

NILS HARTMANN
<https://nilshartmann.net>

GraphQL

APIs mit Spring Boot

Slides (PDF): <https://graphql.schule/jax2022>

NILS HARTMANN

nils@nilshartmann.net

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

Java
JavaScript, TypeScript
React
GraphQL

Trainings & Workshops



<https://reactbuch.de>

HTTPS://NILSHARTMANN.NET

TEIL 1

GraphQL

Grundlagen

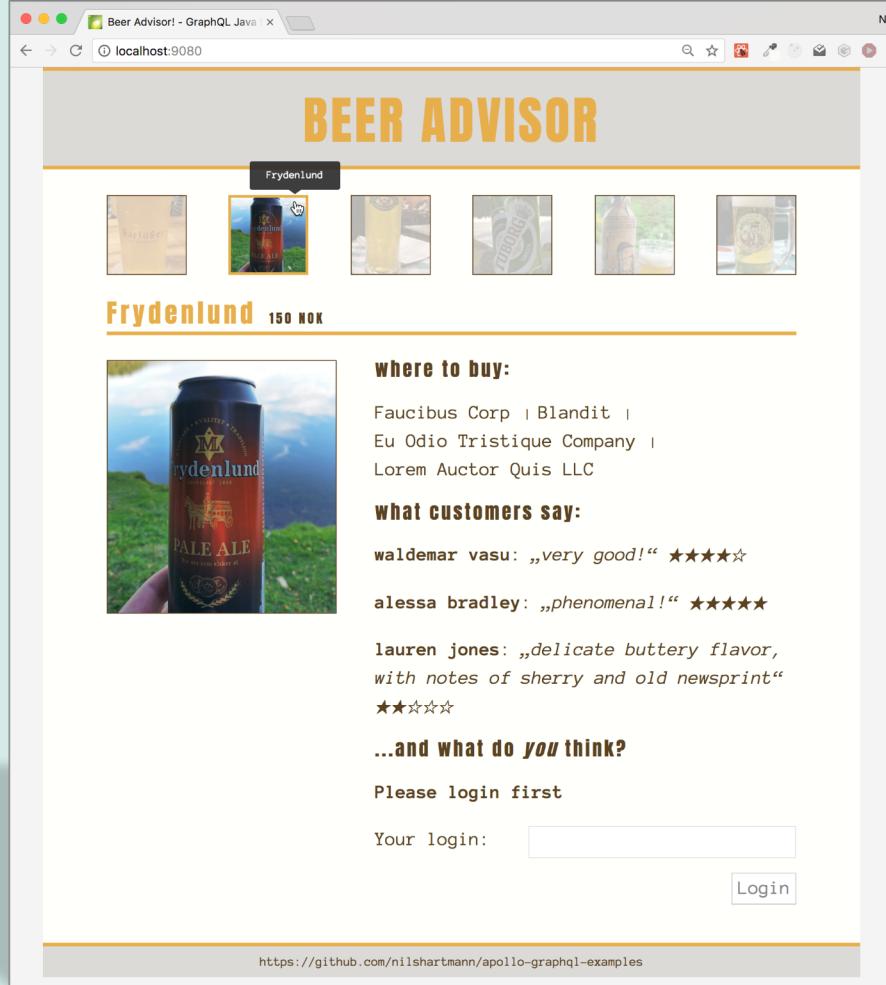
*"GraphQL is a **query language for APIs** and a **runtime for fulfilling those queries** with your existing data"*

- <https://graphql.org>

GraphQL

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

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



Beispiel Anwendung

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

The screenshot shows the GraphiQL interface running on a local host. The left panel displays a GraphQL query named 'BeerAppQuery' which retrieves data about beers, their ratings, and a comment. The right panel shows the resulting JSON data, which includes a list of beers with their names, prices, and associated ratings from different authors.

```
query BeerAppQuery {  
  beers {  
    id  
    name  
    price  
    ratings {  
      id  
      beerId  
      author  
      comment  
    }  
  }  
}  
  
beers  
beer  
ratings  
ping  
__schema  
__type  
Returns all beers in our store
```

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

Demo: GraphiQL

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

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The left sidebar shows a project named "graphql-java-example" with a "backend-tools" module. Inside "backend-tools", there's a "resources" folder containing "public" and "schema" subfolders. The "schema" folder contains "auth.graphqls", "rating.graphqls", "shop.graphqls", "application.properties", "application-postgres.prc", "banner.txt", "shops.csv", and "shops_complete.csv".
- Code Editor:** The main editor window displays a GraphQL query:

```
query {
  beer(beerId: "B1") {
    price
    name
  }
  shops {
    address
    beers
    id
    name
    ...
    __typename
  }
}
```

A tooltip is open over the word "beers", providing documentation: "All Beers this shop sells [Beer!]!".
- Query Result:** Below the editor, the "Query result" tab shows the JSON response from the GraphQL endpoint:

```
{"data": { "beer": { "price": "3,80 EUR", "ratings": [ { "author": { "login": "waldemar" } }, { "author": { "login": "karl" } } ] } }}
```
- Bottom Bar:** The footer includes tabs for Git, Find, TODO, Problems, Profiler, GraphQL, Terminal, Build, Services, Endpoints, Dependencies, Spring, and Event Log. It also shows the current file path: "Get GraphQL schema from endpoint now? Introspect 'Default Introspection Endpoint' to update the local schema file. // Introspect 'http://localhost:9000/graphql' // Open sc... (19 minutes ago)". The status bar indicates the file is 6:13 LF, UTF-8, 4 spaces, and is on branch master.

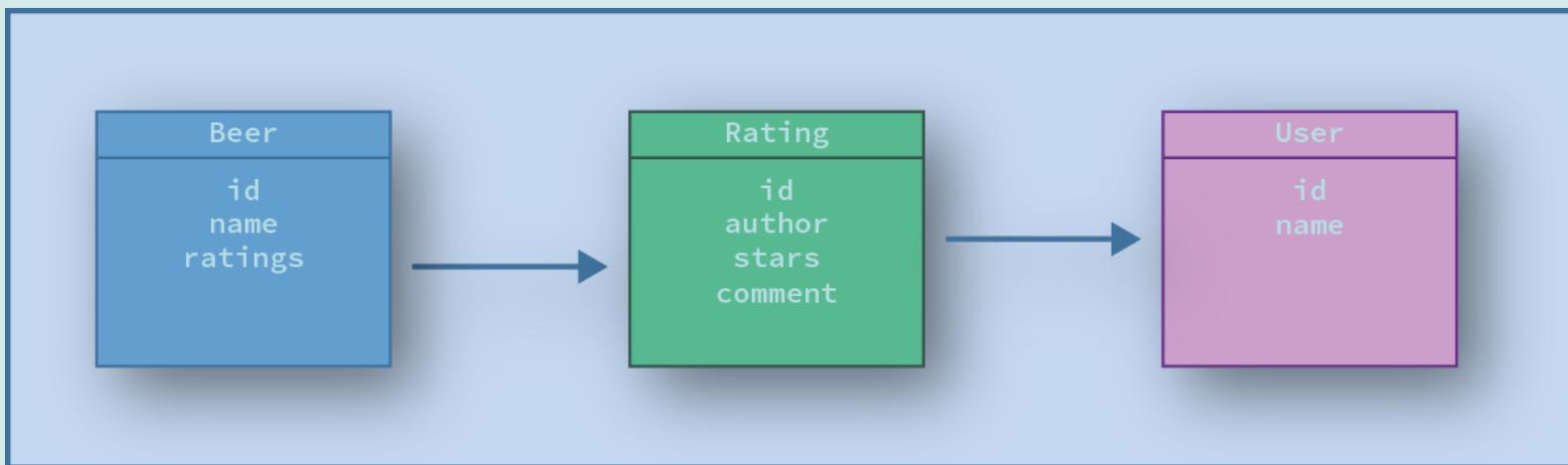
Demo: IntelliJ

<https://plugins.jetbrains.com/plugin/8097-js-graphq>

Vergleich mit REST

BEERADVISOR DOMAINE

"Domain-Model"

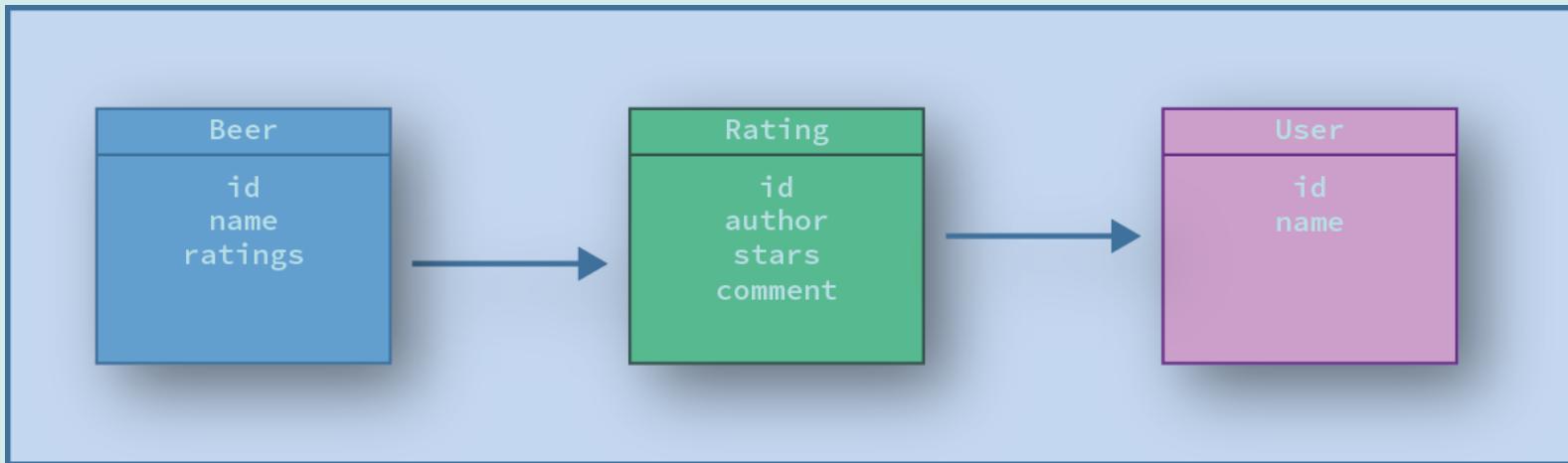


ABFRAGEN MIT REST

REST-Zugriff

- Exemplarisch und vereinfacht: Der **Autor** eines bestimmten **Ratings** eines bestimmten **Biers**

GET /beer/1

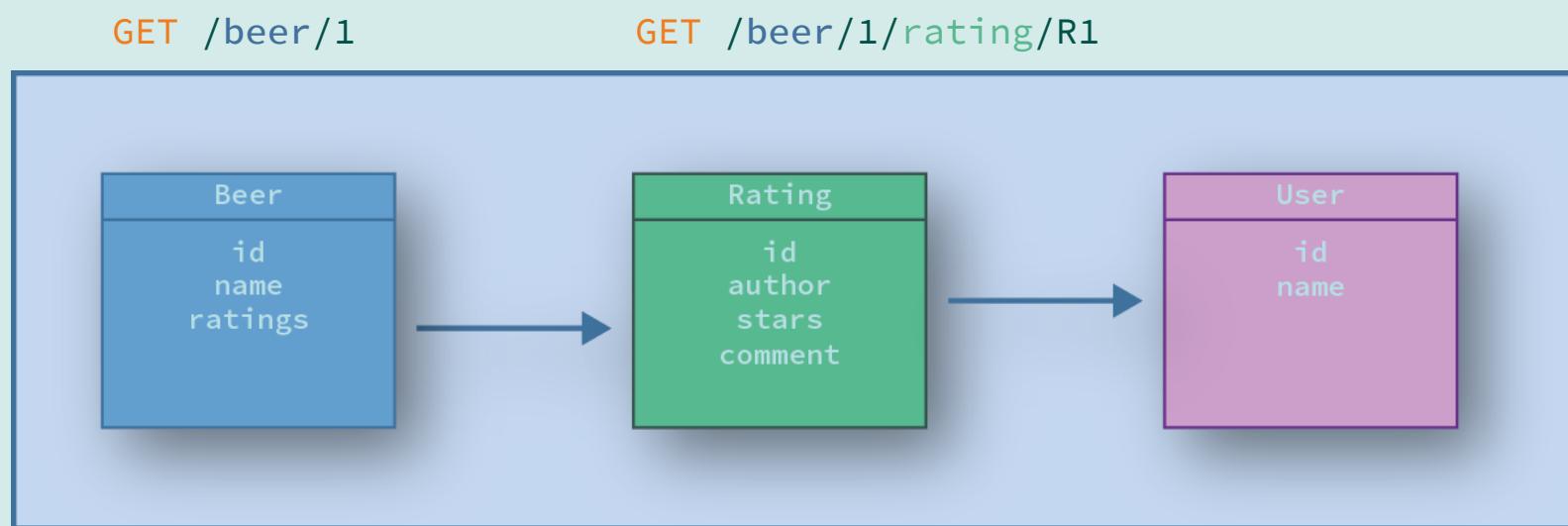


```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

ABFRAGEN MIT REST

REST-Zugriff

- Exemplarisch und vereinfacht: Der **Autor** eines bestimmten **Ratings** eines bestimmten **Biers**



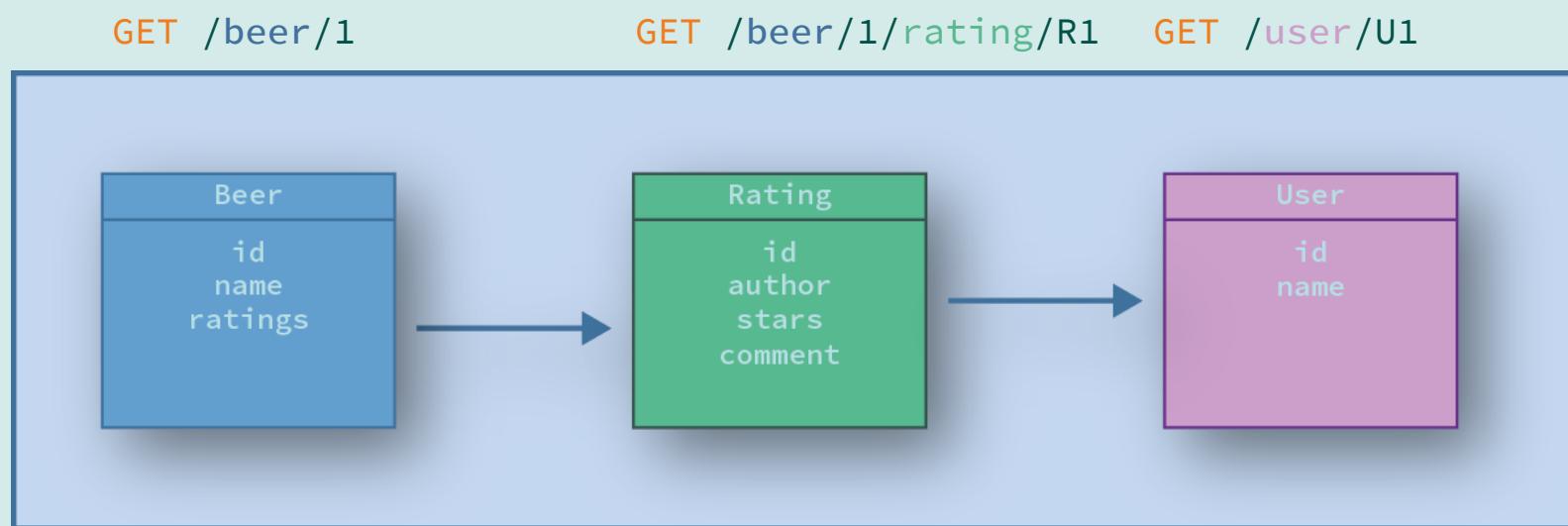
```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

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

ABFRAGEN MIT REST

REST-Zugriff

- Exemplarisch und vereinfacht: Der **Autor** eines bestimmten **Ratings** eines bestimmten **Biers**



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

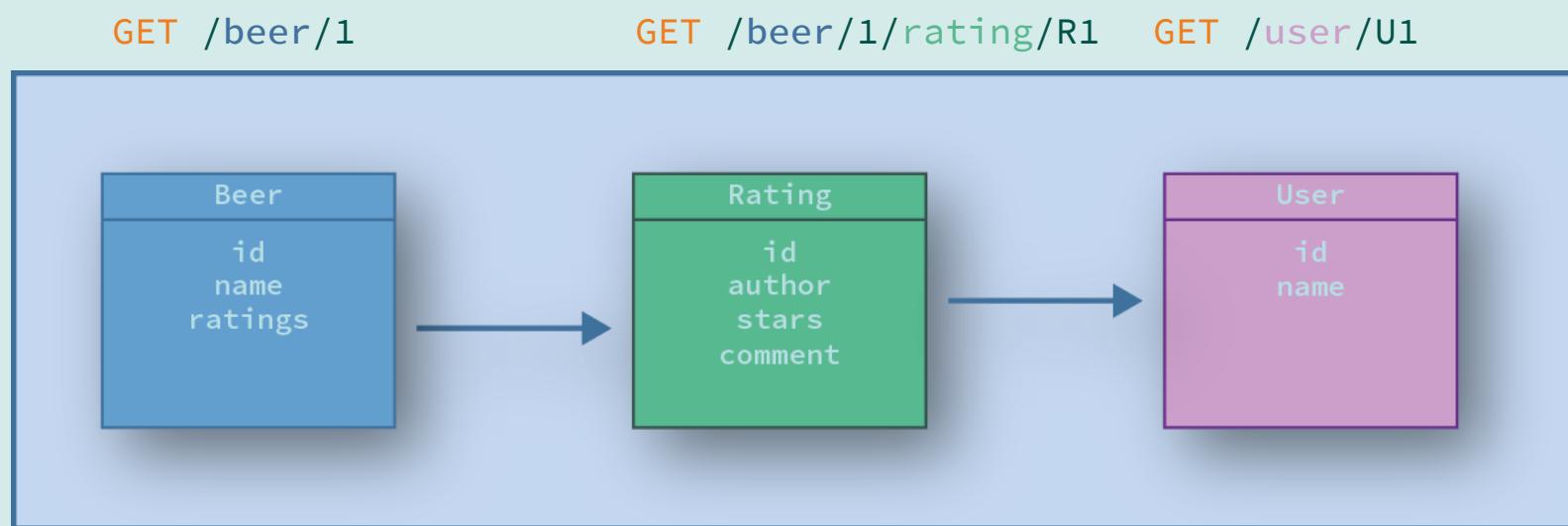
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

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

ABFRAGEN MIT REST

REST-Zugriff

- Exemplarisch und vereinfacht: Der **Autor** eines bestimmten **Ratings** eines bestimmten **Biers**
- Ebenfalls vereinfacht: es kommt immer ein ganzes Objekt zurück



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

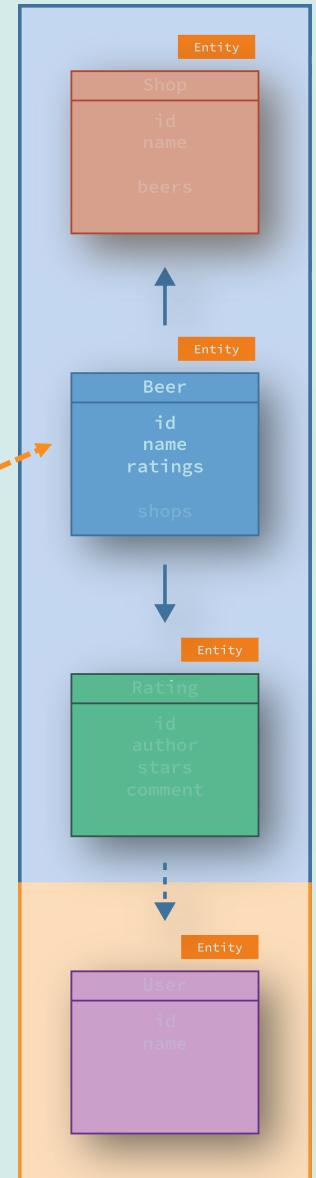
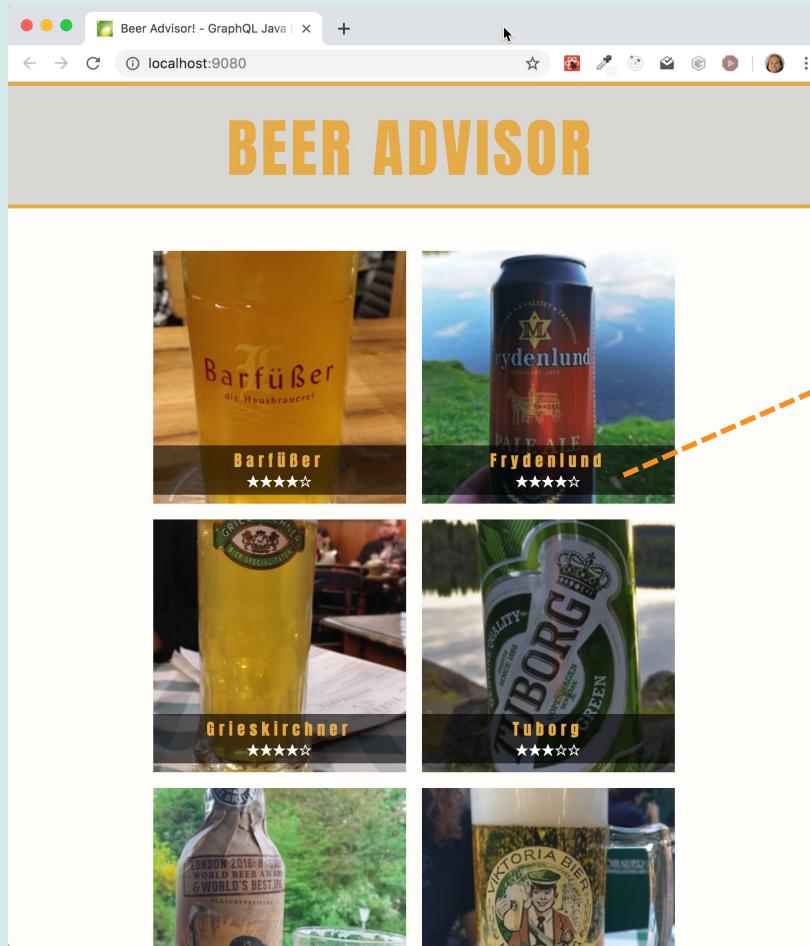
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

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

GRAPHQL QUERIES

Use-Case spezifische Abfragen – 1

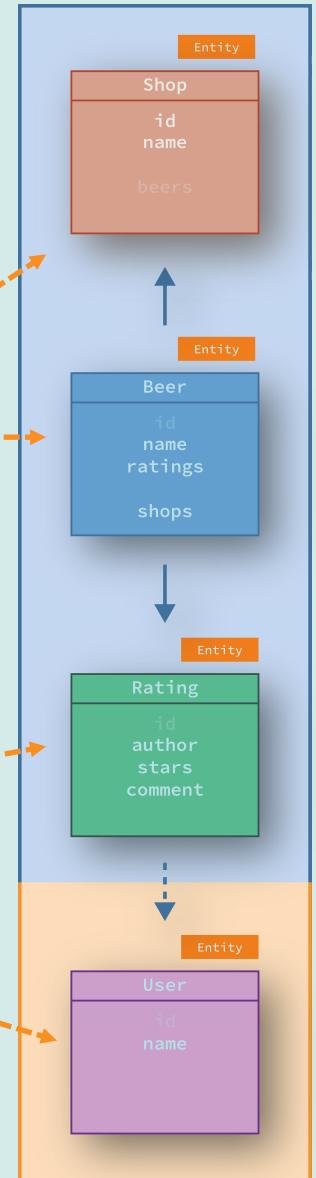
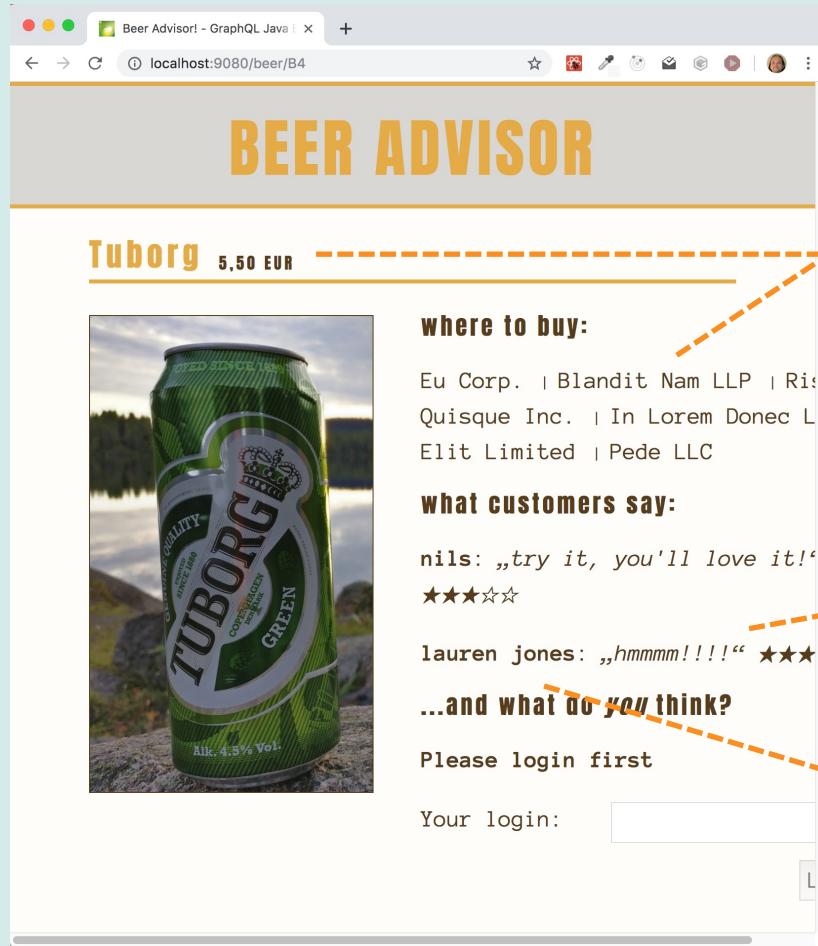
```
{ beer {  
    id  
    name  
    averageStars  
}
```



GRAPHQL QUERIES

Use-Case spezifische Abfragen – 2

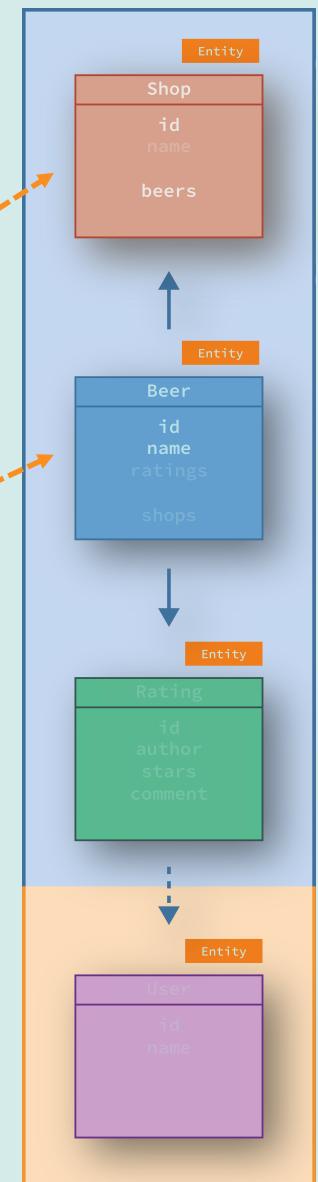
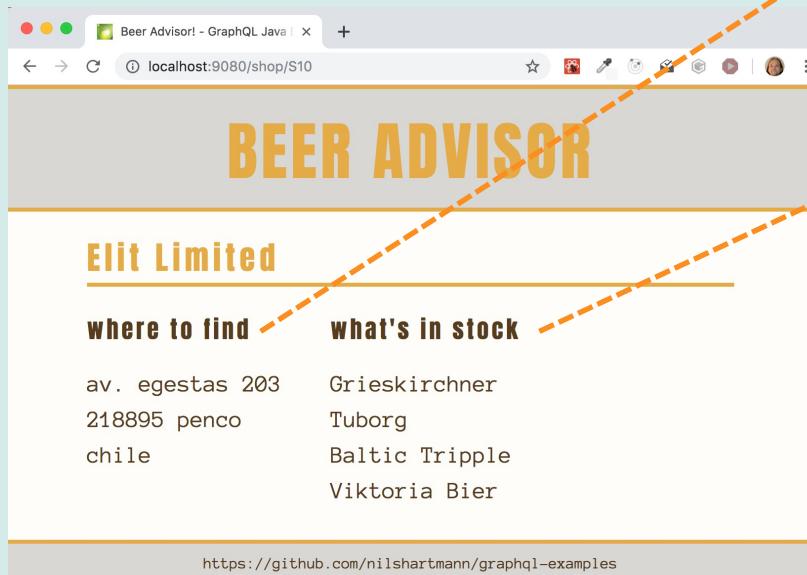
```
{ beer(beerId: "B1" {  
    name  
    price  
    ratings {  
        stars  
        comment  
        author {  
            name  
        }  
    }  
    shops { name }  
}
```



GRAPHQL QUERIES

Use-Case spezifische Abfragen – 3

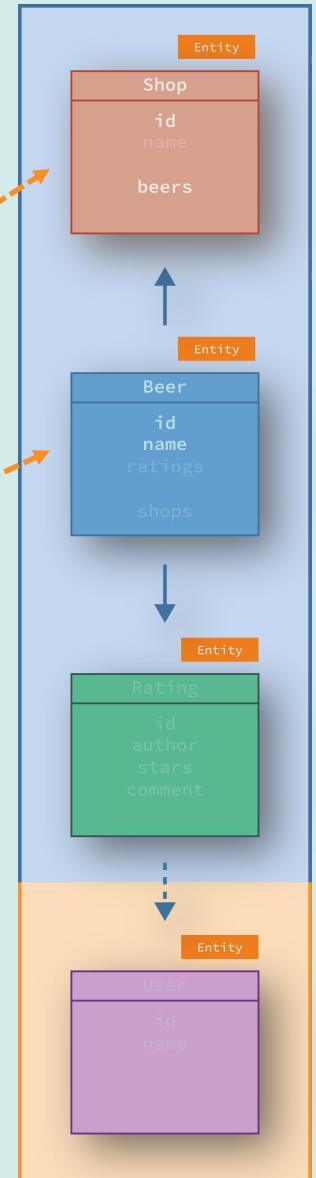
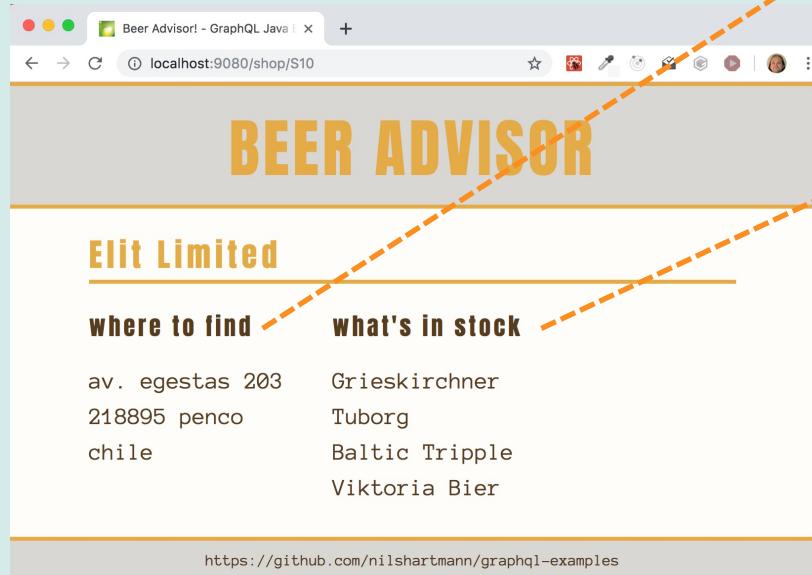
```
{ shop(shopId: "S3") {  
    name  
    address { street city }  
    beers { id name }  
}
```



GRAPHQL QUERIES

Use-Case spezifische Abfragen – 3

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



Abgefragt werden Daten, nicht Endpunkte 😈

Wir veröffentlichen mit REST 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

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

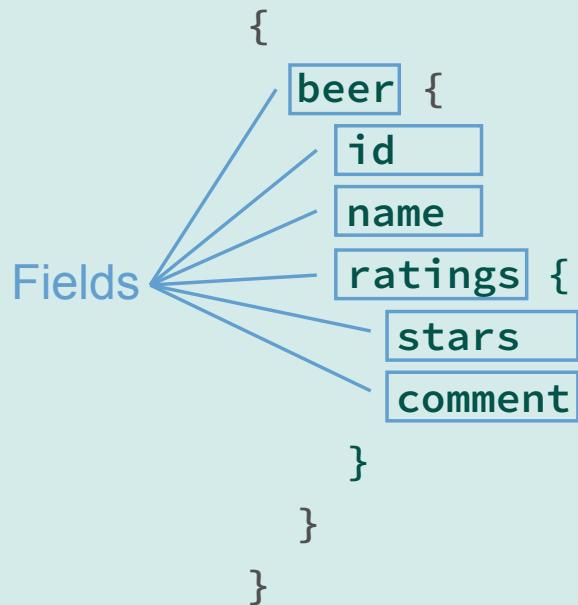
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

GraphQL

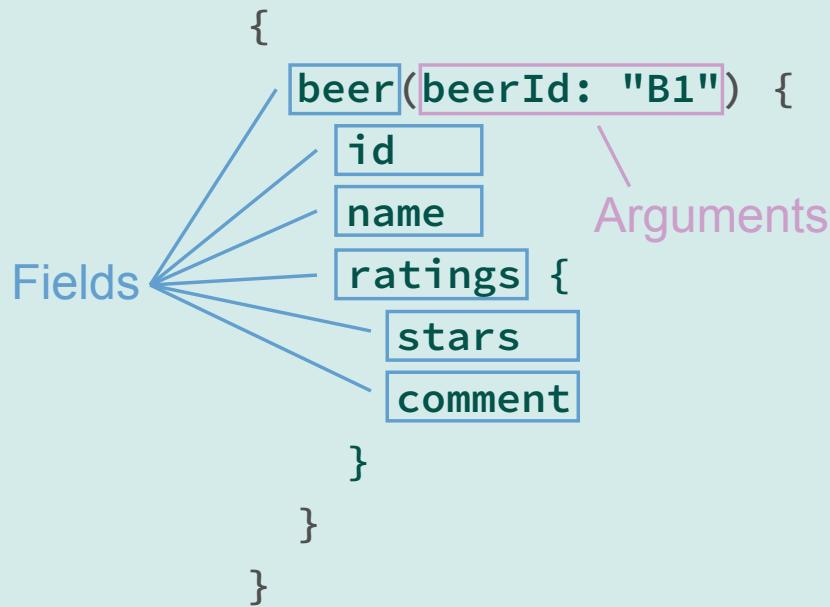
Die GraphQL Query Sprache

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>

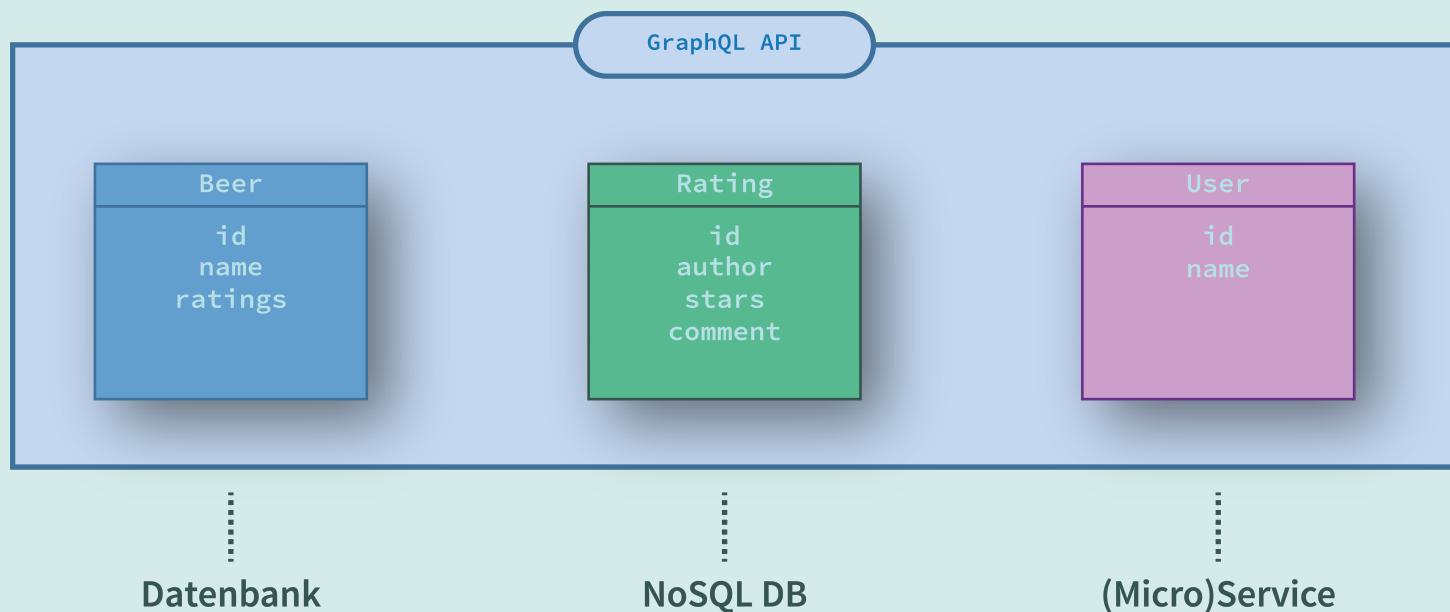
GraphQL Server

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

Object Type ----- **type Rating {**
Fields |
 |**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! ----- 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);  
}  
}
```



Vielen Dank!

Slides: <https://graphql.schule/jax2022> (PDF)

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

Kontakt: nils@nilshartmann.net