



NILS HARTMANN

<https://nilshartmann.net>

GraphQL

mit Spring

vom Schema bis
zur Implementierung

Slides (PDF): <https://graphql.schule/entwickler-de>

NILS HARTMANN

nils@nilshartmann.net

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

Java, Spring, GraphQL, TypeScript, React



<https://graphql.schule/video-kurs>



<https://reactbuch.de>

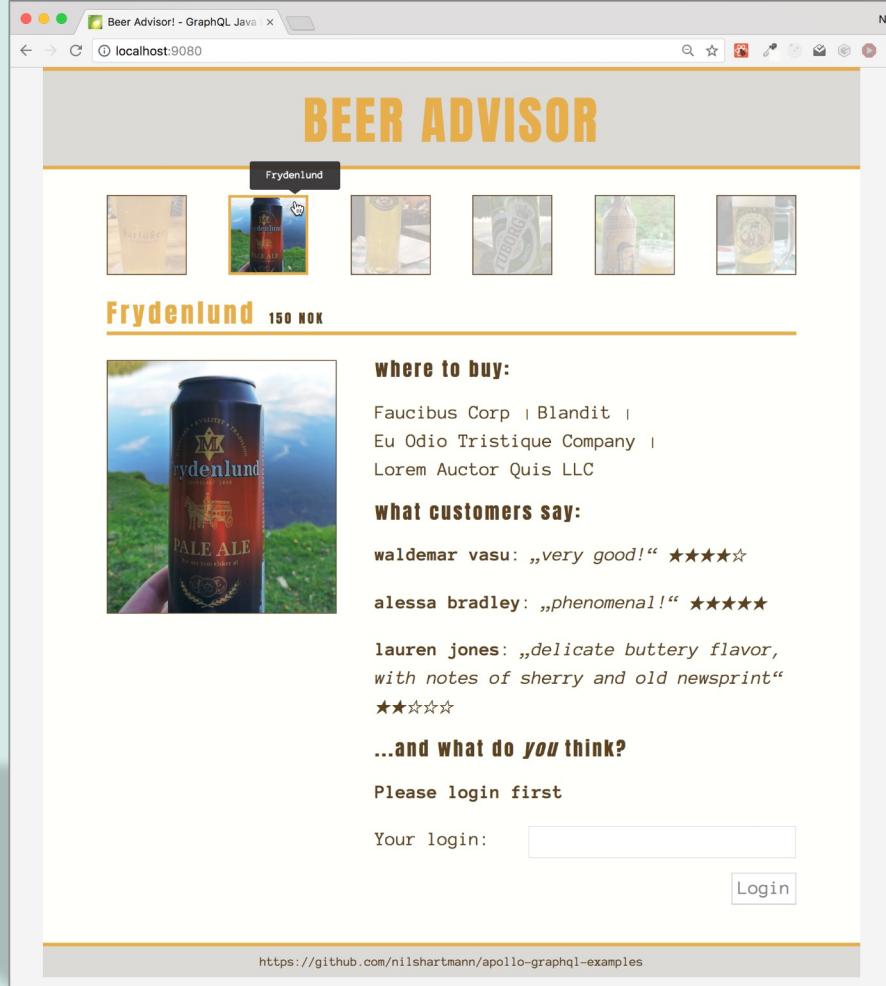
HTTPS://NILSHARTMANN.NET

GraphQL

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

- <https://graphql.org>

GraphQL



Beispiel Anwendung

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

EINE API FÜR DEN BEERADVISOR

Ansatz 1: Backend bestimmt Aussehen der Endpunkte / Daten

/api/beer

| Beer |
|----------------|
| id |
| name |
| price |
| ratings |
| shops |

/api/shop

| Shop |
|--------|
| id |
| name |
| street |
| city |
| phone |

/api/rating

| Rating |
|---------|
| id |
| author |
| stars |
| comment |

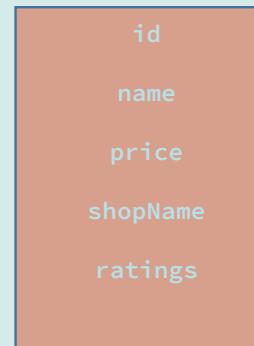
EINE API FÜR DEN BEERADVISOR

Ansatz 2: Client diktiert die API nach seinen Anforderungen

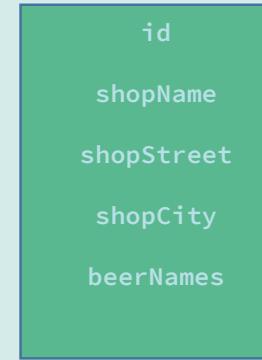
/api/home



/api/beer-view



/api/shopdetails



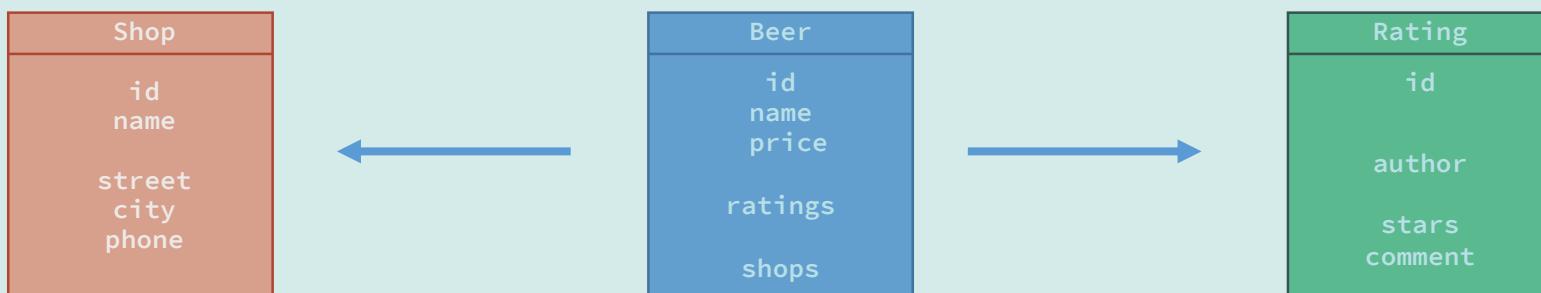
EINE API FÜR DEN BEERADVISOR

Ansatz 3: GraphQL...

EINE API FÜR DEN BEERADVISOR

Ansatz 3: GraphQL...

- Aus Ansatz 1: Server bestimmt, wie Datenmodell aussieht

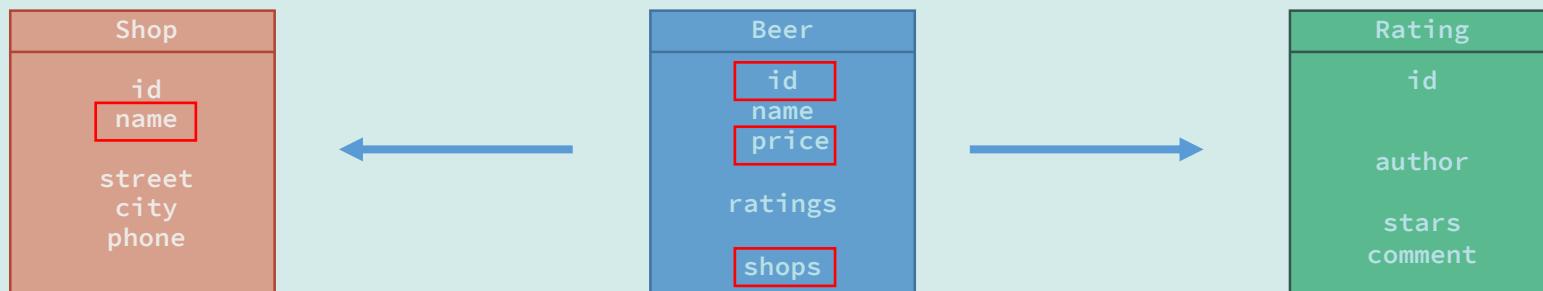


EINE API FÜR DEN BEERADVISOR

Ansatz 3: GraphQL...

- Aus Ansatz 1: Server bestimmt, wie Datenmodell aussieht
- ...aber Client kann pro Ansicht wählen, welche Daten er daraus benötigt

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



The screenshot shows the GraphiQL interface running at localhost:9000/graphiql?operationName=BeerAppQuery&query=quer.... The left panel displays a GraphQL query for a "BeerAppQuery". The right panel shows the resulting JSON data and a detailed schema pane.

GraphQL Query:

```
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
```

Result Data:

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

Schema Panel:

- 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 running process

Demo

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

Spezifikation: <https://graphql.org/>

- Umfasst:
 - Query Sprache und -Ausführung
 - Schema Definition Language
- Kein fertiges Produkt

Wir veröffentlichen mit GraphQL eine fachliche API

- Welche Daten wir zur Verfügung stellen ist unsere Aufgabe
- Wir legen fest, in welcher Form die Daten zur Verfügung gestellt werden

👉 Wir legen damit explizit selbst fest, wie unsere API aussehen soll

Wir veröffentlichen mit GraphQL eine fachliche API

- Welche Daten wir zur Verfügung stellen ist unsere Aufgabe
- Wir legen fest, in welcher Form die Daten zur Verfügung gestellt werden

👉 Wir legen damit explizit selbst fest, wie unsere API aussehen soll

👉 Auch GraphQL erzeugt die API nicht auf „magische“ Weise selbst

- API und API-Zugriffe sind typsicher
- Sehr gutes Tooling vorhanden
- Viel aus einer Hand

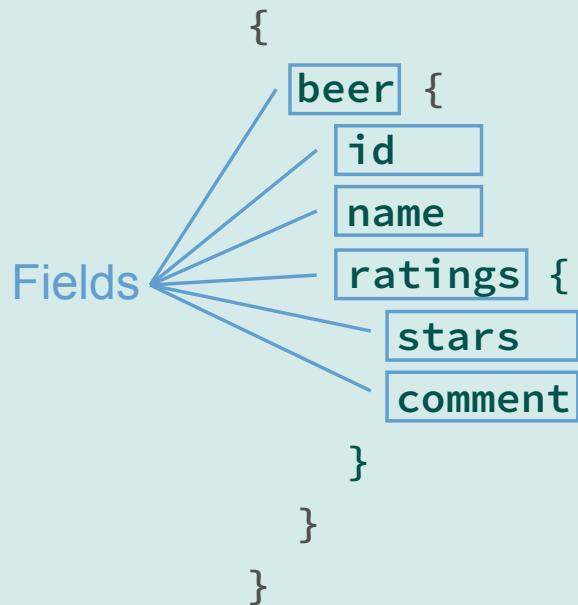
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

GraphQL

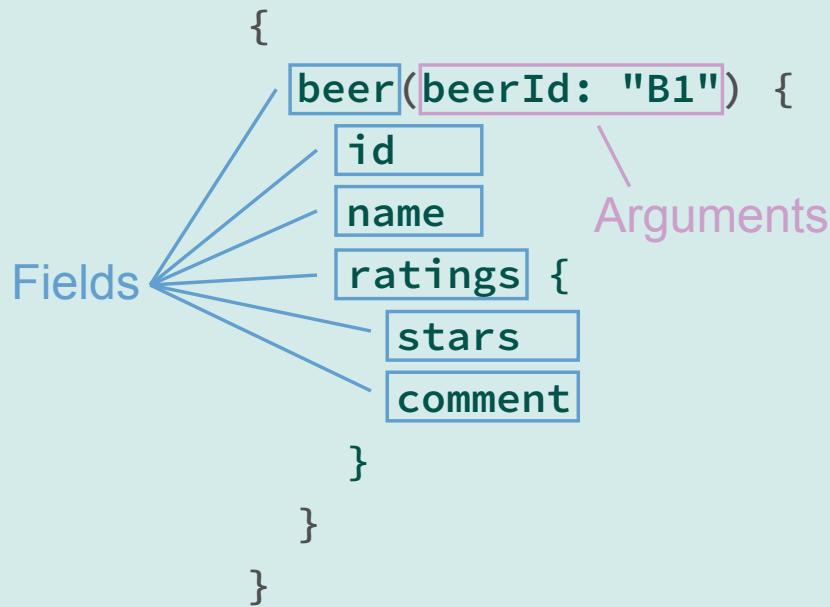
Die GraphQL Query Sprache

QUERY LANGUAGE



- Strukturierte Sprache, um Daten von der API abzufragen
- Abgefragt werden **Felder** von (verschachtelten) Objekten

QUERY LANGUAGE



- Strukturierte Sprache, um Daten von der API abzufragen
- Abgefragt werden **Felder** von (verschachtelten) Objekten
- Felder können **Argumente** haben

QUERY LANGUAGE

Ergebnis

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

- Identische Struktur wie bei der Abfrage
- *Query ist ein String, kein JSON!*

QUERY LANGUAGE: OPERATIONS

Operation: beschreibt, was getan werden soll

- query, mutation, subscription

Operation type

 | Operation name (optional)

 |
 query GetMeABeer {

 beer(beerId: "B1") {

 id

 name

 price

 }

}

QUERY LANGUAGE: MUTATIONS

Mutations

- Mutation wird zum Verändern von Daten verwendet
- Entspricht POST, PUT, PATCH, DELETE in REST
- Rückgabe Wert kann frei definiert werden (z.B. neue Entität)

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

- Automatische Benachrichtigung bei neuen Daten
- API definiert Events (mit Feldern), aus denen der Client auswählt

Operation type

 |

 Operation name (optional)

 |

 subscription **NewRatingSubscription** {

 newRating: onNewRating {

 |

 Field alias id

 beerId

 author

 comment

 }

 }

QUERIES AUSFÜHREN

Queries werden über HTTP ausgeführt

- „Normaler“ HTTP Endpunkt
 - Queries üblicherweise per POST
 - Ein *einzelner* Endpunkt, z.B. /graphql
 - HTTP Verben spielen keine Rolle

QUERIES AUSFÜHREN

Queries werden über HTTP ausgeführt

- „Normaler“ HTTP Endpunkt
 - Queries üblicherweise per POST
 - Ein *einzelner* Endpunkt, z.B. /graphql
 - HTTP Verben spielen keine Rolle
- Der GraphQL-Endpunkt kann parallel zu anderen Endpunkten bestehen
 - REST und GraphQL kann problemlos gemischt werden
- Wie die Anbindung aussieht hängt vom Framework und Umgebung (Spring / JEE) ab

TEIL II

GraphQL Server

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

- <https://graphql.org>

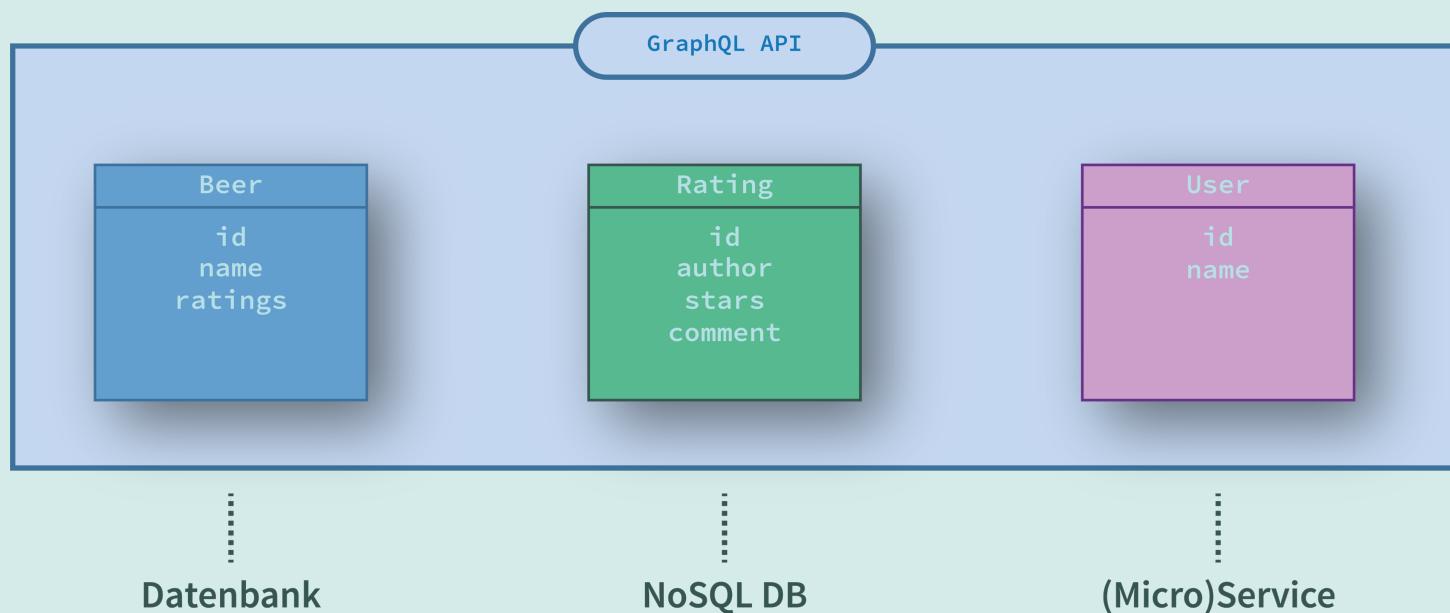
GraphQL Server

RUNTIME (AKA: YOUR APPLICATION)

GRAPHQL APIs

GraphQL macht keine Aussage, wo die Daten herkommen

- 👉 Ermittlung der Daten ist unsere Aufgabe
- 👉 Müssen nicht aus einer Datenbank kommen



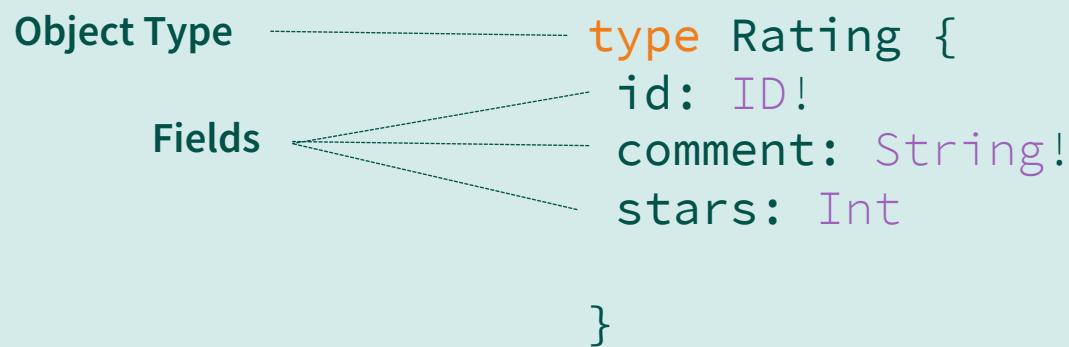
GRAPHQL SCHEMA

Die GraphQL API muss in einem *Schema* beschrieben werden

- Eine GraphQL API muss mit einem *Schema* beschrieben werden
- Schema legt fest, welche *Types* und *Fields* es gibt
- Nur Anfragen und Ergebnisse, die Schema-konform sind werden ausgeführt bzw. zurückgegeben
- **Schema Definition Language** (SDL)

GRAPHQL SCHEMA

Schema Definition per SDL



GRAPHQL SCHEMA

Schema Definition per SDL

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

GRAPHQL SCHEMA

Schema Definition per SDL

```
type Rating {  
    id: ID!  
    comment: String!  
    stars: Int  
    author: User! ----- Referenz auf anderen Typ  
}  
  
type User {  
    id: ID!  
    name: String!  
}
```



GRAPHQL SCHEMA

Schema Definition per 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 per 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: Einstiegspunkte in die 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> | |

Spring for GraphQL

- <https://spring.io/projects/spring-graphql>
- “Offizielle” Spring Lösung für GraphQL in Spring
 - Verbindet graphql-java mit with Spring Boot (Konzepten)
 - Stellt GraphQL Endpunkt über Spring WebMVC oder Spring WebFlux zur Verfügung
 - Support für Subscriptions über WebSockets
 - Alle Spring-Features in GraphQL-Schicht wie gewohnt nutzbar
- Enthalten in Spring Boot 2.7 (aktuell RC1)

Annotated Controllers

- Annotation-basiertes Programmiermodell, ähnlich wie in REST Controllern von Spring

@Controller

```
public class BeerQueryController {  
  
    BeerQueryController(BeerRepository beerRepository) { ... }  
  
@QueryMapping                                Mapping auf das Schema mit Namenskonventionen  
public List<Beer> beers() {  
    return beerRepository.findAll();  
}  
  
@MutationMapping  
public Rating addRating(@Argument AddRatingInput input) {  
    return ratingService.createRating(input);  
}
```

}

Annotated Controllers

- Annotation-basiertes Programmiermodell, ähnlich wie in REST Controllern von Spring

@Controller

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

@QueryMapping

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

Argumente via Methoden Parameter

@MutationMapping

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

Annotated Controllers

- Annotation-basiertes Programmiermodell, ähnlich wie in REST Controllern von Spring

@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);  
}
```

Eltern-Element als Methoden Parameter

@SchemaMapping

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

Performance-Optimierung

- Handler-Funktionen können asynchron sein

@Controller

```
public class RatingController {  
  
    RatingController(...) { ... }
```

Beispiel: Reaktiver Zugriff auf Micro-Service per HTTP

@SchemaMapping

```
public Mono<User> author(Rating rating) {  
    return userService.findUser(rating.getUserId());  
}
```

Beispiel: Zugriff auf asynchronen Spring-Service
(@Async)

@SchemaMapping

```
public CompletableFuture<Float> averageRating(Beer beer) {  
    return ratingService.calculateAvgRating(beer.getRatings());  
}
```

```
}
```

Security

- GraphQL Requests kommen über "normale" Spring Endpunkte
- Integration mit Spring Security
- HTTP-Endpunkt absichern und/oder einzelne Handler-Funktionen und/oder Domain-Schicht (ähnlich wie bei REST)

@Controller

```
public class BeerQueryController {  
  
    BeerQueryController(BeerRepository beerRepository) { ... }  
  
    @PreAuthorize("hasRole('EDITOR')")  
    @MutationMapping  
    public Rating addRating(@Argument AddRatingInput input) {  
        return ratingService.createRating(input);  
    }  
  
}
```

Validation

- Argumente können mit Bean Validation validiert werden
- Zum Beispiel für Größen- oder Längenbeschränkungen

```
record AddRatingInput(  
    String beerId,  
    String userId,  
    @Size(max=128) String comment,  
    @Max(5) int stars) { }  
}
```

@Controller

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

@MutationMapping

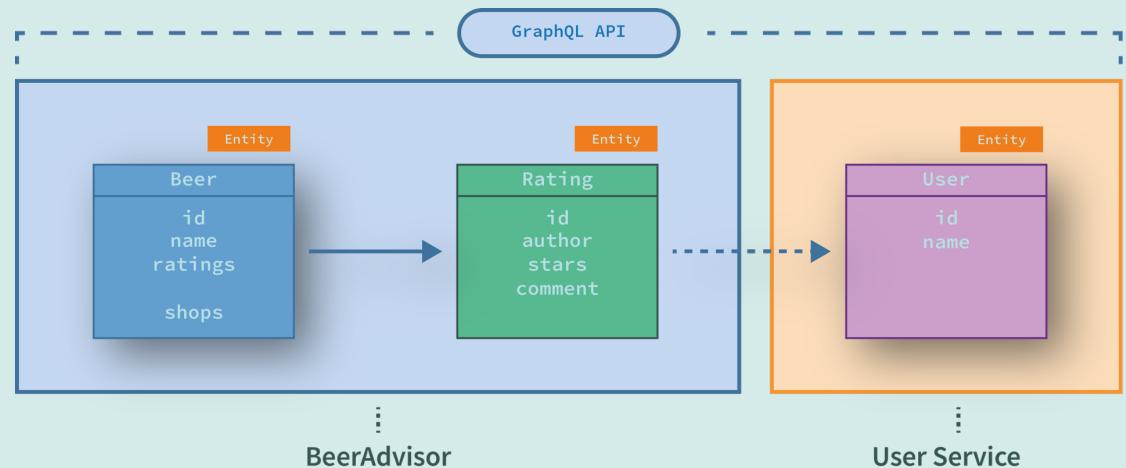
```
public Rating addRating(@Valid @Argument AddRatingInput input) {  
    return ratingService.createRating(input);  
}  
}
```

DATA LOADER



Was gibt es bei der Ausführung dieses Querys für ein Problem?

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



DATA LOADER (GRAPHQL-JAVA)

Beispiel: Zugriff auf (Remote-)Services

1. Beer-DataFetcher liefert Beer zurück
(ein DB-Aufruf)

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

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

DATA LOADER (GRAPHQL-JAVA)

Beispiel: Zugriff auf (Remote-)Services

1. Beer-DataFetcher liefert Beer zurück
(ein DB-Aufruf)

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

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

2. Am Beer hängen n **Ratings** (werden im selben SQL-Query aus der DB als Join mitgeladen)

DATA LOADER (GRAPHQL-JAVA)

Beispiel: Zugriff auf (Remote-)Services

1. Beer-DataFetcher liefert Beer zurück
(ein Aufruf)

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

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

2. Am Beer hängen n-Ratings (werden im selben SQL-Query aus der DB als Join mitgeladen)
3. author-DataFetcher liefert User *pro Rating* zurück
(n-Aufrufe zum Remote-Service)

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

Remote-Call!

DATA LOADER (GRAPHQL-JAVA)

Beispiel: Zugriff auf (Remote-)Services

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

1. Beer-DataFetcher liefert Beer zurück
(ein Aufruf)

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

2. Am Beer hängen n-Ratings (werden im selben SQL-Query aus der DB als Join mitgeladen)
3. author-DataFetcher liefert User *pro Rating* zurück
(n-Aufrufe zum 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 😭

Optimieren und Cachen von Zugriffen mit DataLoader

DataLoader kommen ursprünglich aus der JavaScript-Implementierung

Ein DataLoader kann:

- Aufrufe zusammenfassen (Batching)
- Ergebnisse cachen
- asynchron ausgeführt werden

DATA LOADER

Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück
(unverändert)

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

DATA LOADER (GRAPHQL-JAVA)

Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück
(unverändert)
2. author-DataFetcher delegiert Ermitteln der Daten
an den DataLoader.
GraphQL verzögert das eigentliche Laden der Daten
so lange wie möglich.

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

```
public DataFetcher authorFetcher() {  
    return env -> {  
        Rating rating = environment.getSource();  
        String userId = rating.getUserId();  
  
        DataLoader<String, User> dataLoader =  
            env.getDataLoader("user");  
  
        return dataLoader.load(userId);  
    };  
}
```

 Sammelt alle load-Aufrufe ein und
führt erst dann den DataLoader aus

DATA LOADER (GRAPHQL-JAVA)

Optimieren und Cachen von Zugriffen mit DataLoader

1. Beer-DataFetcher liefert Beer zurück
(unverändert)
2. author-DataFetcher delegiert Ermitteln der Daten
an den DataLoader.
GraphQL verzögert das eigentliche Laden der Daten
so lange wie möglich.

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

```
public DataFetcher authorFetcher() {  
    return env -> {  
        Rating rating = environment.getSource();  
        String userId = rating.getUserId();  
  
        DataLoader<String, User> dataLoader =  
            env.getDataLoader("user");  
  
        return dataLoader.load(userId);  
    };  
}
```

 Sammelt alle load-Aufrufe ein und führt erst dann den DataLoader aus

=> 1 (Beer) + 1 (Remote)-Call 😊

DATA LOADER (GRAPHQL-JAVA)

Optimieren und Cachen von Zugriffen mit DataLoader

Die eigentlichen Daten werden dann gesammelt in einem **BatchLoader** geladen

```
public BatchLoader userBatchLoader = new BatchLoader<String, User>() {  
  
    public CompletableFuture<List<User>> load(List<String> userIds) {  
        return CompletableFuture.supplyAsync(() -> userService.findUsersWithId(userIds));  
    }  
  
};
```

Wird von GraphQL aufgerufen mit einer *Menge* von Ids,
die aus einer *Menge* von DataFetcher-Aufrufen stammen

DATA LOADER (SPRING FOR GRAPHQL)

Spring for GraphQL bietet Unterstützung für graphql-java DataLoader

- DataLoader kann als Parameter in einer Mapping-Funktion angegeben werden
- Funktion muss dann CompletableFuture zurückliefern

```
@SchemaMapping
public CompletableFuture<User> author(Rating rating, DataLoader<String, User> dataloader) {
    String userId = rating.getUserId();
    return dataloader.load(userId);
};
```

DATA LOADER (SPRING FOR GRAPHQL)

Registrieren des DataLoaders

- DataLoader werden mit der BatchLoaderRegistry von Spring for GraphQL registriert
- Eine Instanz der BatchLoaderRegistry steht als Bean zur Verfügung
- Registriert wird ein DataLoader für ein "Typ-Pair", das aus dem Typen eines Keys (z.B. String) und dem Typen des zugehörigen Objektes besteht (z.B. User)
- Die DataLoader-Instanz muss ein Flux (BatchLoader) oder Mono<Map>-Objekt (MappedBatchLoader) zurückliefern)

```
class BeerAdvisorController {  
    public BeerAdvisorController(BatchLoaderRegistry registry) {  
        registry.forTypePair(String.class, User.class)  
            .registerBatchLoader(  
                (List<String> keys, BatchLoaderEnvironment env) -> {  
                    log.info("Loading Users with keys {}", keys);  
  
                    Flux<User> users = userService.findUsersWithIds(keys);  
                    return users;  
                }  
            );  
        // ...  
    }  
}
```



Vielen Dank!

Slides: [https://graphql.schule/entwickler-de\(PDF\)](https://graphql.schule/entwickler-de(PDF))

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

Kontakt: nils@nilshartmann.net