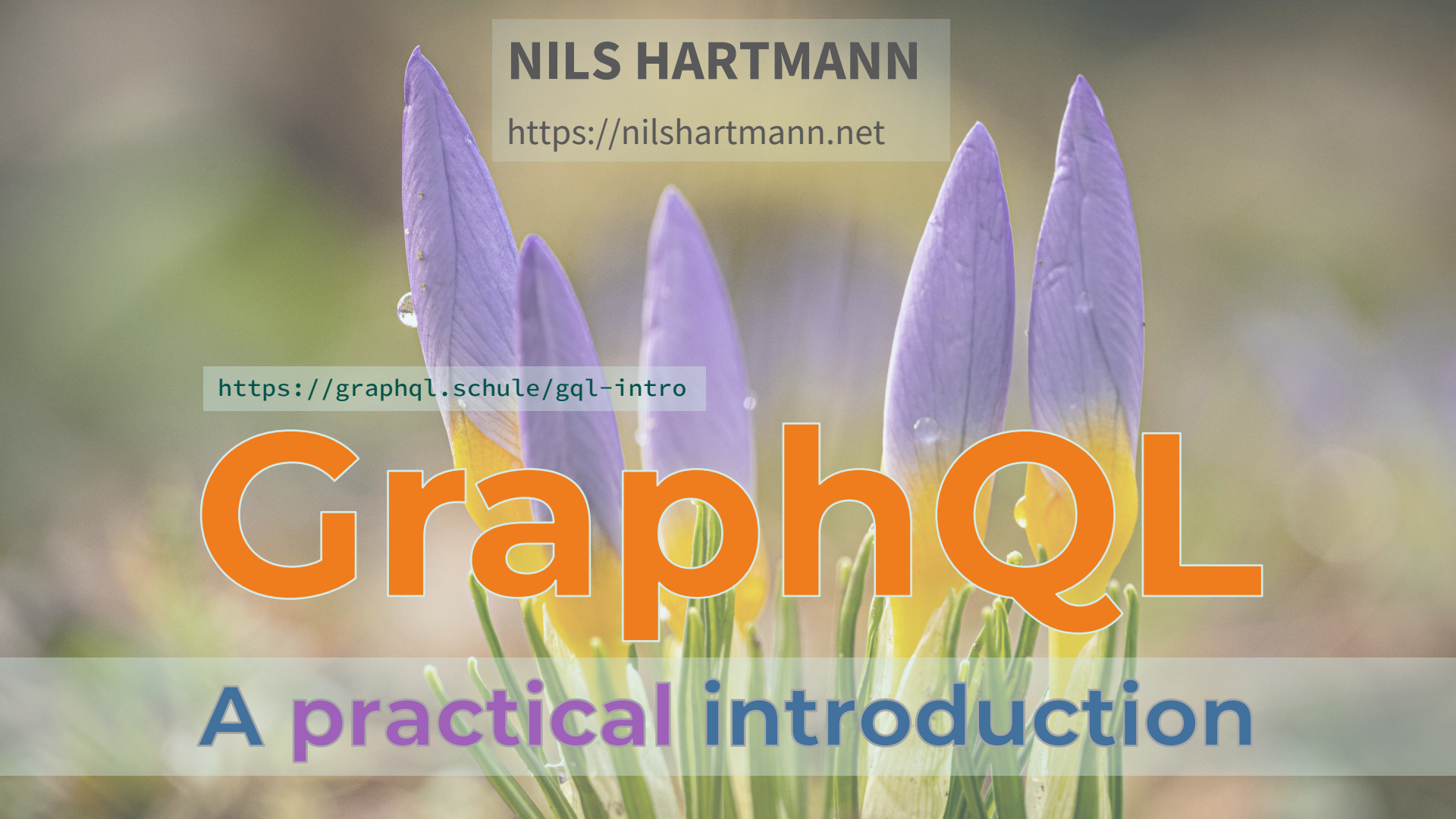
NILS HARTMANN

https://nilshartmann.net

https://graphql.schule/gql-intro



GraphQL

A practical introduction

NILS HARTMANN

nils@nilshartmann.net

Software Developer, Architect and Coach from Hamburg

Java, Spring, GraphQL, TypeScript, React

https://graphql.schule/video-kurs

https://reactbuch.de

GraphQL

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

- https://graphql.org

GraphQL

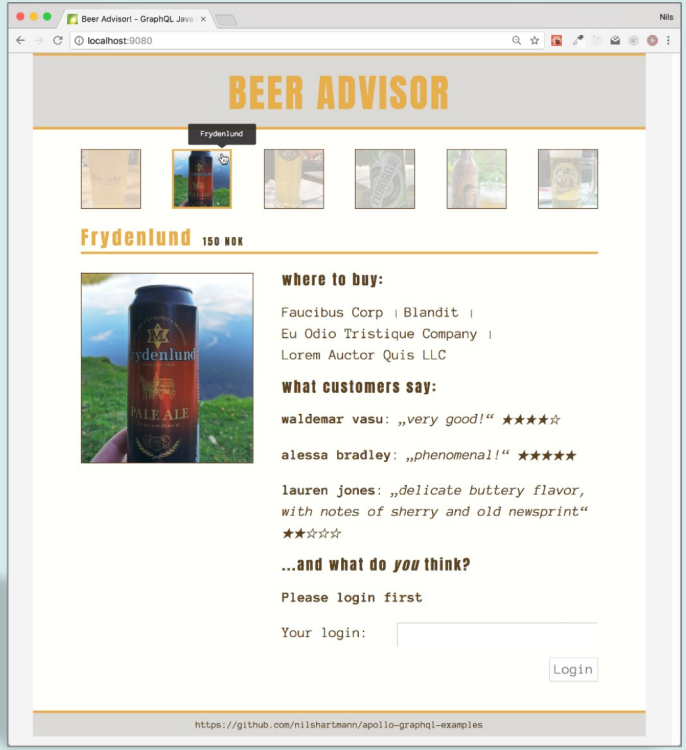
Specifikation: https://graphql.org

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

Specifikation: https://graphql.org

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

- Not a database
- Not a "product"



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

/api/shop

/api/rating

AN API FOR THE BEERADVISOR

Approach 1: Backend defines the API / data

HTTP APIs / REST

/api/beer

/api/shop

/api/rating

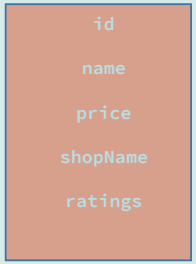
AN API FOR THE BEERADVISOR

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

Backend for Frontend (BfF)

/api/home

/api/beer-view



/api/shopdetails

Approach 3: GraphQL...

Approach 3: GraphQL...

- As approach 1: Server defines the data model

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 } }
```

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**

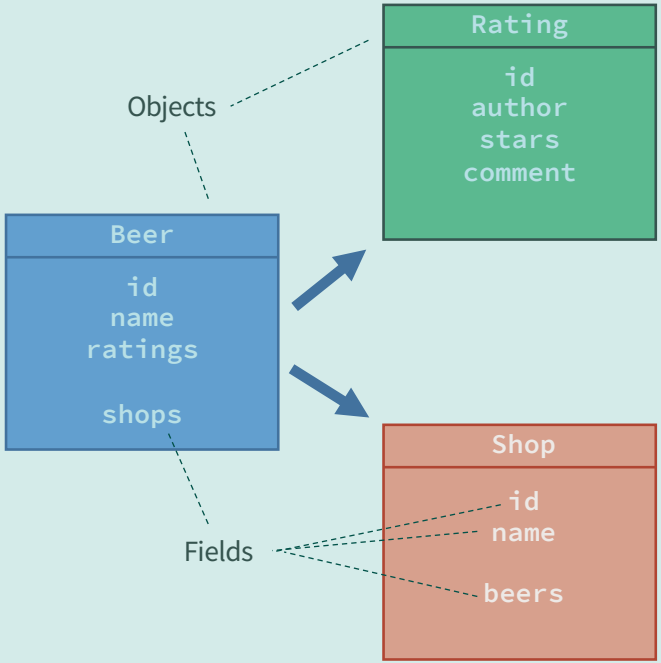
👉 **GraphQL does not create an API "magically" for us**

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

- https://graphql.org

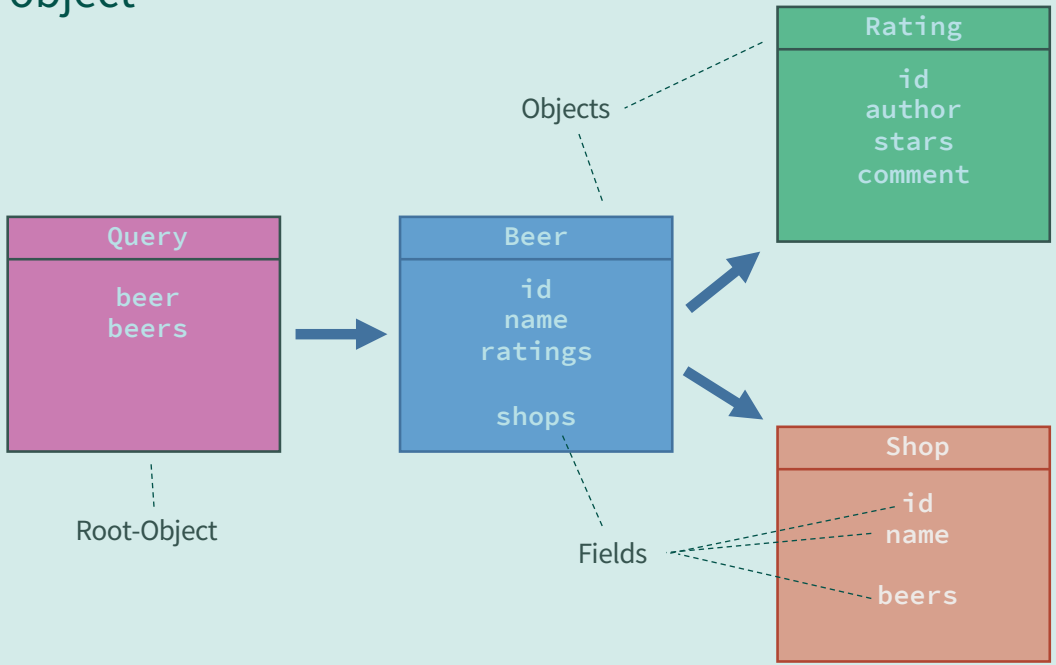
GraphQL

With the query language, you select fields from objects



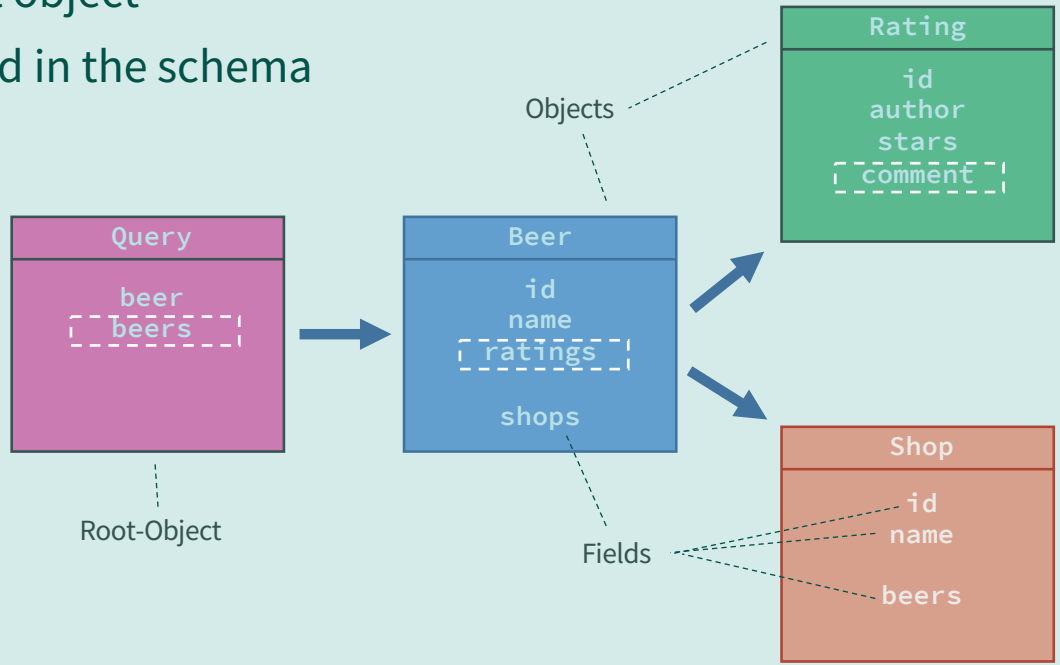
With the query language, you select fields from objects

- All queries start on a special root object



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

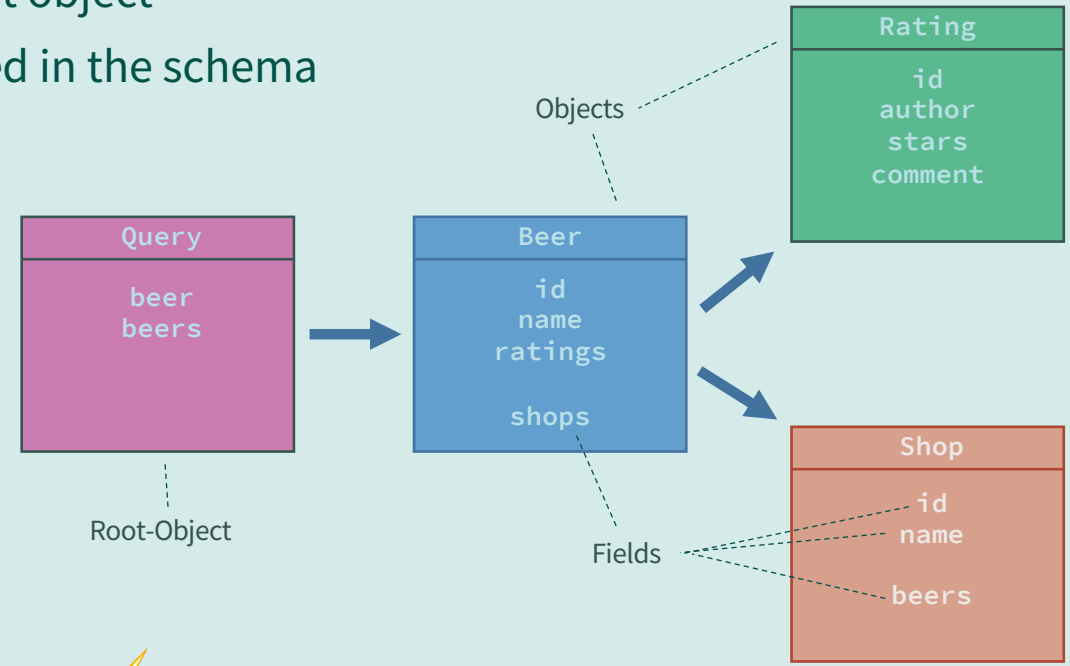



```
query { beers { ratings { comment } } }
```

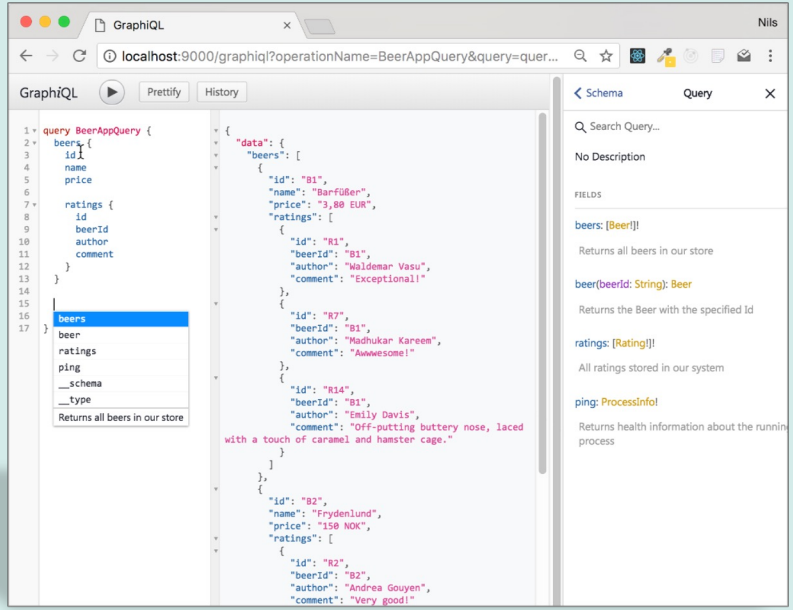
QUERY LANGUAGE

With the query language, you select fields from objects

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



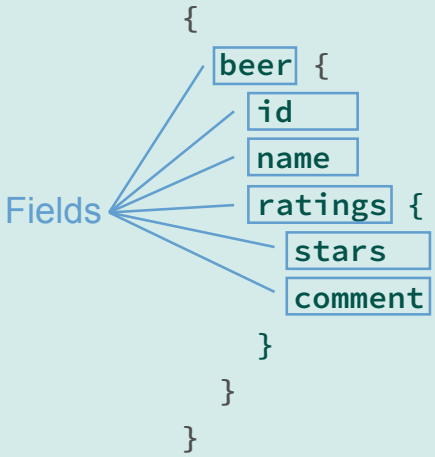

```
query { shops { id } }
```



Demo Query Language

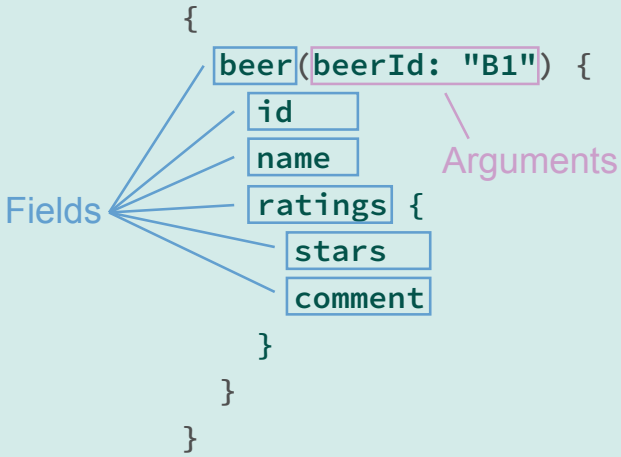
https://github.com/graphql/graphiql

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
    }
  }
}
```

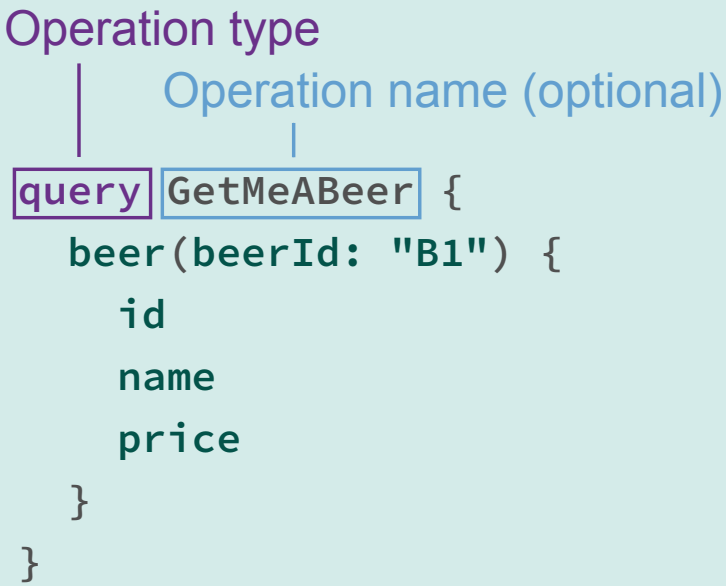
```
"data": {
  "beer": {
    "id": "B1"
    "name": "Barfüßer"
    "ratings": [
      {
        "stars": 3,
        "comment": "grate taste"
      },
      {
        "stars": 5,
        "comment": "best beer ever!"
      }
    ]
  }
}
```

- Identical structure as your query

QUERY LANGUAGE: OPERATIONS

Operation: describe, what the query should do

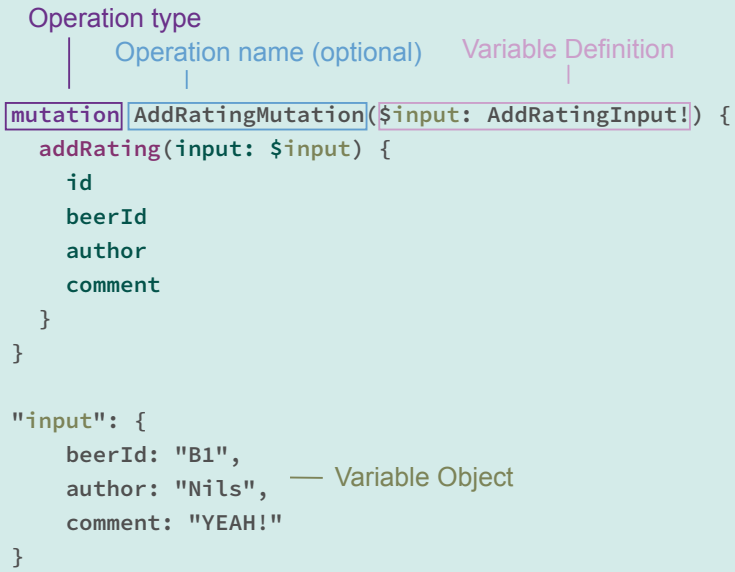
- query, mutation, subscription



QUERY LANGUAGE: MUTATIONS

Mutations

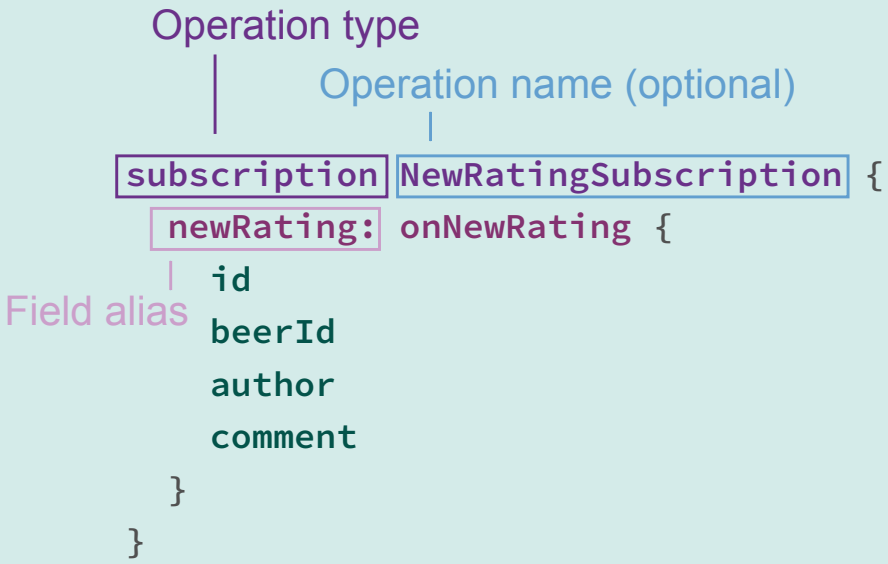
- Mutations can be used to modifiy data
- (would be POST, PUT, PATCH, DELETE in REST)



QUERY LANGUAGE: MUTATIONS

Subscription

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



EXECUTING QUERIES

Queries 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

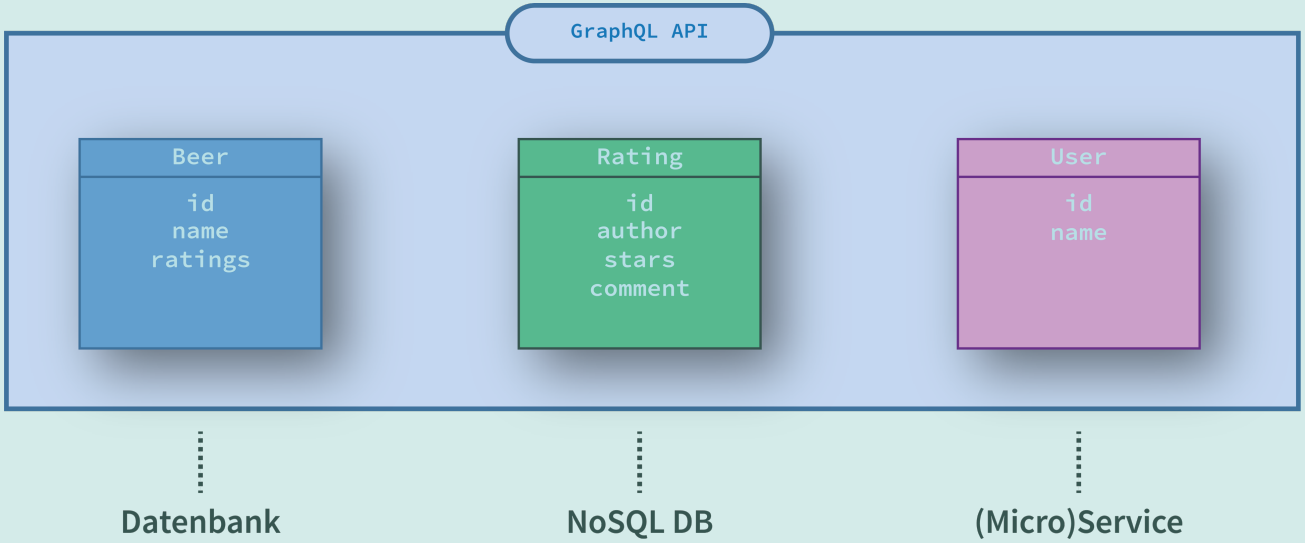
RUNTIME (AKA: YOUR APPLICATION)

GRAPHQL 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*



Implementing a GraphQL API

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

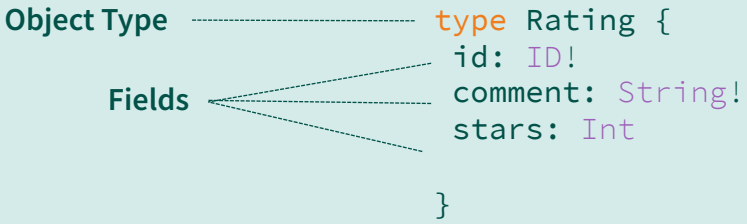
GRAPHQL SCHEMA

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)

Schema Definition with SDL



Schema Definition with SDL

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

}
```

Return Type (non-nullable)

Return Type (nullable)

Schema Definition with SDL

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

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

Referenz auf anderen Typ

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

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!]!
}
```

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!
}
```

Step 2: Implement your 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

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...

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

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

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

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

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

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

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);
}
```

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 class SubscriptionDataFetchers {
  DataFetcher<Publisher<Rating>> onNewRating = new DataFetcher<>() {
    public Publisher<Rating> get(DataFetchingEnvironment env) {
      Publisher<Rating> publisher = getRatingPublisher();

    return publisher;
    }
  };
}
```

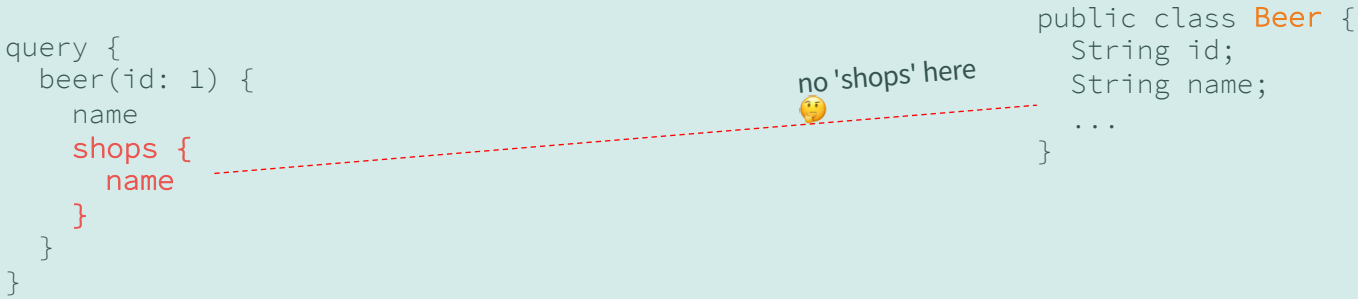
DataFetcher for own Types (not Root Types)

- 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

DataFetcher for own Types (not Root Types)

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



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
    }
  }
}
```

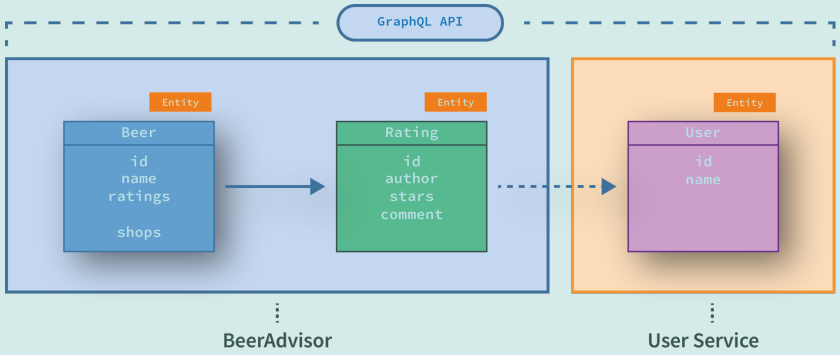
```
public class BeerDataFetchers {
  DataFetcher<List<Shop>> shops = new DataFetcher<>() {
    public String get(DataFetchingEnvironment env) {
      Beer parent = env.getSource();
      String beerId = parent.getId();

      return shopRepository.findShopsSellingBeer(beerId);
    }
  };
}
```

DATA LOADER

🤔 What is problematic with this query?

```
{ beer (id: 3) {
    ratings {
        comment
        author {
            name
        }
    }
}
```



Example: accessing remote services from resolver functions

```
{ beer (id: 3) {
    ratings {
        comment
        author {
            name
        }
    }
}
```

1. Beer-DataFetcher returns a Beer from the DB (one DB call)

```
public DataFetcher<Beer> beerFetcher() {
    return env -> {
      String beerId = env.getArgument("beerId");
      return beerRepository.getBeer(beerId);
    };
  }
```

Example: accessing remote services from resolver functions

```
{ beer (id: 3) {
    ratings {
        comment
        author {
            name
        }
    }
}
```

1. Beer-DataFetcher returns a Beer from the DB (one DB call)

```
public DataFetcher<Beer> beerFetcher() {
    return env -> {
      String beerId = env.getArgument("beerId");
      return beerRepository.getBeer(beerId);
    };
  }
```

2. The Beer has *n* Ratings (will be fetched in the same SQL query using a JOIN condition)

Example: accessing remote services from resolver functions

```
{ beer (id: 3) {
    ratings {
        comment
        author {
            name
        }
    }
}
```

1. Beer-DataFetcher returns a Beer from the DB (one DB call)

```
public DataFetcher<Beer> beerFetcher() {
    return env -> {
      String beerId = env.getArgument("beerId");
      return beerRepository.getBeer(beerId);
    };
  }
```

2. The Beer has *n* Ratings (will be fetched in the same SQL query using a JOIN condition)

3. author-DataFetcher returns User *per Rating* (n calls to the remote service)

```
public DataFetcher<User> authorFetcher() {
  return env -> {
    Rating rating = environment.getSource();
    String userId = rating.getUserId();

    return userService.getUser(userId);
  }
}
```

=> 1 (Beer) + n (User)-Calls 😢

DataLoader

- Concept from the JS reference implementation
- Not part of the spec
- Almost all frameworks support them or have similiar concepts

A DataLoader is able to:

- Batch requests to datasources together
- Cache data from datasources for the lifetime of the GraphQL request
- run asynchronously

Example

```
{ beer (id: 3) {
    ratings {
        comment
        author {
            name
        }
    }
}
```

1. Beer DataFetcher still returns Beer from the DB (as seen before)

DATA LOADER

Example

```
{ beer (id: 3) {
    ratings {
        comment
        author {
            name
        }
    }
}
```

1. Beer DataFetcher still returns Beer from the DB (as seen before)

2. author DataFetcher delegates loading of the data to a DataLoader.
   DataLoader deferes loading of the data as long as possible.

```
public DataFetcher authorFetcher() {
  return env -> {
    Rating rating = environment.getSource();
    String userId = rating.getUserId();

    DataLoader<String, User> dataLoader =
      env.getDataLoader("user");

    return dataLoader.load(userId);

  };
}
```

Collects all ids that then can be requested in one call to the remote service

DATA LOADER

Example

```
{ beer (id: 3) {
    ratings {
        comment
        author {
            name
        }
    }
}
```

1. Beer DataFetcher still returns Beer from the DB (as seen before)

2. author DataFetcher delegates loading of the data to a DataLoader.
   DataLoader deferes loading of the data as long as possible.

```
public DataFetcher authorFetcher() {
  return env -> {
    Rating rating = environment.getSource();
    String userId = rating.getUserId();

    DataLoader<String, User> dataLoader =
      env.getDataLoader("user");

    return dataLoader.load(userId);

  };
}
```

Collects all ids that then can be requested in one call to the remote service

=> 1 (Beer) + 1 (Remote)-Call 😊

Implementing the dataloader

The dataloader is a specific implementation that gets all ids that are collected in the data fetcher

```
public BatchLoader userBatchLoader = new BatchLoader<String, User>() {

  public CompletableFuture<List<User>> load(List<String> userIds) {
    return CompletableFuture.supplyAsync(() -> userService.findUsersWithId(userIds));
  }

};
```

Ids that are collected with "load" in the DataFetcher

One call to remote service

Thank you!

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

Contakt: nils@nilshartmann.net