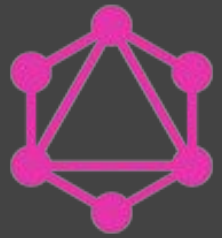


GraphQL

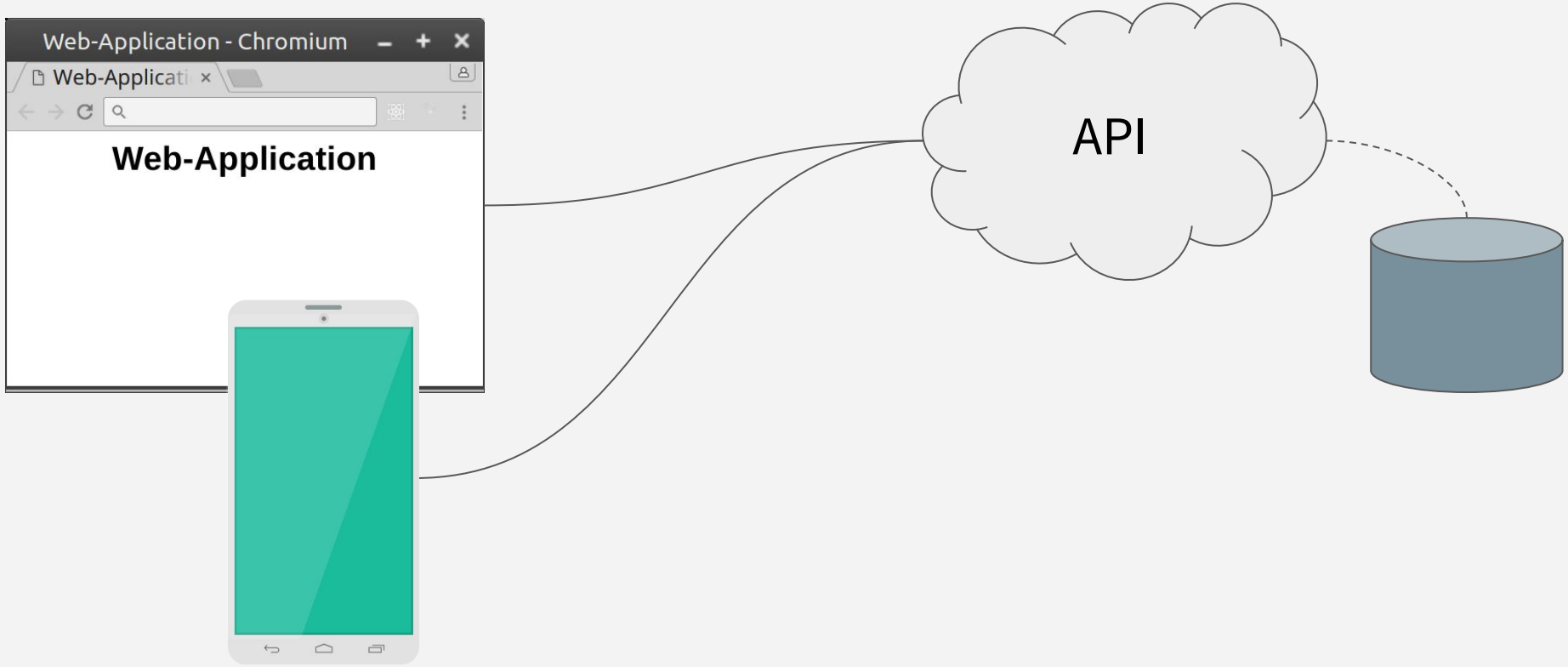
als Alternative zu REST

Manuel Mauky
@manuel_mauky

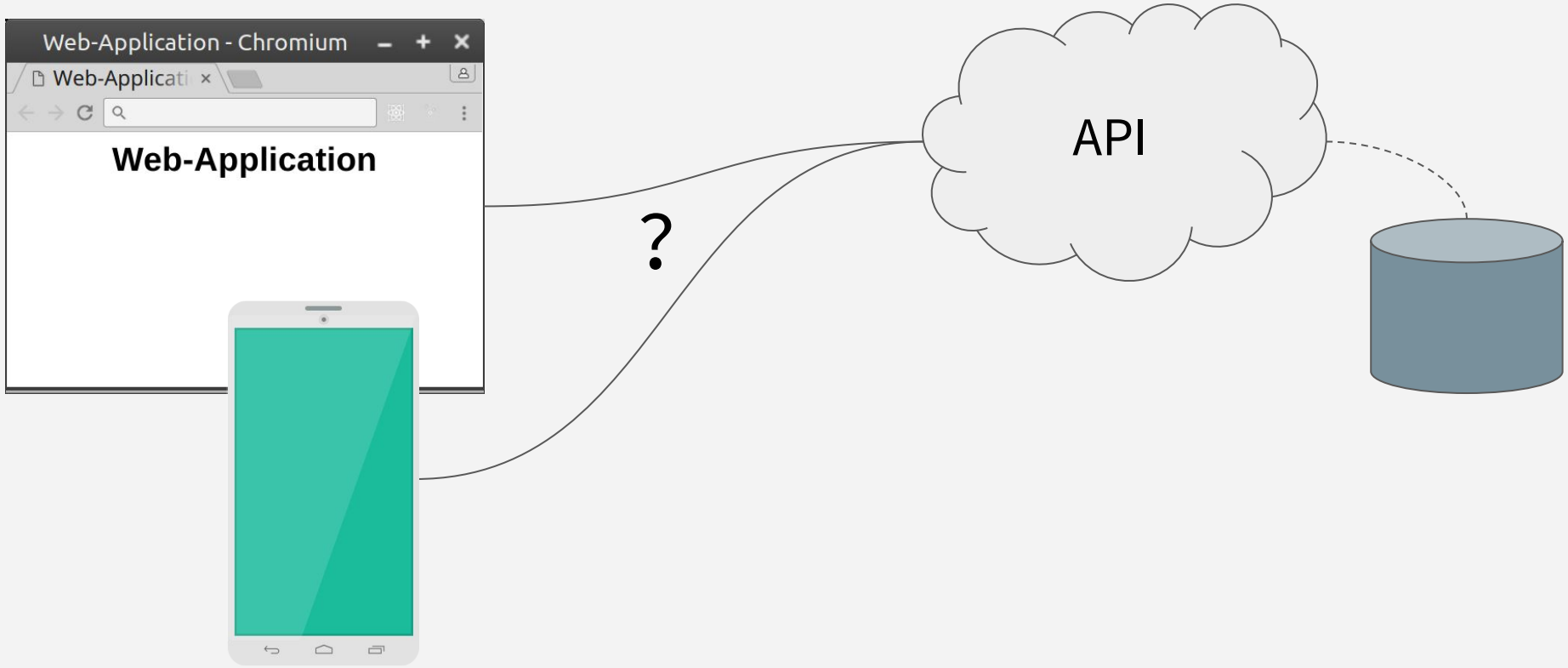
JUG Görlitz

Saxonia Systems
So geht Software.

Single-Page-Apps und mobile Apps



Wie kommen die Daten zum Client?



Klassisch: REST/Hypermedia

- zahlreiche Ressourcen, jeweils über URIs erreichbar
- Hyperlinks zwischen Ressourcen
- HTTP als Protokoll
- verschiedene Repräsentationen, z.B. JSON

Motivation

Wir haben doch REST, warum also mit GraphQL zu beschäftigen?

Gründe, sich mit GraphQL zu beschäftigen

- Löst einige Schwierigkeiten von REST bei Client-Server-Kommunikation
- Netzwerk-Performance
- Ermöglicht neue Ideen im Frontend
- "Rethinking Best-Practices"

Clintr

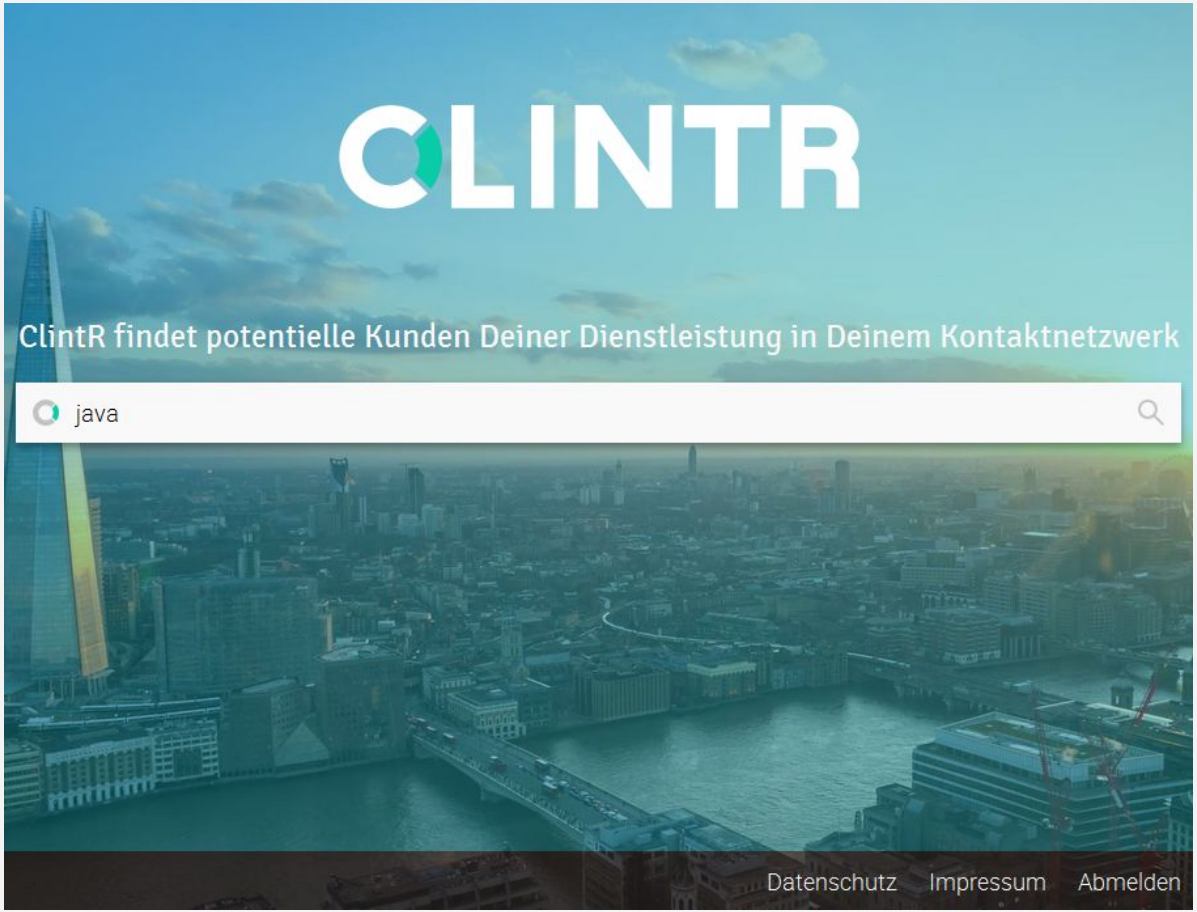
Webanwendung für Dienstleister

Suche nach angebotener Dienstleistung

Finden von potentiellen Kunden

Aufzeigen von Kontaktmöglichkeiten zum Kunden

Quelle: Xing Job-Suche



CLINTR

java

Dein **Kontakt** kennt

Distributed Applications

Senior Softwareentwickler

Du hast keinen Kontakt

zu **Mitarbeitern** dieser Firma

Du hast keinen Kontakt

zu **Mitarbeitern** dieser Firma

Dein **Kontakt** kennt

Test Manager

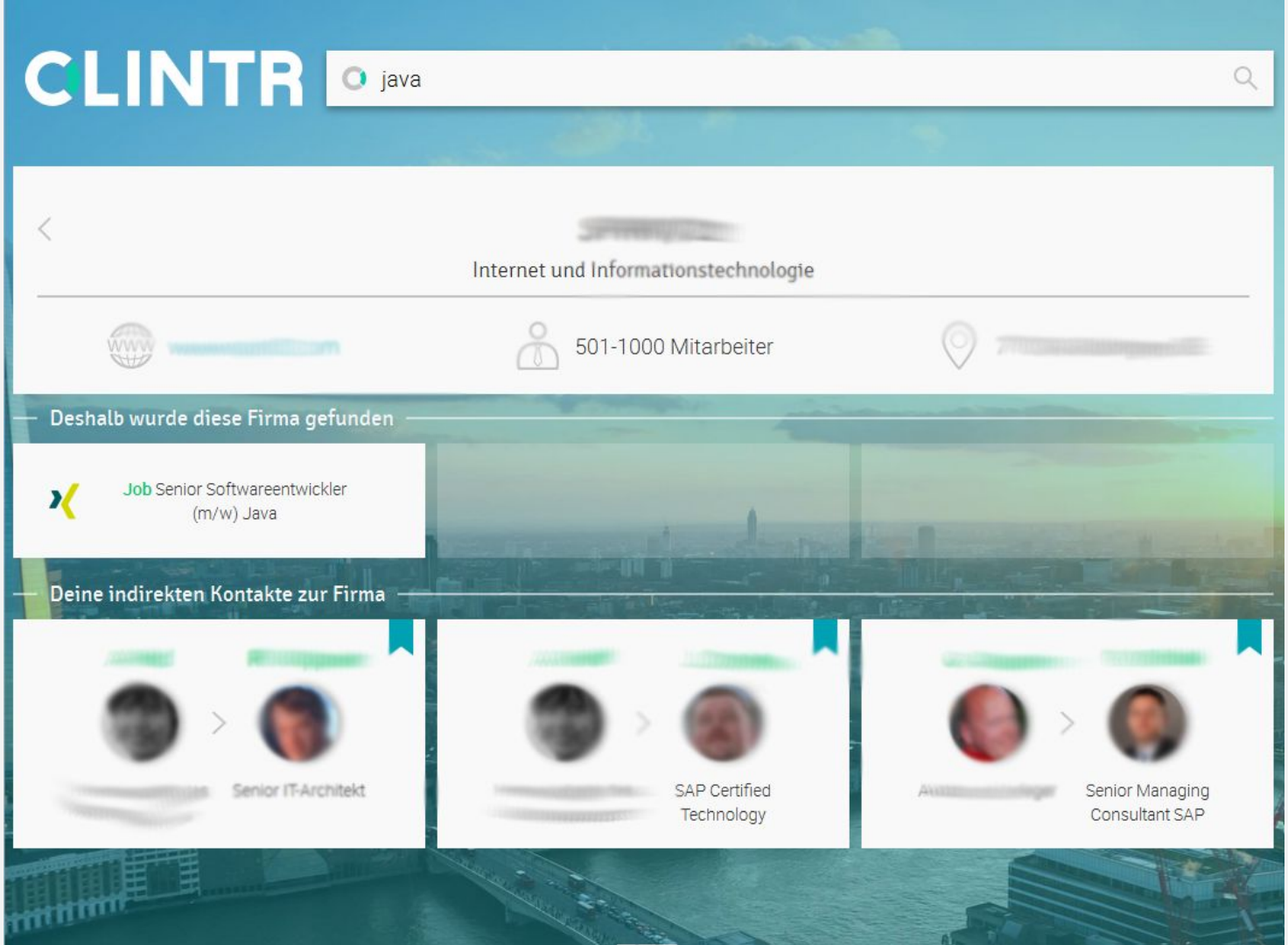
Senior Test Architect

Du hast keinen Kontakt

zu **Mitarbeitern** dieser Firma

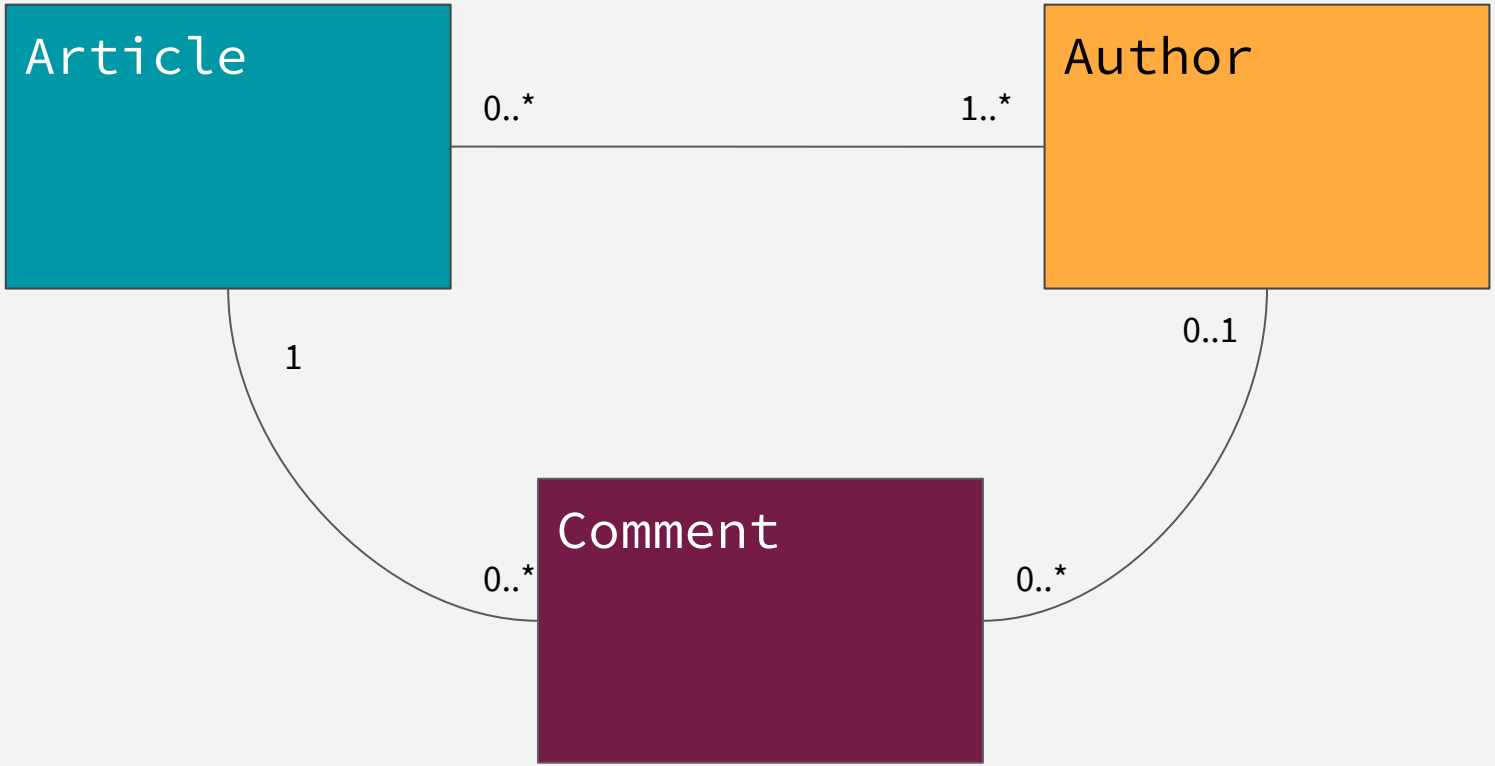
Dein **Kontakt** kennt

Senior IT-Architekt


CLINTR
java
Internet und Informationstechnologie
501-1000 Mitarbeiter
Deshalb wurde diese Firma gefunden
Job Senior Softwareentwickler (m/w) Java
Deine indirekten Kontakte zur Firma
Senior IT-Architekt
SAP Certified Technology
Senior Managing Consultant SAP

Beispiel

Beispiel: Blog-System



Article

id: String

permalink: String

teaser: String

text: String

releaseDate: Datetime

authors: [Author]

comments: [Comment]

Author

id: String

name: String

comments: [Comment]

Comment

id: String

text: String

author: Author?

guestAuthor: String?

REST+Hypermedia

Use-Case: Artikel-Übersicht (10 Artikel)

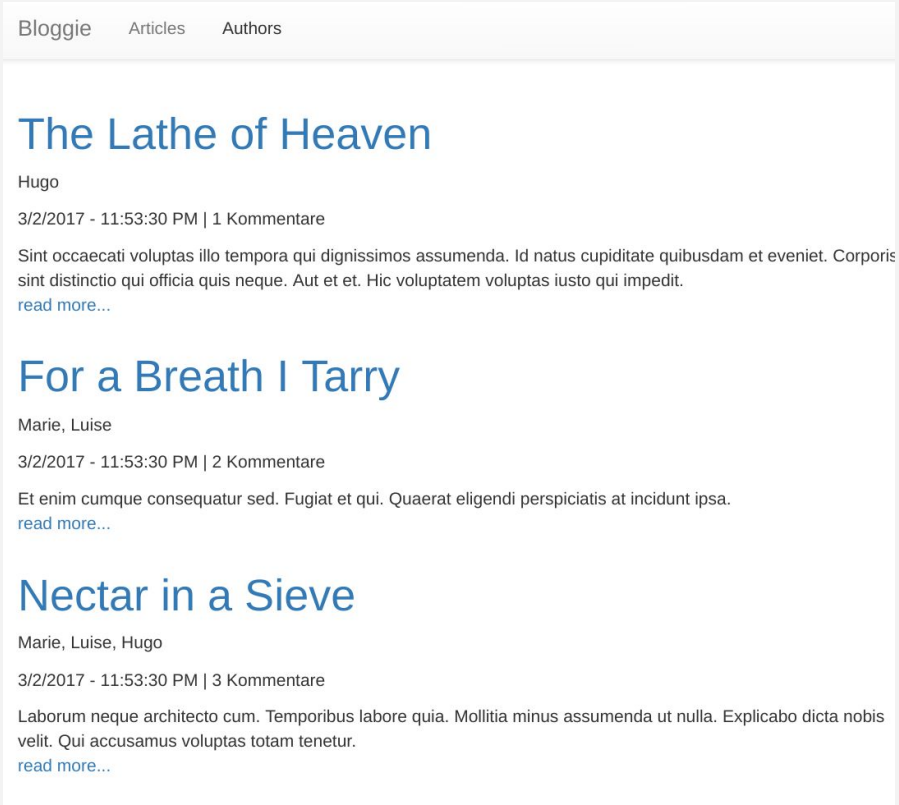
Benötigte Daten:

Je Artikel:

Titel, Permalink, Teasertext, releasedate

Autoren: Name

Anzahl Kommentare



Use-Case 2: Artikel-Details

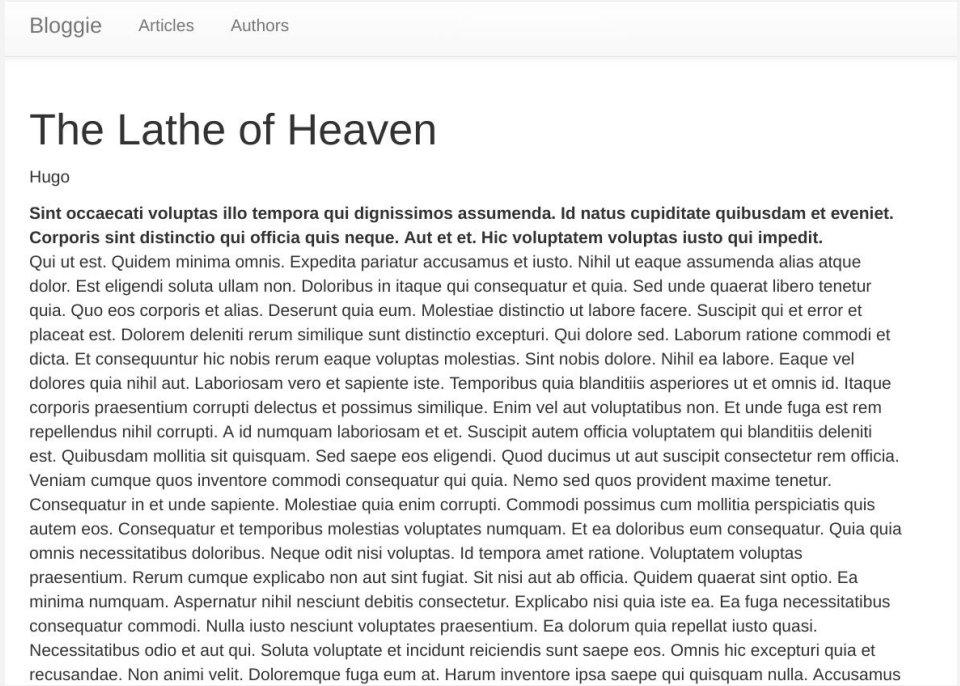
Benötigte Daten:

Artikel:

Titel, Permalink, Teasertext, releasedate, **Text**

Autoren: Name

Kommentare: **Autorname, Text**



REST+Hypermedia

GET /api/ → Liste von verfügbaren Ressourcen

GET /api/article → Liste von Artikeln mit Titel, Permalink, Teaser, ReleaseDate

REST+Hypermedia

Je Artikel:

GET /api/article/:id/authors → Autoren des Artikels

GET /api/article/:id/comments → Kommentare

REST+Hypermedia

Detail-Ansicht:

GET /api/article/:id → Text

REST+Hypermedia

- viele Requests (im Beispiel 22 Requests für Startseite)
- Teils zu wenig Daten in einem Request (Underfetching)
- Teils zu viele Daten in einem Request (Overfetching)

Naive "Lösung":

~~Hypermedia~~ → Alle Daten "embedden"

Lösung: Alle Daten "embedden"

Idee: Artikel-Ressource enthält direkt alle Daten

Nur noch 1 Request

Lösung: Alle Daten "embedden"

Idee: Artikel-Ressource enthält direkt alle Daten

Nur noch 1 Request

Problem: Sehr viele unnötige Daten werden übertragen

Query-Parameter / Projections

Query-Parameter

/api/article?withAuthor=true&withComments=true

Projections

/api/article?projection=short-article

View-Spezifische Ressourcen

View-Spezifische Ressourcen

/api/article_overview

/api/article_details/:id

/api/author_list

/api/authors_comments_list/:id

Verschiedene Repräsentationen

Verschiedene Repräsentationen

GET /api/article ACCEPT: application/short+json

GET /api/article ACCEPT: application/full+json

GET /api/article ACCEPT: application/with.comments+json

Problem:
Hypermedia-Ansatz wird verwässert
⟶ Kopplung Server und Client

Problem: Kopplung Server und Client

- Neue Ansichten im Client benötigen neue Ressourcen auf dem Server
- Änderungen an Ressourcen auf dem Server haben Auswirkungen auf bestehende Ansichten im Client

Herausforderung: Fehlende Typisierung

Herausforderung: Fehlende Typisierung

/api/article/:id

Welche Daten kommen eigentlich als Antwort?

Herausforderung: Versionierung

Herausforderung: Versionierung

/api/article/:id

Welche Daten kommen eigentlich als Antwort?

Welche Daten kommen in 2 Monaten als Antwort?


GraphQL

GraphQL

- Abfragesprache für (Web-)APIs
- Entwickelt von Facebook
- Seit 2012 bei Facebook Mobile App im Einsatz
- Seit Ende 2015 OpenSource

GraphQL - Technisches

- ein einziger Endpunkt
- statisch typisiertes Schema
- Schema erlaubt Introspection
- Queries per HTTP POST oder GET
- Trennung von Query und Mutation → Ansätze von CQRS

Beispiel: Blog-System

Use-Case: Artikel-Übersicht (10 Artikel)

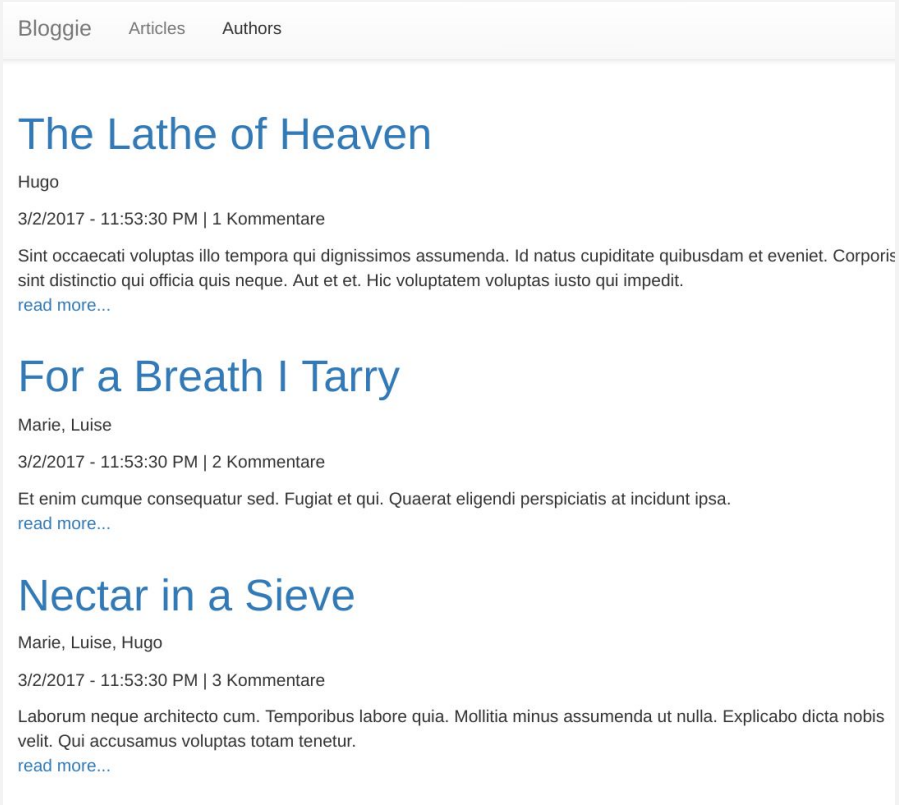
Benötigte Daten:

Je Artikel:

Titel, Permalink, Teasertext, releasedate

Autoren: Name

Anzahl Kommentare



Query:

```
{
  Article {
    title
  }
}
```

Response:

```
{
  "data": {
    "Article": [
      {
        "title": "Some Title"
      },
      {
        "title": "Other.."
      }
    ]
  }
}
```

```
{
  Article {
    id
    title
    permalink
    teaser
    releaseDate
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum..",
        "releaseDate": "2017-02.."
      },
      {
        "id": "sdöf234ds2",
        "title": "Other..",
        ...
      }
    ]
  }
```

```
{
  Article {
    id
    title
    permalink
    teaser
    releaseDate
    authors {
       name
    }
    comments {
        id
    }
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum..",
        "releaseDate": "2017-02..",
        "authors": [
           {
              "name": "Hugo",
           }
        ],
        "comments": [
          {
            "id": "lsldsö2fö2",
          }
        },
        {
          "id": "sdöf234ds2",
          "title": "Other ",
```

Use-Case 2: Artikel-Details

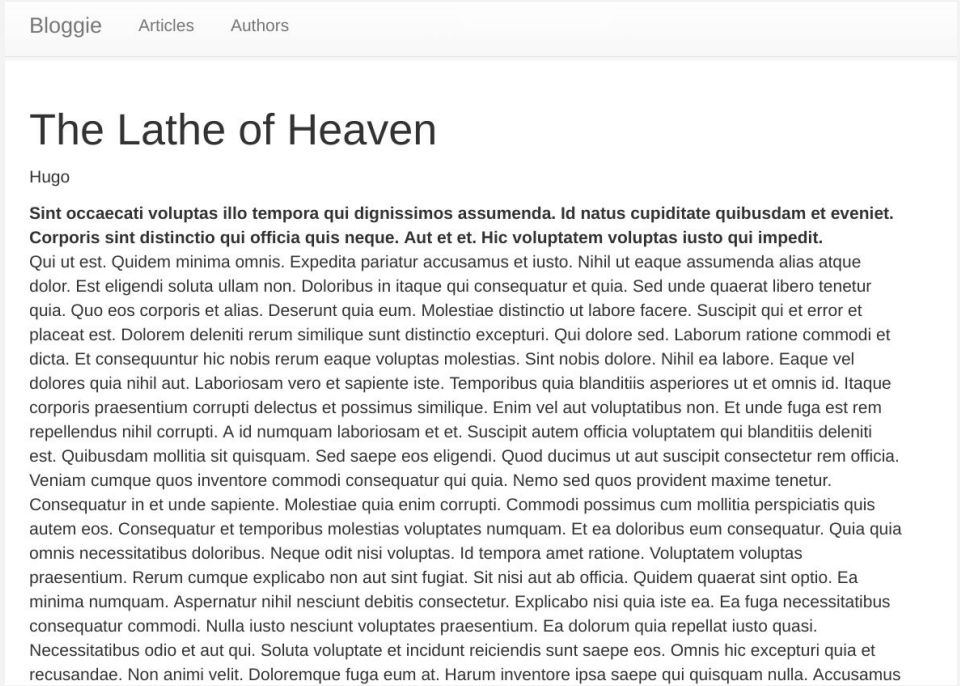
Benötigte Daten:

Artikel:

Titel, Permalink, Teasertext, releasedate, **Text**

Autoren: Name

Kommentare: **Autorname, Text**



```
{
  Article(id: "sd342") {
    id
    text
    ...
    authors {
      name
    }
    comments {
      id
      text
      author {
        name
      }
    }
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd342",
        ...
        "authors": [
           {
              "name": "Hugo",
           }
        ],
        "comments": [
          {
            "id": "lsldsö2fö2",
            "text": "Erster!!111!",
            "author": {
                "name": "Horsti"
            }
          }
        ]
      },
      {
```

Mutations

```
mutation {
  addAuthor(username: "Horsti") {
    id
    name
  }
}
```

```
{
  "data": {
    "addAuthor": {
      "id": "lsdö34",
      "name": "Horsti"
    }
  }
}
```

Zwischenfazit

- Server stellt statisch typisiertes Schema bereit
- Schema definiert, welche Daten abfragbar sind
- Clients können genau die Daten abfragen, die für sie notwendig sind
- Beliebige Schachtelung ist möglich mit einem Request

Neue Ideen im Frontend

Query-Optimierung + Caching

Schritt 1: Artikel-Übersicht

```
{
  Article {
    id
    title
    permalink
    teaser
    releaseDate
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum..",
        "releaseDate": "2017-02.."
      },
      {
        "id": "sdöf234ds2",
        "title": "Other..",
        ...
      }
    ]
  }
```

Schritt 2: Artikel-Details

```
{
  Article (id: "sd34..") {
    id
    title
    text
    permalink
    teaser
    releaseDate
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "text": "some long text…",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum..",
        "releaseDate": "2017-02.."
      }
    ]
  }
}
```

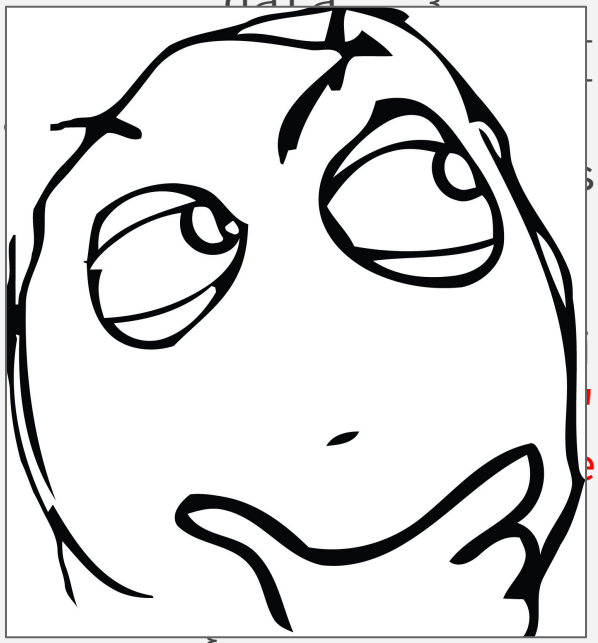
Schritt 2: Artikel-Details

```
{
  Article (id: "sd34..") {
    id
    title
    text
    permalink
    teaser
    releaseDate
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "text": "some long text…",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum..",
        "releaseDate": "2017-02.."
      }
    ]
  }
}
```

Schritt 2: Artikel-Details

```
{
  Article (id: "sd34
    id
    title
    text
    permalink
    teaser
    releaseDate
  }
}
```



```
{
  "data": {
                                    sd3423s32",
                                     "Some Title",
                                   "some long text…",
                                   nk": "some_title",
                                   ": "Lorem Ipsum..",
                                   eDate": "2017-02.."
    }
  }
```

Schritt 2: Artikel-Details

```
{
  Article (id: "sd34..") {
    id
    title
    text
    permalink
    teaser
    releaseDate
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "text": "some long text…",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum..",
        "releaseDate": "2017-02.."
      }
    ]
  }
}
```

Schritt 2: Artikel-Details

```
{
  Article (id: "sd34..") {
    id
    text
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "text": "some long text…",
      }
    ]
  }
}
```

Schritt 2: Artikel-Details

```
{
  Article (id:
    id
    text
  }
}
```

```
{
                                    3s32",
                                    e long text…",
```

Schritt 3: Zurück zur Übersicht

```
{
  Article {
    id
    title
    permalink
    teaser
    releaseDate
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum..",
        "releaseDate": "2017-02.."
      },
      {
        "id": "sdöf234ds2",
        "title": "Other..",
        ...
      }
    ]
  }
```

Schritt 3: Zurück zur Übersicht

```
{
  Article {
    id
    title
    permalink
    teaser
    releaseDate
  }
}
```

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum..",
        "releaseDate": "2017-02.."
      },
      {
        "id": "sdöf234ds2",
        "title": "Other..",
        ...
      }
    ]
}
```

Query-Optimierung und Caching

- Entwickler beschreibt Daten-Anforderungen je Ansicht
- Framework optimiert Queries und übernimmt Caching

Query co-location

Startseite - Chromium

Startseite

www.jug-gr.de

Java User Group Görlitz | Aktuell | Vorträge | Unterstützer | About | Mitmachen

Nächstes Treffen:

Angular 2 für Java Entwickler

von Sven Hoffmann | 22 March 2017

Mit Angular bietet Google ein komponenten-basiertes Webframework an, welches den Anspruch erhebt, die Anforderungen an zeitgemäße Softwareentwicklung ganzheitlich zu unterstützen.

Seit dem Release von Angular 2 war es für gestandene Java Entwickler noch nie so einfach in die Webentwicklung einzusteigen. Mit Klassen, Dependency Injection, Annotations und Data-Binding bietet Angular dem versierten Desktop Entwickler ein vertrautes Umfeld, in dem man sich schnell zurecht findet.

In dieser Veranstaltung möchte ich am lebenden Beispiel zeigen, wie es sich heutzutage anfühlt eine Webanwendung, konkret mit Angular 2, zu erstellen.

Sven Hoffmann

Sven Hoffmann ist seit über 8 Jahren für die Saxonia Systems AG tätig. Er beschäftigte sich in dieser Zeit mit einer Vielzahl von Backend und Frontend-Technologien. Zu seinen technologischen Hauptschwerpunkten zählen Webanwendungen und JavaEE.

Datum: 22.03.2017, 19:00 Uhr

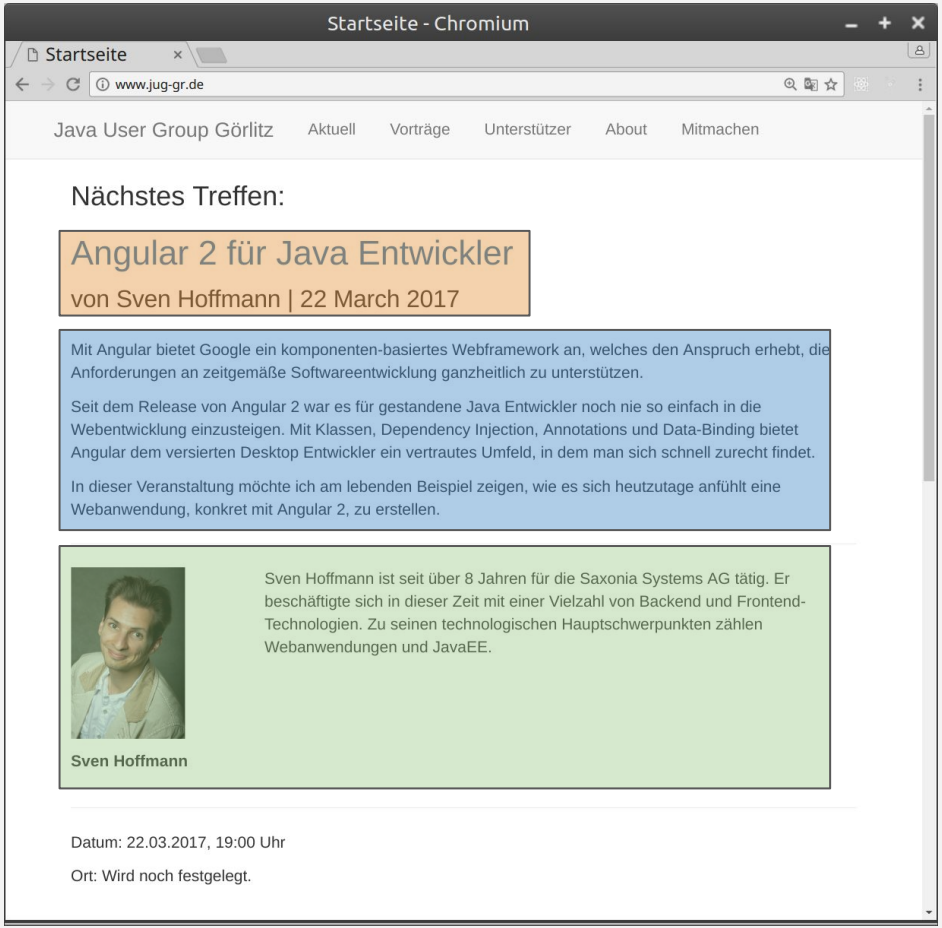
Ort: Wird noch festgelegt.

Header-Komponente

```
{
  Article(id: "123") {
    title
    authors {
      name
    }
    releaseDate
}
```

Autor-Komponente

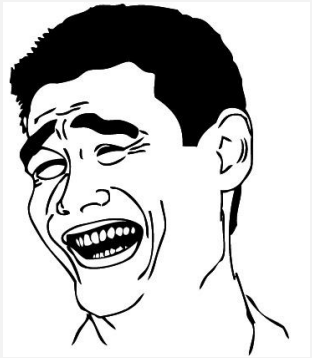
```
{
  Article(id: "123") {
    authors {
      name
      pictureUrl
      description
    }
}
```



Content-Komponente

```
{
  Article(id: "123") {
    text
  }
}
```

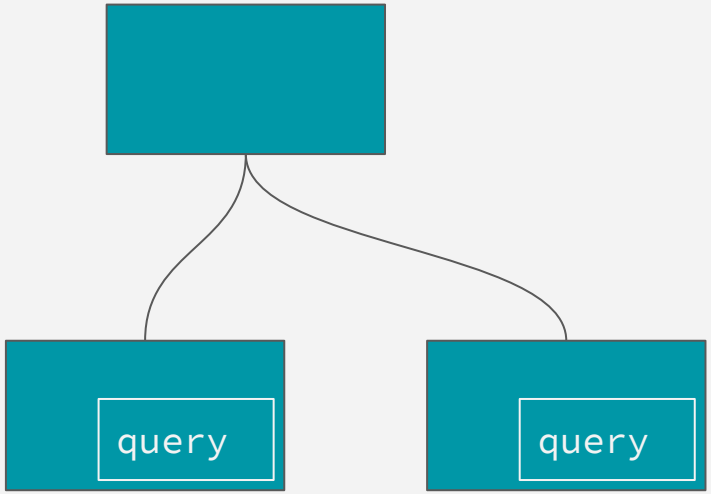
Schlechte Idee:
Jede UI-Komponente führt eigene Requests aus



Idee

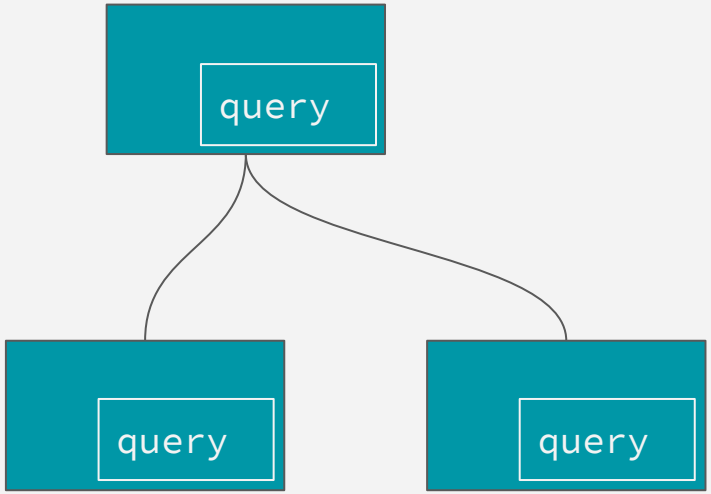
- Jede UI-Komponente deklariert die benötigten Daten als GraphQL-Fragment

Idee



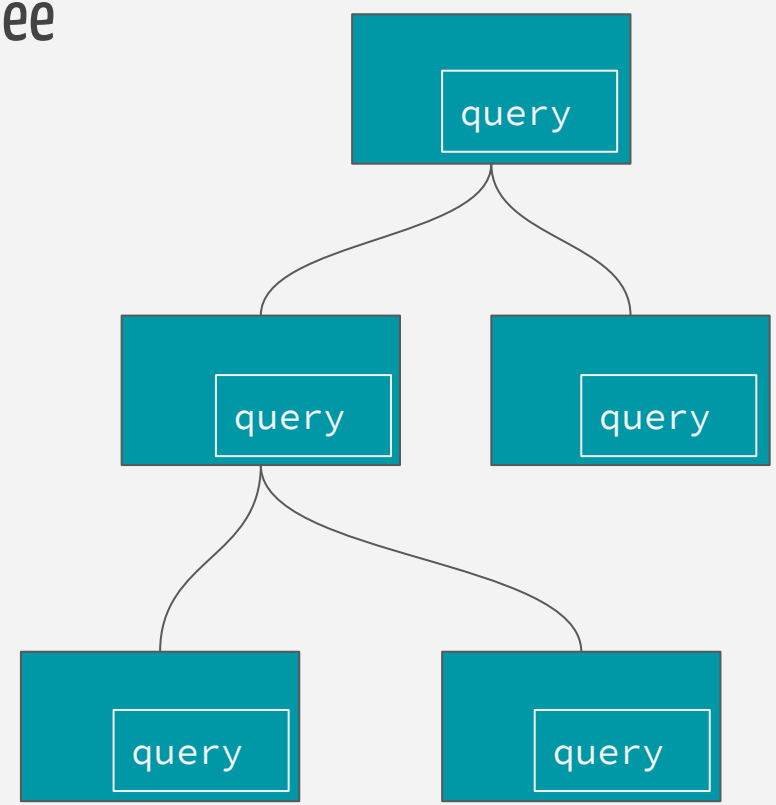
- Jede UI-Komponente deklariert die benötigten Daten als GraphQL-Fragment
- Parents sammeln Queries von Childs

Idee



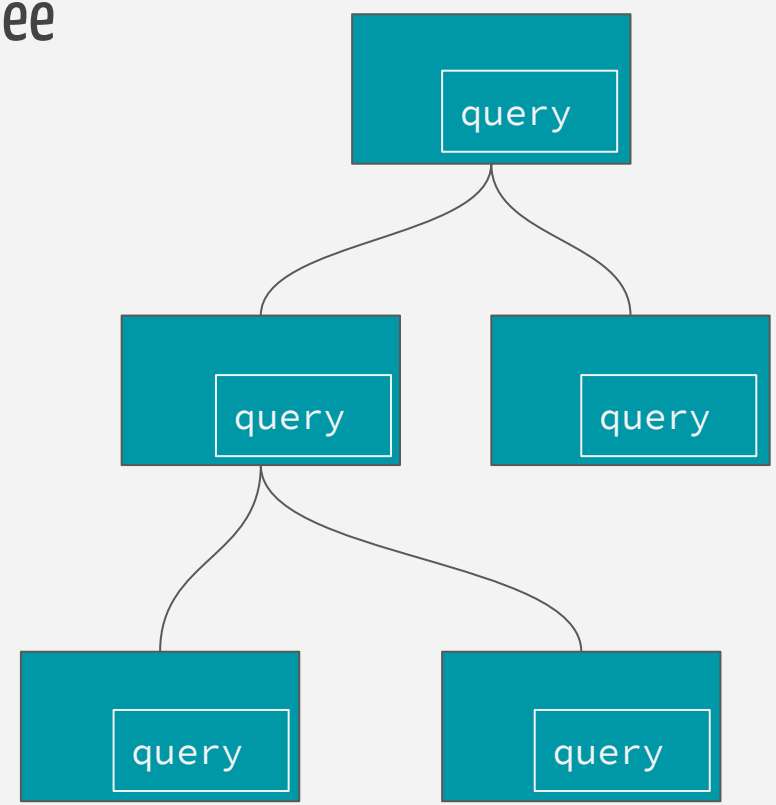
- Jede UI-Komponente deklariert die benötigten Daten als GraphQL-Fragment
- Parents sammeln Queries von Childs
- Parents "mergen" Queries

Idee



- Jede UI-Komponente deklariert die benötigten Daten als GraphQL-Fragment
- Parents sammeln Queries von Childs
- Parents "mergen" Queries
- Root-Komponente führt Gesamt-Query aus

Idee



- Jede UI-Komponente deklariert die benötigten Daten als GraphQL-Fragment
- Parents sammeln Queries von Childs
- Parents "mergen" Queries
- Root-Komponente führt Gesamt-Query aus
- Root reicht Daten an Childs durch

Vorteile

- Childs sind unabhängig von ihren Parents
- Parents sind Unabhängig von Daten-Bedürfnissen der Childs
    - Neues Feature (Daten) in Child → keine Code-Änderung in Parent
- Queries können optimiert werden
- Caching

GraphQL Schema

Wie sieht der Server aus?

GraphQL Server

1. Wie definiert man ein Schema?
2. Wo kommen die Daten her?

Wie definiert man ein Schema?

- Option 1: GraphQL Schema Language
- Option 2: programmatisch mit GraphQL-Bibliothek
- Option 3: Generieren (z.B. JPA-Entities, Postgres-Datenbank,...)

GraphQL Schema Language

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

```
type Author {
  id: ID!
  name: String!
  articles: [Article]
}


type Comment {
  id: ID!
  text: String!
  author: Author
]
```

```
type Article {
  id: ID!
  releaseDate: String
  teaser: String
  text: String
  permalink: String
  authors: [Author]
  comments: [Comment]
}
```

```
type QueryType {
  Author(id: String): [Authors]
  Article(id: String, permalink: String): [Article]
  Comment: [Comment]
}


schema {
  query: QueryType
  mutation: MutationType
}
```

Programmatisch (graphql-java)

```
GraphQLObjectType authorType = GraphQLObjectType.newObject()
                .name("Author")
                .field(GraphQLFieldDefinition.newFieldDefinition()
                        .name("id")
                        .type(GraphQLID))
                .field(GraphQLFieldDefinition.newFieldDefinition()
                        .name("name")
                        .type(GraphQLString))
                .field(GraphQLFieldDefinition.newFieldDefinition()
                        .name("articles")
                        .type(new GraphQLList(
                                new GraphQLTypeReference("Article")
                        ))
                .build();
```

Programmatisch (graphql.js)

```
const AuthorType = new GraphQLObjectType({
    name: 'Author',
    fields: {
        id: { type: GraphQLID },
        name: { type: GraphQLString },
        articles: { type: new GraphQLList(ArticleType) }
    }
}
```

Wo kommen die Daten her?

Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```

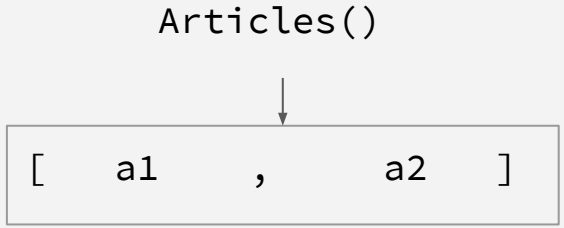
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```

```
Articles()
```

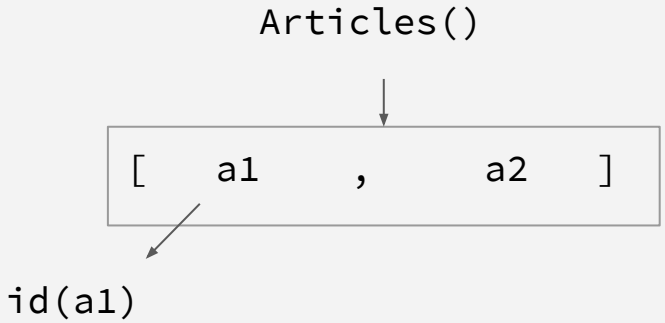
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



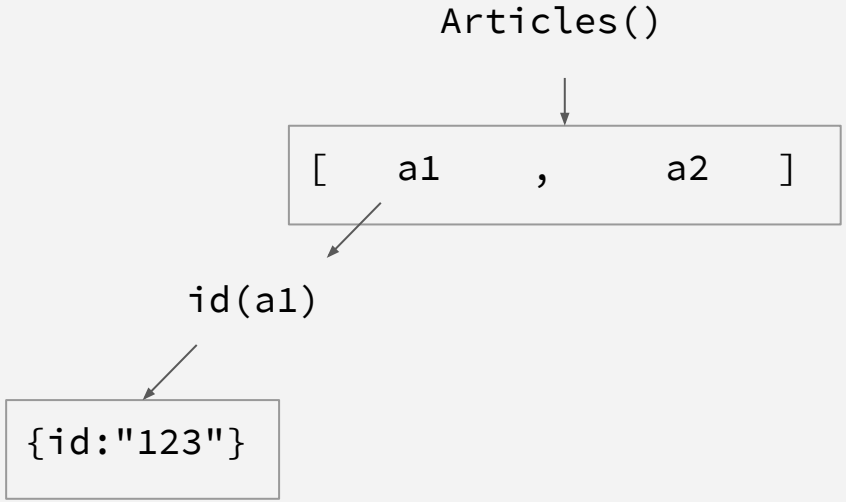
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



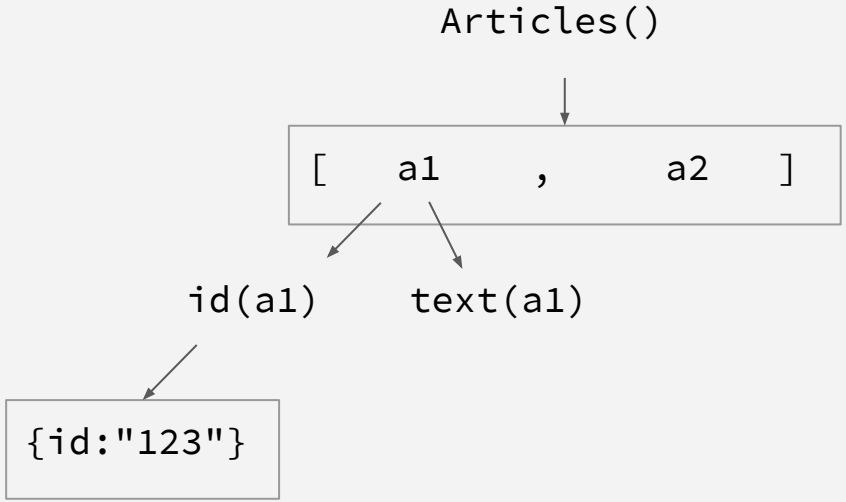
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



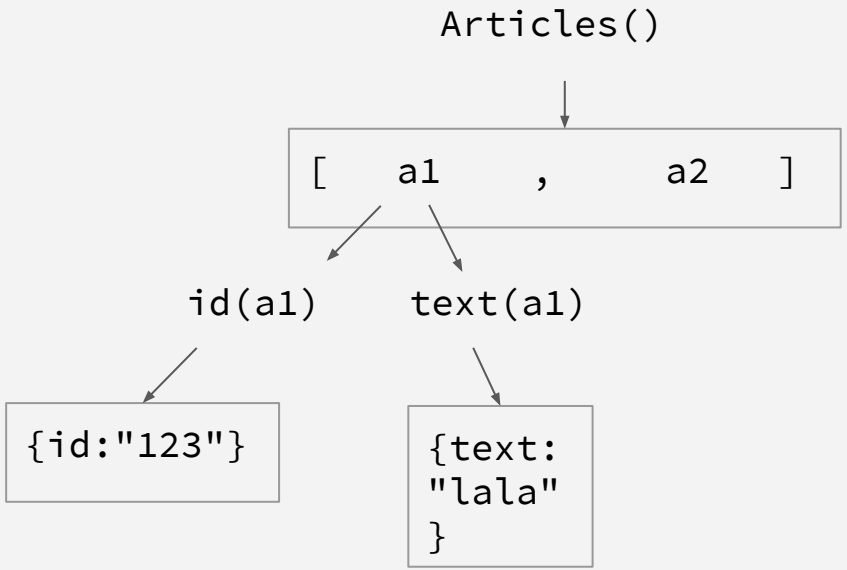
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



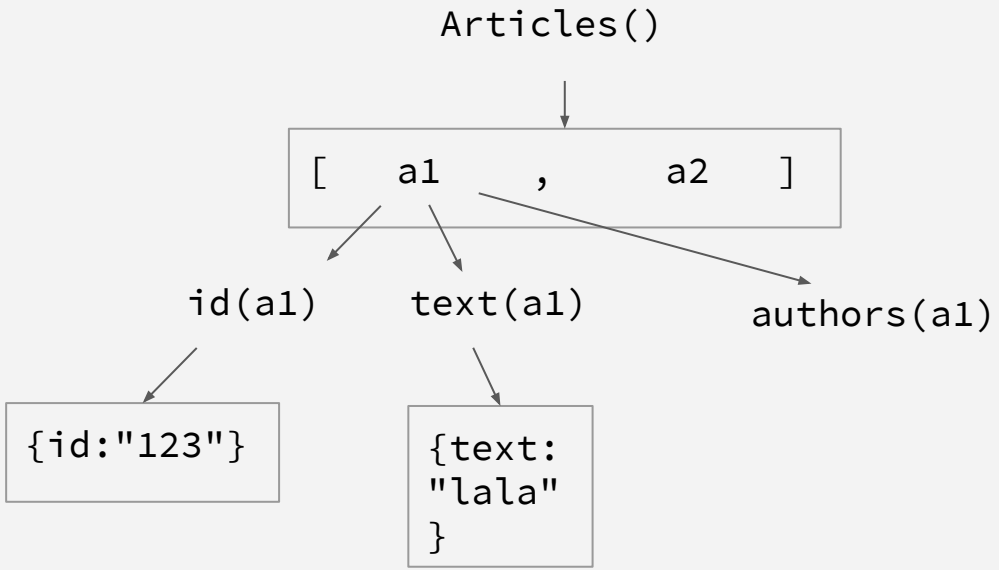
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



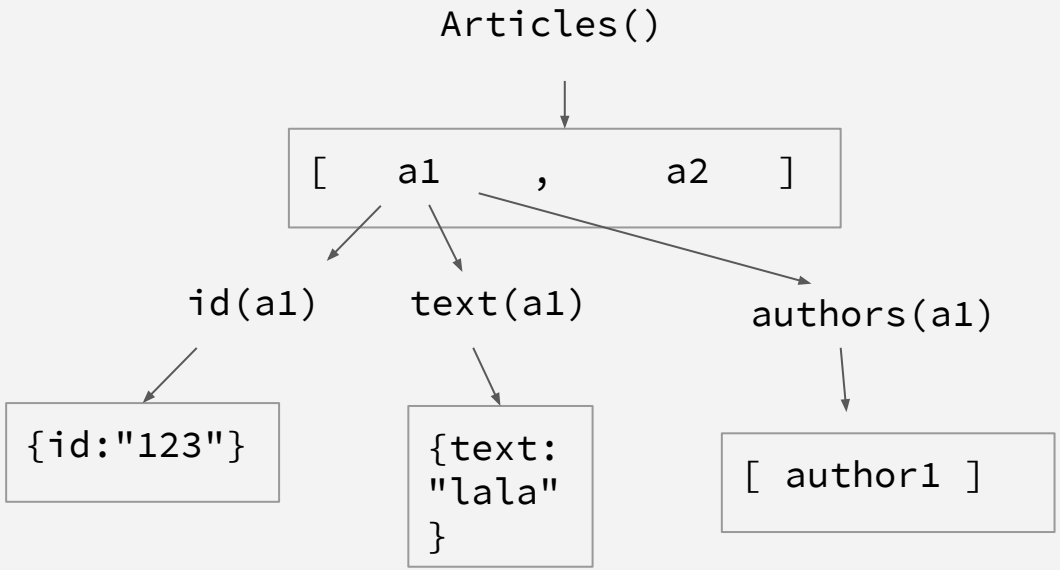
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



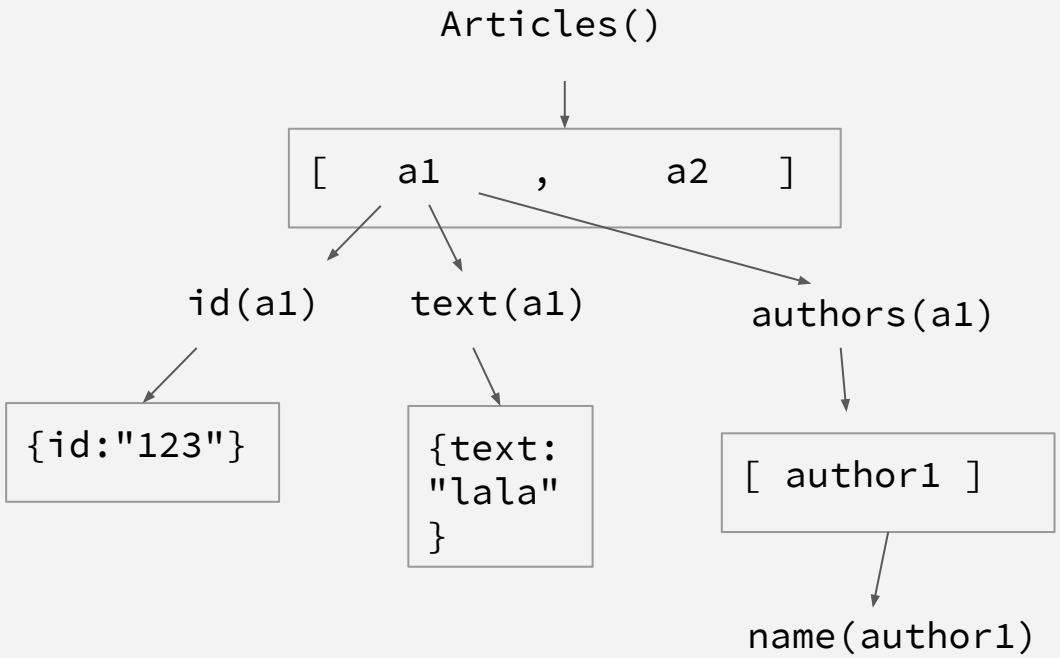
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



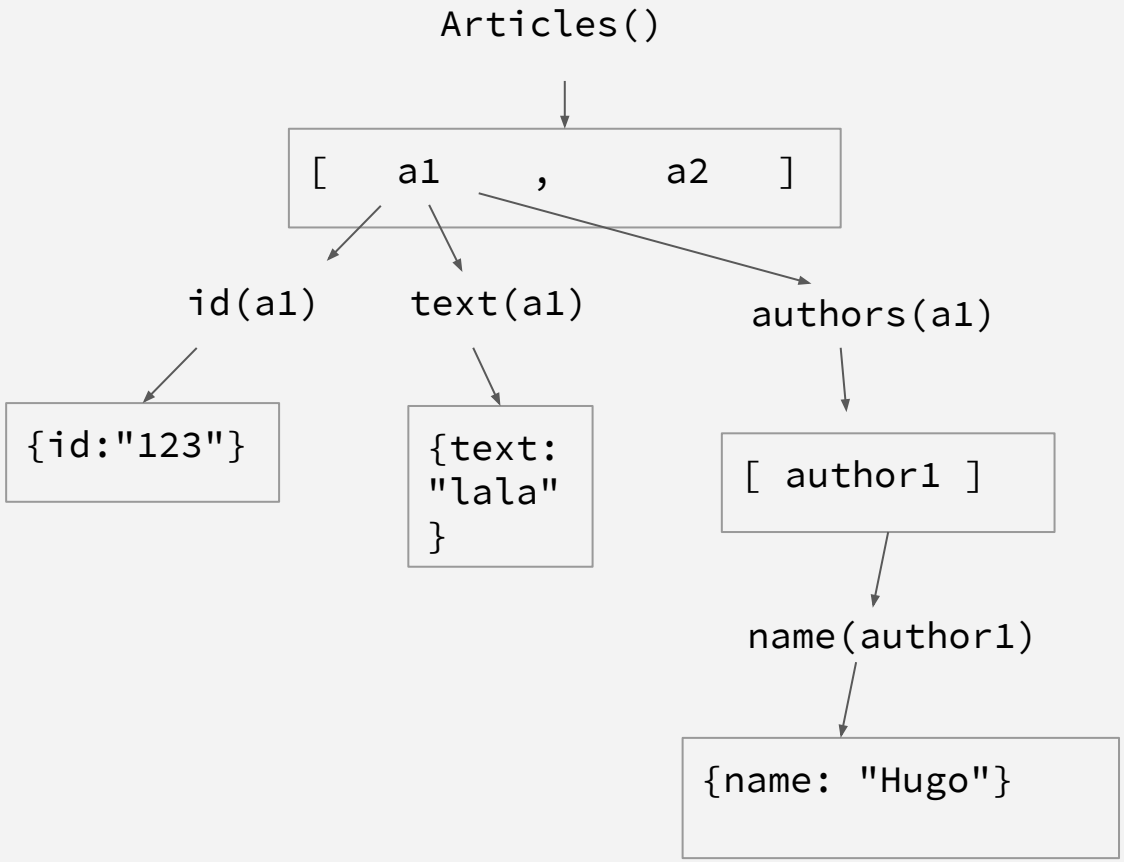
Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



Resolver

```
{
    Articles {
        id
        text
        authors {
            name
        }
    }
}
```



```
const AuthorType = new GraphQLObjectType({
    name: 'Author',
    fields: {
        id: { type: GraphQLID },
        name: { type: GraphQLString },
        articles: { type: new GraphQLList(ArticleType) }
    }
}
```

```
const AuthorType = new GraphQLObjectType({
    name: 'Author',
    fields: {
        id: { type: GraphQLID },
        name: {
            type: GraphQLString,
            resolve: function (parent) {
                return parent.getName();
            }
        },
        articles: { type: new GraphQLList(ArticleType) }
    }
}
```

```
const AuthorType = new GraphQLObjectType({
    name: 'Author',
    fields: {
        id: { type: GraphQLID },
        name: { type: GraphQLString },
        articles: { type: new GraphQLList(ArticleType) }
    }
}
```

```
const AuthorType = new GraphQLObjectType({
    name: 'Author',
    fields: {
        id: { type: GraphQLID },
        name: { type: GraphQLString },
        articles: {
            type: new GraphQLList(ArticleType),
            resolve: function(parent) {
                return sql(`SELECT a.* FROM
                            Article a
                            WHERE a.authorId = $(parent.id)`)
            }
        }
    }
}
```

Wo kommen die Daten her?

- für jedes Feld existiert eine Resolver-Funktion
  - skalare Typen → "Getter" by default
  - komplexe Typen → selbst implementieren bzw. Frameworks einsetzen
- Dadurch beliebige Daten-Quellen nutzbar
- Automatisierung durch Frameworks und Libraries

GraphQL JPA

- https://github.com/jcrygier/graphql-jpa

- Bibliothek generiert GraphQL-Schema aus JPA Entities inkl. Resolver

- Allerdings: relativ wenig Weiterentwicklung

Fazit

Aktueller Stand

- GraphQL-Spec existiert
- Mehrere PAAS-Anbieter existieren
- Ein paar Firmen migrieren (z.B. Github)
- Referenz-Implementierung in JavaScript wird aktiv entwickelt
- Implementierungen in vielen Sprachen verfügbar
- Java-Implementierungen erfahren etwas weniger Aufmerksamkeit

Vorteile

- Minimierung von Netzwerk-Requests
- Minimierung von übertragenen Daten
- Interessante Möglichkeiten für Entwickler
  - Statische Typisierung des Schemas
  - API-Explorer
- Neue Frontend-Patterns ermöglichen saubere Komponenten-Trennung

Nachteile / Offene Fragen

- Framework-Support außerhalb von JavaScript
- Microservices? API-Gateway?
- Requests sind größer
- N+1 Problem zum Server verschoben
  - Lösung: DataFetcher (Sammeln und Optimieren der DB-Queries auf Serverseite)
- Rekursive Queries / Nested-Queries

Rekursive Queries

```
{
  Article {
    authors {
       articles {
          authors {
             articles {
               authors {
                 …
               }
             }
           }
         }
```

Nachteile / Offene Fragen

- Framework-Support außerhalb von JavaScript
- Microservices? API-Gateway?
- Requests sind größer
- Server-Last eventuell höher
- Keine Subscription/Reaktivität in der Spec
- N+1 Problem zum Server verschoben
  - Lösung: DataFetcher
- Rekursive Queries
  - potentielles Sicherheits/Performance-Risiko
  - Lösung: Nesting-Level begrenzen

Vorteile von REST+Hypermedia

- Etablierte Technologie
- Breite Tool- und Framework-Unterstützung
- Caching auf Netzwerkebene
- Server kann Client steuern → Update auf Server ändert Client-Verhalten
- Dynamische Änderungen

"REST" ohne Hypermedia? 

Fragen?

@manuel_mauky
github.com/lestard
www.lestard.eu

Saxonia Systems
So geht Software.