

**NILS HARTMANN**  
<https://nilshartmann.net>

# Fullstack GraphQL

**Workshop mit Apollo und React**

Slides (PDF): <https://nils.buzz/ejs-graphql-workshop>

# NILS HARTMANN

nils@nilshartmann.net

**Freiberuflischer Entwickler, Architekt, Trainer aus Hamburg**

Java  
JavaScript, TypeScript  
React  
GraphQL

**Trainings & Workshops**

<https://nilshartmann.net/react-workshops>

**HTTPS://NILSHARTMANN.NET**

# AGENDA

- **sdfasdf**

# GraphQL

# Grundlagen

TEIL 1

*"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
- Weitere Entwicklung seit 2018 in GraphQL Foundation
- Umfasst:
  - 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!
- 
- *Ersetzt weder Backend noch Datenbank*

# *GraphQL != JavaScript*

- Populär in JS, aber auch außerhalb

## *GraphQL != (nur) Lösung für Over- oder Under-fetching*

- Häufig genanntes technisches Argument für GraphQL
- M.E. aber nicht das wichtigste
- Flexible, fachliche Abfragen möglich
- Gute Möglichkeit, Domainmodel abzubilden und zur Verfügung zu stellen

## *GraphQL != Mainstream*

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



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>

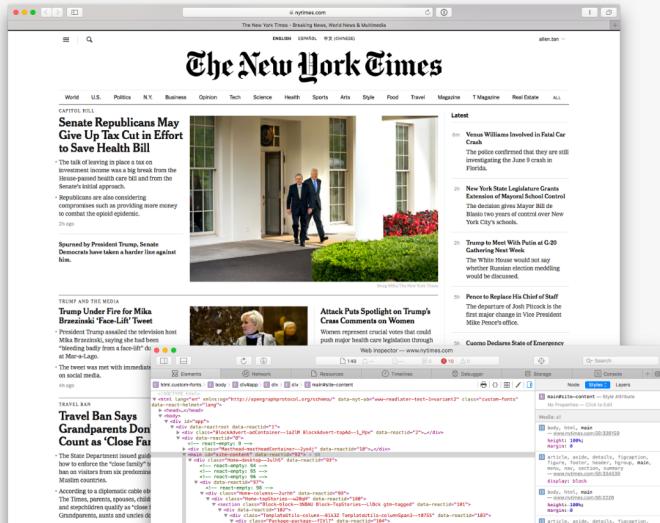
GITHUB



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



Lee Byron

@leeb

Folgen



While most discussion of [@GraphQL](#) centers around web apps, for the last 7 years Facebook only really used GraphQL for mobile.

Very excited for the new “FB5” version of [fb.com](#), powered entirely by React, GraphQL, and of course: Relay.

Tweet übersetzen

22:41 - 30. Apr. 2019

<https://twitter.com/leeb/status/1123326647552266241>

## FACEBOOK 5

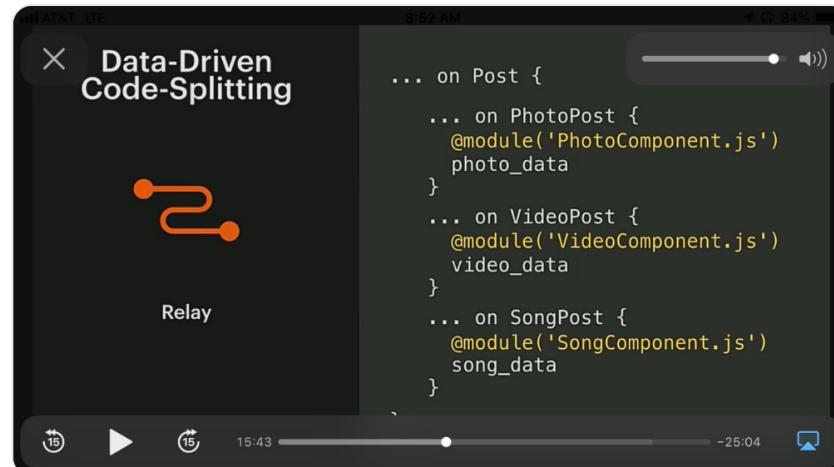


**Nick Schrock**  
@schrockn

Folgen

From the talk about the rewrite of fb using Relay and GraphQL. This feature is so amazing and intuitive. Deliver js only if the graphql query returns data that requires that js.

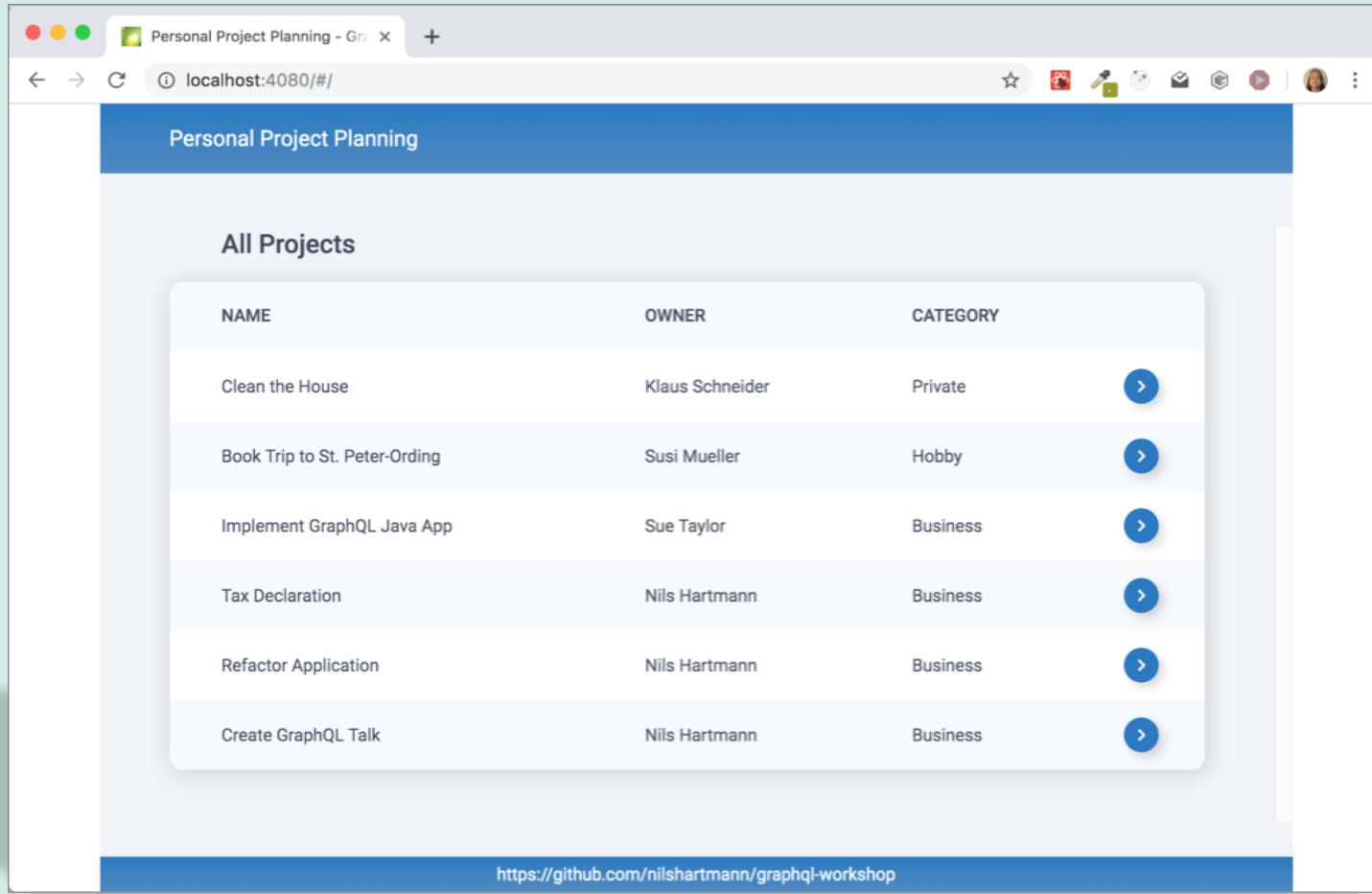
Tweet übersetzen



18:06 - 1. Mai 2019

<https://twitter.com/schrockn/status/1123619660732047360>

## NEXT GEN GRAPHQL?



# GraphQL praktisch

Source-Code: [graphql-workshop/app](https://github.com/nilshartmann/graphql-workshop/app)

The screenshot shows the Prisma GraphQL Playground interface at [localhost:4000](http://localhost:4000). The left panel displays a GraphQL query for 'AllProjectsQuery':

```
1 * query AllProjectsQuery {  
2 *   projects {  
3     id  
4     title  
5     description  
6     owner {  
7       name  
8     }  
9   }  
10 }  
11 }  
12 }
```

The 'login' field is highlighted with a blue selection bar, and a tooltip below it states: "String! The login is used by the user to log in to our System". The right panel contains the Prisma schema documentation, including type details for Project, mutations like addTask and updateTaskState, and subscriptions like onNewTask and onTaskChange.

# Demo: Playground

<https://github.com/prisma/graphql-playground>

<http://localhost:4000>



A screenshot of the IntelliJ IDEA IDE interface. The main editor window shows a piece of GraphQL code:

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

${...}
```

The cursor is positioned at the end of the first line of the query block. A tooltip is displayed over the word 'Beer' in the line 'Beer - Returns the Beer with the specified Id'. The tooltip contains the following information:

- f beer - Returns the Beer with the specified Id
- 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 tooltip, the code continues with:

```
ratings {
  id
  beerId
  author
  comment
}
```

At the bottom right of the editor window, the file name 'BeerPage.tsx' is visible.

# Demo: IDE Support

Beispiel: IntelliJ IDEA

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

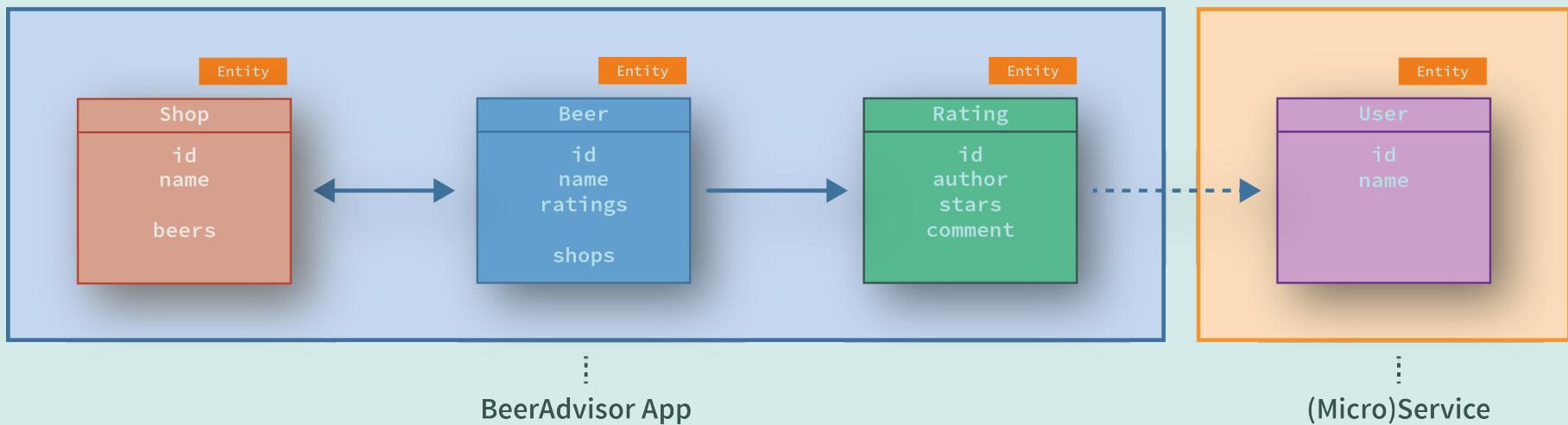
- <https://graphql.org>

# GraphQL

**TEIL 1: ABFRAGEN UND SCHEMA**

# GRAPHQL EINSATZSzenariEN

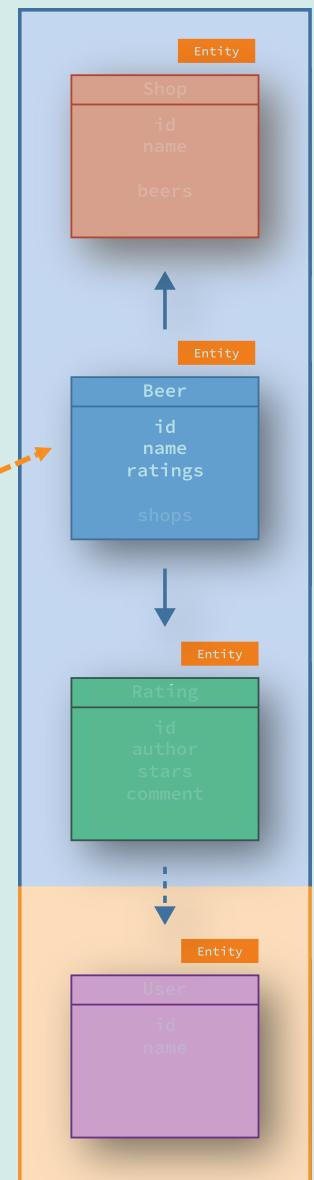
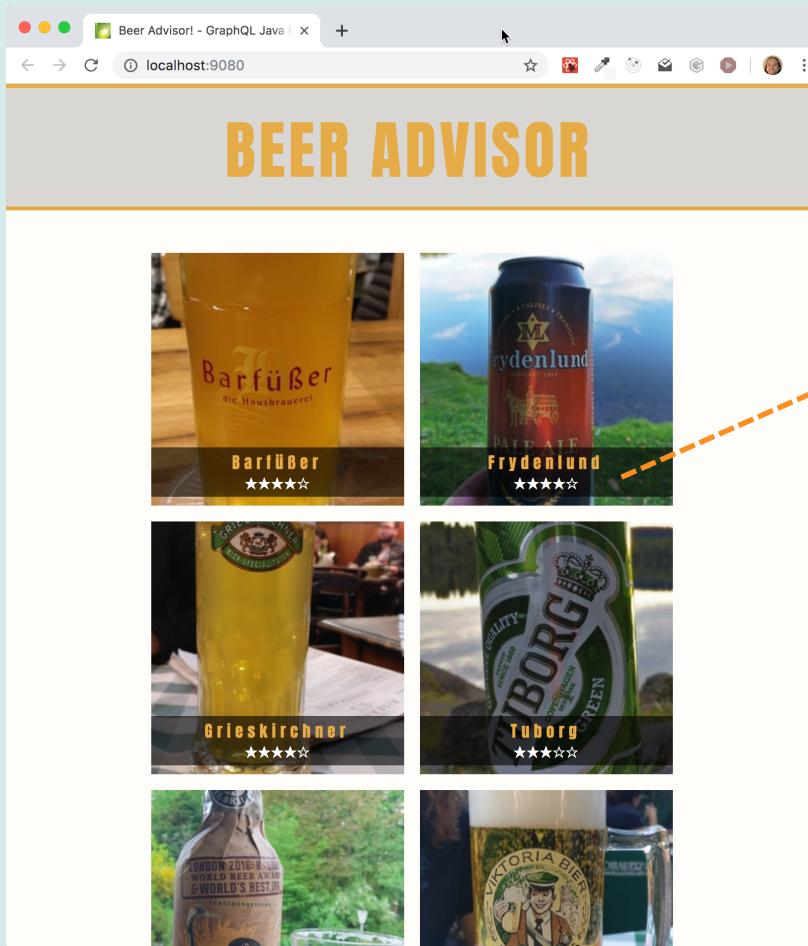
## "Architektur" Beer Advisor



# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 1

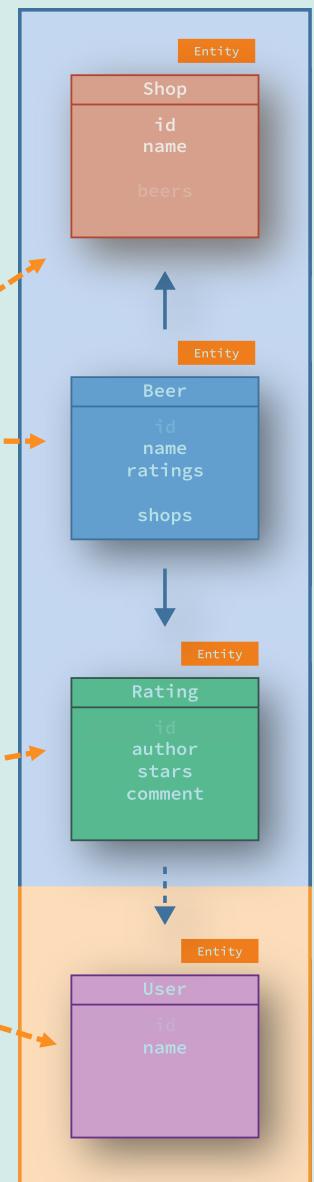
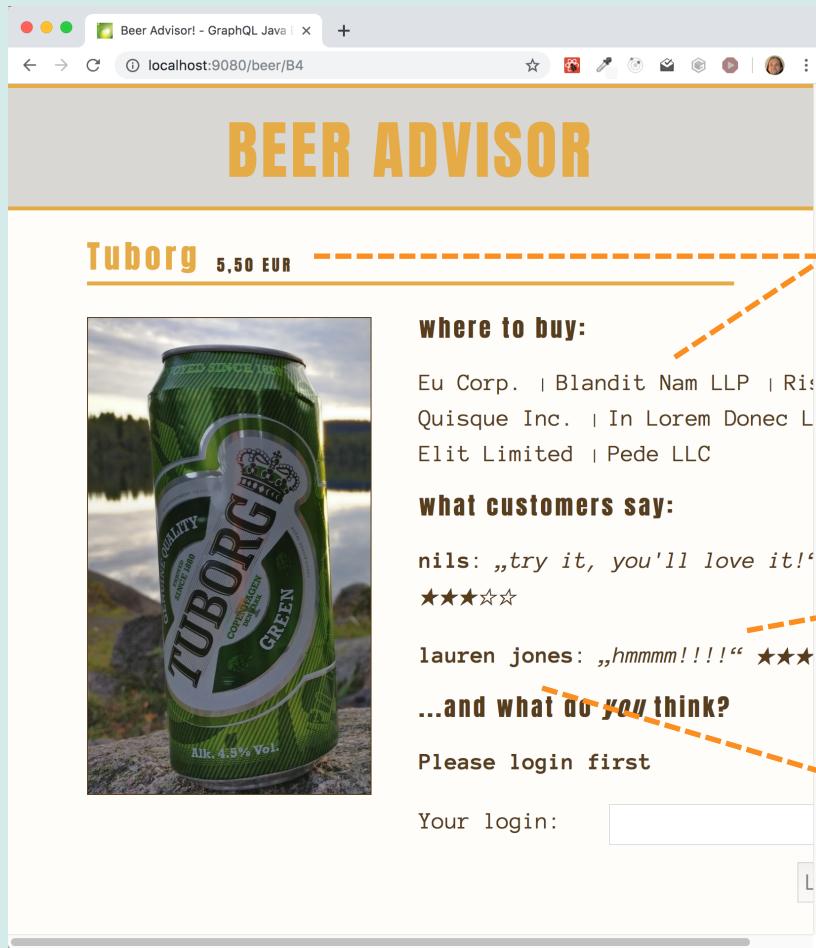
```
{ beer {  
    id  
    name  
    averageStars  
}
```



# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 2

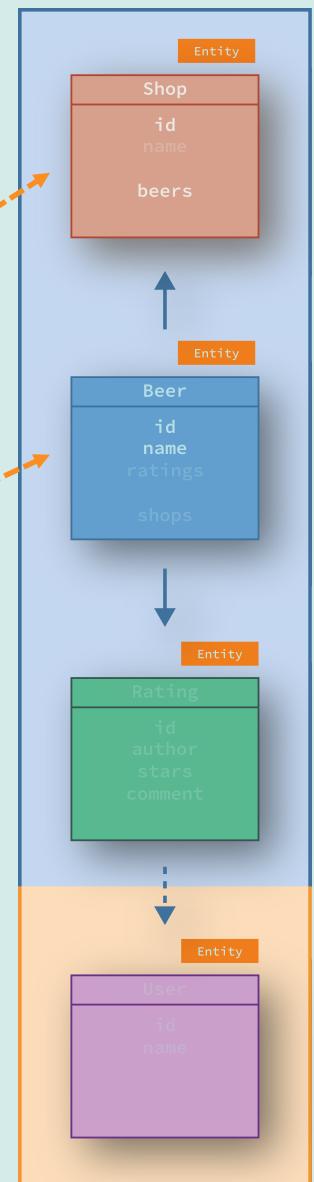
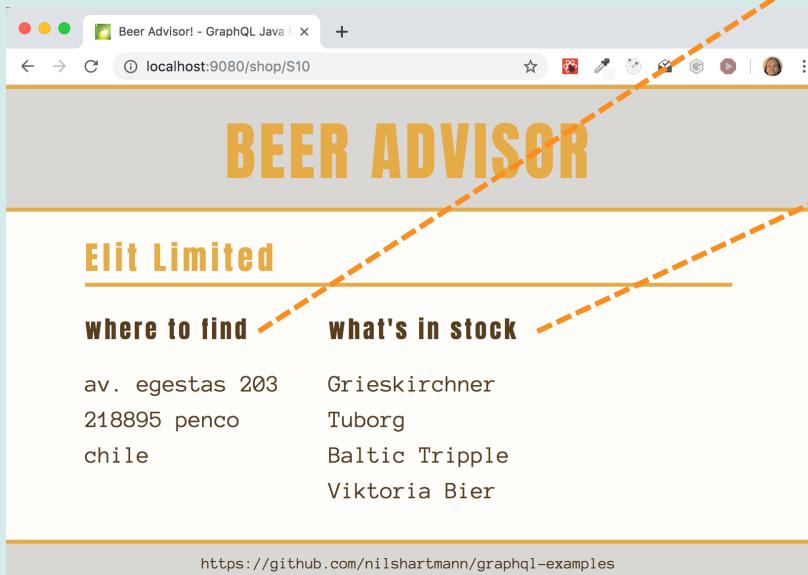
```
{ beer(beerId: "B1" {  
    name  
    price  
    ratings {  
        stars  
        comment  
        author {  
            name  
        }  
    }  
    shops { name }  
}
```



# GRAPHQL EINSATZSzenarien

## Use-Case spezifische Abfragen – 3

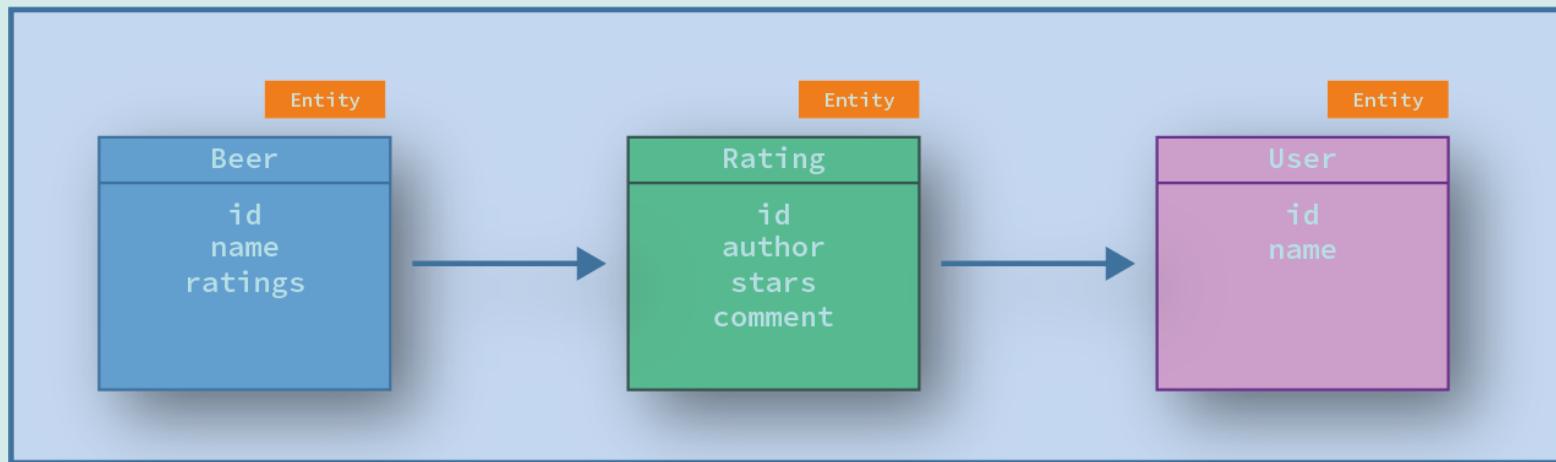
```
{ shop(shopId: "S3") {  
    name  
    address { street city }  
    beers { id name }  
}
```



# ABFRAGEN MIT GRAPHQL

## GraphQL

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



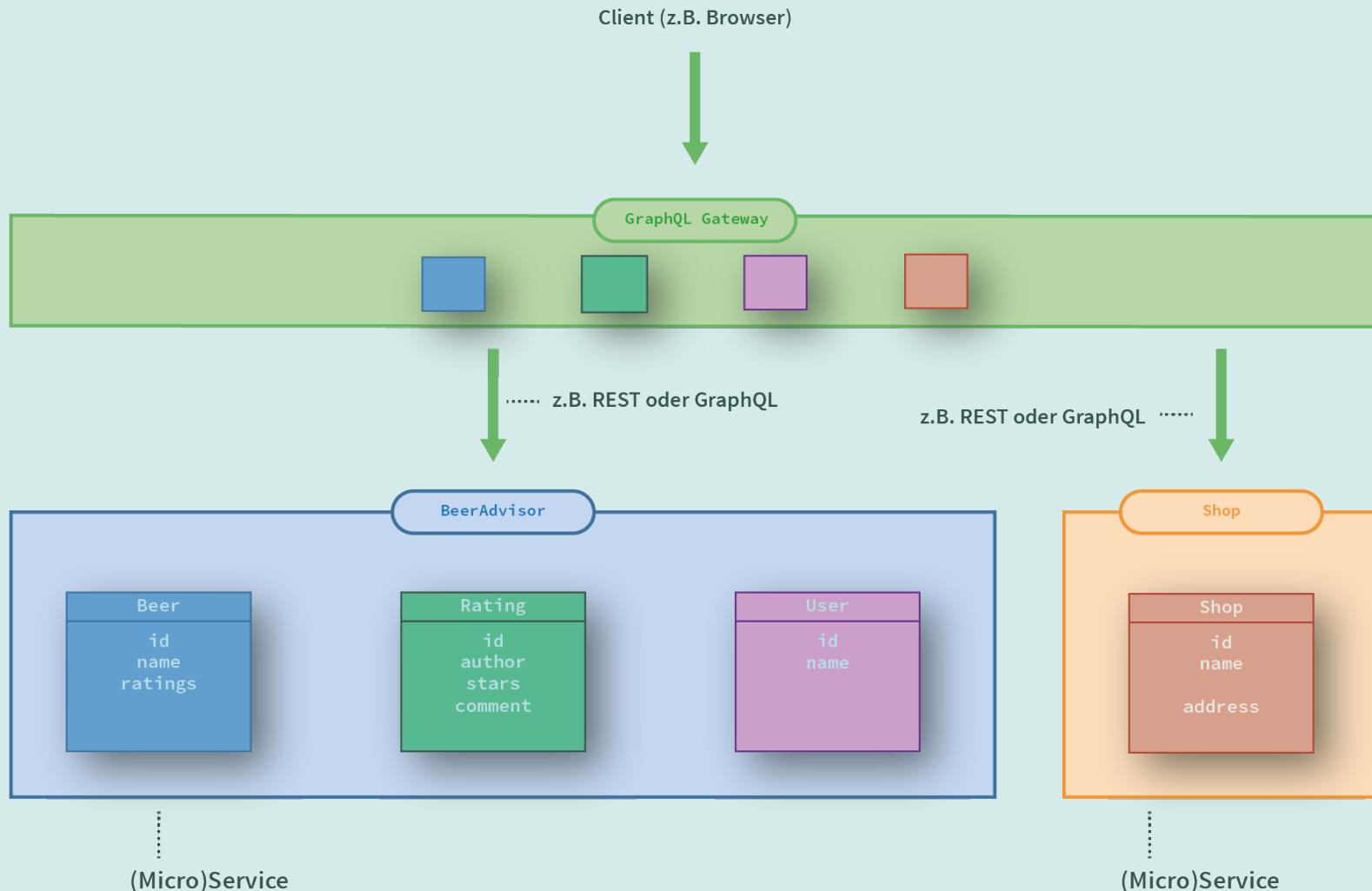
```
{  
  "name": "Barfüßer",  
  "ratings": {  
    "stars": 3,  
    "comment": "good",  
    "author": { "name": "Klaus" }  
  }  
}
```

### Gründe für den Einsatz von GraphQL

- Viele unterschiedliche Use-Cases, die unterschiedliche Daten benötigen
  - Unterschiedliche Ansichten im Frontend
  - Unterschiedliche Clients
  - Flexible Architektur im Client
- Einheitliche Gesamt-Sicht auf Domaine erwünscht
- Typ-sichere API erfordert
- Im Gegensatz zu REST (mehr) standardisiert und aus einer Hand

# EINSATZSzenarien

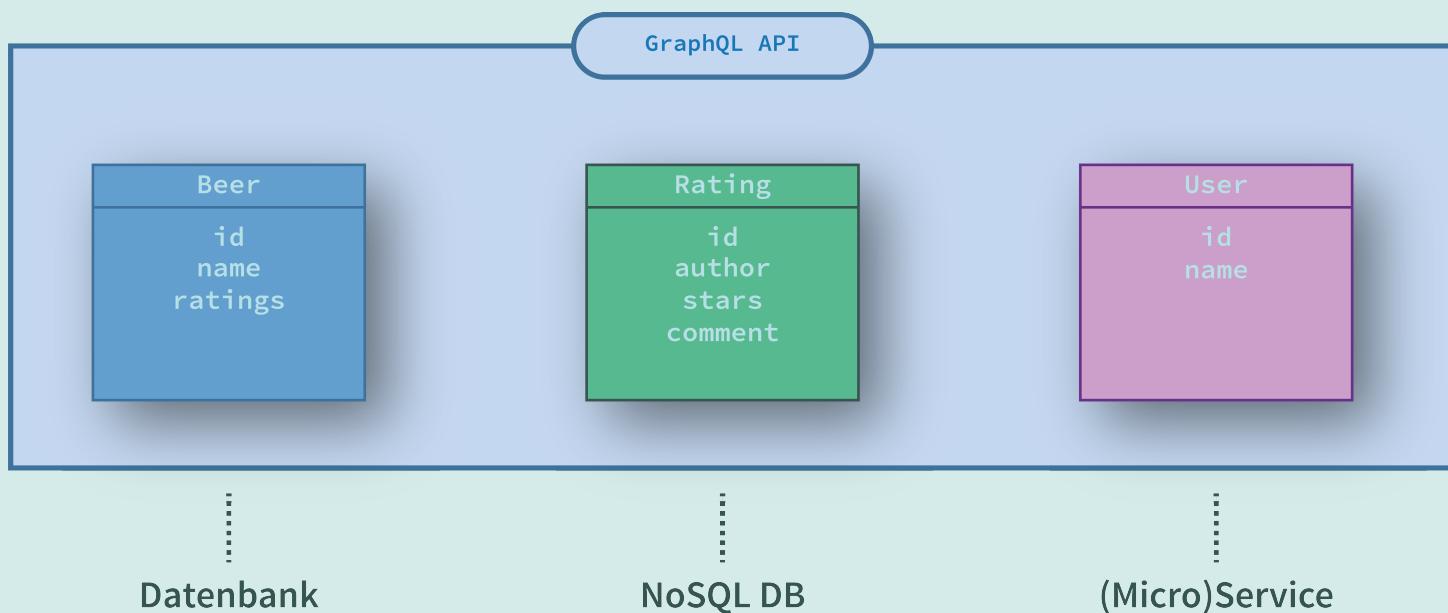
- Gateway für Frontend zu mehreren Backends



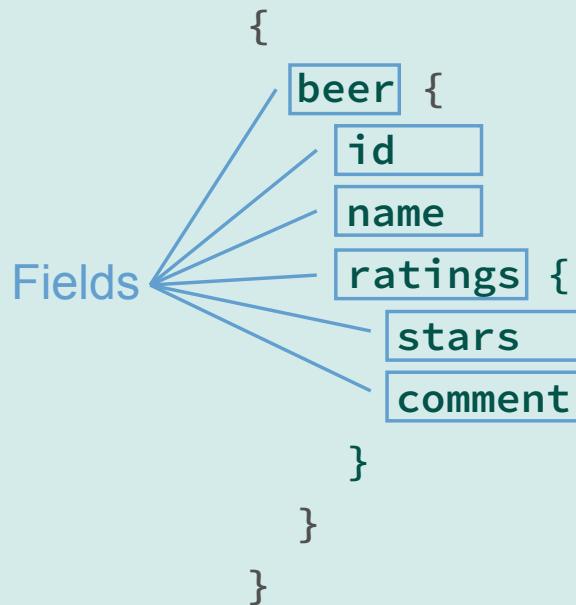
# DATEN QUELLEN

## GraphQL macht keine Aussage, wo die Daten herkommen

- Versteckt unterschiedliche APIs/Services
- Gesamt-Sicht auf die Domain/Anwendung
  - Fachliche Abfragen möglich
- *Ermittlung der Daten ist unsere Aufgabe*

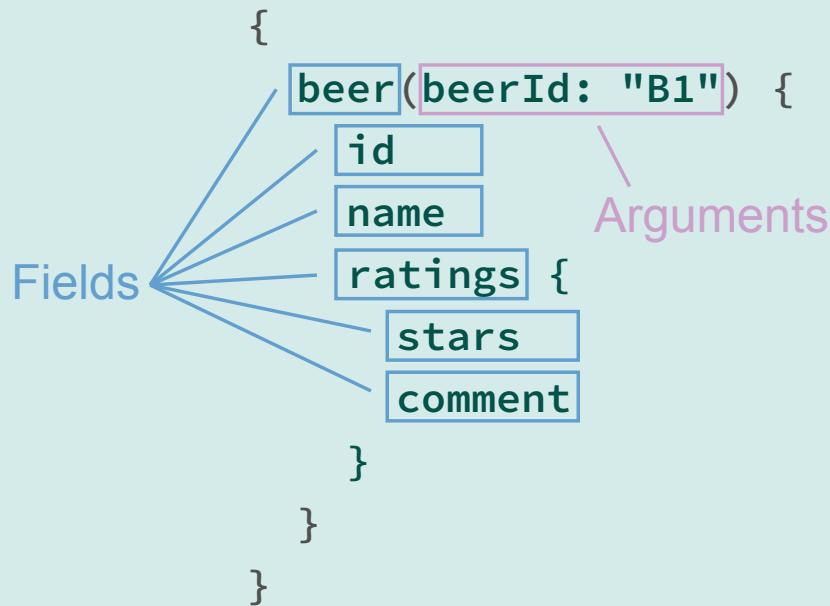


# QUERY LANGUAGE



- 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  
    }  
  }  
}
```



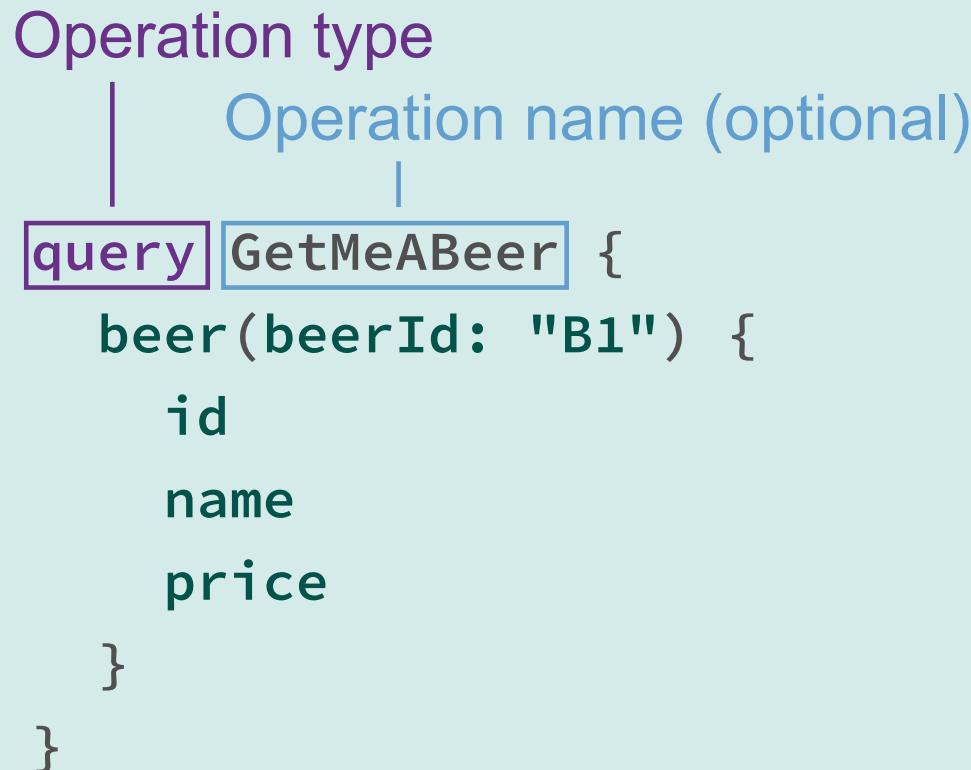
```
"data": {  
  "beer": {  
    "id": "B1"  
    "name": "Barfüßer"  
    "ratings": [  
      {  
        "stars": 3,  
        "comment": "grate taste"  
      },  
      {  
        "stars": 5,  
        "comment": "best beer ever!"  
      }  
    ]  
  }  
}
```

- Identische Struktur wie bei der Abfrage

# QUERY 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

## 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":"{ projects { title } }}'" \
  http://localhost:4000/graphql
```

```
{"data":  
  {"projects": [  
    {"title": "Create GraphQL Talk"},  
    {"title": "Book Trip to St. Peter-Ording"},  
    {"title": "Clean the House"},  
    {"title": "Refactor Application"},  
    {"title": "Tax Declaration"},  
    {"title": "Implement GraphQL Java App"}  
  ]}  
}
```

# QUERIES AUSFÜHREN

## Antwort vom Server

- Grundsätzlich HTTP 200
- (JSON-)Map mit max. drei Feldern

```
{  
  "errors": [  
    { "message": "Could not read User with ID 123",  
      "locations": [ . . . ],  
      "path": [ "project", "task", "assignee" ]  
    }  
  ],  
  "data": {"projects": [ . . . ] },  
  "extensions": { . . . }  
}
```

## **Übung:**

*Starte den Server ("backend") und den REST-Service (user)*

*Öffne den Playground (<http://localhost:4000>)*

*ggf. meinen Rechner benutzen*

*Mach dich mit der API des Projektes vertraut*

*Führe einen Query aus, mit dem Du alle Projekte und alle Benutzer findest*

*Füge einem Projekt eine neue Aufgabe ("Task") hinzu*

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

- <https://graphql.org>

# GraphQL Server

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

*"A community building flexible open source **tools for**  
**GraphQL.**"*

- <https://github.com/apollographql>

# Apollo

## Apollo-Server

**Apollo Server:** <https://www.apollographql.com/docs/apollo-server/>

- Basiert auf JavaScript GraphQL Referenzimplementierung
- "All-inclusive"-Lösung
  - Eingebauter Webserver plus Adapter (Connect, Express, Hapi, ...)
  - Playground zur Query Ausführung
  - Caching

# APOLLO-SERVER

**Apollo Server:** <https://www.apollographql.com/docs/apollo-server/>

- Konfiguration und Start
- Server läuft auf Port 4000 für Playground und Queries

Server Konfiguration

```
const { ApolloServer } = require("apollo-server");

const server = new ApolloServer({
  typeDefs: ....,
  resolvers: ....,
  context: ....,
  dataSources: ...
});
```

Server Start

```
server.listen()
  .then( info => console.log("Running"))
;
```

# GRAPHQL SERVER MIT APOLLO

## Aufgaben

1. Schema definieren
2. Resolver für das Schema implementieren
  - Wie/woher kommen die Daten für eine Anfrage
3. DataSources für Zugriff auf (externe) Daten
4. Server konfigurieren und starten (wie gesehen)

## Schema

- 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** <https://graphql.org/learn/schema/>

Object Type

Fields

```
type Project {  
  id: ID!  
  title: String!  
  description: String
```

```
}
```

# GRAPHQL SCHEMA

## Schema Definition per SDL

```
type Project {  
    id: ID! ----- Return Type (non-nullable)  
    title: String!  
    description: String ----- Return Type (nullable)  
}  
}
```

### Eingebaute skalare Typen:

- **Int**
- **Float**
- **String**
- **Boolean**
- **ID** (wird als String gelesen und geschrieben. Wert wird in der Anwendung nicht "interpretiert")

# GRAPHQL SCHEMA

## Schema Definition per SDL

```
type Project {  
    id: ID!  
    title: String!  
    description: String  
    owner: User! ----- Referenz auf anderen Typ  
}  
  
type User {  
    id: ID!  
    name: String!  
}
```



# GRAPHQL SCHEMA

## Schema Definition per SDL

```
type Project {  
    id: ID!  
    title: String!  
    description: String  
    owner: User!  
    tasks: [Task!]! ----- Return Type  
    }                                Liste / Array  
  
type User {  
    id: ID!  
    name: String!  
    }  
  
type Task { <--  
    id: ID!  
    }
```

# GRAPHQL SCHEMA

## Schema Definition per SDL

```
type Project {  
    id: ID!  
    title: String!  
    description: String  
    owner: User!  
    tasks: [Task!]!  
    task(taskId: ID!): Task  
}
```

**Argumente**

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

```
type Task {  
    id: ID!  
}
```

# GRAPHQL SCHEMA

## Schema Definition per SDL

```
enum TaskState {  
    NEW  
    RUNNING  
    FINISHED  
}
```

# GRAPHQL SCHEMA

## Schema Definition per SDL

```
enum TaskState {  
    NEW  
    RUNNING  
    FINISHED  
}
```

Aufzählungstyp

```
input AddTaskInput {  
    title: String!  
    description: String!  
    toBeFinished: String!  
    assigneeId: ID!  
}
```

Input-Typ  
(für Argumente)

# GRAPHQL SCHEMA

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

Root-Type  
("Query")

```
type Query {  
    projects: [Project!]!  
    project(projectId: ID!): Project  
}
```

Root-Fields

# GRAPHQL SCHEMA

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

Root-Type  
("Query")

```
type Query {  
    projects: [Project!]!  
    project(projectId: ID!): Project  
}
```

Root-Fields

Root-Type  
("Mutation")

```
type Mutation {  
    addTask(newTask: AddTaskInput): Task!  
}
```

Input Type

# GRAPHQL SCHEMA

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

The diagram illustrates the three root types of a GraphQL schema:

- Root-Type ("Query")**:  
A schema block containing:

```
type Query {  
    projects: [Project!]!  
    project(projectId: ID!): Project  
}
```

A callout labeled "Root-Fields" points to the field definitions (projects, project).
- Root-Type ("Mutation")**:  
A schema block containing:

```
type Mutation {  
    addTask(newTask: AddTaskInput): Task!  
}
```

A callout labeled "Input Type" points to the argument "newTask".
- Root-Type ("Subscription")**:  
A schema block containing:

```
type Subscription {  
    onNewTask: Task!  
    onTaskChange(projectId: ID!): Task!  
}
```

## SCHEMA WEITERENTWICKLUNG

### Nur eine Version: Felder werden immer explizit abgefragt

- Es können "ohne Schaden" neue Felder hinzugefügt werden
- Alte Felder können 'deprecated' werden
- Verwendung der Felder kann einzeln getrackt werden

Neues Feld -----

```
type Query {  
    projects: [Project!]!  
    project(projectId: ID!): Project @deprecated  
    getProjectById(projectId: ID!): Project  
}
```

# DAS SCHEMA IN APOLLO SERVER

## Type-Definition in Apollo Server über Schema-Definition-Language

- Erfolgt in der Regel in eigener Datei/eigenen Dateien

```
// schema.js
```

### Schema Definition

```
module.exports = `  
  type Project {  
    id: ID!  
    title: String!  
    description: String!  
    tasks: [Task!]!  
  }  
  
  type Task { . . . }
```

### Root-Fields (erforderlich)

```
  type Query {  
    projects: [Project!]!  
    project(projectId: ID!): Project  
  }  
`;
```

## DAS SCHEMA IN APOLLO SERVER

### Type-Definition in Apollo Server über Schema-Definition-Language

- Dokumentation in Markdown-Syntax mit """ hinzugefügt werden

```
// schema.js

module.exports = `

    """
        A **Project** consists of **Tasks**
    """

    type Project {
        id: ID!
        ...
    }

    type Query {
        """Get a project by its ID or null if not found"""
        project(projectId: ID!): Project
    }
`;
```

## DAS SCHEMA IN APOLLO SERVER

### Type-Definition in Apollo Server über Schema-Definition-Language

- Schema wird beim Server-Start übergeben

```
// server.js

const { ApolloServer } = require("apollo-server");
const typeDefs = require("./schema");

const server = new ApolloServer({
  typeDefs,
  resolvers: ...,
  context: ...,
  dataSources: ...
});

server.listen();
```

Konfiguration des Servers

# DAS SCHEMA IN APOLLO SERVER

## Modulare Schema

- Schema kann in mehrere Dateien aufgeteilt werden

```
// project.js
module.exports = `type Project { ... } `;

// query.js
module.exports = `type Query { ... }`;

// server.js
const projectTypes = require("./projects");
const queryTypes = require("./query");

const server = new ApolloServer({
  typeDefs: [projectTypes, queryTypes],
  ...
});
```

## ÜBUNG 1: SCHEMA DEFINIEREN

**Vervollständige das Schema der Beispiel-Anwendung**

# ÜBUNG 1: SCHEMA DEFINIEREN

## Vervollständige das Schema der Beispiel-Anwendung

### ◀ GRAPHQL-WORKSHOP

- ▶ app
- ◀ code-backend
  - ▶ 01\_schema\_fertig
  - ▶ 02\_resolver\_fertig
  - ▶ 03\_datasource\_fertig
  - ▶ userservice
  - ▶ workspace
- { } package-lock.json
- { } package.json
- ▶ code-frontend
- 🔗 graphql-workshop-enterjs.pdf

### Das Workshop Repository

Lösungen für die Übungen

Fertiger userservice (nur starten)

Verzeichnis für **Übungen** mit Ausgangsmaterial  
(in IDE/Editor öffnen)

Verzeichnis für React-Übungen (2. Teil)

Slides

# ÜBUNG 1: SCHEMA DEFINIEREN

## Vorbereitung: Installation und Starten (gemeinsam)

### ◀ GRAPHQL-WORKSHOP

- ▷ app
- ◀ code-backend
  - ▷ 01\_schema\_fertig
  - ▷ 02\_resolver\_fertig
  - ▷ 03\_datasource\_fertig
  - ▷ userservice
  - ▷ workspace
- { } package-lock.json
- { } package.json
- ▷ code-frontend
- 🔗 graphql-workshop-enterjs.pdf

### Das Workshop Repository

- hier "npm install" ausführen
- hier "npm start" ausführen
- hier "npm start" ausführen
- Achtung, Windows-Benutzer: entweder
  - yarn start
  - Bash/Linux Subsystem verwenden oder
  - in package.json "/" durch "\\" ersetzen
- Öffnen in der IDE/Editor
  - Bei Änderungen re-startet Server automatisch
  - Playground/API: <http://localhost:4000>

## ÜBUNG 1: SCHEMA DEFINIEREN

### Vervollständige das Schema der Beispiel-Anwendung

- Vorbereitung (gemeinsam): Starten aller Prozesse
  1. In "code-backend": "npm install"
  2. In "code-backend/userservice": "npm start"
  3. In "code-backend/workspace": "npm start"
  4. Playground sollte jetzt über <http://localhost:4000> erreichbar sein
- Um die Übungen zu machen, am Besten "code-backend/workspace" in deiner IDE/Editor öffnen
  - Nach dem ändern/speichern von Code wird der Server automatisch neugestartet
- Lösungen der Übungen: code-backend/01\_..., 02\_..., 03\_...

## ÜBUNG 1: SCHEMA DEFINIEREN

### Vervollständige das Schema der Beispiel-Anwendung

1. Der Project-Type muss definiert werden
  2. Der Query-Type muss um zwei Felder erweitert werden
- 
- In der Datei `workspace/src/schema.js` stehen TODOs drin
  - Nach Änderungen am Schema (speichern der Datei) könnt ihr im Playground Eure API-Änderungen sehen
    - <http://localhost:4000/>
    - (Auf "Docs" am rechten Rand klicken)
    - Hinweis: Ausführen der Queries funktioniert noch nicht

## SCHRITT 2: RESOLVER

## SCHRITT 2: RESOLVER

### Resolver-Funktion

*Ein Resolver liefert einen Wert für ein angefragtes Feld in einer Query*

- Zwingend erforderlich für jedes Root-Field (Query, Mutation, Subscription)
  - ab da per "Property-Pfad" weiter (root.projects.task.assignee)
  - oder per speziellem Resolver
- Eingehende Argumente und Rückgabewert wird validiert
  - Nur gültige Queries werden an Resolver gegeben
  - Nur gültige Antworten kommen an den Client zurück

## SCHRITT 2: RESOLVER

### Resolver-Funktionen

Schema Definition

```
type Query {  
  ping: String!  
}
```

Query

```
query { ping }
```



```
"data": {  
  "ping": "Hello World"  
}
```

## SCHRITT 2: RESOLVER

### Resolver-Funktionen

- Werden in einem Objekt angegeben. Key ist der Name des Objektes
- Darin die Funktionen für dieses Objekt (Key = Name des Feldes)

Schema Definition

```
type Query {  
  ping: String!  
}
```

Query

```
query { ping } → "data": {  
  "ping": "Hello World"  
}
```

Resolver-Map

```
const resolvers = {  
  Query: {  
    ping: () => "Hello World",  
  },  
}
```

Resolver für 'ping'-Feld

## SCHRITT 2: RESOLVER

### Beispiel: Root-Resolver mit Argumenten

Schema Definition

```
type Query {  
  ping(msg: String!): String!  
}
```

```
query {  
  ping(msg: "EnterJS") → "data": {  
    "ping": "Hello, EnterJS"  
}
```

## SCHRITT 2: RESOLVER

### Beispiel: Root-Resolver mit Argumenten

- Argumente werden der Resolver-Funktion als 2. Parameter (Objekt) übergeben
- Es werden nur gültige Werte übergeben werden (gemäß Schema)

Schema Definition

```
type Query {  
  ping(msg: String!): String!  
}
```

```
query {  
  ping(msg: "EnterJS") → "data": {  
    "ping": "Hello, EnterJS"  
  }  
}
```

Resolver mit Argumenten

```
const resolvers = {  
  Query: {  
    ping: (_, { msg }) => `Hello, ${msg}`  
  }  
}
```

## SCHRITT 2: RESOLVER

### Resolver: Der Context

- Context ist ein beliebiges Objekt, das für jeden Request erzeugt wird
- Wird den Resolver-Funktionen zur Verfügung gestellt
- Kann z.B. DataSources enthalten oder aktuellen Benutzer
  - DataSources werden automatisch unter 'dataSources' in den Context gelegt

## SCHRITT 2: RESOLVER

### Resolver: Der Context

- Context ist ein beliebiges Objekt, das für jeden Request erzeugt wird
- Wird den Resolver-Funktionen zur Verfügung gestellt
- Kann z.B. DataSources enthalten oder aktuellen Benutzer
  - DataSources werden automatisch unter 'dataSources' in den Context gelegt

#### Context Definition

```
const context = (req) => (  
  { user: req.headers.user }  
) ;
```

## SCHRITT 2: RESOLVER

### Resolver: Der Context

- Context ist ein beliebiges Objekt, das für jeden Request erzeugt wird
- Wird den Resolver-Funktionen zur Verfügung gestellt
- Kann z.B. DataSources enthalten oder aktuellen Benutzer
  - DataSources werden automatisch unter 'dataSources' in den Context gelegt

#### Context Definition

```
const context = (req) => (
  { user: req.headers.user }
);
```

#### DataSources

```
const dataSources = (req) => (
  { projectDataSource: new ProjectDataSource() }
);
```

## SCHRITT 2: RESOLVER

### Resolver: Der Context

- Context ist ein beliebiges Objekt, das für jeden Request erzeugt wird
- Wird den Resolver-Funktionen zur Verfügung gestellt
- Kann z.B. DataSources enthalten oder aktuellen Benutzer
  - DataSources werden automatisch unter 'dataSources' in den Context gelegt

#### Context Definition

```
const context = (req) => (
  { user: req.headers.user }
);
```

#### DataSources

```
const dataSources = (req) => (
  { projectDataSource: new ProjectDataSource() }
);
```

#### Server-Konfiguration

```
const server = new ApolloServer({
  typeDefs,
  context,
  dataSources
});
```

## SCHRITT 2: RESOLVER

### Resolver: Der Context

- Der Context wird jedem Resolver als 3. Parameter übergeben

#### Context Definition

```
const context = (req) => (
  { user: req.headers.user }
);
```

#### Resolver mit Context

```
const resolvers = {
  Query: {
    ping: (_, _args, { user }) => `Hello, ${user}`
  }
}
```

## SCHRITT 2: RESOLVER

### Resolver: Der Context

- Definierte DataSources werden dem Context automatisch hinzugefügt

#### Schema Definition

```
type Query {  
  projects: [Project!]!  
}
```

#### DataSources

```
const dataSources = (req) => (  
  { projectDataSource: new ProjectDataSource() }  
)
```

#### Resolver mit Context

```
const resolvers = {  
  Query: {  
    projects: (_, _args, { dataSources }) => {  
      return dataSources  
        .projectDataSource.getAllProjects();  
    }  
  }  
}
```

## SCHRITT 2: RESOLVER

### Resolver für ein Feld eines *eigenen Types*

- Erlaubt individuelle Behandlung für einzelne Felder
- Zum Beispiel laden von Daten
- Source-Objekt wird als 1. Parameter an den Resolver übergeben

#### Schema Definition

```
type Project {  
    tasks: [Tasks!]! ----- tasks müssen aus DB geladen werden!  
    ...  
}
```

## SCHRITT 2: RESOLVER

### Resolver für ein Feld eines *eigenen Types*

- Erlaubt individuelle Behandlung für einzelne Felder
- Zum Beispiel laden von Daten
- Source-Objekt wird als 1. Parameter an den Resolver übergeben

#### Schema Definition

```
type Project {  
    tasks: [Tasks!]! ----- tasks müssen aus DB geladen werden!  
    . . .  
}
```

#### Resolver

```
const resolvers = {  
    Query: { . . . },  
    Project: {  
        tasks: (project, _, {dataSources}) => {  
            return dataSources.projectDatasource.getTasks  
                (project._taskId)  
        }  
    }  
}
```

## SCHRITT 2: RESOLVER

### Resolver für Mutations

- Mutations können nur auf top-level-Ebene definiert werden
- Resolver analog zu Query, selbe API, Datenänderungen möglich

#### Schema Definition

```
type Mutation {  
    updateTaskState(taskId: ID!, newState: TaskState!): Task!  
}
```

#### Mutation-Resolver

```
const resolvers = {  
    Query: { . . . },  
    Project: { . . . },  
    Mutation: {  
        updateTaskState(_, {taskId, newState}, {dataSources}) => {  
            return dataSources  
                .projectDataSource.updateTaskState(taskId, newState);  
        }  
    }  
}
```

## SCHRITT 2: RESOLVER

### Die Resolver-Methode - Zusammenfassung

- Resolver werden in "Resolver-Map" gruppiert
- Auf oberster Ebene für Objekte, darunter Funktionen für Felder

```
const resolvers = {
  Query: {
    ping: () => ...
    projects: () => ...
  },
  Mutation: { updateTask: () => ... }
  Projects: { tasks: () => ... }
}
```

Signatur: `fieldname(source, args, context, info): Wert`

- source nicht bei Root-Resolvern befüllt
- args und context jeweils Objekt mit Key-Value-Paaren
- info enthält Weitere Meta-Daten zum aktuellen Query

## SCHRITT 2: RESOLVER

### Resolver beim Server anmelden

- Resolver müssen ein Objekt sein, dessen Keys jeweils so heißen, wie das Objekt, für das sie Funktionen definieren (Query, Mutation, Project...)

Schema Definition

```
const resolvers = {
  Query: {
    ping: (_, { msg }) => `Hello, ${msg}`
  },
  Mutation: { ... },
  Subscription: { ... },
  Project: { ... }
}
```

Root-Resolver

```
const server = new ApolloServer({
  typeDefs,
  resolvers
});
```

Root-Resolver mit Argumenten

## SCHRITT 2: RESOLVER

### Resolver modularisieren

Aufteilung z.B. nach Domainen

```
// query.js
modules.export = {
  ping: () => ...
  projects: () => ...
}

// projects.js
modules.export = { . . . }

// server.js
const query = require("./query");
const project = require("./project");

const server = new ApolloServer({
  typeDefs, context, dataSources,
  resolvers: {
    Query: query,
    Project: project
  }
});
```

## ÜBUNG 2: RESOLVER IMPLEMENTIEREN

### Implementiere fehlende Resolver für unsere Anwendung

- Am Query: Felder **projects** und **project**
- Am Task: Felder **tasks** und **task**

Die Änderungen müssen in **query.js** und **project.js** vorgenommen werden

- dort sind entsprechende TODOs eingetragen
- (Falls Du mit Übung 1 nicht fertig geworden bist, einfach Dateien aus **01\_schema\_fertig** in deinen Workspace kopieren)

Wenn die Resolver implementiert sind, kannst Du über den Playground Queries ausführen

- Funktionierende Queries zum Testen auf der nächsten Slide
- Zugriff auf User (owner bzw. assignee) werden noch nicht funktionieren

## ÜBUNG 2: RESOLVER IMPLEMENTIEREN

**Implementiere fehlende Resolver für unsere Anwendung**

Nach dem Implementieren sollten folgende Queries funktionieren:

```
query {  
  projects {  
    id title  
    tasks {  
      id title  
    }  
  }  
}
```

```
query {  
  project(id: "1") {  
    id title  
    task(id: "2002") {  
      id title  
    }  
  }  
}
```

## SCHRITT 3: DATASOURCES

## SCHRITT 3: DATASOURCES

### DataSources

*DataSources können für den Zugriff auf externe Systeme verwendet werden*

- Zum Beispiel REST-Service, Datenbank etc
- Fertige Implementierung für REST-Services
- Community-Implementierungen u.a. für Postgres und MySQL

## SCHRITT 3: DATASOURCES

### DataSources: Zugriff auf Datenbank

- Leider keine Standard-Lösung von Apollo
- Zugriff auf Datenbank in unserer Anwendung in ProjectSQLiteDataSource
- Implementierung hämdsärmlig und naiv
  - Kein Caching, keine optimierten Queries
  - In echten Leben bitte "bessere" Implementierung verwenden
- These: Zugriff auf REST deutlich öfter als auf Datenbank
  - Weil per GraphQL Microservices "aggregiert" werden

## SCHRITT 3: DIE REST DATASOURCE

**Die REST Data Source** <https://www.apollographql.com/docs/apollo-server/features/data-sources/>

- Fester Bestandteil von Apollo (apollo-datasource-rest)

## SCHRITT 3: DIE REST DATASOURCE

### Die REST Data Source <https://www.apollographql.com/docs/apollo-server/features/data-sources/>

- Fester Bestandteil von Apollo (apollo-datasource-rest)

```
const { RESTDataSource } = require("apollo-datasource-rest");
```

Klasse definieren

```
class UserRESTDataSource extends RESTDataSource {  
  constructor() {  
    super();  
    this.baseURL = "http://localhost:4010/";  
  }
```

URL des Services  
(kann auch dynamisch gesetzt werden)

Zugriff auf Service  
(auch POST, PUT, ... möglich)

```
  listAllUsers() {  
    return this.get("users");  
  }  
  
  getUser(id) {  
    return this.get(`users/${id}`)  
      .catch(err => {  
        if (getStatusCode(err) === 404) return null;  
        throw err;  
      });  
  }  
}
```

→ (GET http://localhost:4010/users)

→ (GET http://localhost:4010/users/ID)

## SCHRITT 3: DIE REST DATASOURCE

### Die REST Data Source

- Base URL wird im Konstruktor gesetzt
- In fachlichen Methoden werden die Requests ausgeführt
- Zum Ausführen der Requests wird fetch-Bibliothek genutzt
  - Entsprechende Methoden this.get, this.post, this.delete, ...
- Request Header etc können ebenfalls gesetzt werden
- Caching

## SCHRITT 3: DIE REST DATASOURCE

### Die REST Data Source: Laufzeitverhalten

#### Resolver

```
const resolvers = {
  Query: {
    projects(_, __, { dataSources }) => {
      return dataSources.projectDataSource.getAllProjects();
    }
  },
  Project: {
    owner(project, __, { dataSources }) => {
      return dataSources.userDataSource.getUser(project._ownerId);
    }
  }
}
```

## SCHRITT 3: DIE REST DATASOURCE

### Die REST Data Source: Laufzeitverhalten

#### Resolver

```
const resolvers = {
  Query: {
    projects(_, __, { dataSources }) => {
      return dataSources.projectDataSource.getAllProjects();
    }
  },
  Project: {
    owner(project, __, { dataSources }) => {
      return dataSources.userDataSource.getUser(project._ownerId);
    }
  }
}
```

#### Query

```
query {
  projects {
    owner {
      name
    }
  }
}
```

*Frage: Was passiert beim Ausführen dieses Queries?* 🤔

## SCHRITT 3: DIE REST DATASOURCE

### Die REST Data Source: Laufzeitverhalten

EIN Datenbankzugriff  
(liefert  $n$  Projekte)

$n$  REST-Aufrufe  
(1x je Project)

```
const resolvers = {
  Query: {
    projects(_, __, { dataSources }) => {
      return dataSources.projectDataSource.getAllProjects();
    }
  },
  Project: {
    owner(project, __, { dataSources }) => {
      return dataSources.userDataSource.getUser(project._ownerId);
    }
  }
}

query {
  projects {
    owner {
      name
    }
  }
}
```

1+n-Problem 😱

## SCHRITT 3: DIE REST DATASOURCE

### Die REST Data Source cached Ergebnisse

- Wird die Rest-Datasource mehrfach mit selber URL aufgerufen, wird das Ergebnis gecached
- Es werden HTTP Aufrufe eingespart

## SCHRITT 3: DIE REST DATASOURCE

### Die REST Data Source cached Ergebnisse

- Wird die Rest-Datasource mehrfach mit selber URL aufgerufen, wird das Ergebnis gecached
- Es werden HTTP Aufrufe eingespart
- Zusätzlich: Caching der Antwort, wenn HTTP Header kommt
  - cache-control Header aus dem Server

## SCHRITT 3: DIE REST DATASOURCE

### Die REST Data Source cached Ergebnisse

- Wird die Rest-Datasource mehrfach mit selber URL aufgerufen, wird das Ergebnis gecached
- Es werden HTTP Aufrufe eingespart
- Zusätzlich: Caching der Antwort, wenn HTTP Header kommt
  - cache-control Header aus dem Server
- Alternative: Batching (mehrere Aufrufe zusammenfassen)
  - Geht auch, muss der Remote-Service aber unterstützen
  - Typische Implementierung: DataLoader
    - <https://github.com/graphql/dataloader>

## (OPTIONAL) ÜBUNG 3: DATA SOURCE IMPLEMENTIEREN

### Die REST DataSource implementieren

- Die REST DataSource soll auf den userservice zugreifen
  - Zwei Methoden werden benötigt:
    - `listAllUsers()`
    - `getUser(id)`
1. Implementiere die fehlenden Methoden in `UserRESTDataSource.js`
  2. Füge die DataSource dem Server hinzu (`server.js`)
    - Dort sind jeweils entsprechende TODOs eingetragen
  3. Prüfe an Hand des Loggings, wieviele Anfragen beim vorherigen Query tatsächlich an den Userservice abgesetzt werden

Bonus:

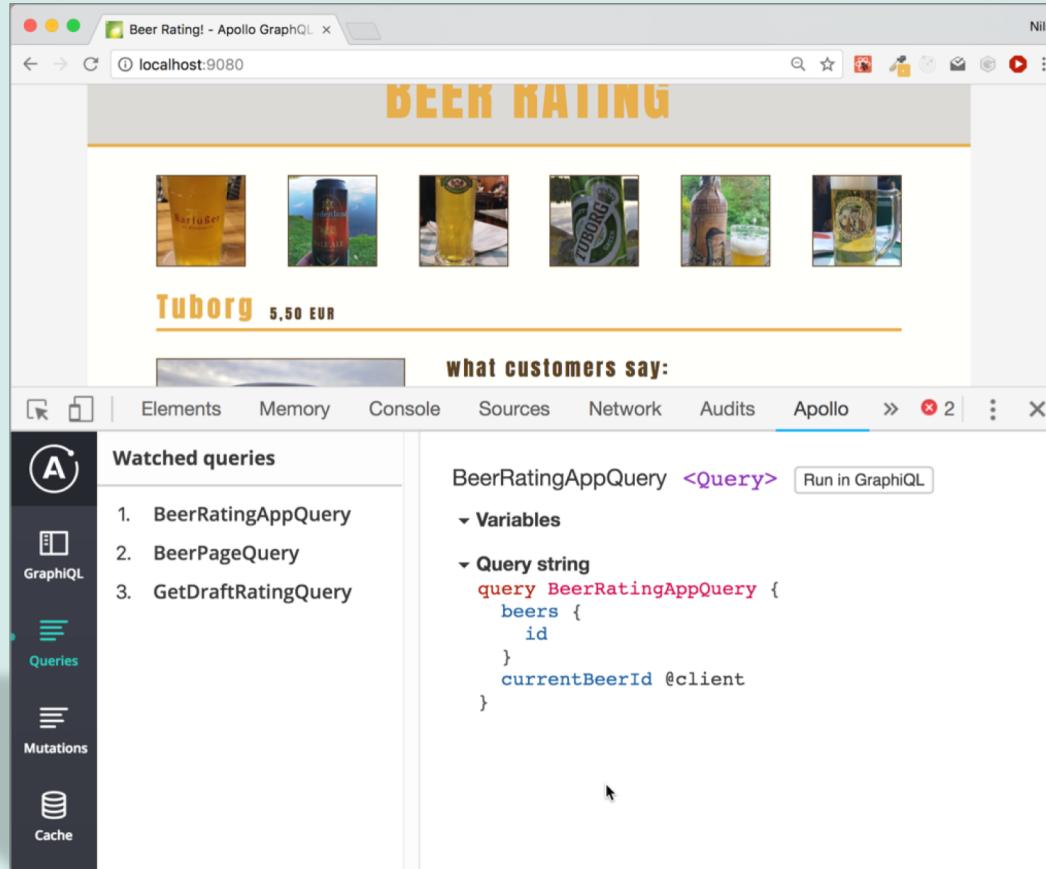
- Kommentiere in `userservice/index.js` die Cache-Header ein (Zeile 41)
- Wie ändert sich das Request Verhalten? (=> Logging!)

**AB HIER TODO**

# GraphQL Clients

TYP-SICHERE CLIENTS MIT REACT, APOLLO UND TYPESCRIPT

# Demo: Apollo Dev Tools



## APOLLO-SERVER

**React Apollo:** <https://www.apollographql.com/docs/react/>

- React-Komponenten zum Zugriff auf GraphQL APIs
  - funktioniert mit allen GraphQL Backends
- Sehr modular aufgebaut, viele npm-Module
- Anwendungsweiter, globaler **Cache** sorgt für konsistente Darstellung der Daten

## SCHRITT 1: ERZEUGEN DES CLIENTS UND PROVIDERS

### ApolloClient ist für Kommunikation mit Backend zuständig

- Zentraler Cache, Authentifizierung, Fehlerbehandlung, ...

#### Bootstrap der React-Anwendung

```
import ApolloClient from "apollo-client";
```

#### Client erzeugen

```
const client = new ApolloClient({  
  link: new HttpLink({uri: "http://..."}),  
  cache: new InMemoryCache()  
});
```

# SCHRITT 1: ERZEUGEN DES CLIENTS UND PROVIDERS

## Provider stellt Client in React Komponenten zur Verfügung

- Zugriff dann in allen Komponenten möglich

### Bootstrap der React-Anwendung

```
import ApolloClient from "apollo-client";
import { ApolloProvider } from "react-apollo";
```

### Client erzeugen

```
const client = new ApolloClient({
  link: new HttpLink({uri: "http://..."}),
  cache: new InMemoryCache()
});
```

### Apollo Provider um Anwendung legen

```
ReactDOM.render(
  <ApolloProvider client={client}>
    <ProjectApp />
  </ApolloProvider>,
  document.getElementById('...'))
);
```

## SCHRITT 2: QUERIES

### Queries

- Werden mittels gql-Funktion angegeben und geparsst

Query parsen

```
import { gql } from "react-apollo";

const BEER_RATING_APP_QUERY = gql`  
  query BeerRatingAppQuery {  
    beers {  
      id  
      name  
      price  
  
        ratings { . . . }  
    }  
  }  
`;
```

## SCHRITT 2: QUERIES

### Query-Komponente

- Führt Query aus, kümmert sich um Caching, Fehlerbehandlung etc

```
import { gql, Query } from "react-apollo";  
  
const BEER_RATING_APP_QUERY = gql`...`;  
  
function BeerRatingApp(props) {  
  return <Query query={BEER_RATING_APP_QUERY}>  
    </Query>  
};
```

React Komponente

## SCHRITT 2: QUERIES

### Query-Komponente

- Führt Query aus, kümmert sich um Caching, Fehlerbehandlung etc
- Ergebnis (Daten, ...) wird per Render-Prop (Children) übergeben

```
import { gql, Query } from "react-apollo";

const BEER_RATING_APP_QUERY = gql`...`;

function BeerRatingApp(props) {
  return <Query query={query}>
    {({ loading, error, data }) => {
      ...
    }}
  </Query>
};
```

Query Ergebnis  
(wird ggf mehrfach  
aufgerufen)

## SCHRITT 2: QUERIES

### Query-Komponente

- Führt Query aus, kümmert sich um Caching, Fehlerbehandlung etc
- Ergebnis (Daten, ...) wird per Render-Prop (Children) übergeben

```
import { gql, Query } from "react-apollo";

const BEER_RATING_APP_QUERY = gql`...`;

function BeerRatingApp(props) {
  return <Query query={query}>
    {({ loading, error, data }) => {
      if (loading) { return <h1>Loading...</h1> }
      if (error) { return <h1>Error!</h1> }

      return <BeerList beers={data.beers} />
    }}
  </Query>
};
```

Ergebnis (samt Fehler)  
auswerten

## SCHRITT 2: QUERIES

### Alternative: useQuery Hook (Alpha!)

- Support für React Hook API in Alpha Version verfügbar

```
import { useQuery } from "@apollo/react-hooks";

const QUERY = gql`...`;

function BeerRatingApp(props) {
  const { loading, error, data } = useQuery(QUERY);
  if (loading) { return <h1>Loading...</h1> }
  if (error) { return <h1>Error!</h1> }

  return <BeerList beers={data.beers} />
}
```

## SCHRITT 2: QUERIES

### Mit TypeScript: Typ-sicherer Zugriff auf Ergebnis

- Wird typisiert mit Query-Resultat und ggf. Variablen
- TS Interfaces können mit apollo client:codegen generiert werden

```
import { gql, Query } from "react-apollo";
import { BeerRatingAppQuery } from "./__generated__/...";
```

```
function BeerRatingApp(props) {
  return <Query<BeerRatingAppQuery> query={query}>
    {({ loading, error, data }) => {
      // . . .
    }}
```

Compile-Fehler!

```
      return <BeerList beers={data.biere} />
    </Query>
};
```

## SCHRITT 3: MUTATIONS

### Mutation-Komponente: Führt Mutations aus

- Mutation wird ebenfalls per gql geparsst

```
import { gql } from "react-apollo";

const ADD_RATING_MUTATION = gql`  
  mutation AddRatingMutation($input: AddRatingInput!) {  
    addRating(ratingInput: $input) {  
      id  
      beerId  
      author  
      comment  
    }  
  }  
`;
```

## SCHRITT 3: MUTATIONS

### Mutation-Komponente: Führt Mutations aus

- Funktionsweise ähnlich wie Query-Komponente

```
import { gql, Mutation } from "react-apollo";

const ADD_RATING_MUTATION = gql`...`;

function AddRating(props) {
  return <Mutation mutation={ADD_RATING_MUTATION}>
    </Mutation>
}
```

## SCHRITT 3: MUTATIONS

### Mutation-Komponente: Führt Mutations aus

- Funktionsweise ähnlich wie Query-Komponente
- Render-Property erhält Funktion zum Ausführen der Mutation

```
import { gql, Mutation } from "react-apollo";

const ADD_RATING_MUTATION = gql`...`;

function AddRating(props) {
  return <Mutation mutation={ADD_RATING_MUTATION}>
    {addRating => {
      }
    }
  </Mutation>
}
```

## SCHRITT 3: MUTATIONS

### Mutation-Komponente: Führt Mutations aus

- Funktionsweise ähnlich wie Query-Komponente
- Render-Property erhält Funktion zum Ausführen der Mutation

```
import { gql, Mutation } from "react-apollo";

const ADD_RATING_MUTATION = gql`...`;

function AddRating(props) {
  return <Mutation mutation={ADD_RATING_MUTATION}>
    {addRating => {
      return <RatingForm onNewRating={
        newRating => addRating({
          variables: {ratingInput: newRating}
        })
      } />
    }
  </Mutation>
}
```

## SCHRITT 3: MUTATIONS

### Mutation-Komponente: Cache aktualisieren

- Callback-Funktionen zum aktualisieren des lokalen Caches
  - Aktualisiert automatisch sämtliche Ansichten

```
function AddRating(props) {  
  return <Mutation mutation={ADD_RATING_MUTATION }  
    update={(cache, {data}) => {  
      // "Altes" Beer aus Cache lesen  
      const oldBeer = cache.readFragment(...);  
  
      // Neues Rating dem Beer hinzufügen  
      const newBeer = ...;  
  
      // Beer im Cache aktualisieren  
      cache.writeFragment({data: newBeer});  
    }}>  
  . . .  
</Mutation>  
}
```



# Vielen Dank!

Beispiel-Code: <https://nils.buzz/oose-graphql-example>

Slides: <https://nils.buzz/oose-graphql>

Mehr GraphQL: <https://nils.buzz/graphql>