



NILS HARTMANN

<https://nilshartmann.net>



Slides

GraphQL

unter der Lupe

Eine kritische Betrachtung

NILS HARTMANN

nils@nilshartmann.net

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

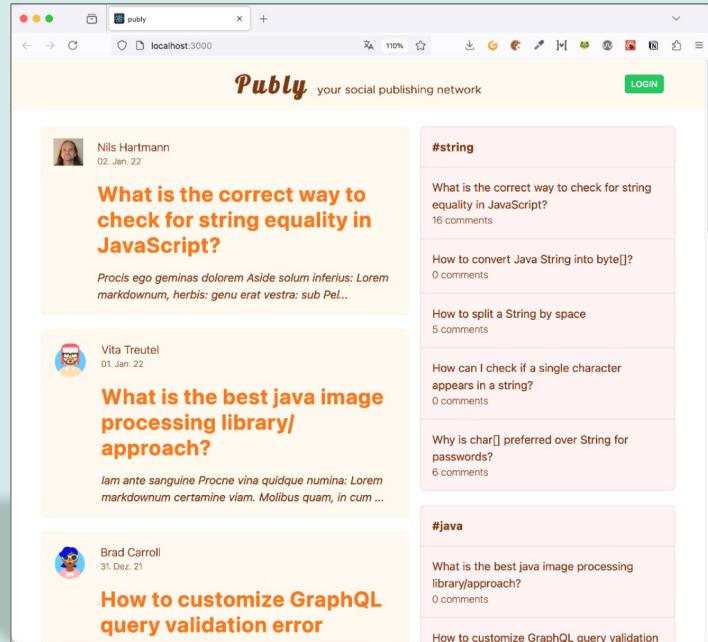
Java, Spring, GraphQL, React, TypeScript



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

<https://reactbuch.de>

HTTPS://NILSHARTMANN.NET



Ein Beispiel...

User

- username

Story

- title
- excerpt

The screenshot shows a web browser window for the 'Publy' social publishing network 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 tags: "#javascript" and "#string". The story content is a placeholder text: "Procis ego geminas dolorem Aside solum inferius". A long 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, a bio ('Software-Developer from Hamburg'), location ('Hamburg'), joining date ('joined 01. Mai 19'), and email ('kontakt@nilshartmann.net').

Below the profile, there are two sections: 'Currently Learning' and 'Recent stories'. The 'Currently Learning' section lists 'How to teach GraphQL'. The 'Recent stories' section lists several posts, each with a title, a timestamp, and a view count:

- 'What is the correct way to check for string equality in JavaScript?' - 02. Jan. 22
- 'Find object by id in an array of JavaScript objects' - 30. Dez. 21
- 'Why use getters and setters/accessors?' - 15. Dez. 21

On the left side of the main content, there are three sections: 'Story', 'Comment', and 'Comment' (repeated). The first 'Story' section has a red dashed box around it, indicating it is the current focus. It contains a link to 'How to teach GraphQL'. The other 'Story' section contains links to 'Skills' and 'Beer and GraphQL'.

The 'Comment' sections show statistics: 5 stories written and 8 comments written. The second 'Comment' section also has a red dashed box around it, indicating it is the current focus. It contains a link to 'Interview question: Check if one string is a rotation of other string' with a timestamp of 21. Dez. 21. Below this, there is some placeholder text: 'Sed laboriosam quis corporis voluptatem amet qui. Suscipit nihil enim et'.

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 } } } }
```



*"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 the structure of a GraphQL schema. On the left, a code snippet defines a 'Comment' type with four fields: 'id', 'comment', 'approved', and 'likes'. A bracket labeled 'Type (Object)' groups these definitions. On the right, a label 'Felder' is positioned, with dashed arrows pointing from each field name ('id', 'comment', 'approved', 'likes') to it, indicating that these are the fields of the 'Comment' type.

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!  
}
```

GRAPHQL SCHEMA

Das Schema

- Es gibt nur eine Version

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

GRAPHQL SCHEMA

Das Schema

- Es gibt nur eine Version
- Schema kann jederzeit erweitert werden, ohne bestehende Clients zu beeinflussen

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

Evolution

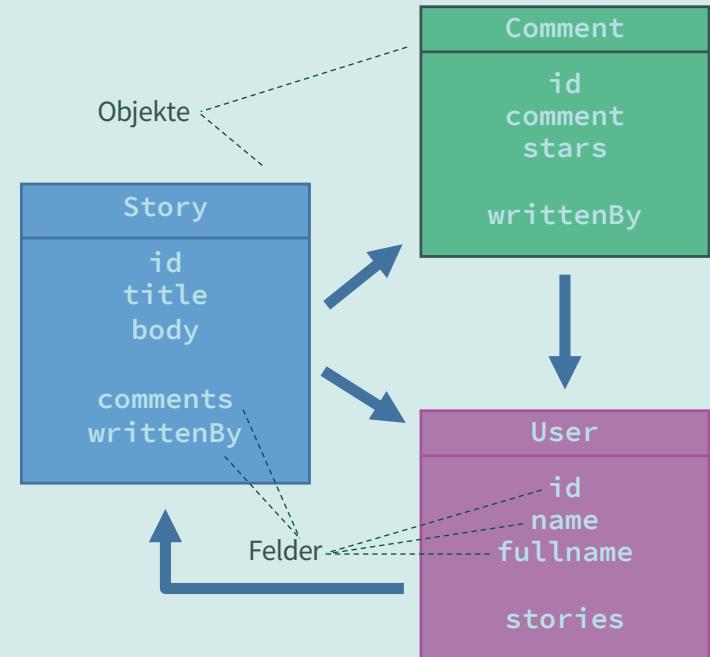
```
type Story {  
    title: String!  
    body: String!  
    state: PublishingState!  
    created: DateTime  
  
    comments: [Comment!]!  
}
```

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

Die
Query
Sprache

QUERY LANGUAGE

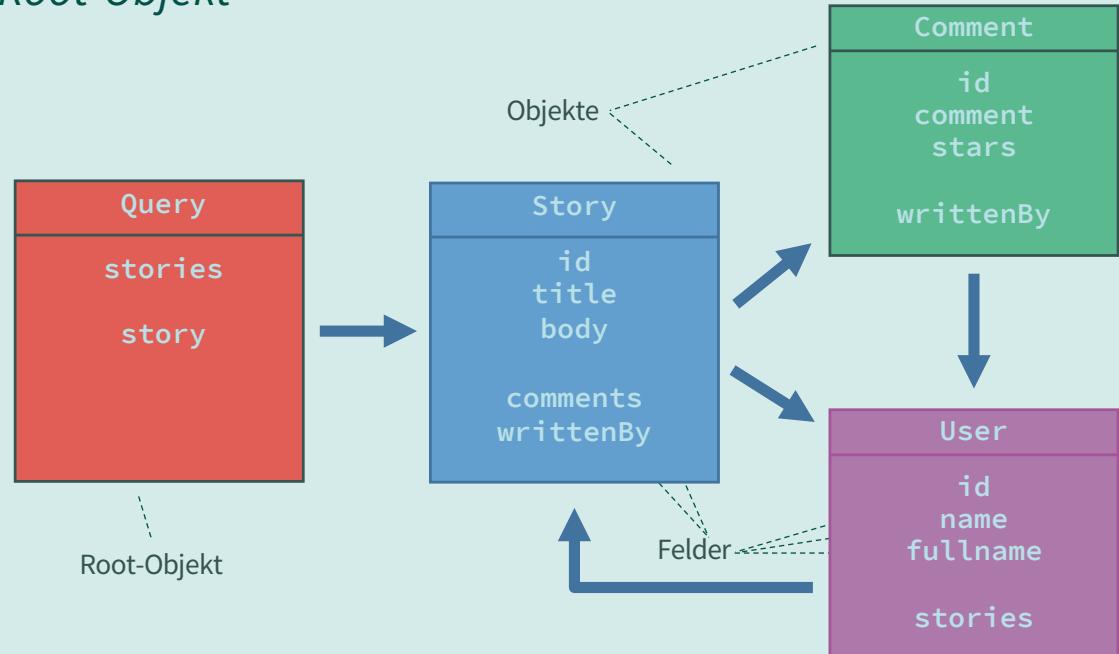
Über eine GraphQL API werden *Objekte mit Feldern* bereitgestellt



QUERY LANGUAGE

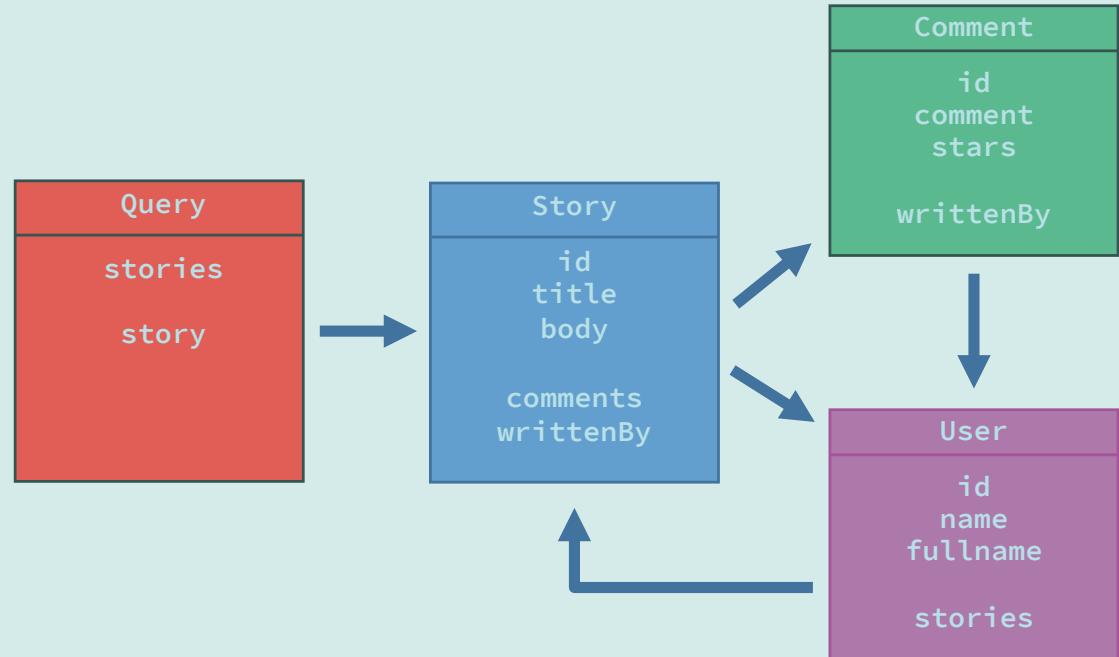
Über eine GraphQL API werden *Objekte mit Feldern* bereitgestellt

- Der Graph beginnt bei einem *Root-Objekt*



QUERY LANGUAGE

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

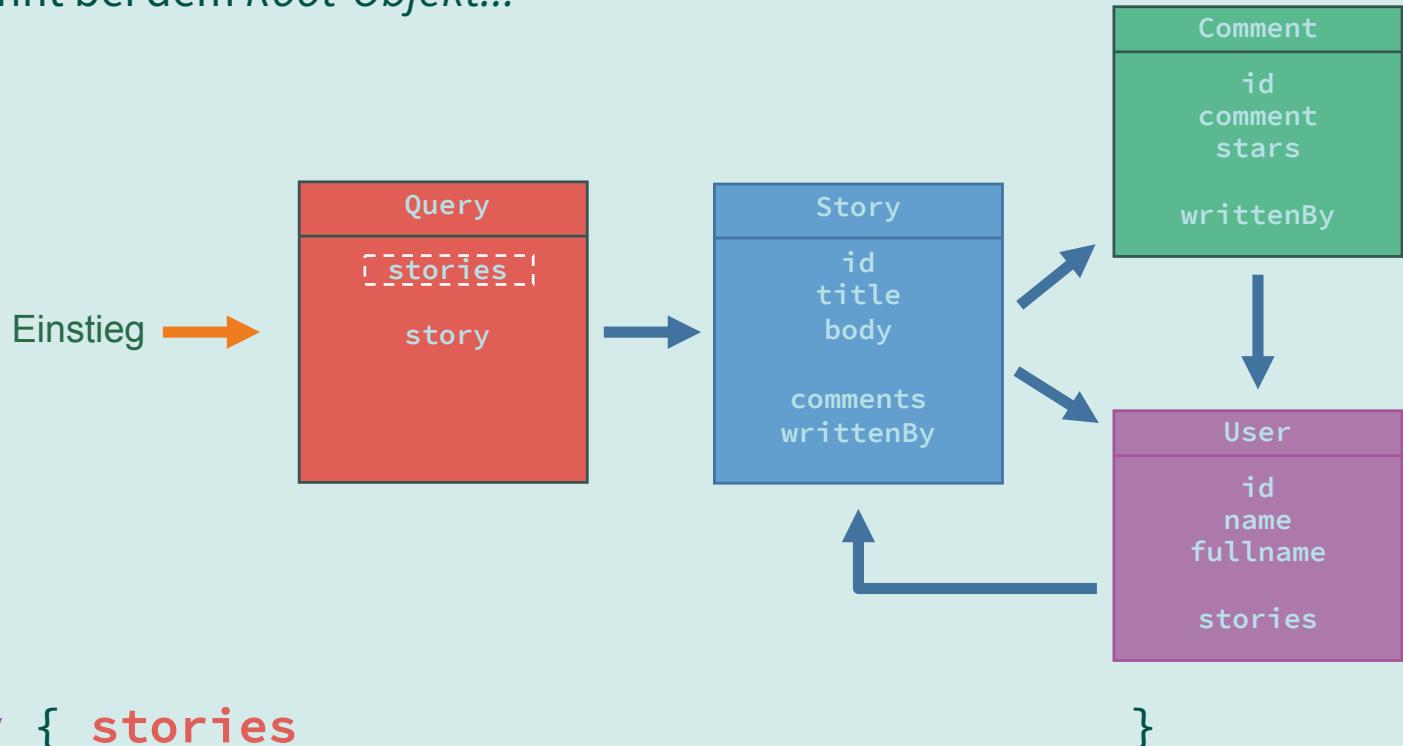


```
query { } 
```

QUERY LANGUAGE

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

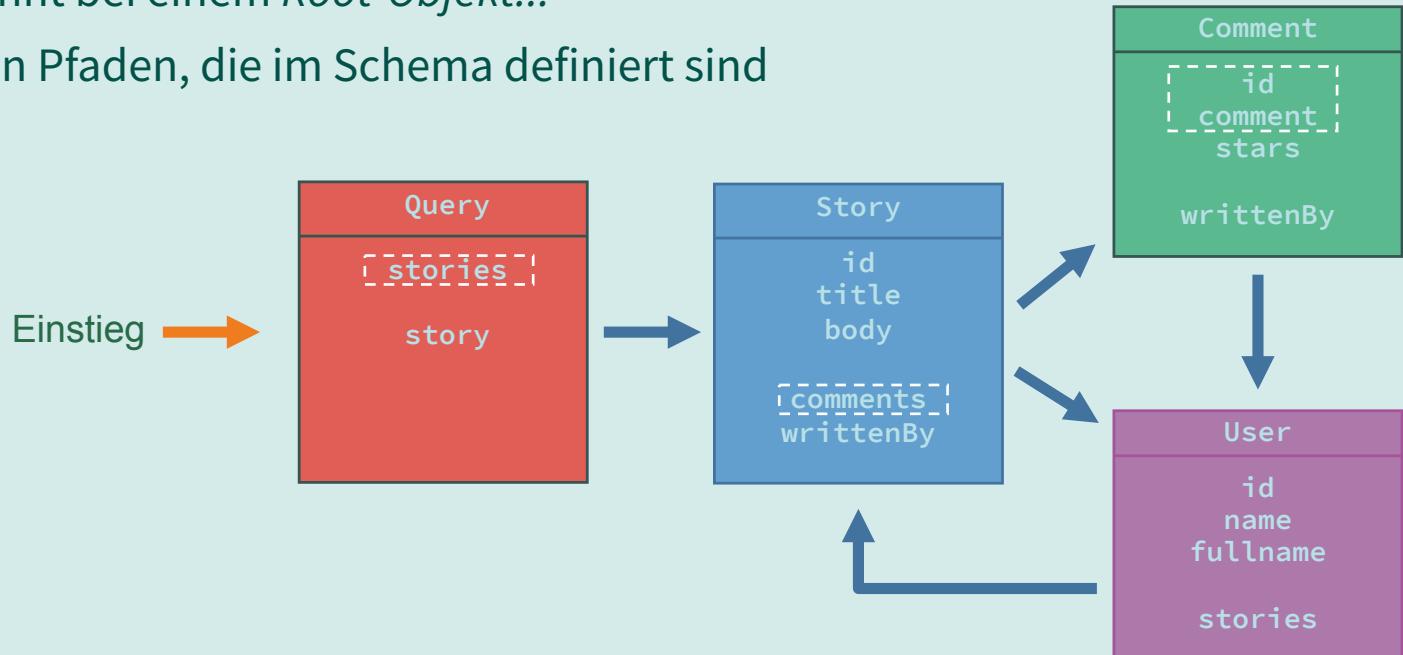
- Der Query beginnt bei dem *Root-Objekt*...



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

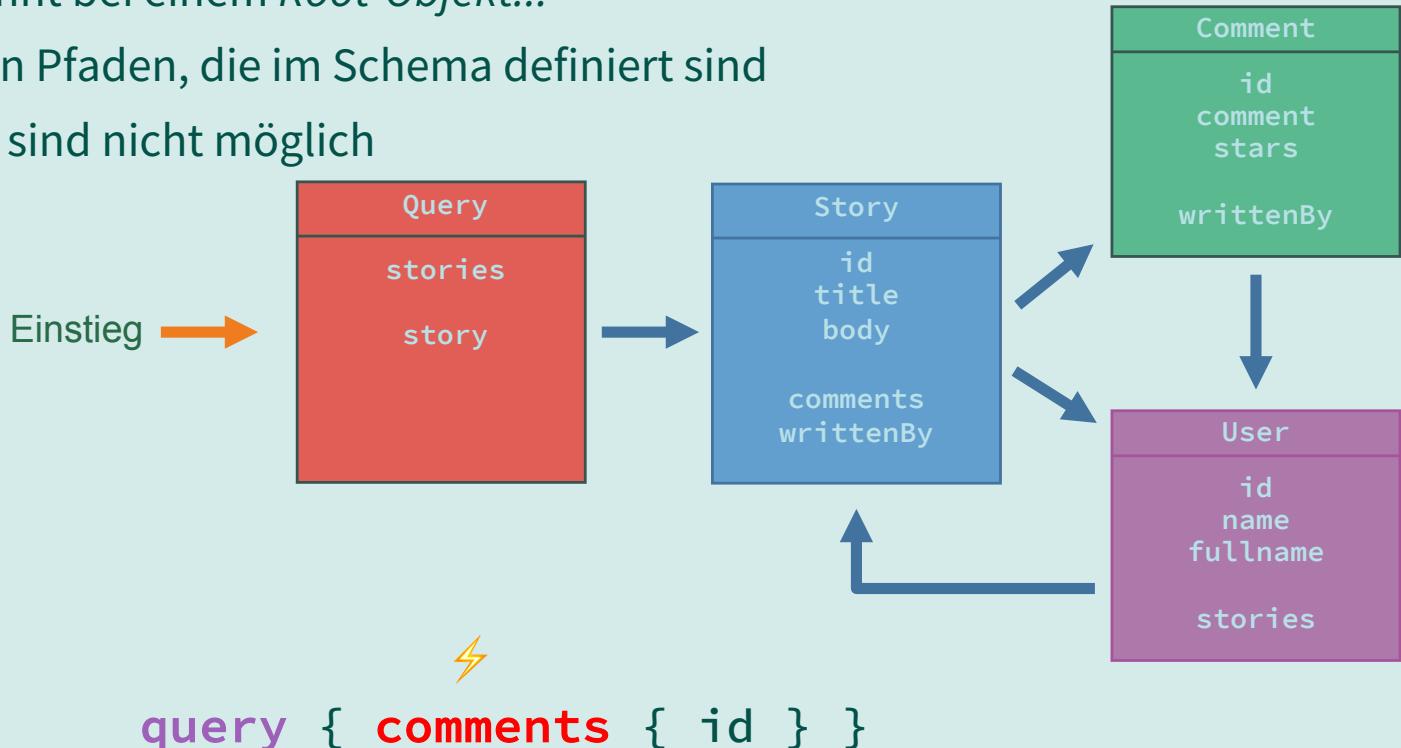


```
query { stories { comments { id comment } } }
```

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 playground interface with the following components:

- Left Panel (Query):** Displays a **Story** type with fields: **id**, **title**, **body**, **excerpt(maxLength: 200)**, and **tags**.
- Middle Panel (Query Editor):** Shows a query for **GetRecentStories**:


```
query GetRecentStories {
  stories {
    id
    title
    createdAt
    excerpt(maxLength: 30)
    writtenBy {
      profileImageUrl
      user {
        name
      }
    }
    comments(first: 3, orderBy: { field: createdAt, direction: DESC }) {
      content
    }
  }
}
```
- Right Panel (GraphiQL):** Displays the JSON response for the query:


```
{
  "data": {
    "stories": [
      {
        "id": "1",
        "title": "Simple way to repeat a string",
        "createdAt": "2021-10-09T04:40:50.027Z",
        "excerpt": "Non lacrimis curvum prominet m...",
        "writtenBy": {
          "profileImageUrl": "http://localhost:8090/avatars/avatar_U19.png",
          "user": {
            "name": "Oren Connolly"
          }
        },
        "comments": [
          {
            "content": "Hello GraphQL!"
          }
        ]
      },
      {
        "id": "2",
        "title": "Parse JSON in JavaScript [duplicate]",
        "createdAt": "2021-10-09T12:02:58.387Z",
        "excerpt": "Factum laticemque Partheniumqu...",
        "writtenBy": {
          "profileImageUrl": "http://localhost:8090/avatars/avatar_U14.png",
          "user": {
            "name": "Bryan Weissnat"
          }
        },
        "comments": [
          {
            "content": "Officia cuniditate \\n\\nEnim"
          }
        ]
      }
    ]
  }
}
```

Ein Query

Response

Dokumentation

Netzwerkverkehr (evtl.)

Demo: Query-Sprache

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

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

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

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 APIs sind komplexer in der Entwicklung als REST APIs"

Das ist in der Absolutheit so nicht richtig

"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

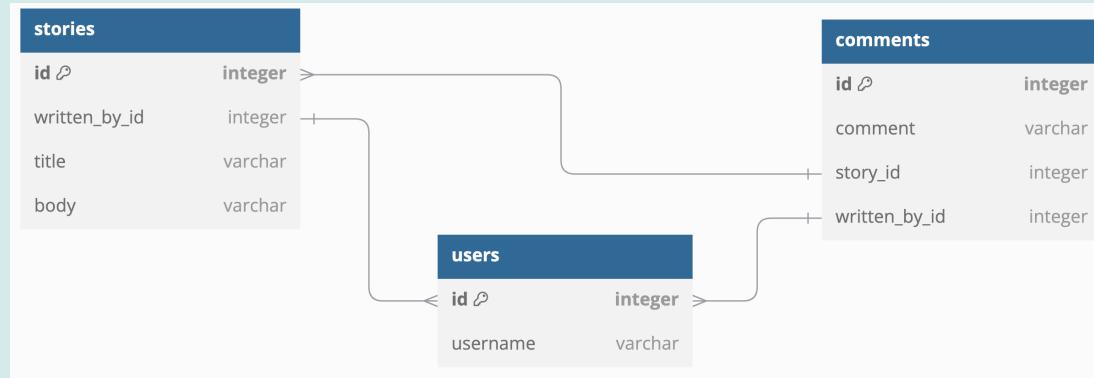
- Extrem Beispiel: "GraphQL API, REST-Style"

```
type Comment { id: ID! comment: String! approved: Boolean likes: Int }
type Story { id: ID! title: String! body: String! writtenBy: ID! comments: [ID!]! }
type User { id: ID! name: 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

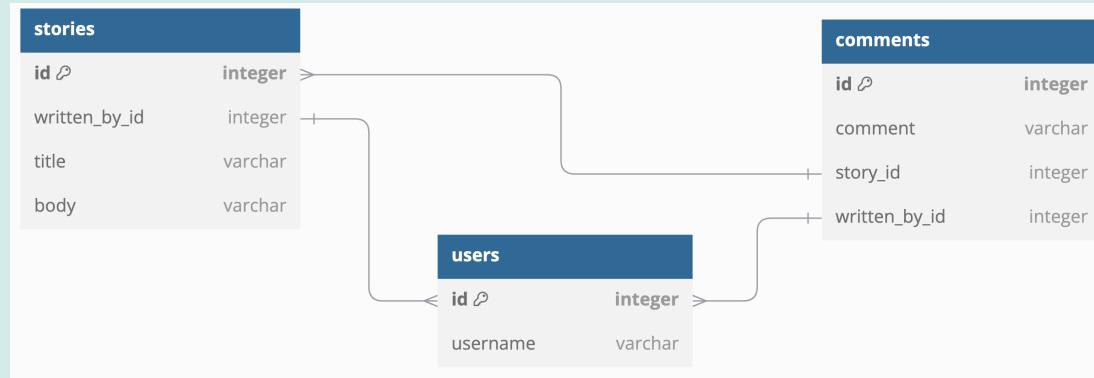
- Stories, User und Kommentare können mit einem SQL Query gelesen werden



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

Probleme der Optimierung #1

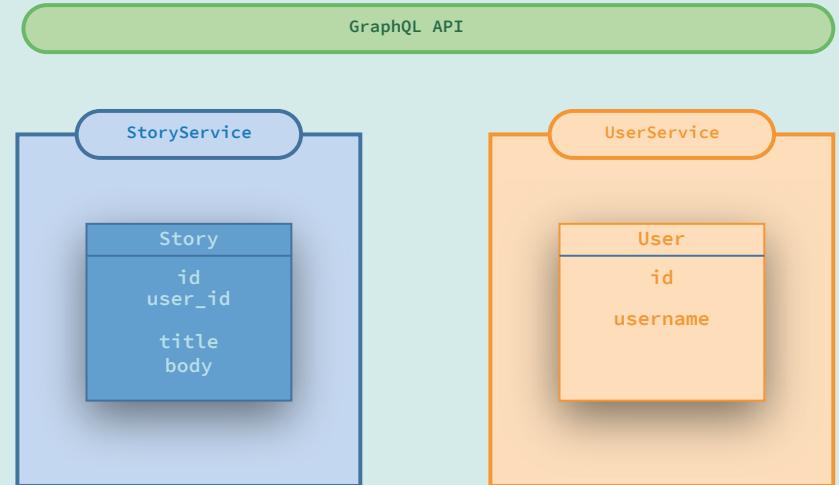
- Stories, User und Kommentare können mit einem SQL Query gelesen werden
- Wir müssen das aber von Query zu Query entscheiden
 - Eventuell sind das dann zu viele Daten



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

Probleme der Optimierung #2

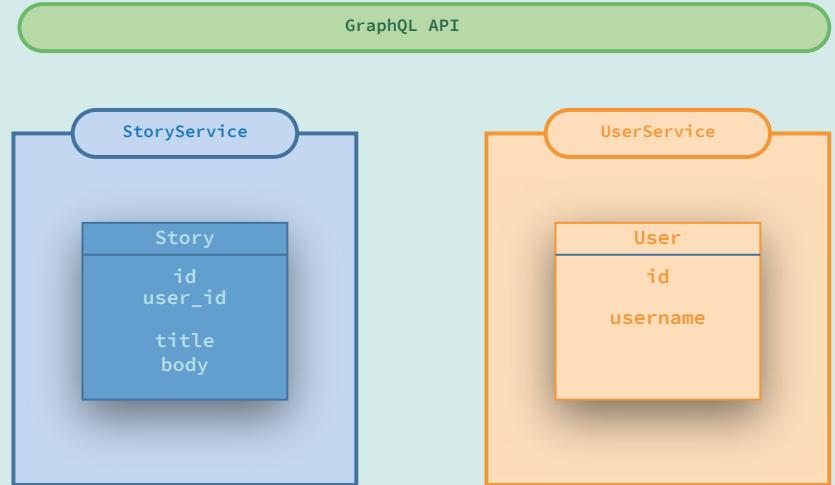
- Stories und User kommen aus unterschiedlichen Services



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

Probleme der Optimierung #2

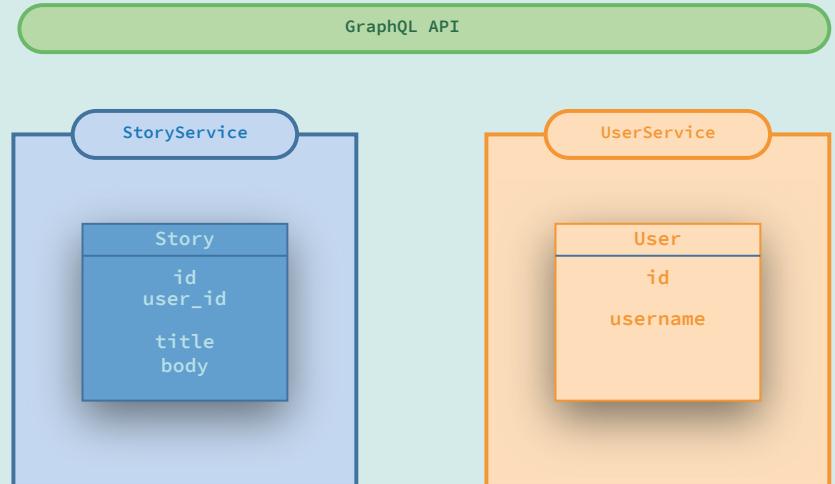
- Stories und User kommen aus unterschiedlichen Services
- Hier kann es zu n+1-Problemen kommen
 - 1 Request liefert n Stories zu der jeweils 1 User abgefragt werden muss



"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

"GraphQL ist SQL für's Frontend"

Das ist falsch

"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.

"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

"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

"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

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

Das ist falsch

"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

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

Das ist falsch

- Suggeriert wird, dass man mit GraphQL einfach alle Daten strukturiert zur Verfügung stellt
- Schema-Design ist aber genauso wie in anderen API-Technologien

"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

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

Das ist falsch

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

"Mit GraphQL gibt es kein Caching"

Das ist übertrieben

- 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
 - Im Entwurf der "GraphQL over HTTP"-Spec ist GET explizit vorgesehen

"Mit GraphQL gibt es kein Caching"

Probleme beim Caching #2

- Wenn mehrere Objekte abgefragt werden, muss die kürzeste Cache-Dauer gewinnen

"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)

"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 Ressource entscheiden, wie lange sie cachbar ist

"GraphQL spart Daten gegen über REST APIs"

Das ist nur teilweise korrekt

"GraphQL spart Daten gegen über REST APIs"

Das ist nur teilweise korrekt

- Richtig ist, das Netzwerk-Requests und damit Latenz gespart werden kann

"GraphQL spart Daten gegen über REST APIs"

Das ist nur teilweise korrekt

- 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

"GraphQL spart Daten gegen über REST APIs"

Das ist nur teilweise korrekt

- 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 alles auch mit REST APIs machen..."

Oftmals korrekt

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

Oftmals korrekt

- Mit Search-Parametern etc. kann man oft die Anzahl der Attribute einschränken

"Das kann ich alles 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

"Das kann ich alles 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

"Das kann ich alles 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
- Mit HATEOS zeigt der Server wo bzw. wie weitere Daten geladen werden können

"Das kann ich alles 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
- Mit HATEOS zeigt der Server wo bzw. wie weitere Daten geladen werden können
- Aber:
 - das alles in GraphQL "eingebaut", man bekommt es aus einer Hand

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe:

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe:

- Over-/Underfetching (meines Erachtens überbewertet)

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe:

- Over-/Underfetching (meines Erachtens überbewertet)
- Schema-Evolution ermöglicht separate Weiterentwicklung der API

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe:

- Over-/Underfetching (meines Erachtens überbewertet)
- Schema-Evolution ermöglicht separate Weiterentwicklung der API
- Tooling und Typsicherheit bieten sehr gute Entwicklungserfahrung

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe:

- Over-/Underfetching (meines Erachtens überbewertet)
- Schema-Evolution ermöglicht separate Weiterentwicklung der API
- Tooling und Typsicherheit bieten sehr gute Entwicklungserfahrung
- Fehlerbehandlung expliziter als mit REST

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe dagegen:

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe dagegen:

- REST ist Mainstream und in der Entwicklung gut verstanden

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe dagegen:

- REST ist Mainstream und in der Entwicklung gut verstanden
- Konsumenten wissen, wie REST funktioniert

"Wann soll ich dann GraphQL überhaupt verwenden?"

Es gibt gute Gründe dagegen:

- REST ist Mainstream und in der Entwicklung gut verstanden
- Konsumenten wissen, wie REST funktioniert
- Caching und Datensparsamkeit (kommt auf Anforderung an)

"Wann soll ich dann GraphQL überhaupt verwenden?"

it depends...

(wie immer 😊)

NILS HARTMANN
<https://nilshartmann.net>



vielen Dank!

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

Fragen & Kontakt: nils@nilshartmann.net

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

