

NILS HARTMANN

Einführung in

# GraphQL

für Java

Slides: <https://bit.ly/bedcon2018-graphql>

# **NILS HARTMANN**

## **Programmierer aus Hamburg**

**JavaScript, TypeScript, React  
Java**

**Trainings, Workshops**

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

**@NILSHARTMANN**

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

- <https://graphql.org>

# GraphQL

*Spezifikation: <https://facebook.github.io/graphql/>*

- 2015 von Facebook erstmals veröffentlicht
- Query Sprache und -Ausführung
- Schema Definition Language
- Nicht: Implementierung
  - Referenz-Implementierung: graphql-js

# *GraphQL != SQL*

- kein SQL, keine "vollständige" Query-Sprache
  - z.B. keine Sortierung, keine (beliebigen) Joins etc
- keine Datenbank!
- kein Framework!

## *GraphQL != Mainstream*

- Implementierungen und Einsatz noch "bleeding edge"
- Wenig erprobte Best-Practices

## *GraphQL != Mainstream*

- Implementierungen und Einsatz noch "bleeding edge"
- Wenig erprobte Best-Practices
- ...dennoch wird es von einigen verwendet!



GitHub

@github

Folge ich



Announcing GitHub Marketplace and the official releases of GitHub Apps and our GraphQL API

Original (Englisch) übersetzen

# GitHub

## GitHub

GitHub is where people build software. More than 23 million people use GitHub to discover, fork, and contribute to over 64 million projects.

[github.com](https://github.com)

11:46 - 22. Mai 2017

<https://twitter.com/github/status/866590967314472960>

GITHUB

The screenshot shows a web browser window with the title "GraphQL API (Alpha) | GitLab". The URL in the address bar is <https://docs.gitlab.com/ee/api/graphql/>. The page content is from the "GitLab Docs" section, specifically the "GitLab Documentation > GitLab API > GraphQL API (Alpha)" page.

**On this page:**

- › [Enabling the GraphQL feature](#)
- › [Available queries](#)
- › [GraphiQL](#)

## GraphQL API (Alpha)

**Introduced** in GitLab 11.0.

GraphQL [↗](#) is a query language for APIs that allows clients to request exactly the data they need, making it possible to get all required data in a limited number of requests.

The GraphQL data (fields) can be described in the form of types, allowing clients to use [clientside GraphQL libraries ↗](#) to consume the API and avoid manual parsing.

Since there's no fixed endpoints and datamodel, new abilities can be added to the API without creating breaking changes. This allows us to have a versionless API as described in [the GraphQL documentation ↗](#).

<https://docs.gitlab.com/ee/api/graphql/>

Sicher | https://docs.atlassian.com/atlassian-confluence/1000.18...

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

## Package com.atlassian.confluence.plugins.graphql.resource

### Class Summary

Class	Description
ConfluenceGraphQLRestEndpoint	Provides the REST API endpoint for GraphQL.
GraphResource	REST API for GraphQL. ←



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

Copyright © 2003–2017 Atlassian. All rights reserved.

<https://docs.atlassian.com/atlassian-confluence/1000.1829.0/overview-summary.html>



Scott Taylor [Follow](#)

Musician. Sr. Software Engineer at the New York Times. WordPress core committer. Married to Allie. Jun 29 · 5 min read

# React, Relay and GraphQL: Under the Hood of the Times Website Redesign

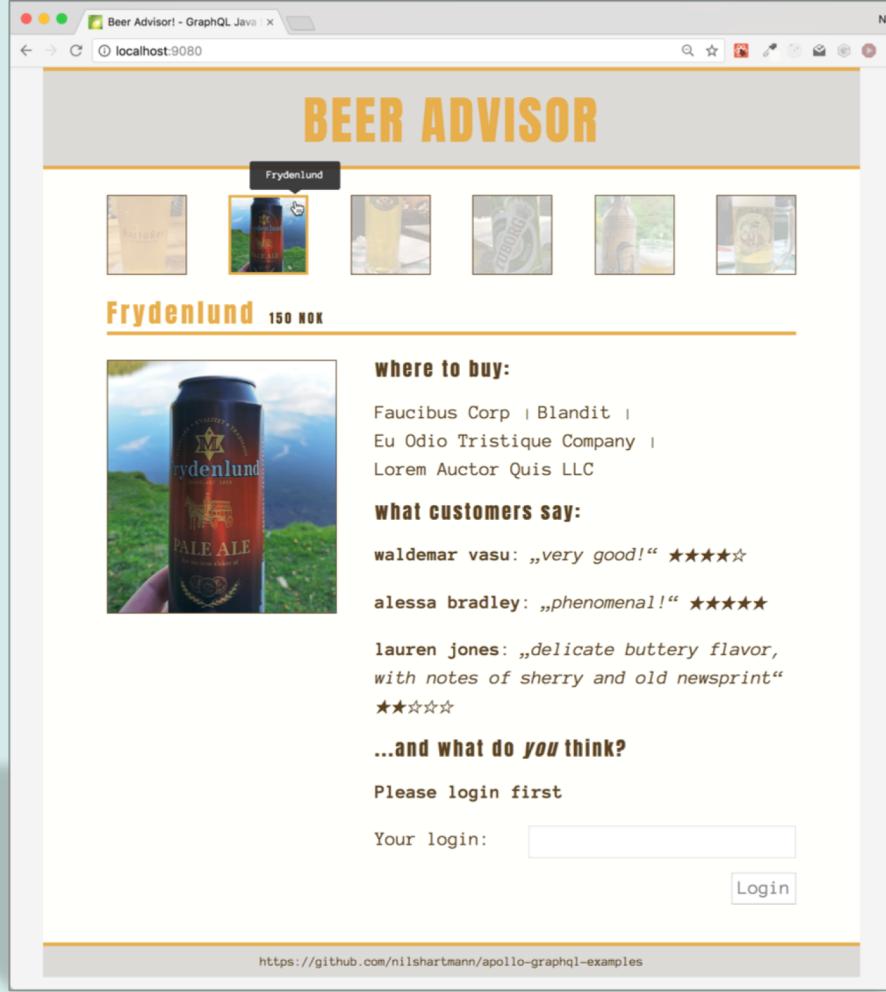


A look under the hood.

The New York Times website is changing, and the technology we use to run it is changing too.

<https://open.nytimes.com/react-relay-and-graphql-under-the-hood-of-the-times-website-redesign-22fb62ea9764>

NEW YORK TIMES



# GraphQL praktisch

Source-Code: <http://bit.ly/bedcon-graphql-example>

The screenshot shows the GraphiQL interface running at [localhost:9000/graphiql](http://localhost:9000/graphiql). The left panel displays a GraphQL query for a 'BeerAppQuery' that retrieves beers, their ratings, and a ping response. The right panel shows the schema definition and the results of the executed query.

```
query BeerAppQuery {
  beers {
    id
    name
    price
    ratings {
      id
      beerId
      author
      comment
    }
  }
}

beers
beer
ratings
ping
__schema
__type
>Returns all beers in our store
```

Schema:

```
data: {
  beers: [Beer!]!
```

Query:

```
beers: [Beer!]!
```

Results:

```
[{"id": "B1", "name": "Barfüßer", "price": "3,88 EUR", "ratings": [{"id": "R1", "beerId": "B1", "author": "Waldemar Vasu", "comment": "Exceptional!"}, {"id": "R2", "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."}], {"id": "B2", "name": "Frydenlund", "price": "158 NOK", "ratings": [{"id": "R2", "beerId": "B2", "author": "Andrea Gouyen", "comment": "Very good!"}, {"id": "R8", "beerId": "B2", "author": "Marketta Glaukos", "comment": "phenomenal!"}, {"id": "R15", "beerId": "B2", "author": "Lauren Jones", "comment": "Delicate buttery flavor, with notes of sherry and old newsprint."}], {"id": "B3", "name": "Grieskirchner", "price": "3,28 EUR", "ratings": [{"id": "R3", "beerId": "B3", "author": "Nils", "comment": "Great beer, great price!"}]}
```

# Demo: GraphiQL

<http://localhost:9000>

A screenshot of the IntelliJ IDEA code editor showing a GraphQL query. The code is as follows:

```
const BEER_RATING_APP_QUERY = gql`query BeerRatingAppQuery {
  backendStatus: ping {
    name
    nodeJsVersion
    uptime
  }
}

${/* Intellisense suggestion box */}
```

The cursor is positioned at the end of the first brace of the 'backendStatus' field. A tooltip box is open, listing several suggestions related to the current context:

- f beer - Returns the Beer with the specified Id [Beer!]!
- f beers - Returns all beers in our store [Beer!]!
- f ping - Returns health information about t... [ProcessInfo!]
- f ratings - All ratings stored in our system [Rating!]!
- f \_\_schema - Access the current type schema of... [\_\_Schema!]
- f \_\_type - Request the type information of a sing... [\_\_Type]

Below the suggestions, a note states: "Dot, space and some other keys will also close this lookup and be inserted into editor".

# Demo: IDE Support

Beispiel: IntelliJ IDEA

The screenshot shows a VS Code interface with a dark theme. A tooltip is displayed over the code editor, prompting the user to "Please enter the value for beerId" and instructing them to press 'Enter' to confirm or 'Escape' to cancel. The code being edited is a TypeScript file named `BeerPage.tsx`, which contains a GraphQL query for a beer page. The query is as follows:

```
const BEER_PAGE_QUERY = gql`query BeerPageQuery($beerId: ID!) {  beer(beerId: $beerId) {    id    name    price    ratings    shops      enum      fra      inp      input      int      inter      typ        id        name  }}
```

The cursor is positioned at the end of the first argument of the `ratings` field. A code completion dropdown is open, listing several options: `id`, `name`, `price`, `ratings`, `shops`, `enum`, `fra`, `inp`, `input`, `int`, `inter`, and `typ`. To the right of the code editor, a large JSON object representing the GraphQL response is shown. It includes fields for beer details, rating data, and a list of reviews.

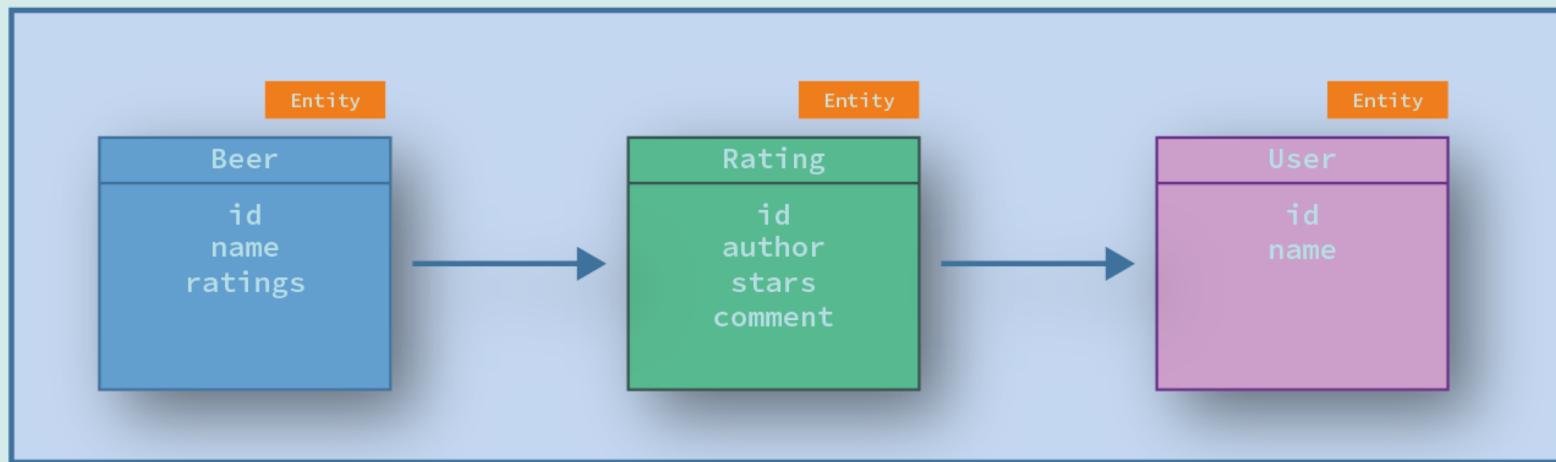
# Demo: IDE Support

Beispiel: VS Code

# **Vergleich mit REST**

# BEERADVISOR DOMAINE

## "Domain-Model"

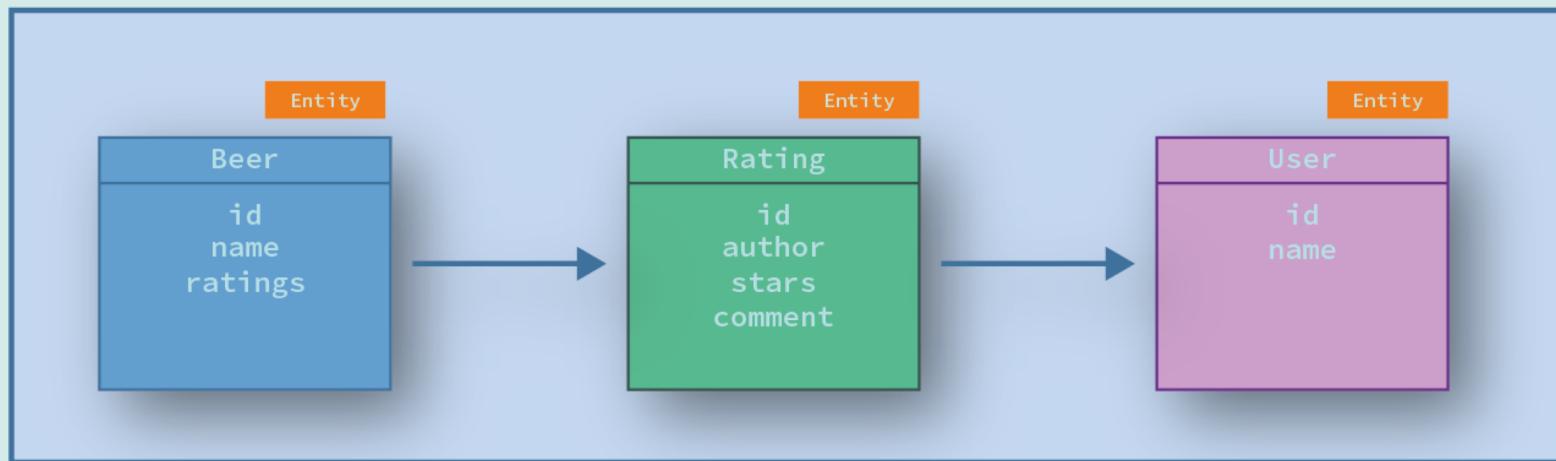


# ABFRAGEN MIT REST

## REST-Zugriff

- Exemplarisch und vereinfacht

GET /beer/1



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

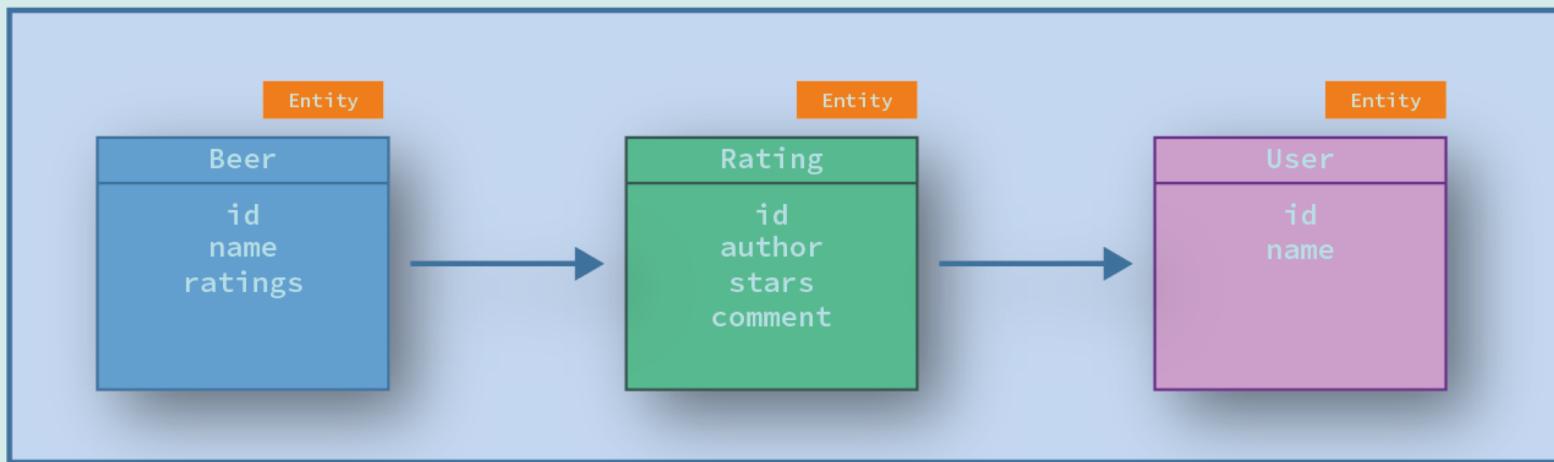
# ABFRAGEN MIT REST

## REST-Zugriff

- Exemplarisch und vereinfacht

GET /beer/1

GET /beer/1/rating/R1



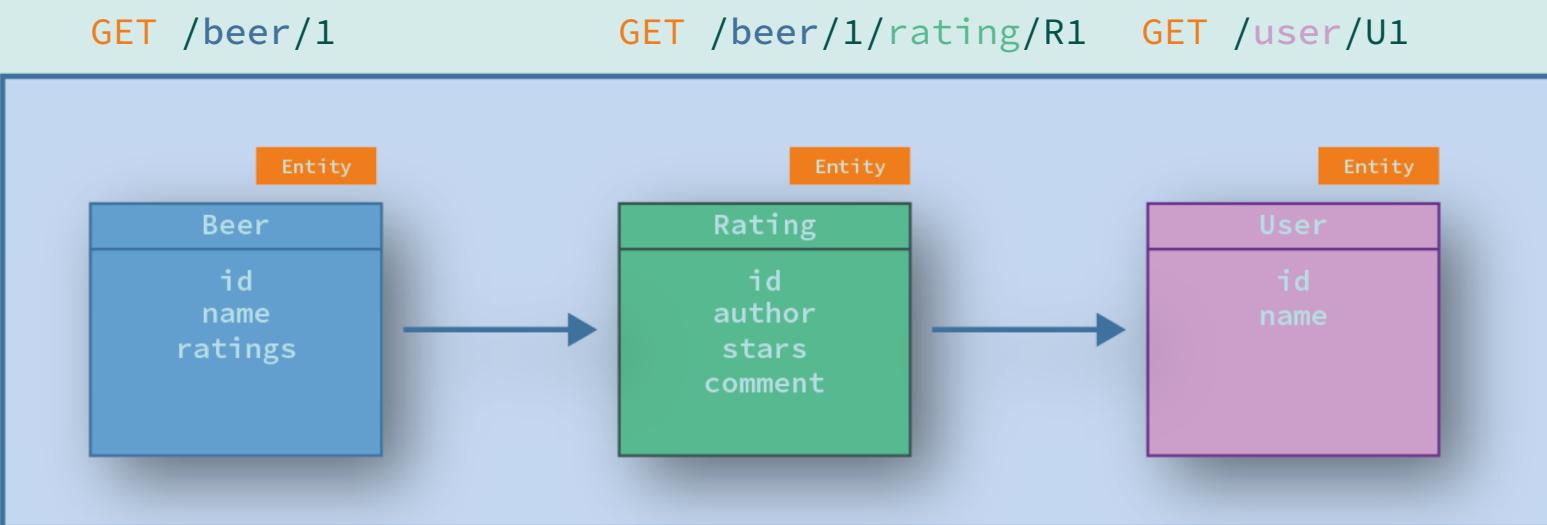
```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

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

# ABFRAGEN MIT REST

## REST-Zugriff

- Pro Ressource eine Abfrage
- Zurückgeliefert wird immer komplette Ressource
- Keine "Gesamt-Sicht" auf Domaine (auf Abfragen bezogen)



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

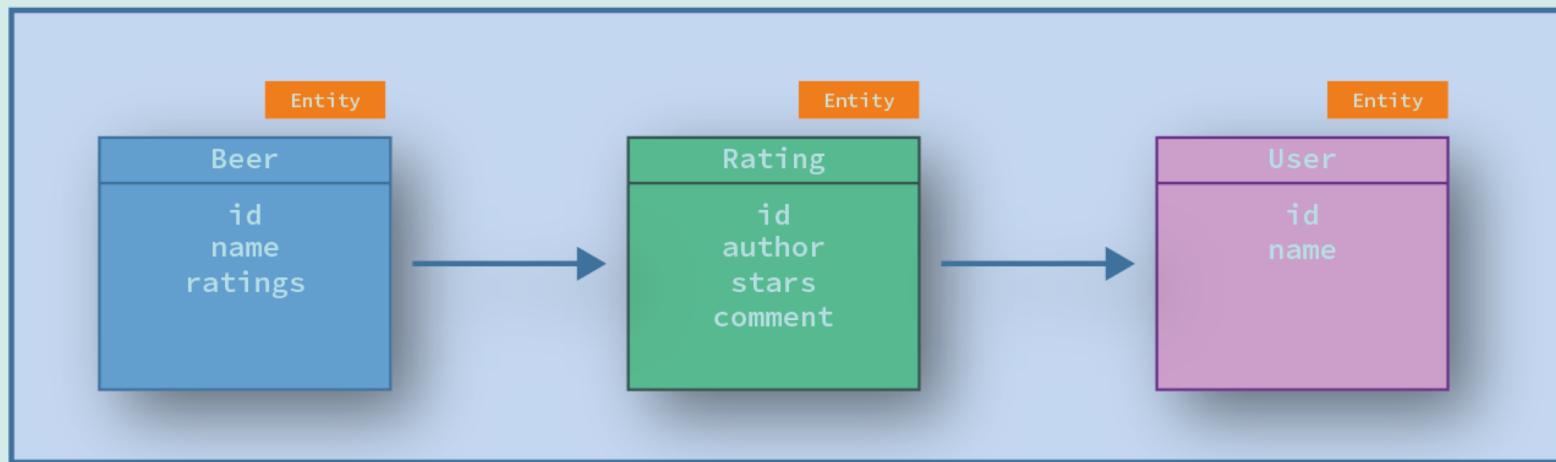
```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

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

# ABFRAGEN MIT GRAPHQL

## GraphQL

```
query { beer
  { name ratings(rid: "R1")
    { stars author { name } }
  }
}
```

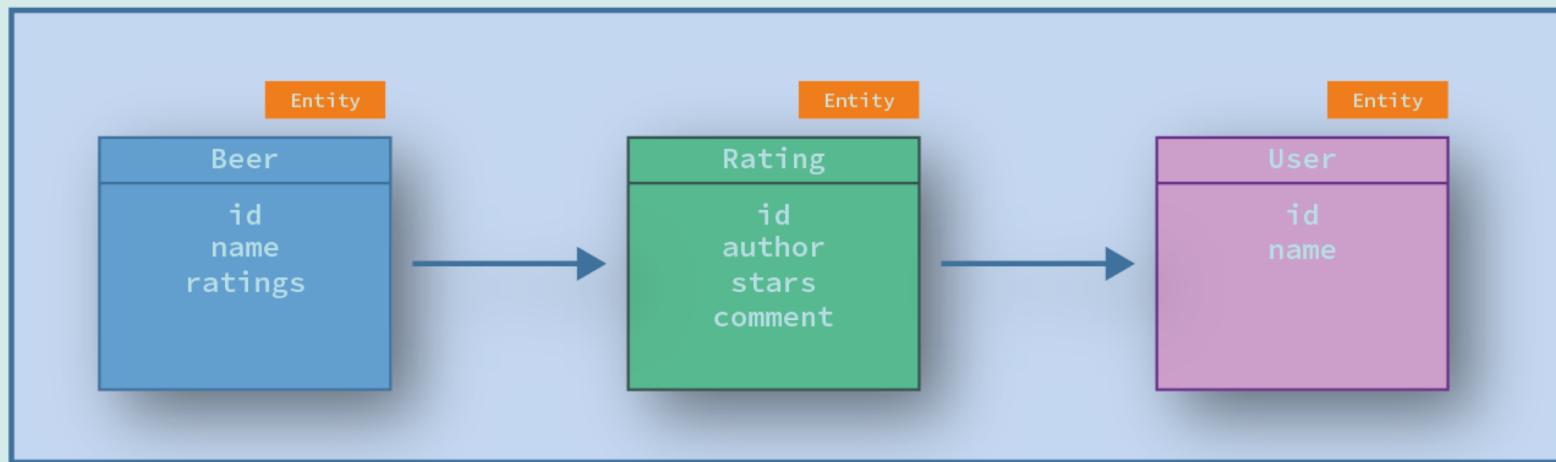


```
{
  "name": "Barfüßer",
  "ratings": {
    "stars": 3,
    "comment": "good",
    "author": { "name": "Klaus" }
  }
}
```

# ABFRAGEN MIT GRAPHQL

## GraphQL

- Beliebige Abfragen über veröffentlichte API
- Kein Widerspruch zu REST, kann als Ergänzung genutzt werden
  - z.B. Login oder File Upload



```
{  
  "name": "Barfüßer",  
  "ratings": {  
    "stars": 3,  
    "comment": "good",  
    "author": { "name": "Klaus" }  
  }  
}
```

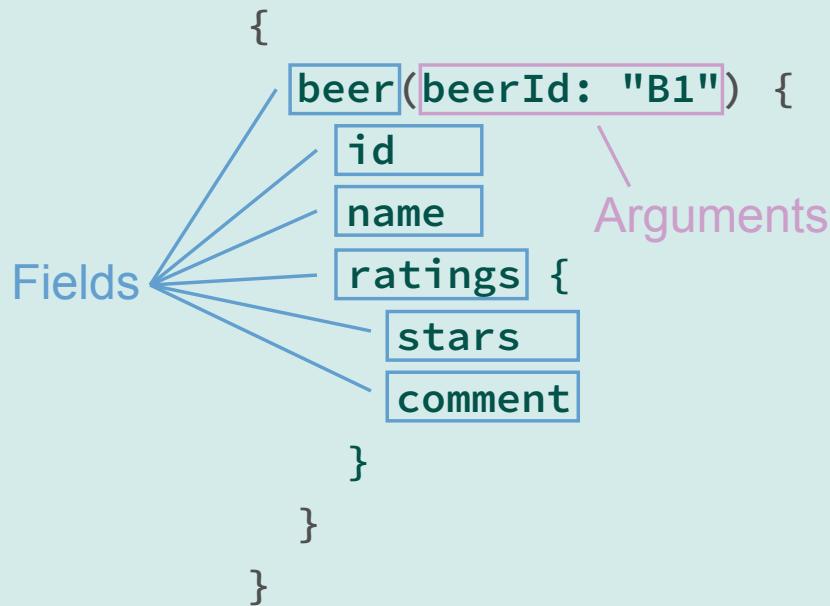
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- <https://graphql.org>

# GraphQL

**TEIL 1: ABFRAGEN UND SCHEMA**

# 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  
    }  
  }  
}
```



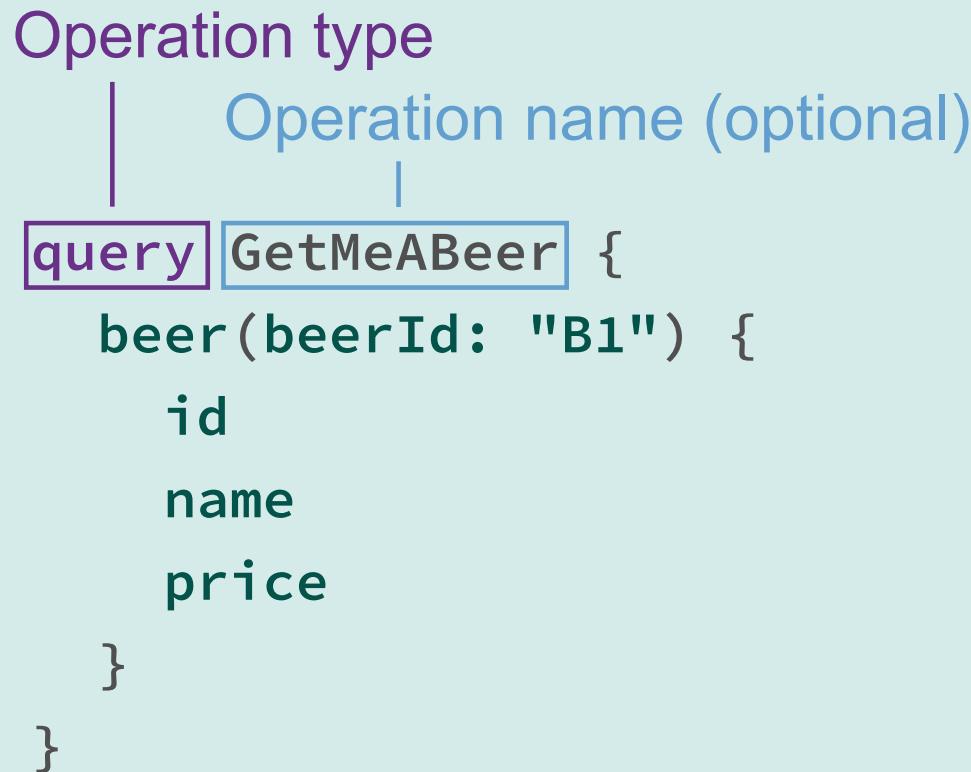
```
"data": {  
  "beer": {  
    "id": "B1"  
    "name": "Barfüßer"  
    "ratings": [  
      {  
        "stars": 3,  
        "comment": "grate taste"  
      },  
      {  
        "stars": 5,  
        "comment": "best beer ever!"  
      }  
    ]  
  }  
}
```

- Identische Struktur wie bei der Abfrage

# QUERY LANGUAGE: OPERATIONS

**Operation:** beschreibt, was getan werden soll

- query, mutation, subscription



# QUERY LANGUAGE: OPERATIONS

## Operation: Variablen

```
query GetMeABeer($bid: ID!) {  
  beer(beerId: $bid) {  
    id  
    name  
    price  
  }  
}
```

Variable Definition  
|  
query GetMeABeer(**\$bid: ID!**) {  
 beer(beerId: **\$bid**) {  
 id  
 name  
 price  
 }  
}  
Variable usage

# 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

## Beispiel: Subscription

- Automatische Benachrichtigung bei neuen Daten

```
Operation type
  |
  | Operation name (optional)
  |
  | subscription NewRatingSubscription {
  |   newRating: onNewRating {
  |     id
  |     beerId
  |     author
  |     comment
  |
  |   }
  |
  | }
```

Field alias

# QUERIES AUSFÜHREN

## Queries werden über HTTP ausgeführt

- Üblicherweise per POST
- Ein einzelner Endpoint, z.B. /graphql

```
$ curl -X POST -H "Content-Type: application/json" \
  -d '{"query":"{ beers { name } }"}' \
  http://localhost:9000/graphql
```

```
{"data":  
  {"beers": [  
    {"name": "Barfüßer"},  
    {"name": "Frydenlund"},  
    {"name": "Grieskirchner"},  
    {"name": "Tuborg"},  
    {"name": "Baltic Tripple"},  
    {"name": "Viktoria Bier"}  
  ]}  
}
```

## Schema

- Eine GraphQL API *muss* mit einem Schema beschrieben werden
- Schema legt fest, welche Types und Fields es gibt
- **Schema Definition Language** (SDL)
  - In der GraphQL Spec seit 2018

# 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!  
    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!  
}
```

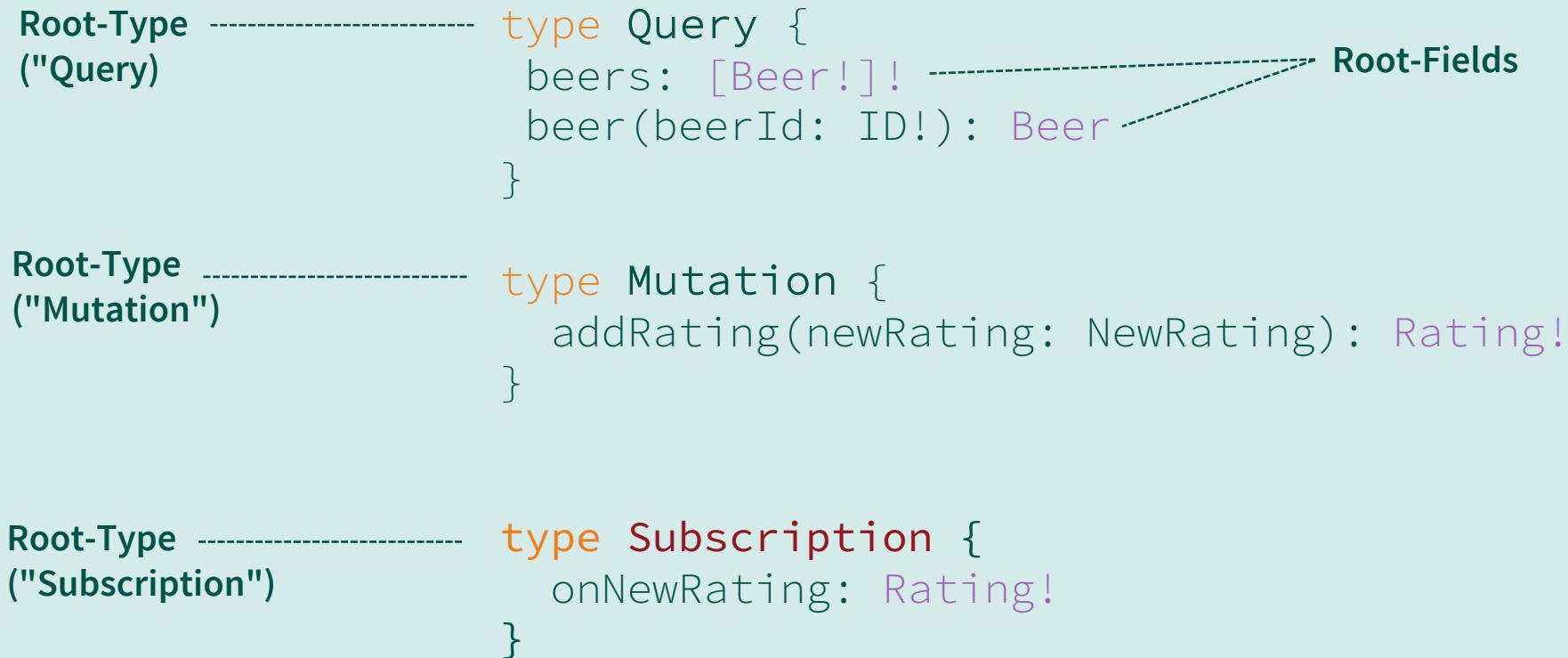
```
type User {  
    id: ID!  
    name: String!  
}
```

```
type Beer {  
    name: String!  
    ratings: [Rating!]!  
    ratingsWithStars(stars: Int!): [Rating!]!  
}
```

Arguments

# GRAPHQL SCHEMA

## Root-Types: Einstiegspunkte in die API (Query, Mutation, Subscription)



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

- <https://graphql.org>

# GraphQL für Java

**TEIL 2: RUNTIME-UMGEBUNG (AKA: EURE ANWENDUNG)**

# GRAPHQL-JAVA PROJEKT FAMILIE

graphql-java

Low-Level API  
SDL Implementierung

<https://github.com/graphql-java>

# GRAPHQL-JAVA PROJEKT FAMILIE

graphql-java-tools

High-Level API insbesondere für Resolver

graphql-java

Low-Level API  
SDL Implementierung

<https://github.com/graphql-java>

# GRAPHQL-JAVA PROJEKT FAMILIE

graphql-java-servlet

Servlet für GraphQL Requests

graphql-java-tools

High-Level API insbesondere für Resolver

graphql-java

Low-Level API  
SDL Implementierung

<https://github.com/graphql-java>

# GRAPHQL-JAVA PROJEKT FAMILIE

graphql-spring-boot-starter

Abstraktion für Spring Boot  
(Einlesen Schemas, Servlet Registrierung,...)

graphql-java-servlet

Servlet für GraphQL Requests

graphql-java-tools

High-Level API insbesondere für Resolver

graphql-java

Low-Level API  
SDL Parser

<https://github.com/graphql-java>

## Schritt 1: Schema definieren

- Inline oder extern per .graphqls-Datei

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Rating {  
    id: ID!  
    beer: Beer!  
    author: User!  
    comment: String!  
    stars: Int!  
}  
  
type Beer {  
    id: ID!  
    name: String!  
    price: String!  
    ratings: [Rating!]!  
    ratingsWithStars(stars: Int!): [Rating!]!  
}  
  
type Query {  
    beer(beerId: ID!): Beer  
    beers: [Beer!]!  
}  
  
input AddRatingInput {  
    beerId: ID!  
    userId: ID!  
    comment: String!  
    stars: Int!  
}  
  
type Mutation {  
    addRating(ratingInput: AddRatingInput):  
        Rating!  
}
```

## Schritt 2: Resolver implementieren

- Ein **Resolver** liefert ein Wert für ein angefragtes Feld
  - Default: per Reflection
- Root-Resolver (für Query-Typ) müssen implementiert werden

## Resolver implementieren

- Beispiel: Root-Resolver (Query)

Interface für  
Query-Resolver

```
import com.coxautodev.graphql.tools.GraphQLQueryResolver;  
  
public class RatingQueryResolver implements  
    GraphQLQueryResolver {
```

```
type Query {  
  beers: [Beer!]!  
}
```

```
}
```

## Resolver implementieren

- Beispiel: Root-Resolver (Query)

Zuordnung über  
Namenskonvention

```
type Query {  
  beers: [Beer!]!  
}
```

```
import com.coxautodev.graphql.tools.GraphQLQueryResolver;  
  
public class RatingQueryResolver implements  
  GraphQLQueryResolver {  
  
  // z.B via DI  
  private BeerRepository beerRepository;  
  
  public List<Beer> beers() {  
    return beerRepository.findAll();  
  }  
  
}
```

## Resolver implementieren

- Beispiel: Root-Resolver (Query)

```
import com.coxautodev.graphql.tools.GraphQLQueryResolver;

public class RatingQueryResolver implements
    GraphQLQueryResolver {

    // z.B via DI
    private BeerRepository beerRepository;

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

    public Beer beer(String beerId) {
        return beerRepository.getBeer(beerId);
    }
}
```

**Argumente als Parameter**

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

## Resolver implementieren

- Beispiel: Root-Resolver (Mutation)
- Ähnlich wie Query-Resolver

```
import com.coxautodev.graphql.tools.GraphQLMutationResolver;

public class RatingMutationResolver implements
    GraphQLMutationResolver {

    // z.B via DI
    private RatingRepository ratingRepository;

type Mutation {
    addRating
        (ratingInput: AddRatingInput): Rating!
}
```

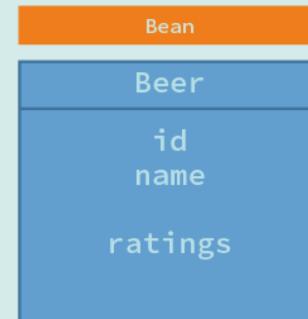
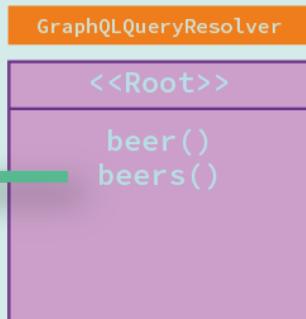
```
    public Rating addRating(AddRatingInput newRating) {
        Rating rating = Rating.from(newRating);
        ratingRepository.save(rating);
        return rating;
    }
}
```

# RESOLVING ZUR LAUFZEIT

graphql-java-tools

- **Schritt 1: Root Resolver**

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

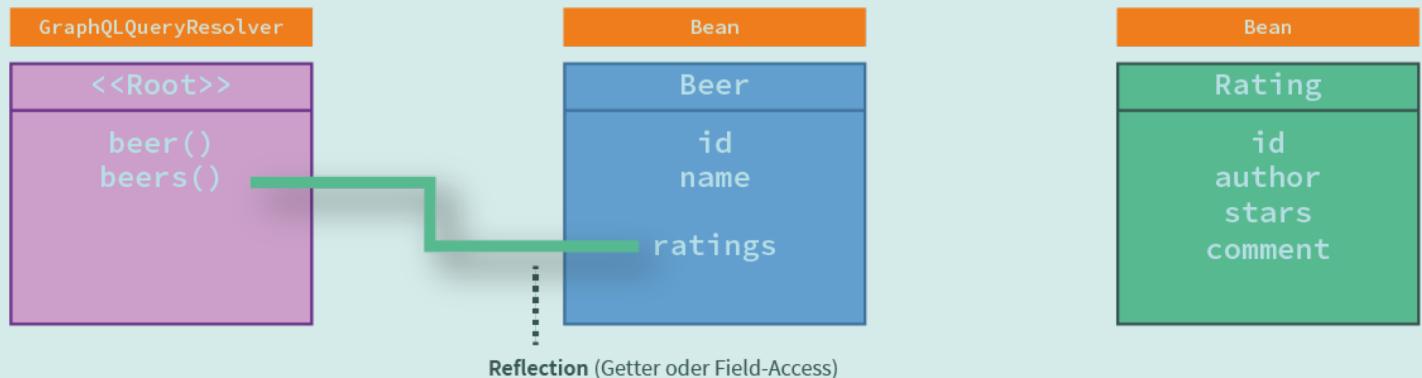


# RESOLVING ZUR LAUFZEIT

graphql-java-tools

- Schritt 2: Field Resolver (Default: Reflection)

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

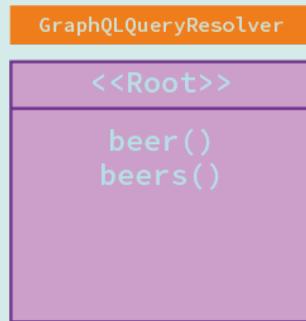


# RESOLVING ZUR LAUFZEIT

graphql-java-tools

- Schritt 3: Field Resolver (Default: Reflection)

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



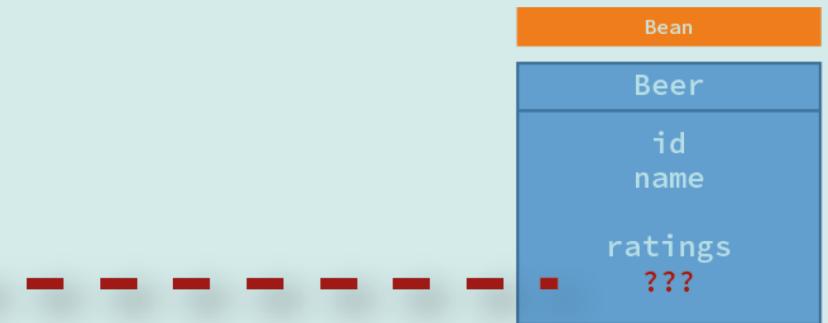
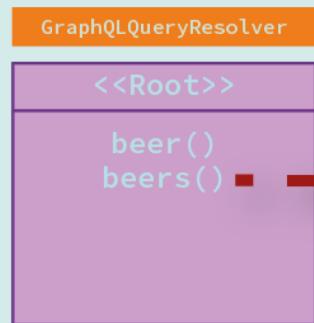
Reflection (Getter oder Field-Access)

# RESOLVING ZUR LAUFZEIT

graphql-java-tools

**Problem:** Mismatch zwischen Java-Klassen und Schema

```
{  
  beers {  
    ratingsWithStars  
    (stars: 3) {  
      comment  
    }  
  }  
}
```

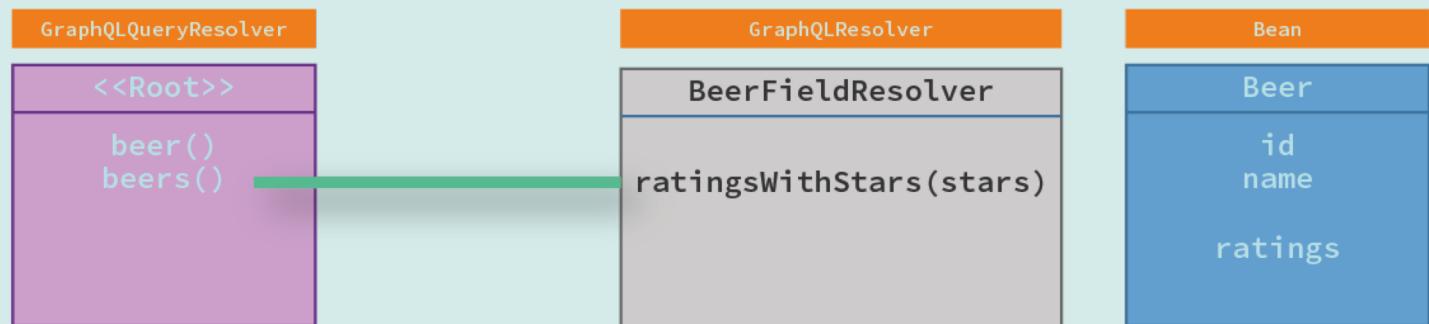


Feld/Methode „ratingWithStars“ nicht in Beer-Klasse vorhanden

## Field Resolver für zusätzliche Felder

- Explizite Zugriffs-Methoden für Felder, die nicht an Klasse vorhanden sind
- Gilt auch für Felder, die zwischen API und Java-Klasse abweichen
  - z.B. anderer Rückgabe-Wert

```
{  
  beers {  
    ratingsWithStars  
    (stars: 3) {  
      comment  
    }  
  }  
}
```



## Field Resolver implementieren

- Liefert Wert für ein Feld in einem Objekt zurück

```
import com.coxautodev.graphql.tools.GraphQLResolver;

public class BeerFieldResolver implements
GraphQLResolver<Beer> {
type Beer {
  ratingsWithStars(stars: Int!): [Rating!]!
}

}
```

## Field Resolver implementieren

- Liefert Wert für ein Feld in einem Objekt zurück

```
import com.coxautodev.graphql.tools.GraphQLResolver;  
  
public class BeerFieldResolver implements  
GraphQLResolver<Beer> {  
  
    public List<Rating> ratingsWithStars(Beer beer, int stars) {  
  
    }  
}
```

The diagram consists of two vertical dashed blue arrows. The left arrow points from the 'beer' parameter in the Java code back up to the 'Beer' type in the GraphQL schema. The right arrow points from the 'stars' parameter in the Java code back up to the 'Int!' type in the GraphQL schema.

Quell-Objekt als Parameter      Argumente als Parameter

```
type Beer {  
    ratingsWithStars(stars: Int!): [Rating!]!  
}
```

## Field Resolver implementieren

- Liefert Wert für ein Feld in einem Objekt zurück

```
type Beer {  
    ratingsWithStars(stars: Int!): [Rating!]!  
}  
  
import com.coxautodev.graphql.tools.GraphQLResolver;  
  
public class BeerFieldResolver implements  
GraphQLResolver<Beer> {  
  
    public List<Rating> ratingsWithStars(Beer beer, int stars) {  
        return beer.getRatings().stream()  
            .filter(r -> r.getStars() == stars)  
            .collect(Collectors.toList());  
    }  
}
```

## Validierung beim Starten

- Alle Resolver müssen vorhanden sein
- Return-Types und Methoden-Parameter der Resolver-Funktionen müssen zum Schema passen

```
com.coxautodev.graphql.tools.FieldResolverError:
```

```
No method found with any of the following signatures, in priority order:
```

```
nh.graphql.beeradvisor.rating.graphql.RatingQueryResolver.beer(~beerId)
nh.graphql.beeradvisor.rating.graphql.RatingQueryResolver.getBeer(~beerId)
```

## Validierung zur Laufzeit

- Resolver werden immer mit korrekten Parametern aufgerufen
  - Argumente haben korrekten Typ
  - Argumente sind ggf. nicht null
  - Keine Annotationen etc notwendig

## Validierung zur Laufzeit

- Resolver werden immer mit korrekten Parametern aufgerufen
  - Argumente haben korrekten Typ
  - Argumente sind ggf. nicht null
  - Keine Annotationen etc notwendig
- Rückgabe-Wert eines Resolvers wird überprüft
  - Client erhält nie ungültige Werte

## Validierung zur Laufzeit

- Resolver werden immer mit korrekten Parametern aufgerufen
  - Argumente haben korrekten Typ
  - Argumente sind ggf. nicht null
  - Keine Annotationen etc notwendig
- Rückgabe-Wert eines Resolvers wird überprüft
  - Client erhält nie ungültige Werte
- Es werden nur Felder herausgegeben, die auch im Schema definiert sind
  - Alle anderen Felder einer Java-Klasse sind "unsichtbar"

## Schritt 3: Schema-Instanz

- GraphQLSchema verbindet SDL mit Resolvern

```
import com.coxautodev.graphql.tools.SchemaParser;
import graphql.schema.GraphQLSchema;

public class BeerSchemaFactory {

    public static GraphQLSchema createSchema() {
        GraphQLSchema graphQLSchema =
            SchemaParser.newParser()
                .file("beer.graphqls")
                .resolvers(new BeerMutationResolver(),
                           new BeerQueryResolver())
                .build()
                .makeExecutableSchema();
    }
}
```

Schema-Datei



.file("beer.graphqls")

Alle Resolver



.resolvers(new BeerMutationResolver(),
 new BeerQueryResolver())

## Schritt 4: GraphQL Servlet

- Vergleichbar mit REST-Controller
- aber: *ein* Endpunkt für komplette API

```
import graphql.servlet.SimpleGraphQLHttpServlet;  
  
GraphQLSchema beerSchema =  
    BeerSchemaFactory.createSchema();  
  
SimpleGraphQLHttpServlet servlet =  
    SimpleGraphQLHttpServlet  
        .newBuilder(beerSchema)  
        .build()  
  
// ... Servlet in Container anmelden ...
```

Query ausführen

http localhost:9000/graphql?query='{beers { name ratings { stars } } }'

# ZUSAMMENFASSUNG

graphql-java

graphql-java-tools

graphql-javaservlet

**Zusammenfassung:** Anwendung mit graphql-java

# ZUSAMMENFASSUNG

graphql-java

graphql-java-tools

graphql-java-servlet

**Zusammenfassung:** Anwendung mit graphql-java

1. Schema mit SDL beschreiben (\*.graphqls)

# ZUSAMMENFASSUNG

graphql-java

graphql-java-tools

graphql-java-servlet

## Zusammenfassung: Anwendung mit graphql-java

1. Schema mit SDL beschreiben (\*.graphqls)
2. Root- und Field-Resolver schreiben

# ZUSAMMENFASSUNG

graphql-java

graphql-java-tools

graphql-java-servlet

## Zusammenfassung: Anwendung mit graphql-java

1. Schema mit SDL beschreiben (\*.graphqls)
2. Root- und Field-Resolver schreiben
3. GraphQLSchema mit SDL und Resolvern erzeugen

# ZUSAMMENFASSUNG

graphql-java

graphql-java-tools

graphql-java-servlet

## Zusammenfassung: Anwendung mit graphql-java

1. Schema mit SDL beschreiben (\*.graphqls)
2. Root- und Field-Resolver schreiben
3. GraphQLSchema mit SDL und Resolvern erzeugen
4. Servlet instantiiieren und in Container deployen

# SPRING BOOT STARTER

graphql-spring-boot-starter

**Vereinfacht das Setup von GraphQL-Anwendungen**

## Vereinfacht das Setup von GraphQL-Anwendungen

- Mergt alle Schema-Dateien im Klassenpfad zusammen (\*.graphqls)

## Vereinfacht das Setup von GraphQL-Anwendungen

- Mergt alle Schema-Dateien im Klassenpfad zusammen (\*.graphqls)
- Resolver werden als Beans annotiert (zB. @Component) und automatisch dem Schema hinzugefügt

## Vereinfacht das Setup von GraphQL-Anwendungen

- Mergt alle Schema-Dateien im Klassenpfad zusammen (\*.graphqls)
- Resolver werden als Beans annotiert (zB. @Component) und automatisch dem Schema hinzugefügt
- Servlet und Servlet-Mapping erfolgt per application.properties

## Vereinfacht das Setup von GraphQL-Anwendungen

- Mergt alle Schema-Dateien im Klassenpfad zusammen (\*.graphqls)
- Resolver werden als Beans annotiert (zB. @Component) und automatisch dem Schema hinzugefügt
- Servlet und Servlet-Mapping erfolgt per application.properties

```
# application.yaml

graphql:
  servlet:
    mapping: /graphql
    enabled: true
    corsEnabled: true
```

## Vereinfacht das Setup von GraphQL-Anwendungen

- Mergt alle Schema-Dateien im Klassenpfad zusammen (\*.graphqls)
- Resolver werden als Beans annotiert (zB. @Component) und automatisch dem Schema hinzugefügt
- Servlet und Servlet-Mapping erfolgt per application.properties
- GraphiQL (API Explorer) kann ebenfalls per Konfiguration aktiviert werden

```
# application.yaml
```

```
graphql:
  servlet:
    mapping: /graphql
    enabled: true
    corsEnabled: true
```

```
# application.yaml
```

```
graphiql:
  mapping: /graphiql
  endpoint: /graphql
  enabled: true
```



# Vielen Dank!

Beispiel-Code: <http://bit.ly/bedcon-graphql-example>

Slides: <https://bit.ly/bedcon2018-graphql>

# SUBSCRIPTIONS

graphql-java-tools

- An Events (z.B. Datenänderungen) anmelden
- Server informiert Client sobald es neue Events gibt
- Kein Pendant in REST (?)

# SUBSCRIPTIONS

graphql-java-tools

- An Events (z.B. Datenänderungen) anmelden
- Server informiert Client sobald es neue Events gibt
- Kein Pendant in REST (?)

```
subscription {  
  onNewRating {  
    id  
    beer {  
      id  
    }  
    author {  
      name  
    }  
    comment  
    stars  
  }  
}
```

Bei jedem  
neuen Rating



```
{  
  "onNewRating": {  
    "id": "R9",  
    "beer": {  
      "id": "B4"  
    },  
    "author": {  
      "name": "Lauren Jones"  
    },  
    "comment": "Very tasteful",  
    "stars": 4  
  }  
}
```

## Schema Definition

- Root-Type "Subscription"
- Felder und Return-Values wie bei Query und Mutation

```
type Subscription {  
    onNewRating: Rating!  
  
    newRatings(beerId: ID!): Rating!  
}
```

# SUBSCRIPTIONS

graphql-java-  
tools

## Resolver implementieren

- Resolver ähnlich wie bei Query- und Mutation-Type

```
import com.coxautodev.graphql.tools.GraphQLSubscriptionResolver;

public class RatingMutationResolver implements
    GraphQLSubscriptionResolver {
}

type Subscription {
    onNewRating: Rating!
}
```

## Resolver implementieren

- Resolver ähnlich wie bei Query- und Mutation-Type
- Rückgabewert muss Reactive Streams Publisher sein

```
import com.coxautodev.graphql.tools.GraphQLSubscriptionResolver;
import org.reactivestreams.Publisher;

public class RatingMutationResolver implements
    GraphQLSubscriptionResolver {

    // z.B via DI
    private RatingPublisher ratingPublisher;

    public Publisher<Rating> onNewRating() {
        return ratingPublisher.getPublisher();
    }
}
```

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

## Publisher implementieren (Beispiel)

- Hängt von verwendeter Technologie und Architektur ab
- Beispiel: Spring

```
type Subscription {  
    onNewRating: Rating!  
}  
  
@Component  
public class RatingPublisher {  
    public RatingPublisher() {  
        // create reactivestreams Emitter and Publisher  
    }  
  
    @TransactionalEventListener  
    public void ratingCreated(RatingCreatedEvent e) {  
        this.emitter.onNext(e.getRating());  
    }  
  
    public Publisher<Rating> getPublisher() {  
        return this.publisher;  
    }  
}
```

# SUBSCRIPTIONS

graphql-java-servlet

graphql-spring-boot-starter

## Veröffentlichen über WebSockets

- Eigenes Servlet in graphql-java-servlet
- Einbinden via Spring Boot Konfiguration

```
# application.yaml

graphql:
    servlet:
        websocket:
            enabled: true
            subscriptions:
                websocket:
                    path: /subscriptions
```