

NILS HARTMANN

<https://nilshartmann.net>

Slides (PDF): <https://graphql.schule/db-tage>

# GraphQL

## Eine praktische Einführung

# NILS HARTMANN

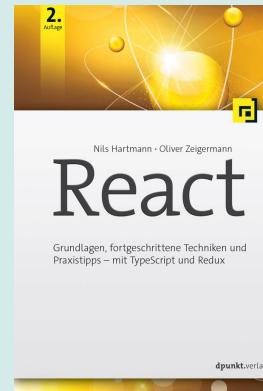
nils@nilshartmann.net

**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

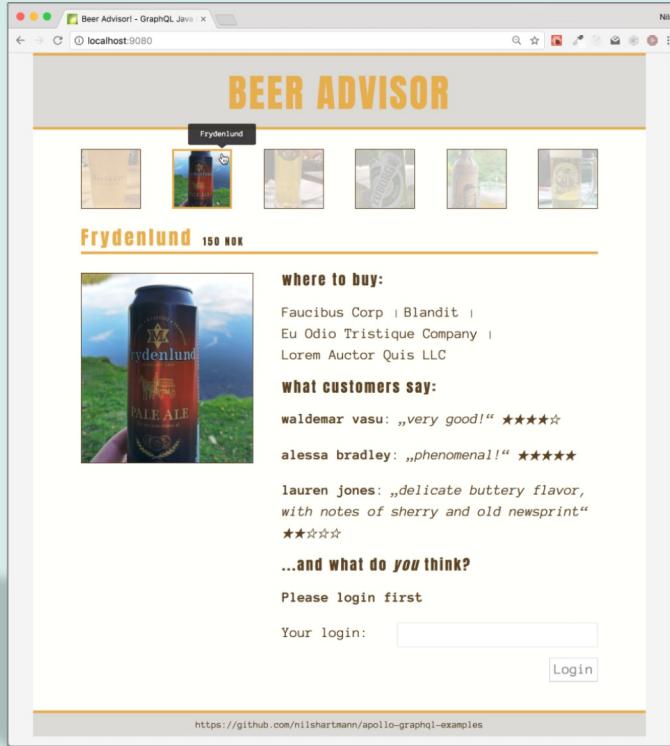
*Spezifikation: <https://graphql.org/>*

- Umfasst:
  - Query Sprache und -Ausführung
  - Schema Definition Language

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

- Umfasst:
  - Query Sprache und -Ausführung
  - Schema Definition Language
- Keine Datenbank, kein fertiges Produkt

A screenshot of a session card from a conference or event. The card has a light gray background with a green header bar at the top. At the top left, there is a small orange circular icon with the number '1'. Below it, the word 'Session' is written in white on an orange button-like background. To the right of this are five blue buttons with white text: 'Architektur', 'Entwicklung', 'Java', 'Spring Boot', and 'Datenbanken'. The 'Datenbanken' button has a large red 'X' drawn over it. In the center, the title 'GraphQL: Eine praktische Einführung' is displayed in a large, bold, blue font. Below the title, the speaker's name 'Nils Hartmann' is shown in a smaller blue font. At the bottom left, there are three icons with corresponding text: a calendar icon followed by 'Mittwoch, 28. Feb 2024', a clock icon followed by '13:00 - 13:45', and a computer monitor icon followed by 'Track 2'.



# 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 1: Backend bestimmt Aussehen der Endpunkte / Daten  
REST / HTTP APIs**

/api/beer

Beer
id
name
price
ratings
shops

/api/shop

Shop
id
name
street
city
phone

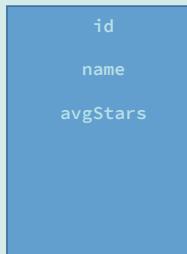
/api/rating

Rating
id
author
stars
comment

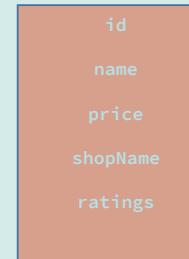
# EINE API FÜR DEN BEERADVISOR

## Ansatz 2: Client diktiert die API nach seinen Anforderungen

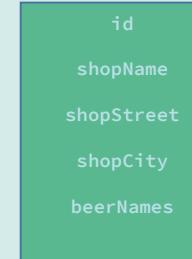
/api/home



/api/beer-view



/api/shopdetails



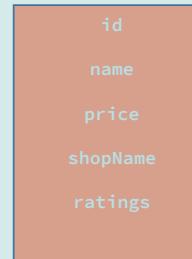
# EINE API FÜR DEN BEERADVISOR

## Ansatz 2: Client diktiert die API nach seinen Anforderungen Backend for Frontend (BFF)

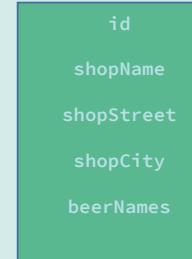
/api/home



/api/beer-view



/api/shopdetails



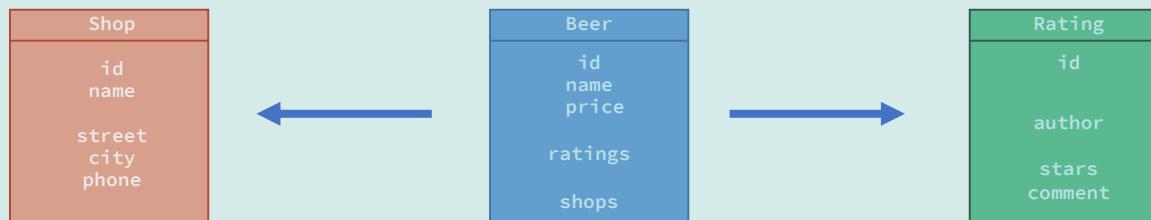
# EINE API FÜR DEN BEERADVISOR

Ansatz 3: GraphQL...

# EINE API FÜR DEN BEERADVISOR

## Ansatz 3: GraphQL...

- Aus Ansatz 1: Server bestimmt, wie Datenmodell aussieht

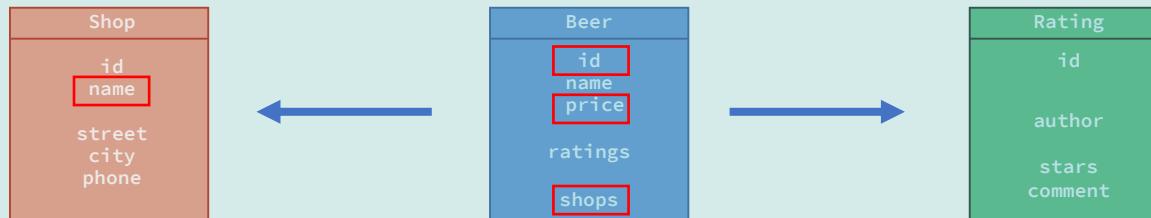


# EINE API FÜR DEN BEERADVISOR

## 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 } } }
```



### 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

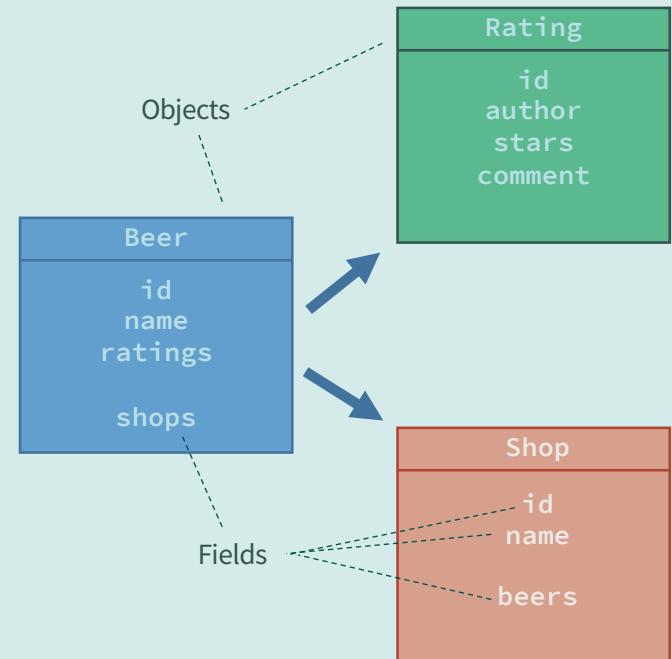
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

# GraphQL

# QUERY LANGUAGE

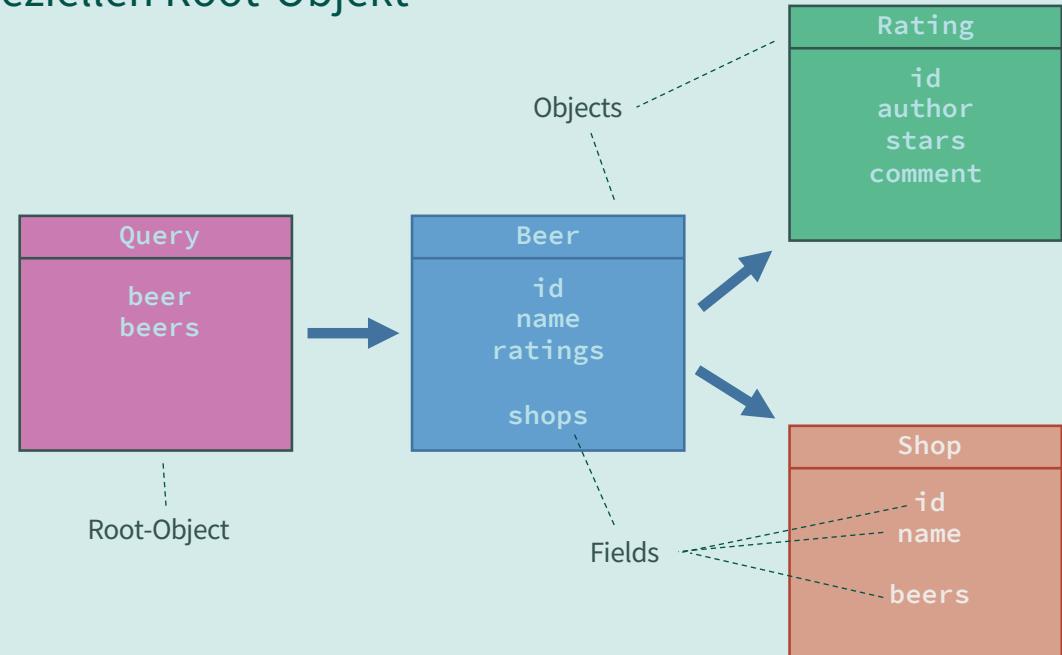
Mit der Query-Sprache werden *Felder von Objekten abgefragt*



# QUERY LANGUAGE

Mit der Query-Sprache werden *Felder von Objekten abgefragt*

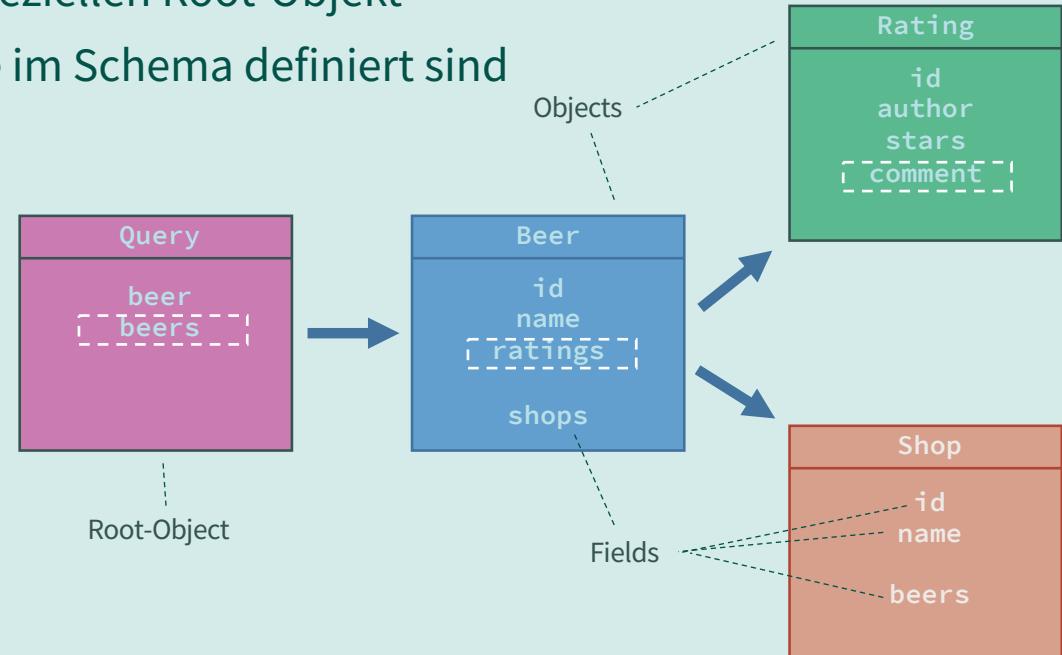
- Alle Queries starten an einem speziellen Root-Objekt



# QUERY LANGUAGE

## Mit der Query-Sprache werden *Felder von Objekten* abgefragt

- Alle Queries starten an einem speziellen Root-Objekt
- Man kann nur Pfade folgen, die im Schema definiert sind

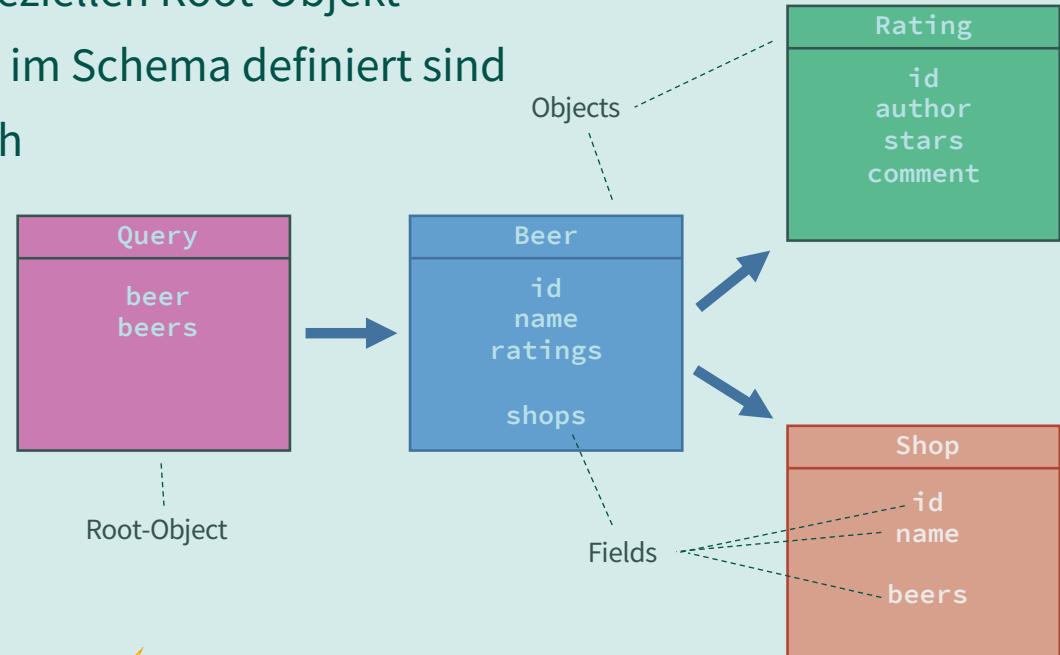


```
query { beers { ratings { comment } } }
```

# QUERY LANGUAGE

## Mit der Query-Sprache werden *Felder von Objekten* abgefragt

- Alle Queries starten an einem speziellen Root-Objekt
- Man kann nur Pfade folgen, die im Schema definiert sind
- Andere "joins" sind nicht möglich



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

The screenshot shows the GraphiQL interface running at [localhost:9000/graphiql](http://localhost:9000/graphiql). The left pane displays a GraphQL query for a 'BeerAppQuery' type:

```
query BeerAppQuery {
  beers {
    id
    name
    price
    ratings {
      id
      beerId
      author
      comment
    }
  }
  beers
  beer
  ratings
  ping
  __schema
  __type
}
```

The word 'beers' is highlighted in blue, indicating it's the currently selected field. The right pane shows the resulting JSON data:

```
{
  "data": {
    "beers": [
      {
        "id": "B1",
        "name": "Barfüßer",
        "price": "3,88 EUR",
        "ratings": [
          {
            "id": "R1",
            "beerId": "B1",
            "author": "Waldemar Vasu",
            "comment": "Exceptional!"
          },
          {
            "id": "R7",
            "beerId": "B1",
            "author": "Madhukar Kareem",
            "comment": "Awesome!"
          },
          {
            "id": "R14",
            "beerId": "B1",
            "author": "Emily Davis",
            "comment": "Off-putting buttery nose, laced with a touch of caramel and hamster cage."
          }
        ],
        "beer": {
          "id": "B2",
          "name": "Frøyenlund",
          "price": "158 NOK",
          "ratings": [
            {
              "id": "R2",
              "beerId": "B2",
              "author": "Andrea Gouyen",
              "comment": "Very good!"
            }
          ]
        }
      }
    ]
  }
}
```

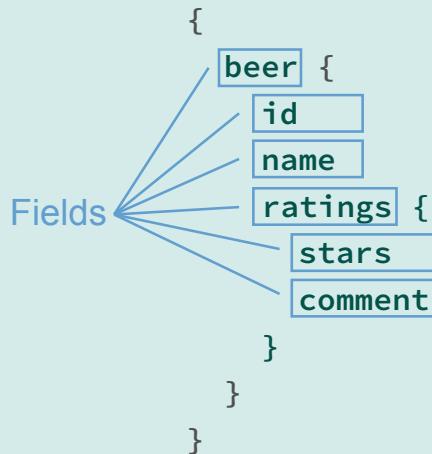
The 'beers' field is described as 'Returns all beers in our store'. The 'beer' field is described as 'Returns the Beer with the specified Id'. The 'ratings' field is described as 'All ratings stored in our system'. The 'ping' field is described as 'Returns health information about the running process'.

# Demo

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

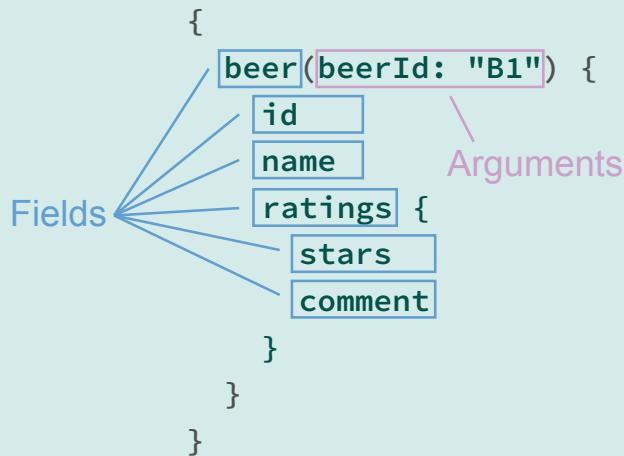
# Die GraphQL Query Sprache

# 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 {
    id
    beerId
    author
    comment
  }
}
```

### Queries werden über HTTP ausgeführt

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

### 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>

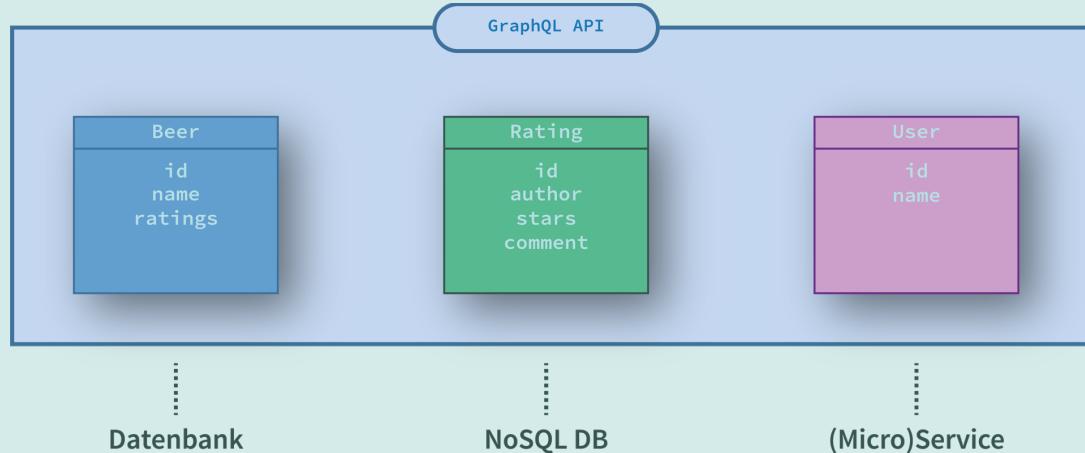
# GraphQL Server

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



### 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

Object Type -----  
Fields ----- `type Rating {  
 id: ID!  
 comment: String!  
 stars: Int  
}`

# GRAPHQL SCHEMA

## Schema Definition per SDL

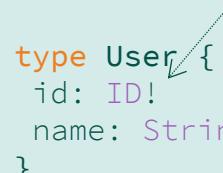
```
type Rating {  
    id: ID!           ----- Return Type (non-nullable)  
    comment: String!  
    stars: Int        ----- Return Type (nullable)  
}
```

# GRAPHQL SCHEMA

## Schema Definition per SDL

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

----- Referenz auf anderen Typ



# 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!]!  
}
```



# GRAPHQL SCHEMA

## Root-Types: Einstiegspunkte in die 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!  
}
```

## 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

## 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);
    }
}
```

Mapping auf das Schema mit Namenskonventionen

## 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);
    }
}
```

Argumente via Methoden Parameter

## 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);
    }

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

Eltern-Element als Methoden Parameter

## Performance-Optimierung

- Handler-Funktionen können asynchron sein

```
@Controller  
public class RatingController {  
  
    RatingController(...) { ... }  
  
    @SchemaMapping  
    public Mono<User> author(Rating rating) {  
        return userService.findUser(rating.getUserId());  
    }  
  
    @SchemaMapping  
    public CompletableFuture<Float> averageRating(Beer beer) {  
        return ratingService.calculateAvgRating(beer.getRatings());  
    }  
}
```

Beispiel: Reaktiver Zugriff auf Micro-Service per HTTP

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

## 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);  
}  
  
}
```



# Vielen Dank!

Slides: <https://graphql.schule/db-tage> (PDF)

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

Kontakt: [nils@nilshartmann.net](mailto:nils@nilshartmann.net)