

NILS HARTMANN

<https://nilshartmann.net>

# GraphQL

unter der Lupe

# Eine kritische Betrachtung

INFODAY MODERNES API-DESIGN | KÖLN, 26. SEPTEMBER 2024 | @NILSHARTMANN

# NILS HARTMANN

nils@nilshartmann.net

**Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg**  
**Java, Spring, GraphQL, React, TypeScript**



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

<https://reactbuch.de>

[HTTPS://NILSHARTMANN.NET](https://nilshartmann.net)

The screenshot shows a web browser window titled "publy" at "localhost:3000". The main content area displays three posts:

- What is the correct way to check for string equality in JavaScript?** by Nils Hartmann (02. Jan. 22). Preview: *Procis ego geminas lorem Aside solum inferius: Lorem markdownum, herbis: genu erat vestra: sub Pel...*
- What is the best java image processing library/approach?** by Vita Treutel (01. Jan. 22). Preview: *Iam ante sanguine Procne vina quidque numina: Lorem markdownum certamine viam. Molibus quam, in cum ...*
- How to customize GraphQL query validation error** by Brad Carroll (31. Dez. 21).

To the right, a sidebar lists topics with associated posts:

- #string**
  - What is the correct way to check for string equality in JavaScript?  
16 comments
  - How to convert Java String into byte[]?  
0 comments
  - How to split a String by space  
5 comments
  - How can I check if a single character appears in a string?  
0 comments
  - Why is char[] preferred over String for passwords?  
6 comments
- #java**
  - What is the best java image processing library/approach?  
0 comments
  - How to customize GraphQL query validation  
How to customize GraphQL query validation error

# Ein Beispiel...

## User

- username

## Story

- title
- excerpt

The screenshot shows a web browser window for the 'Publy' platform at localhost:3000. The interface includes a header with the Publy logo and a 'LOGIN' button. Below the header, there are three main story cards, each with a user profile picture, name, date, title, and a truncated excerpt. A red dashed box highlights the first story card.

**User Story 1 (highlighted):**  
Nils Hartmann  
02. Jan. 22  
**Title:** What is the correct way to check for string equality in JavaScript?  
**Excerpt:** *Procis ego geminas dolorem Aside solum inferius: Lorem markdownum, herbis: genu erat vestra: sub Pel...*

**User Story 2:**  
Vita Treutel  
01. Jan. 22  
**Title:** What is the best java image processing library/approach?  
**Excerpt:** *Iam ante sanguine Procne vina quidque numina: Lorem markdownum certamine viam. Molibus quam, in cum ...*

**User Story 3:**  
Brad Carroll  
31. Dez. 21  
**Title:** How to customize GraphQL query validation error

**Sidebar (Left):** #string

- What is the correct way to check for string equality in JavaScript?  
16 comments
- How to convert Java String into byte[]?  
0 comments
- How to split a String by space  
5 comments
- How can I check if a single character appears in a string?  
0 comments
- Why is char[] preferred over String for passwords?  
6 comments

**Sidebar (Right):** #java

- What is the best java image processing library/approach?  
0 comments
- How to customize GraphQL query validation

EIN BEISPIEL ...

The screenshot shows a web browser window with the URL `localhost:3000/s/100`. The page title is "Publy your social publishing network". A green "LOGIN" button is visible in the top right corner. On the left, there's a sidebar with a user profile picture and the name "Nils Hartmann" followed by "Posted on 02. Jan. 22". The main content area features a large orange header: "What is the correct way to check for string equality in JavaScript?". Below it are two small orange tags: "#javascript" and "#string". The story content is a placeholder text: "Procis ego geminas dolorem Aside solum inferius". A larger block of Latin placeholder text follows. At the bottom, there's a section titled "16 Comments" with two visible comments. The first comment is from "Ryleigh Herman" on "06. Nov. 23" with the text "Cool post, thx a ton". The second comment is from "Brad Carroll" on "18. Okt. 23" with the text "Boring content, disappointed!". Red dashed boxes highlight the "Story" section, the "User" profile, and the "Comment" section.

**Story**

- title
- tags
- body

**User**

- username
- bio
- location

**Comment**

- writtenBy
- content

EIN BEISPIEL ...

## User

- username
- bio
- location
- joined
- contact

The screenshot shows a web browser window for the 'publy' website at `localhost:3000/u/1`. The page features a header with the logo 'Publy' and the tagline 'your social publishing network'. A green 'LOGIN' button is visible in the top right corner.

The main content area displays a user profile for 'Nils Hartmann'. The profile includes a photo of a man with long hair, his name 'Nils Hartmann', his bio 'Software-Developer from Hamburg', his location 'Hamburg', the date he joined 'joined 01. Mai 19', and his email 'kontakt@nilshartmann.net'. A red dashed box highlights the bio and location information.

Below the profile, there are two sections: 'Currently Learning' and 'Recent stories'. The 'Currently Learning' section contains a card for 'How to teach GraphQL'. The 'Recent stories' section contains cards for 'What is the correct way to check for string equality in JavaScript?' (posted on 02. Jan., 22 views), 'Find object by id in an array of JavaScript objects' (posted on 30. Dez. 21), and 'Why use getters and setters/accessors?' (posted on 15. Dez. 21). A red dashed box highlights the first story card.

On the left side of the page, there are three sections: 'Story' (with a 'title' entry), 'Comment' (with a 'content' entry), and a summary section showing '5 stories written' and '8 comments written'. A red dashed box highlights the 'Comment' section.

EIN BEISPIEL ...

# EINE API FÜR DIE BEISPIEL-ANWENDUNG

## Ansatz 1: Backend bestimmt Aussehen der Endpunkte / Daten

/api/story

Story
title
excerpt
body
comments
writtenBy

/api/comment

Comment
id
content
writtenBy

/api/user

User
id
fullname
location
bio

# EINE API FÜR DIE BEISPIEL-ANWENDUNG

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

/api/story

Story
title
excerpt
body
comments
writtenBy

/api/comment

Comment
id
content
writtenBy

/api/user

User
id
fullname
location
bio

## EINE API FÜR DIE BEISPIEL-ANWENDUNG

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

/api/feed

title

excerpt

username

/api/story-view

title

body

tags

username

userLocation

userBio

/api/user-detail

username

userBio

userLocation

commentContent

storyTitle

## EINE API FÜR DIE BEISPIEL-ANWENDUNG

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

/api/feed

title

excerpt

username

/api/story-view

title

body

tags

username

userLocation

userBio

/api/user-detail

username

userBio

userLocation

commentContent

storyTitle

## EINE API FÜR DIE BEISPIEL-ANWENDUNG

Ansatz 3: GraphQL...

# EINE API FÜR DIE BEISPIEL-ANWENDUNG

## Ansatz 3: GraphQL...

Aus Ansatz 1: Server bestimmt, wie Datenmodell aussieht



# EINE API FÜR DIE BEISPIEL-ANWENDUNG

## Ansatz 3: GraphQL...

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

```
{ stories { title excerpt { user { fullname } } } }
```



## TODO:

- Ende-zu-Ende Typsicherheit (eventuell in der Demo)
- Gute IDE Unterstützung
- Bei den Aussagen: "Wann soll ich GraphQL denn nun verwenden"

# GraphQL

The screenshot shows a GraphQL interface running on `localhost:9000/graphiql?operationName=BeerAppQuery&query=query`. The left pane displays a GraphQL query for a `BeerAppQuery`:

```
1+ query BeerAppQuery {  
2+   beers {  
3+     id  
4+     name  
5+     price  
6+     ratings {  
7+       id  
8+       beerId  
9+       author  
10+      comment  
11+    }  
12+  }  
13+}  
14+  
15+  |  
16+  beers  
17+ }  
  beer  
  ratings  
  ping  
  __schema  
  __type  
  Returns all beers in our store
```

The right pane shows the results of the query:

```
{  
  "data": {  
    "beers": [  
      {  
        "id": "B1",  
        "name": "Barfüßer",  
        "price": "3,88 EUR",  
        "ratings": [  
          {  
            "id": "R1",  
            "beerId": "B1",  
            "author": "Moldemar Vasu",  
            "comment": "Exceptional!"  
          },  
          {  
            "id": "R7",  
            "beerId": "B1",  
            "author": "Madhukar Kareem",  
            "comment": "Awwesome!"  
          },  
          {  
            "id": "R14",  
            "beerId": "B1",  
            "author": "Eainly Davis",  
            "comment": "Off-putting buttery nose, laced  
with a touch of caramel and hamster cage."  
          }  
        ]  
      },  
      {  
        "id": "B2",  
        "name": "Frøydenlund",  
        "price": "150 NOK",  
        "ratings": [  
          {  
            "id": "R2",  
            "beerId": "B2",  
            "author": "Andrea Gouyen",  
            "comment": "Very good!"  
          }  
        ]  
      }  
    ]  
  }  
}  
No Description
```

Below the results, there are descriptions for each field:

- `beers: [Beer!]!`: Returns all beers in our store
- `beer(beerId: String): Beer`: Returns the Beer with the specified Id
- `ratings: [Rating!]!`: All ratings stored in our system
- `ping: ProcessInfo!`: Returns health information about the running process

# Demo

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

- <https://graphql.org>

# GraphQL

**Die GraphQL API muss in einem *Schema* beschrieben werden**

- Das Schema legt fest, welche *Types* und *Fields* es gibt
- **Schema Definition Language (SDL)**

## GRAPHQL SCHEMA

Schema Definition per SDL  
Objekte

```
type Comment {  
}  
          ^ Type (Object)
```

# GRAPHQL SCHEMA

## Schema Definition per SDL Objekte und Felder

```
type Comment {  
    id  
    comment  
    approved  
    likes  
}
```

Type (Object)

Felder

The diagram illustrates a GraphQL schema definition. On the left, a code snippet defines a 'Comment' type with four fields: 'id', 'comment', 'approved', and 'likes'. On the right, the word 'Type (Object)' is positioned above the opening brace of the type definition. The word 'Felder' is positioned below the closing brace. Four dashed arrows originate from the field names 'id', 'comment', 'approved', and 'likes', each pointing towards the word 'Felder'.

# GRAPHQL SCHEMA

## Schema Definition per SDL Objekte und Felder

```
type Comment {  
    id: ID!  
    comment: String! ----- Return Type (non-nullable)  
    approved: Boolean  
    likes: Int ----- Return Type (nullable)  
}
```

# GRAPHQL SCHEMA

## Schema Definition per SDL

### Objekte und Felder

```
type Comment {  
    id: ID!  
    comment: String! ----- Return Type (non-nullable)  
    approved: Boolean  
    likes: Int ----- Return Type (nullable)  
}
```

👉 **Felder sind konzeptionell Funktionen, die Werte zurückliefern**

# GRAPHQL SCHEMA

## Schema Definition per SDL

### Datentypen

```
type Comment {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    likes: Int  
}
```

Skalare Datentypen (Blätter)

# GRAPHQL SCHEMA

## Schema Definition per SDL

### Datentypen

```
type Comment {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    likes: Int  
}  
  
enum PublishingState { draft, in_review, published }
```

```
type Story {  
    title: String!  
    body: String!  
    state: PublishingState!  
}
```

Aufzählungstypen

# GRAPHQL SCHEMA

## Schema Definition per SDL Datentypen

```
type Comment {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    likes: Int  
}  
  
enum PublishingState { draft, in_review, published }  
  
scalar DateTime  
  
type Story {  
    title: String!  
    body: String!  
    state: PublishingState!  
    created: DateTime  
}
```

**Eigene Skalare**

# GRAPHQL SCHEMA

## Schema Definition per SDL Datentypen

```
type Comment {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    likes: Int  
}  
  
enum PublishingState { draft, in_review, published }  
  
scalar DateTime  
  
type Story {  
    title: String!  
    body: String!  
    state: PublishingState!  
    created: DateTime  
    comments: [Comment!] }  
}
```

Listen

# GRAPHQL SCHEMA

## Schema Definition per SDL Argumente

```
type Comment {  
    id: ID!  
    comment: String!  
    approved: Boolean  
    likes: Int  
}  
  
enum PublishingState { draft, in_review, published }  
  
scalar DateTime  
  
type Story {  
    title: String!  
    body: String!  
    state: PublishingState!  
    created: DateTime  
    comments: [Comment!]!  
    excerpt(maxLength: Int! = 120): String!  
}
```

Argumente

# GRAPHQL SCHEMA

## Schema Definition per SDL Dokumentation

```
"""
A `Story` is the main object in our service.

After creating the `Story` it must be **reviewed** before it
can be **published**.

"""

type Story {
    title: String!
    body: String!
    comments: [Comment!]!

    "Returns an excerpt of this story."
    excerpt(
        "Specifies the maximum number of chars for the excerpt"
        maxLength: Int! = 120
    ): String!
}
```

# GRAPHQL SCHEMA

## Root-Typen

Einstiegspunkte in den Graphen

Root-Type -----  
("Query")

```
type Query {  
    stories(page: Int, size: Int): [Story!]!  
    storyById(storyId: ID!): Story  
}
```

# GRAPHQL SCHEMA

## Root-Typen

Einstiegspunkte in den Graphen

Root-Type  
("Query")

```
type Query {  
    stories(page: Int, size: Int): [Story!]!  
    storyById(storyId: ID!): Story  
}
```

```
input NewStory { title: String! body: String! }
```

Root-Type  
("Mutation")

```
type Mutation {  
    addStory(newStory: NewStory!): Story  
}
```

# GRAPHQL SCHEMA

## Root-Typen

### Einstiegspunkte in den Graphen

Root-Type -----  
("Query")

```
type Query {  
    stories(page: Int, size: Int): [Story!]!  
    storyById(storyId: ID!): Story  
}
```

```
input NewStory { title: String! body: String! }
```

Root-Type -----  
("Mutation")

```
type Mutation {  
    addStory(newStory: NewStory!): Story  
}
```

Root-Type -----  
("Subscription")

```
type Subscription {  
    onNewComment(storyId: ID): Comment!  
}
```

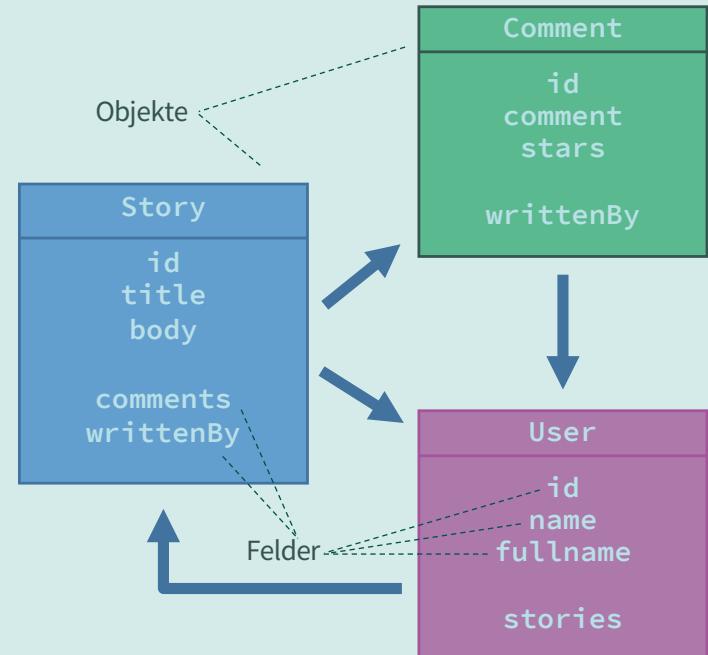
### Das Schema

- kann per GraphQL abgefragt werden ("Introspection Query")
- Basis für Tooling (Code-Generatoren, IDE-Support etc.)

Die  
Query  
Sprache

# QUERY LANGUAGE

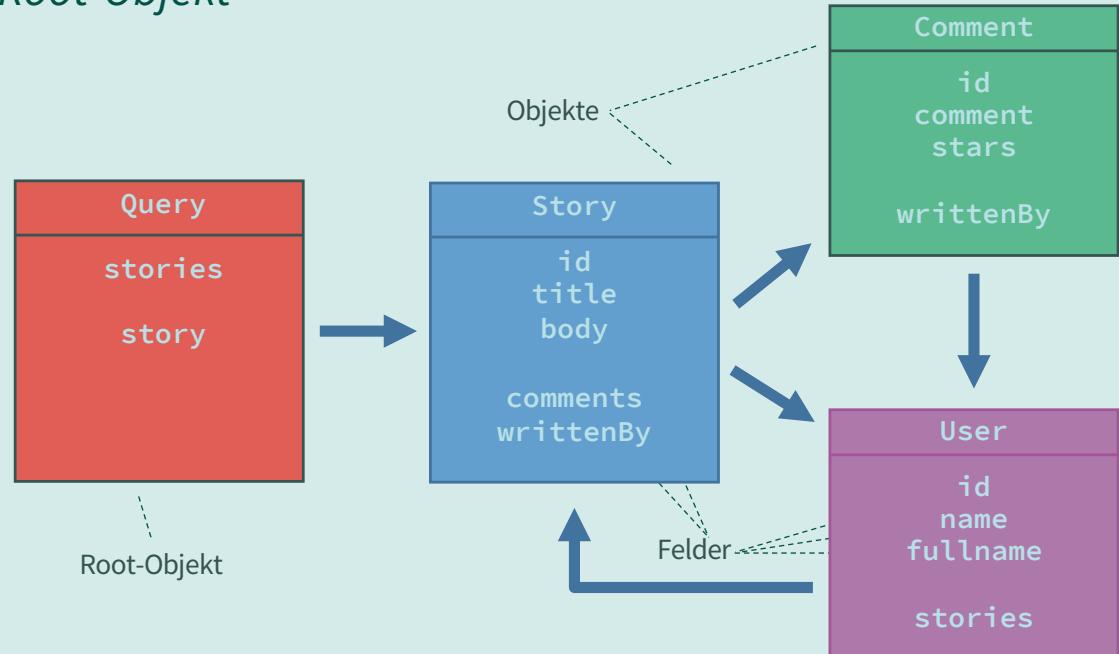
Über eine GraphQL API werden *Objekte mit Feldern* bereitgestellt



# QUERY LANGUAGE

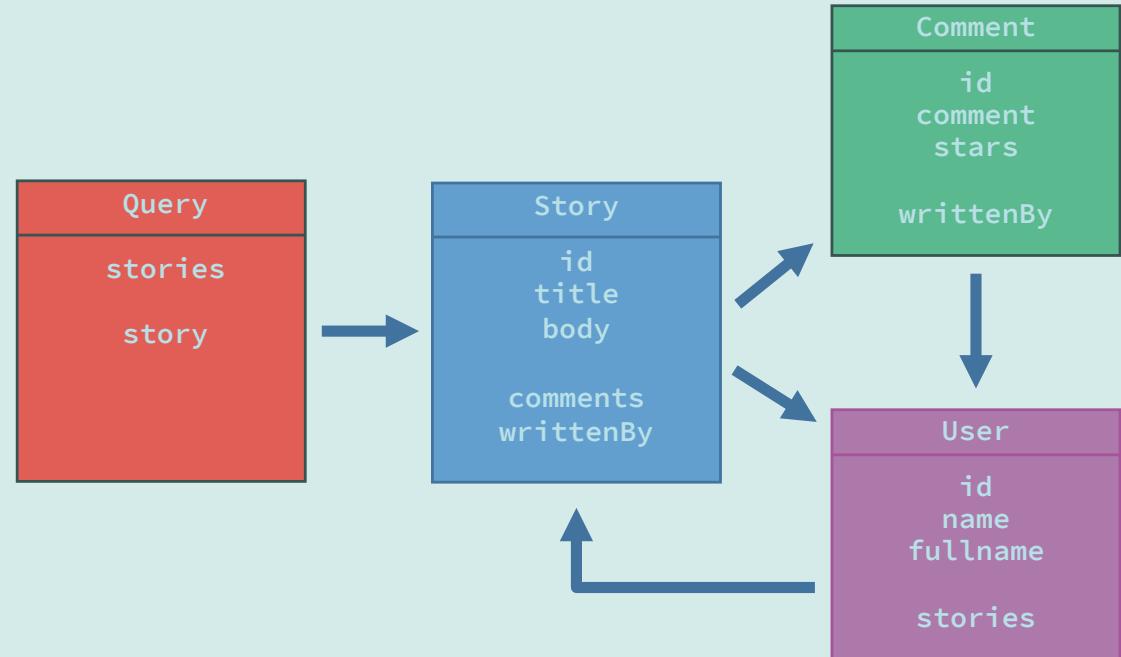
Über eine GraphQL API werden *Objekte mit Feldern* bereitgestellt

- Der Graph beginnt bei einem *Root-Objekt*



# QUERY LANGUAGE

Mit der *Query-Sprache* werden aus dem Graphen **Felder** ausgewählt

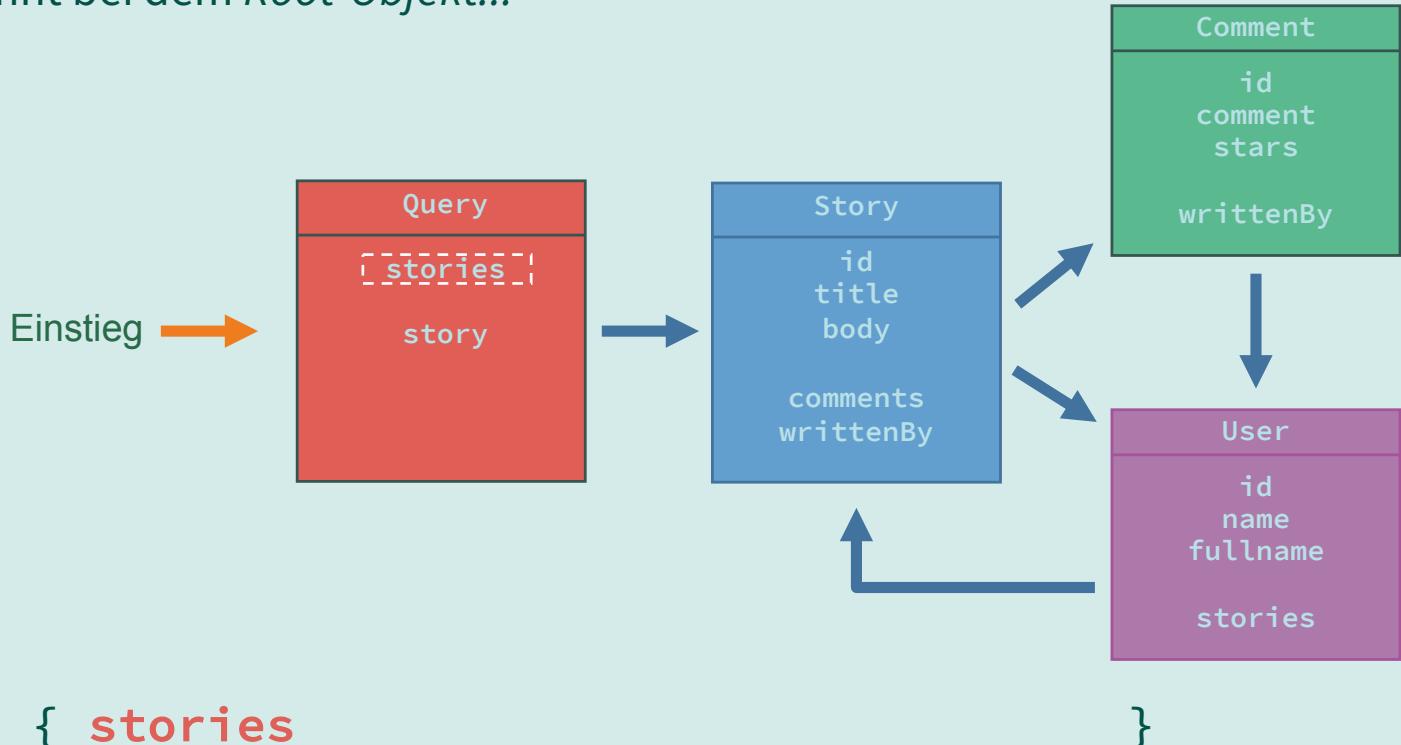


```
query { } 
```

# QUERY LANGUAGE

Mit der *Query-Sprache* werden aus dem Graphen *Felder* ausgewählt

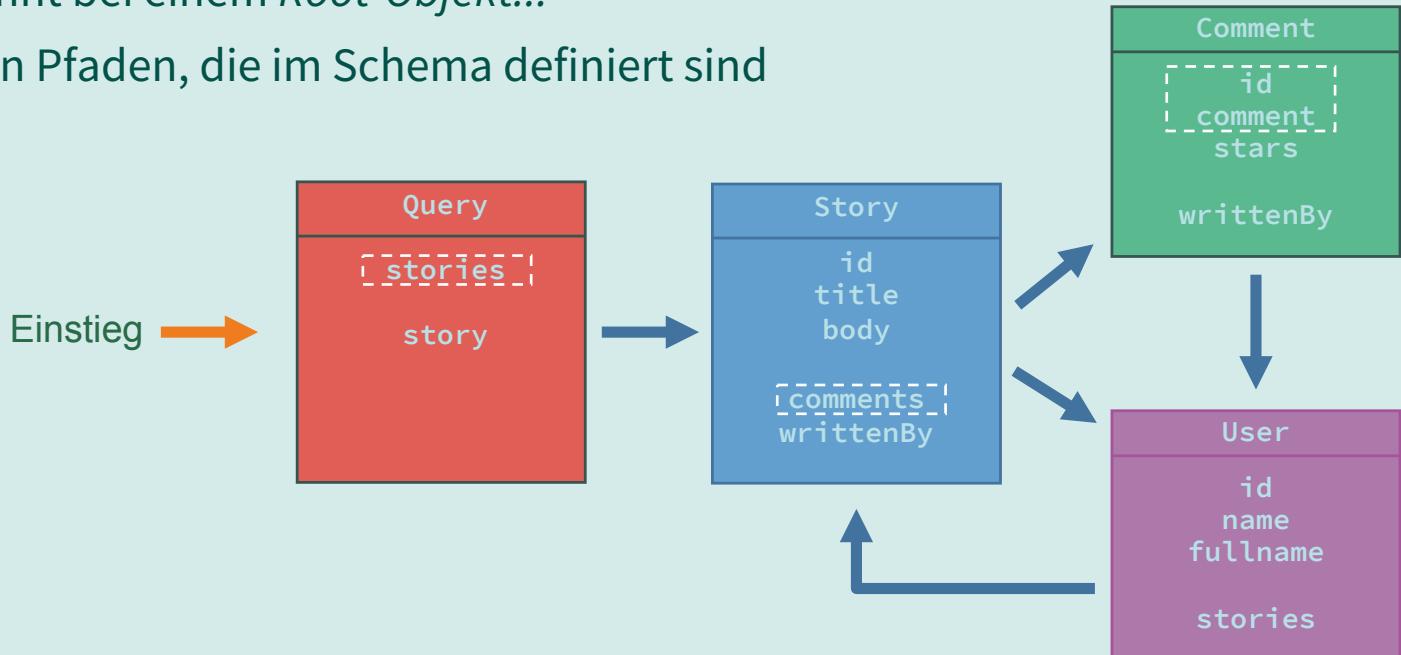
- Der Query beginnt bei dem *Root-Objekt*...



# QUERY LANGUAGE

Mit der *Query-Sprache* werden aus dem Graphen *Felder* ausgewählt

- Der Query beginnt bei einem *Root-Objekt*...
- ...folgt dann den Pfaden, die im Schema definiert sind

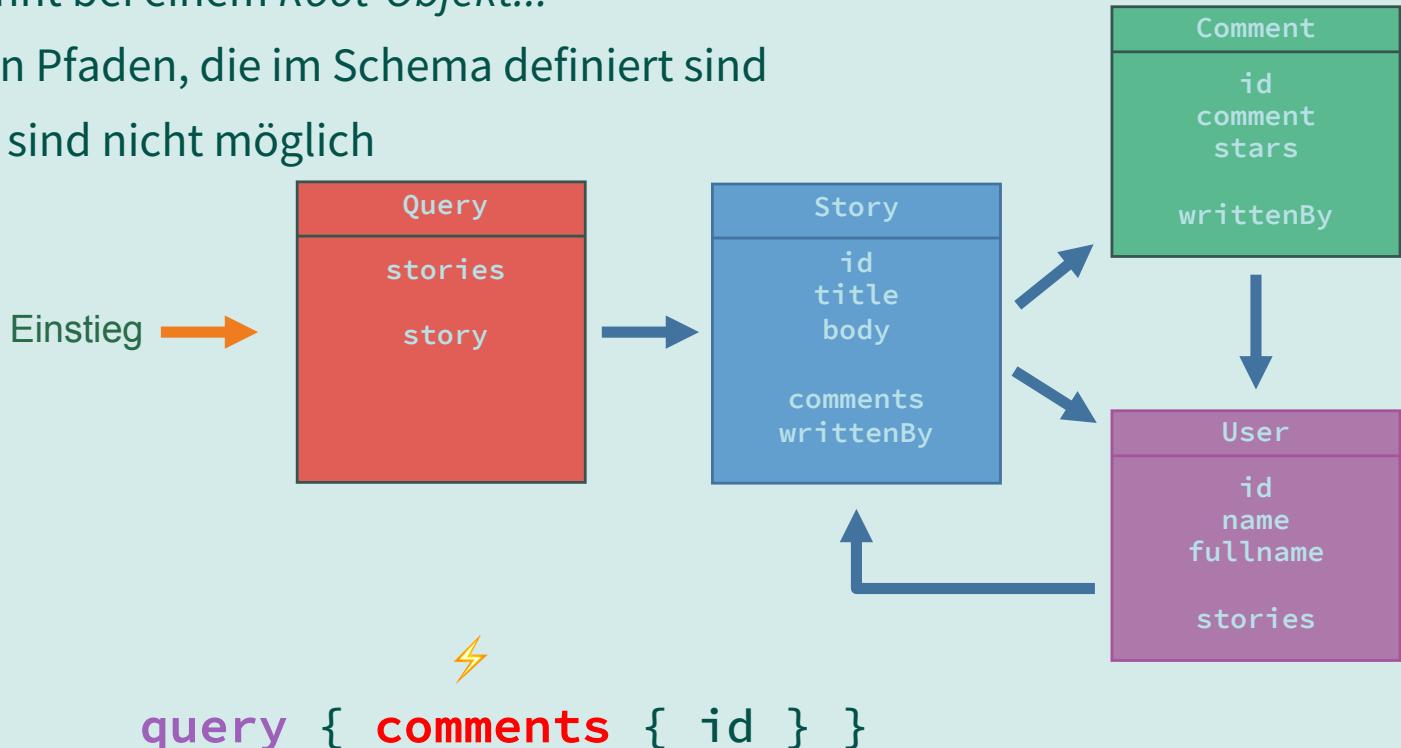


```
query { stories { story } }
```

# QUERY LANGUAGE

Mit der *Query-Sprache* werden aus dem Graphen *Felder* ausgewählt

- Der Query beginnt bei einem *Root-Objekt*...
- ...folgt dann den Pfaden, die im Schema definiert sind
- Andere "Joins" sind nicht möglich



The screenshot shows a GraphQL interface with the following components:

- Left Panel (GraphiQL):** Displays a GraphQL query for a "BeerAppQuery". The query includes fields for "beers", "beer", "ratings", "ping", and introspection fields like "\_\_schema" and "\_\_type". A tooltip below the "beers" field says: "Returns all beers in our store".
- Middle Panel (Results):** Shows the JSON response from the query. It includes a "data" object with a "beers" array containing several beer objects. One beer object is highlighted with a blue background.
- Right Panel (Documentation):** Contains four entries:
  - beers: [Beer]!**: Returns all beers in our store.
  - beer(beerId: String): Beer**: Returns the Beer with the specified Id.
  - ratings: [Rating]!**: All ratings stored in our system.
  - ping: ProcessInfo!**: Returns health information about the running process.



Ein Query



Response



Dokumentation



Netzwerkverkehr (evtl.)

# Demo

## QUERIES AUSFÜHREN

**Die Spec schreibt nicht vor, wie Queries auszuführen sind**

- Typisch: über HTTP API
- Request ist dann (meist) ein HTTP Post Request
- Response ein JSON-Objekt mit den gelesenen Daten
- HTTP-Status-Codes spielen fast keine Rolle
- Transportschicht ist eher Implementierungsdetail

# Implementierung

### Implementieren von GraphQL APIs

- In der Regel muss man die Logik selbst implementieren
- Es gibt Frameworks für fast alle Programmiersprachen

## IMPLEMENTIERUNG

**Die Spec schreibt nicht vor, wie eine GraphQL Implementierung aussehen muss**

- Es gibt aber deutliche Hinweise
- Fast alle Frameworks arbeiten danach

### "Felder sind konzeptionell Funktionen"

- Ein abgefragtes Feld entspricht konzeptionell einem "Funktionsaufruf"
- Ein GraphQL-Backend muss für jedes Feld eine Funktion bereitstellen
- Die Funktionen im Backend werden **Resolver**-Funktionen genannt

## Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt
8. Die Antwort wird an den Client gesendet

👉 Die Implementierung der Resolver-Funktionen ist unsere Aufgabe

## Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt
8. Die Antwort wird an den Client gesendet

👉 Die Implementierung der Resolver-Funktionen ist unsere Aufgabe

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt
8. Die Antwort wird an den Client gesendet

👉 Die Implementierung der Resolver-Funktionen ist unsere Aufgabe

## IMPLEMENTIERUNG

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen

## IMPLEMENTIERUNG

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt
8. Die Antwort wird an den Client gesendet

### Ablauf (stark vereinfacht)

1. HTTP Request mit Anfrage kommt an
2. Framework validiert das Dokument mit der Anfrage (Syntax und Schema)
3. Ungültige Dokumente werden abgewiesen
4. Für alle Felder werden die entsprechenden Resolver-Funktionen aufgerufen
5. Die Resolver-Funktionen ermitteln die Daten für ein Feld
6. Das Framework validiert die ermittelten Daten (Schema)
7. Das Framework erzeugt das Antwort-Objekt
8. Die Antwort wird an den Client gesendet

👉 Die Implementierung der Resolver-Funktionen ist unsere Aufgabe

# IMPLEMENTIERUNG

## Beispiel

Java mit Spring Boot

```
query {  
  story(storyId: "1")  
  {  
    writtenBy  
    {  
      name  
    }  
  }  
}
```

```
@Controller  
public class StoryController {  
  
  private final StoryRepository storyRepository;  
  private final UserMicroService userMicroService;  
  
  public StoryController(StoryRepository storyRepository,  
                        UserMicroService userMicroService) {...}  
  
  @QueryMapping  
  public Story story(@Argument String storyId) {  
    return storyRepository.findById(storyId);  
  }  
  
  @SchemaMapping  
  public User writtenBy(Story story) {  
    return userMicroService.fetchUser(story.getWrittenBy().getUserId());  
  }  
}
```

# Aussagen über GraphQL

**"GraphQL ist nur für JavaScript"**

Das ist falsch

- Es gibt GraphQL-Frameworks für fast alle Programmiersprachen

***"GraphQL ist nur für JavaScript"***

Das ist falsch

- Es gibt GraphQL-Frameworks für fast alle Programmiersprachen
- Erste GraphQL-Implementierung war für PHP (!)

**"GraphQL ist nur für JavaScript"**

Das ist falsch

- Es gibt GraphQL-Frameworks für fast alle Programmiersprachen
- Erste GraphQL-Implementierung war für PHP (!)
- Richtig ist, dass die Clients oft in JavaScript geschrieben sind (z.B. mit React)

**"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"**

REST API sind konzeptionell relativ "einfach": Jeder Request liefert *eine* Ressource

- Was genau abgefragt wird, ist bereits zur *Entwicklungszeit* bekannt
- Wird vorgegeben durch das Backend

### **"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"**

REST API sind konzeptionell relativ "einfach": Jeder Request liefert *eine* Ressource

- Was genau abgefragt wird, ist bereits zur *Entwicklungszeit* bekannt
- Wird vorgegeben durch das Backend

GraphQL kann mehrere Objekte liefern

- Was genau abgefragt wird, ist erst zur *Laufzeit* bekannt
- Wird vorgegeben vom Client

### **"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"**

REST API sind konzeptionell relativ "einfach": Jeder Request liefert *eine* Ressource

- Was genau abgefragt wird, ist bereits zur *Entwicklungszeit* bekannt
- Wird vorgegeben durch das Backend

GraphQL kann mehrere Objekte liefern

- Was genau abgefragt wird, ist erst zur *Laufzeit* bekannt
- Wird vorgegeben vom Client

👉 Optimierungen sind dadurch schwieriger als in REST APIs

***"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"***

Aufwand der Entwicklung hängt an mehreren Faktoren:

- Größe und Komplexität des Schemas
- Anzahl und Art der Datenquellen
- Wie gut passen die vorhandenen Daten zum Schema

## "GraphQL APIs sind komplexer in der Entwicklung als REST APIs"

### Größe und Komplexität des Schemas

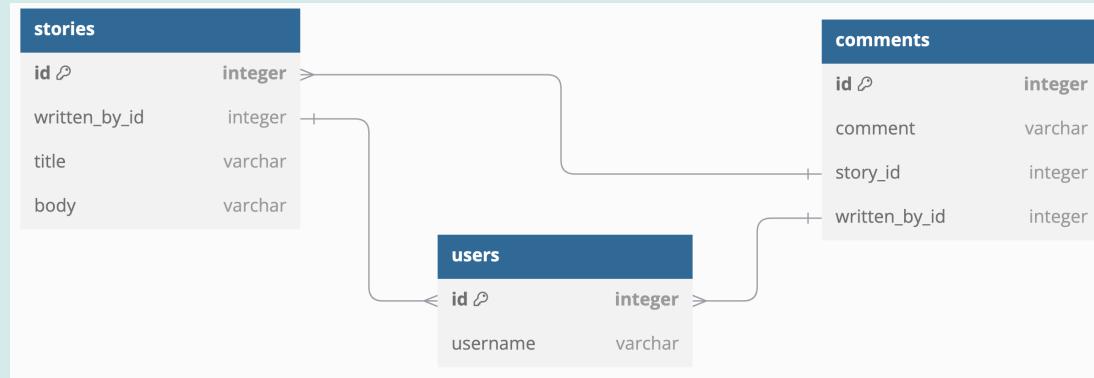
- Extrem Beispiel: "GraphQL API, REST-Style"

```
type Comment {           type Story {           type User {           type Query {  
    id: ID!             id: ID!  
    comment: String!       title: String!  
    approved: Boolean      body: String!  
    likes: Int            writtenBy: ID!  
  }                         comments: [ID!]!  
}  
}  
  
type Story {           type User {           type Query {  
    id: ID!             id: ID!  
    name: String!         name: String!  
    roles:[String!]!       roles:[String!]!  
  }                         }  
}  
  
type User {           type Query {  
    id: ID!             id: ID!  
    name: String!         name: String!  
    roles:[String!]!       roles:[String!]!  
  }                         }  
  
type Query {  
  storyById(id: ID!): Story  
  commentById(id: ID!): Comment  
  userById(id: ID!): User  
}
```

**"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"**

## Probleme der Optimierung #1

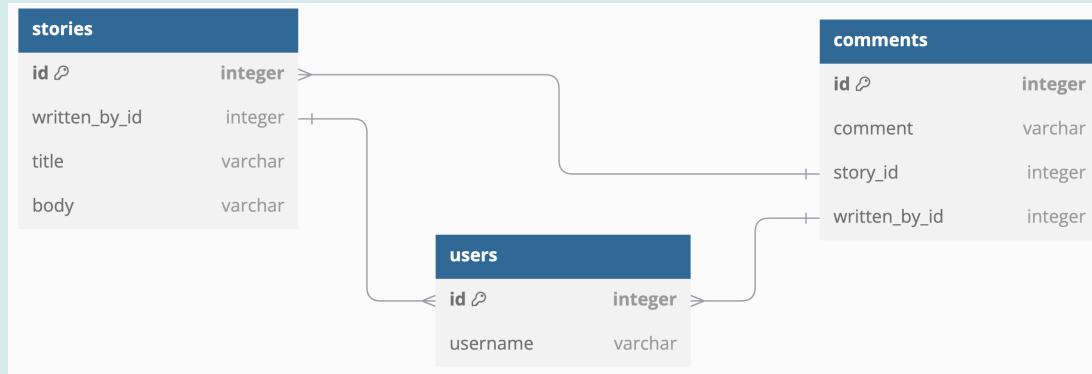
- Stories, User und Kommentare könnten mit einem SQL Query gelesen werden



**"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"**

## Probleme der Optimierung #1

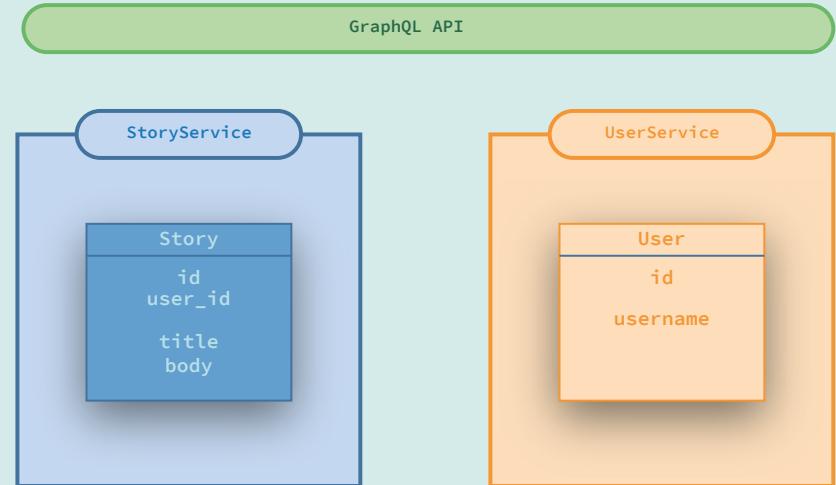
- Stories, User und Kommentare könnten mit einem SQL Query gelesen werden
- Wir müssen das aber von Query zu Query entscheiden
  - Eventuell sind das dann zu viele Daten



**"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"**

## Probleme der Optimierung #2

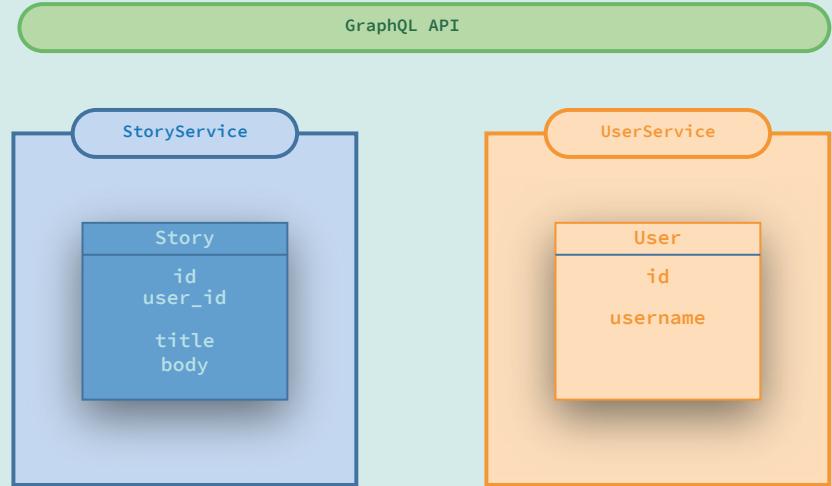
- Stories und User kommen aus unterschiedlichen Services



**"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"**

## Probleme der Optimierung #2

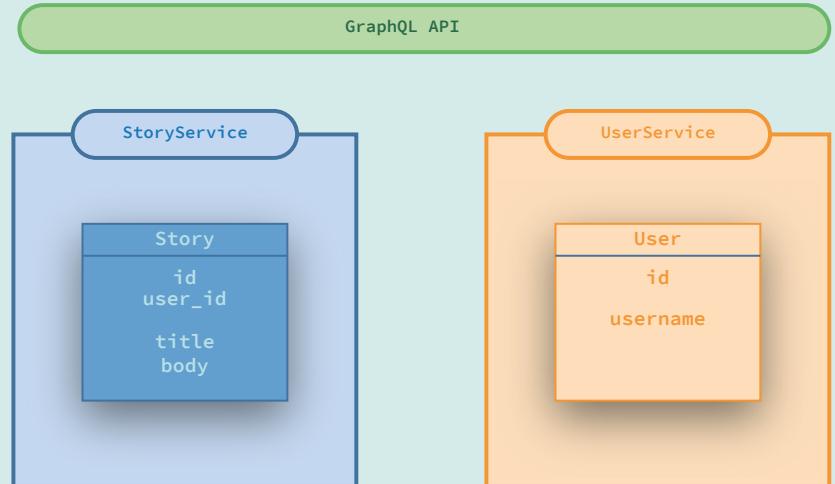
- Stories und User kommen aus unterschiedlichen Services
- Hier kann es zu n+1-Problemen kommen
  - 1 Request liefert n Stories zu der jeweils 1 User abgefragt werden muss



## "GraphQL APIs sind komplexer in der Entwicklung als REST APIs"

### Probleme der Optimierung #2

- Stories und User kommen aus unterschiedlichen Services
- Hier kann es zu n+1-Problemen kommen
  - 1 Request liefert n Stories zu der jeweils 1 User abgefragt werden muss
- Frameworks helfen hier mit dem *DataLoader*-Pattern



**"GraphQL APIs sind komplexer in der Entwicklung als REST APIs"**



Cal Irvine @Cal\_Irvine

...

I feel this way about many arguments against GraphQL. “X is hard in GraphQL”, drop the “in GraphQL” and the statement remains equally true.

[Post übersetzen](#)

11:39 nachm. · 13. März 2024 · 238 Mal angezeigt

[https://x.com/Cal\\_Irvine/status/1768044262124323175](https://x.com/Cal_Irvine/status/1768044262124323175)

**"GraphQL APIs sind für Faule, die kein Bock auf API Design haben"**

Das ist falsch

- Suggeriert wird, dass man mit GraphQL einfach alle Daten strukturlos zur Verfügung stellt
- Schema-Design ist aber genauso wie in anderen API Technologien
- Wie die API aussieht, bestimmen wir und nicht GraphQL
- Man sich wie bei allem anderen Mühe geben... oder eben nicht

**"Mit GraphQL gibt es kein Caching"**

Das stimmt so absolut nicht

- Aussage bezieht sich in der Regel auf GraphQL Anfragen per HTTP

**"Mit GraphQL gibt es kein Caching"**

## Probleme beim Caching #1

- HTTP Post Requests nicht cachebar
  - Theoretisch ginge aber auch GET

### **"Mit GraphQL gibt es kein Caching"**

#### Probleme beim Caching #2

- Wenn mehrere Objekte abgefragt werden, muss die kürzeste Cache-Dauer gewinnen
- Beispiel: Story und Benutzer
  - Benutzer ist wahrscheinlich "stabil", gut cachebar
  - Story vielleicht nicht, weil sich daran Dinge ändern können (z.B. Likes)
- In REST kann jede Resource entscheiden, wie lange sie cachbar ist

### **"GraphQL spart Daten gegen über REST APIs"**

**Das ist so absolut nicht richtig**

- Richtig ist, das Netzwerk-Requests und damit Latenz gespart werden kann
- Daten werden nicht automatisch de-dupliziert oder normalisiert
  - Eine Abfrage von X Stories mit denselben Autoren liefert X mal die abgefragten Daten der Autoren
  - Das kann sparsamer sein als in REST, muss es aber nicht
- Eine "unmodified"-Antwort einer REST Ressource liefert überhaupt keine Daten

**"Das kann ich auch mit REST APIs machen..."**

## Oftmals korrekt

- Mit Search-Parametern etc. kann man oft die Anzahl der Attribute einschränken
- Ein Schema kann man zum Beispiel mit OpenAPI beschreiben
- Auch mit einer REST-API können Unterobjekte geliefert werden
- Alternativ kann mit HATEOS ein Client gewissermaßen automatisch Daten nachladen
- Aber:
  - das alles in GraphQL "eingebaut", man bekommt es aus einer Hand

**"GraphQL ist SQL für's Frontend"**

Das ist falsch

- SQL ist viel mächtiger als die GraphQL Sprache
  - Wir haben kein SORT BY, LIMIT, JOIN etc.
- Es werden keine fertigen Datenbank-Schema oder –Inhalte ungefiltert ausgeliefert
- Daten müssen gar nicht aus einer Datenbank kommen
- Was und wie ausgeliefert wird entscheidet das Schema bzw. die Resolver-Funktionen

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



# vielen Dank!

Slides: <https://graphql.schule/api-design>

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

Twitter: [@nilshartmann](https://twitter.com/nilshartmann)

