

NILS HARTMANN
<https://nilshartmann.net>

Fullstack

GraphQL

mit Apollo und React

git clone <https://github.com/nilshartmann/graphql-workshop>

npm run setup

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

- 1. GraphQL Grundlagen: wieso, weshalb, warum**
- 2. Ein GraphQL Backend mit Apollo**
- 3. GraphQL Frontends mit React und TypeScript**

Jederzeit: Fragen, und Diskussionen!
Traut euch 😊!

TEIL 1

GraphQL

Grundlagen

*"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

11:46 - 22. Mai 2017

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

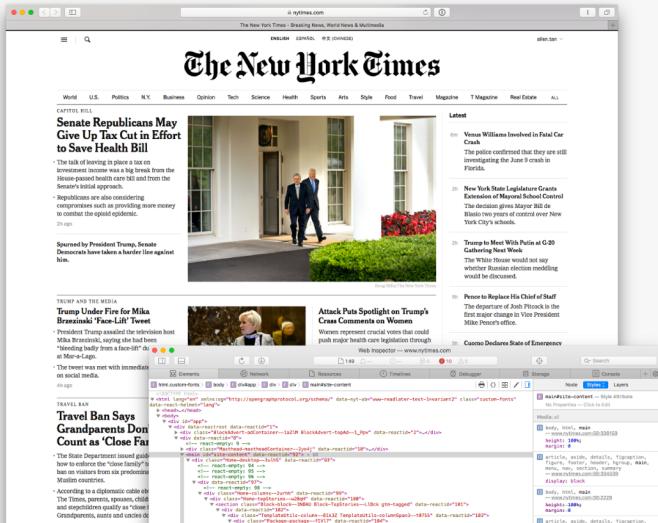
GITHUB



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

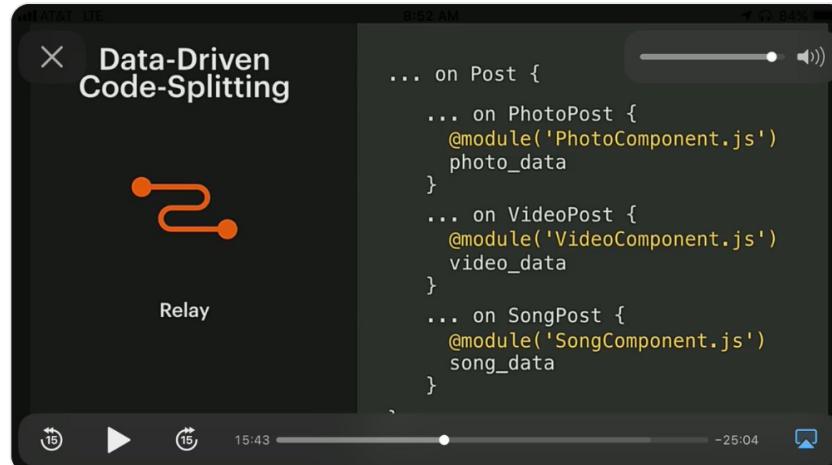


Nick Schrock
@schrockn

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.

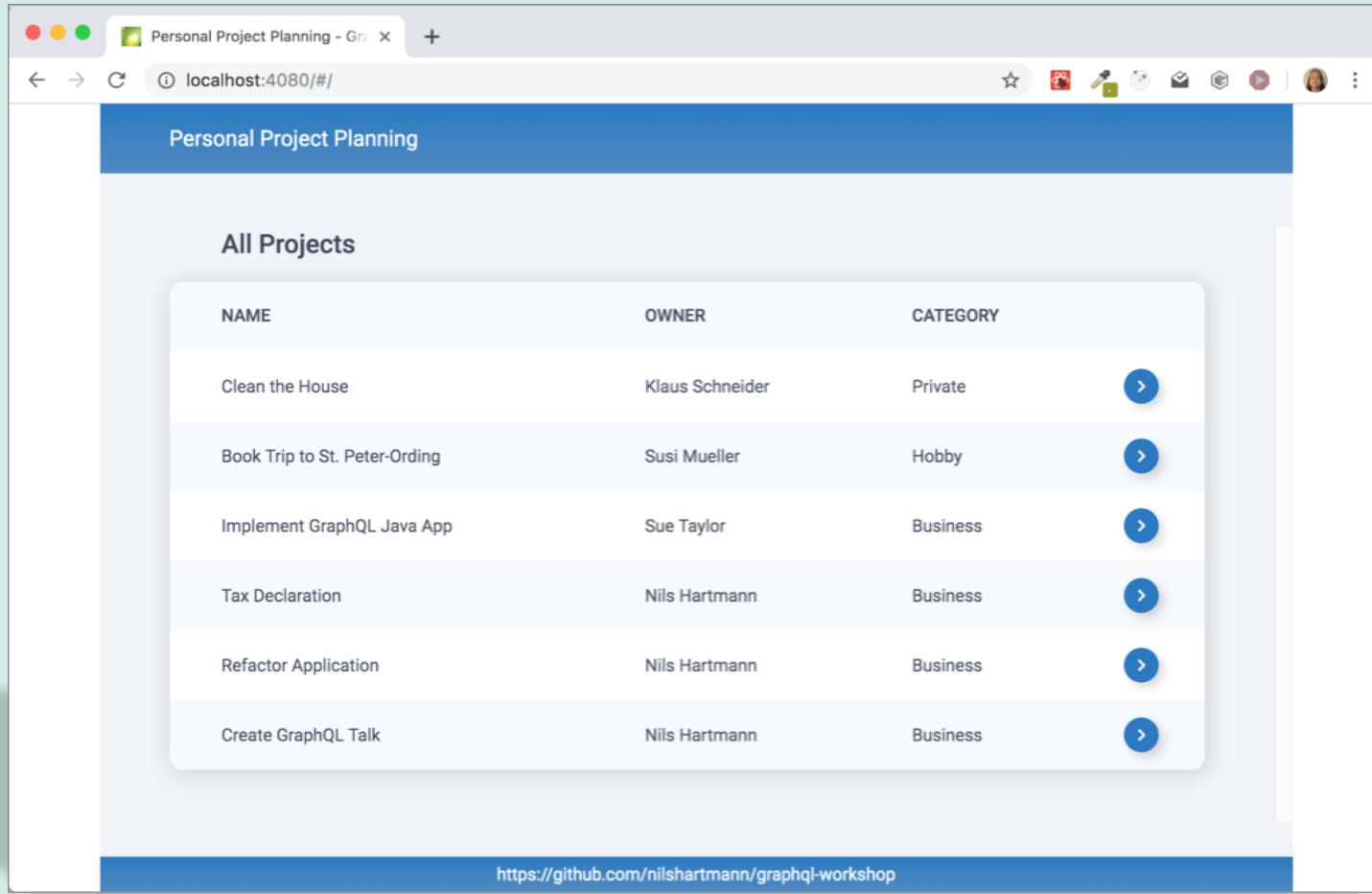
Tweet übersetzen



18:06 - 1. Mai 2019

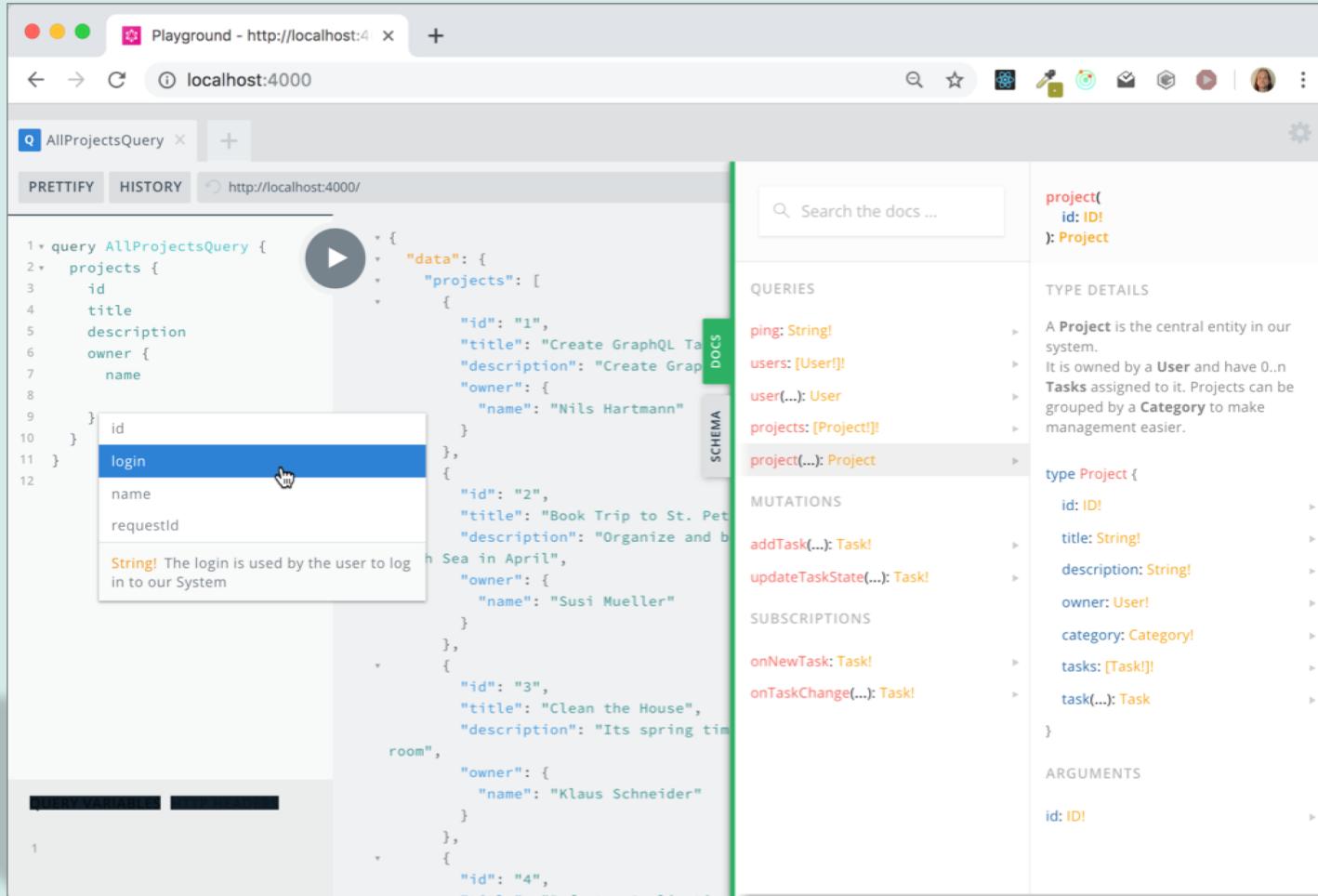
<https://twitter.com/schrockn/status/1123619660732047360>

NEXT GEN GRAPHQL?



Die Beispiel Anwendung

<http://localhost:4080>



Demo: Playground

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

<http://localhost:4000>

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

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

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

The cursor is positioned at the end of the first line of the query. A tooltip box is open, listing suggestions for the word 'beer'. The suggestions are:

- f beer - Returns the Beer with the specified Id [Beer!]!
- f beers - Returns all beers in our store [Beer!]!
- f ping - Returns health information about t... [ProcessInfo!]
- f ratings - All ratings stored in our system [Rating!]!
- f __schema - Access the current type schema of... [__Schema!]
- f __type - Request the type information of a sing... [__Type]

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

At the bottom right of the code 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

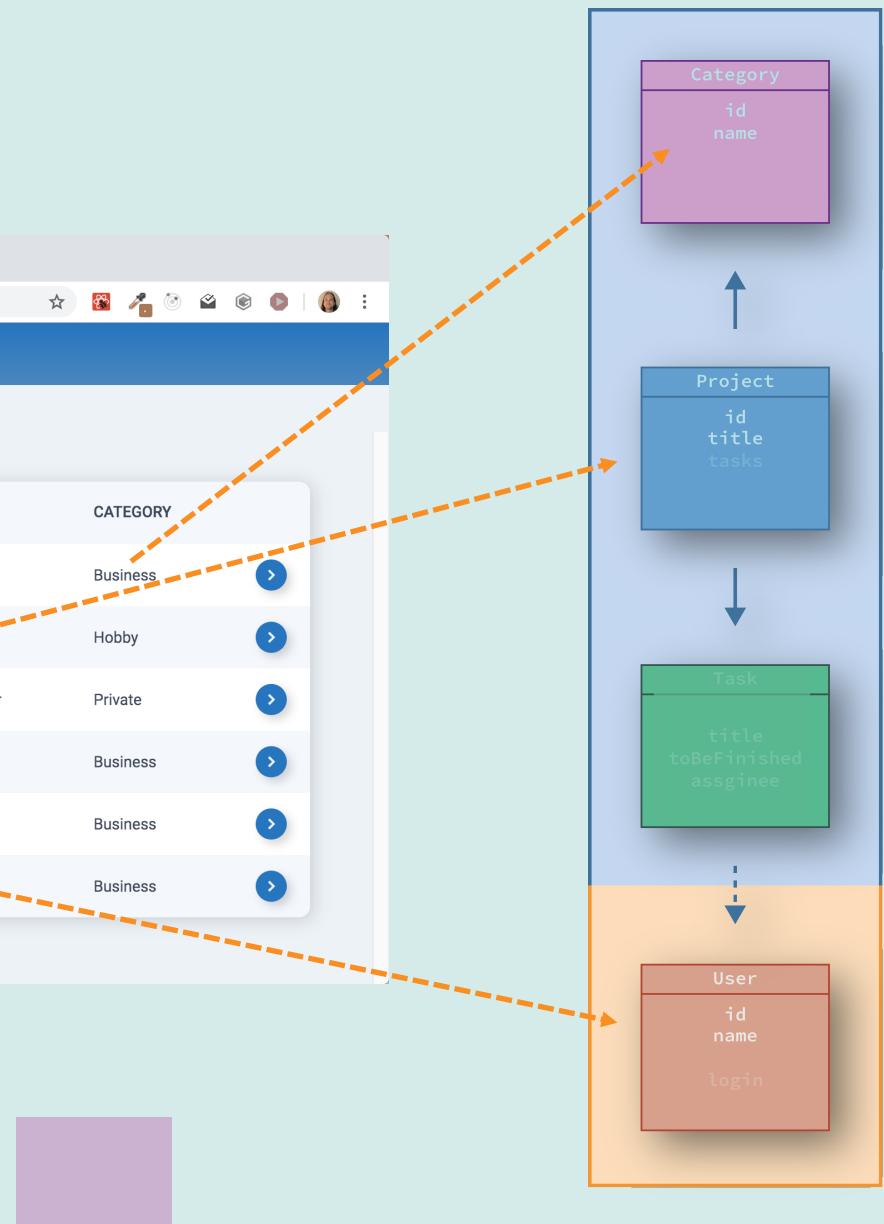
TEIL 1: ABFRAGEN UND SCHEMA

GRAPHQL EINSATZSzenariEN

Use-Case spezifische Abfragen – 1

```
{ projects {  
  id  
  name  
  owner { name }  
  category { name }  
}
```

NAME	OWNER	CATEGORY
Create GraphQL Talk	Nils Hartmann	Business
Book Trip to St. Peter-Ording	Susi Mueller	Hobby
Clean the House	Klaus Schneider	Private
Refactor Application	Nils Hartmann	Business
Tax Declaration	Nils Hartmann	Business
Implement GraphQL Java App	Sue Taylor	Business



GRAPHQL EINSATZSzenariEN

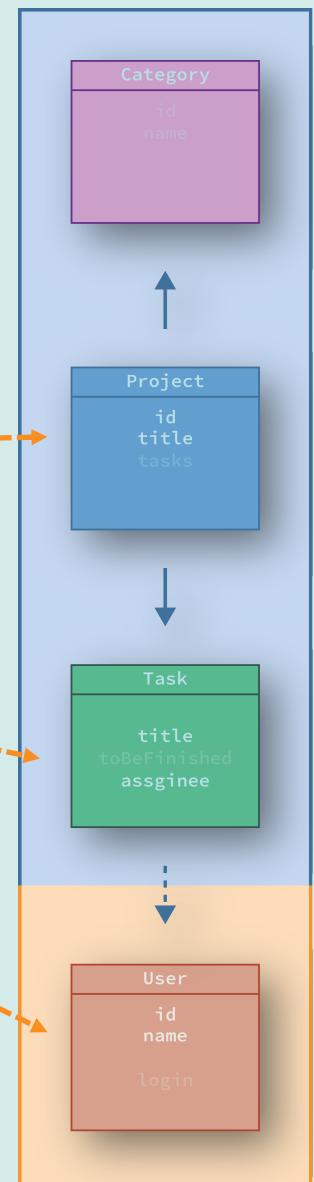
Use-Case spezifische Abfragen – 2

```
{ project(...) {  
  title  
  tasks {  
    name  
    assignee { name }  
    state  
  }  
}
```

A screenshot of a web browser displaying a GraphQL query results table titled "All Projects > Create GraphQL Talk Tasks". The table has three columns: NAME, ASSIGNEE, and STATE. It lists three tasks:

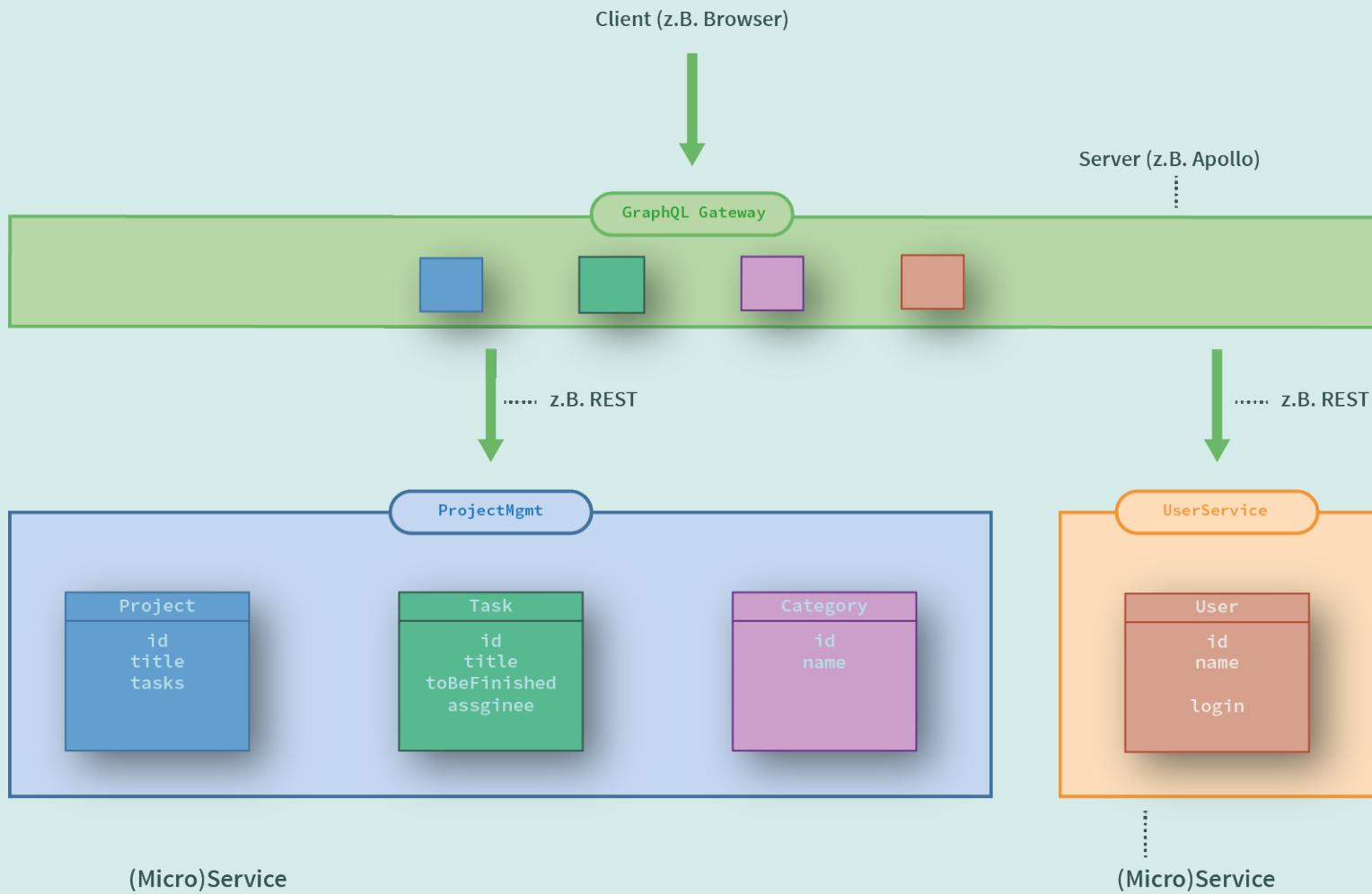
NAME	ASSIGNEE	STATE
Create a draft story	Nils Hartmann	In Progress
Finish Example App	Susi Mueller	In Progress
Design Slides	Nils Hartmann	New

An "Add Task >" button is located at the bottom right of the table. Dashed orange arrows point from the "NAME" column of each row to the "name" field in the GraphQL query above.



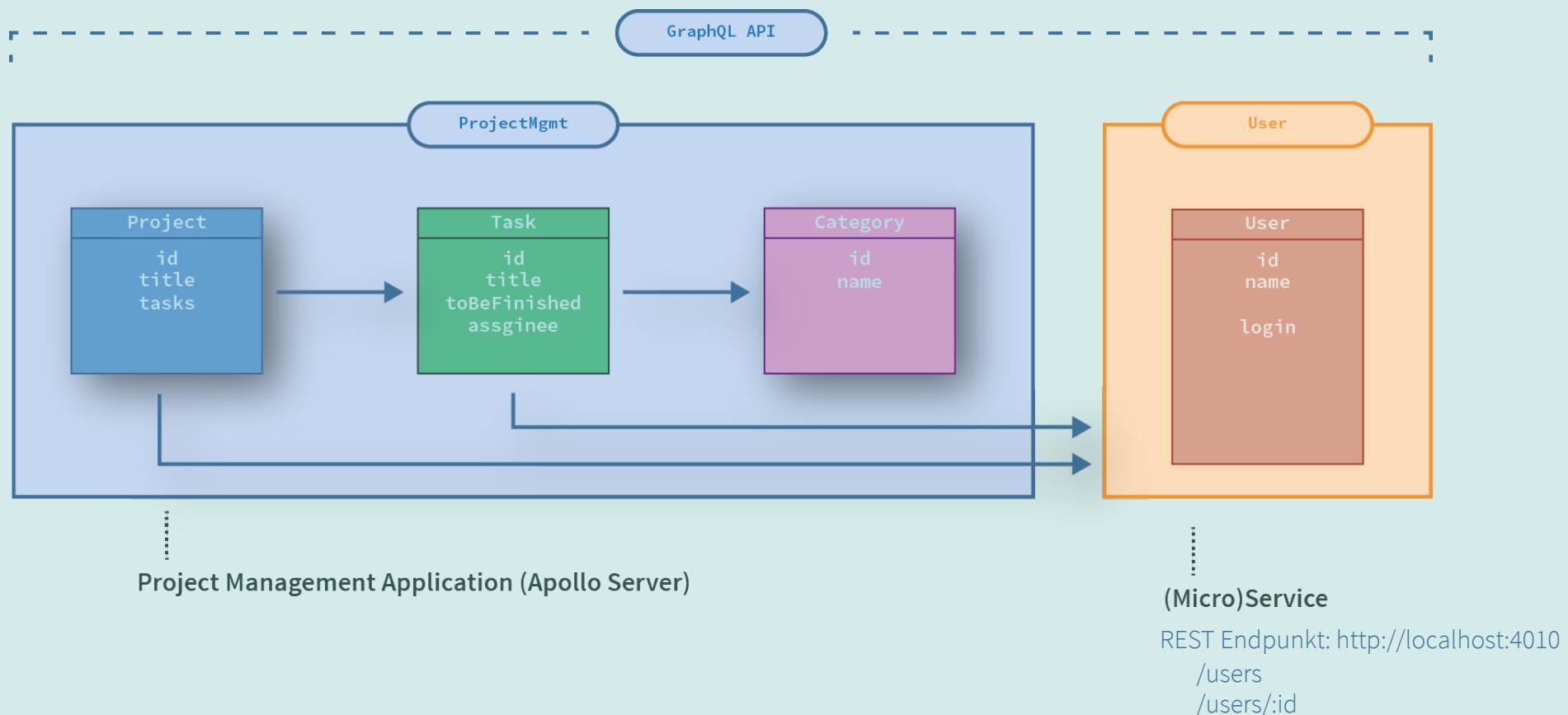
EINSATZSzenariEN

- Gateway für Frontend zu mehreren Backends



GRAPHQL EINSATZSzenariEN

"Architektur" Project Beispiel Anwendung



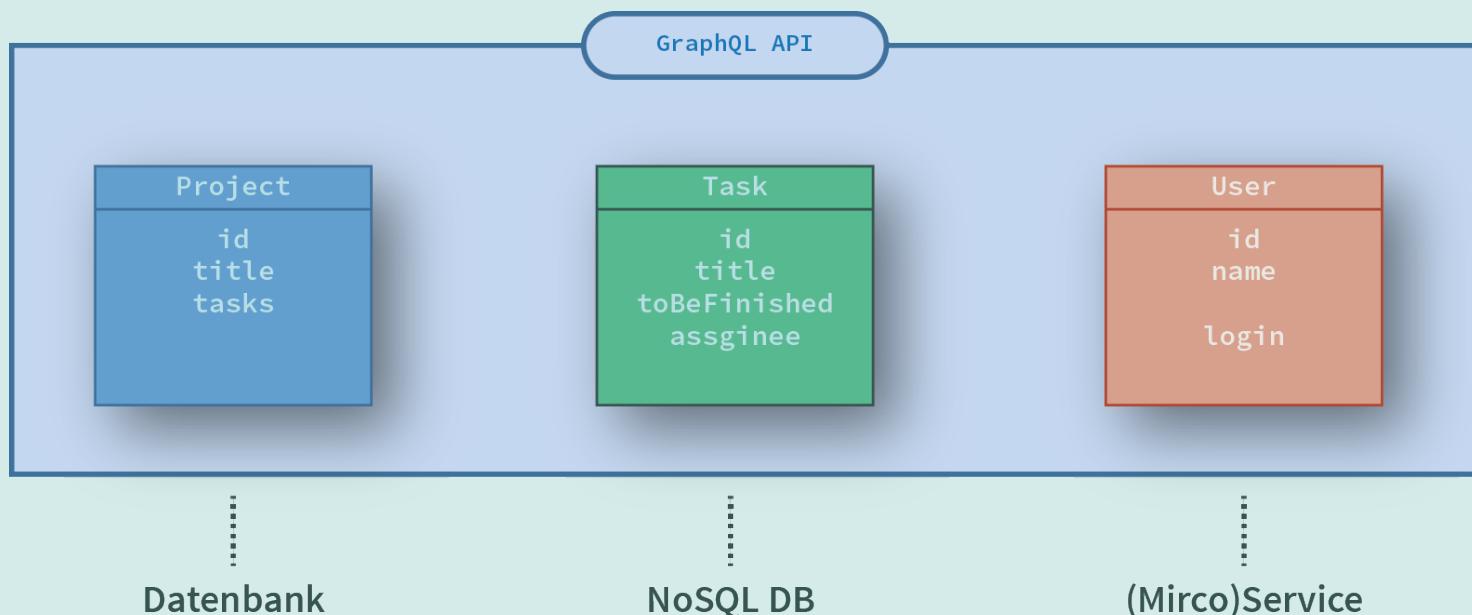
Gründe für den Einsatz von GraphQL

- 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

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*



Die GraphQL Query Sprache

QUERY LANGUAGE

```
{  
  project : {  
    id :  
    title :  
    tasks : {  
      title :  
      state :  
    }  
  }  
}
```

The diagram illustrates a query language structure. A blue arrow labeled "Fields" points from the left towards a JSON-like object. The object starts with an opening brace {}, followed by a key "project" enclosed in a blue box, which is preceded by another brace {}, indicating it's a nested object. Inside "project", there are three keys: "id", "title", and "tasks". The "tasks" key is also enclosed in a blue box and has its own brace {}, suggesting it might be a list or another object. Inside "tasks", there are two more keys: "title" and "state", both enclosed in blue boxes and preceded by braces {}, indicating they are individual fields.

- Strukturierte Sprache, um Daten von der API abzufragen
- Abgefragt werden **Felder** von (verschachtelten) Objekten

QUERY LANGUAGE

```
{  
  project(projectId: "P1") {  
    id  
    title  
    tasks {  
      title  
      state  
    }  
  }  
}
```

Fields

Argumente

- Strukturierte Sprache, um Daten von der API abzufragen
- Abgefragt werden **Felder** von (verschachtelten) Objekten
- Felder können **Argumente** haben

QUERY LANGUAGE

Ergebnis

```
{  
  project(projectId: "P1") {  
    id  
    title  
    tasks {  
      title  
      state  
    }  
  }  
}
```



```
"data": {  
  "project": {  
    "id": "P1"  
    "title": "GraphQL Talk"  
    "tasks": [  
      {  
        "state": "IN_PROGRESS",  
        "title": "Create Story"  
      },  
      {  
        "state": "NEW",  
        "title": "Finish Example"  
      }  
    ]  
  }  
}
```

- Identische Struktur wie bei der Abfrage

QUERY LANGUAGE: OPERATIONS

Operation: beschreibt, was getan werden soll

- query, mutation, subscription

Operation type

Operation name (optional)

```
query GetProject {  
  project(projectId: "P1") {  
    id  
    title  
    owner { name }  
  }  
}
```

QUERY LANGUAGE: OPERATIONS

Operation: Variablen

```
query GetProject($pid: ID!) {  
  project(projectId: $pid) {  
    id  
    title  
    owner { name }  
  }  
}
```

Variable Definition
|
query GetProject(**\$pid: ID!**) {
 project(projectId: **\$pid**) {
 id
 title
 owner { name }
 }
}
}|
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 AddTaskMutation(pid: ID!, $input: AddTaskInput!) {
  addTask(projectId: ID!, input: $input) {
    id
    title
    state
  }
}

"input": { — Variable Object
  title: "Create GraphQL Example",
  description: "Simple example application",
  author: "Nils",
  toBeFinishedAt: "2019-07-04T22:00:00.000Z",
  assigneeId: "U3"
}
```

QUERY LANGUAGE: MUTATIONS

Subscription

- Automatische Benachrichtigung bei neuen Daten

```
Operation type
  |
  | Operation name (optional)
  |
subscription NewTaskSubscription {
  newTask: onNewTask {
    Field alias
    | id
    title
    assignee { id name }
    description
  }
}
```

QUERIES AUSFÜHREN

Queries werden über HTTP ausgeführt

- Üblicherweise per POST
- Ein einzelner Endpoint (bei Apollo: /, sonst meist /graphql)

```
$ curl -X POST -H "Content-Type: application/json" \
-d '{"query":"{ projects { title } }}'" \
http://localhost:4000/
```

```
{"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: QUERIES AUSFÜHREN

Mach dich mit dem Playground und der Query-Sprache vertraut

- Öffne den Playground auf meinem Computer (IP steht auf der Tafel)
1. Mach' dich mit der API des Projektes vertraut
 2. Führe einen Query aus, mit dem Du alle Projekte und alle Benutzer (insb. jeweils die IDs) erhältst
 3. Führe eine Mutation aus, mit der Du eine neue Aufgabe ("Task") einem bestehenden Projekte ("Project") hinzufügst

TEIL II

GraphQL Server

*"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
 - Das ist bei anderen GraphQL-Frameworks anders

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
 - Im Playground sollte auch das Schema automatisch aktualisiert werden
- 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/>
 - Schema sollte automatisch aktualisiert werden
 - (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
- Parent-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
- Parent-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, dieselbe 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**: Source-Objekt (oder ROOT_QUERY bei Root-Feldern)
- **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

Root-Resolver mit
Argumenten

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

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();  
  
  }  
  
}
```

-----► (GET http://localhost:4010/users)

-----► (GET http://localhost:4010/users/**ID**)

```
}
```

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)

```
}
```

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, delete, ... 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 und Payload 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 gemäß cache-control Header
 - Service kann Cache-Dauer bestimmen (HTTP Standard)

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 gemäß cache-control Header
 - Service kann Cache-Dauer bestimmen (HTTP Standard)
- 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

(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!)

TEIL III

GraphQL Clients

TYP-SICHERE CLIENTS MIT REACT, APOLLO UND TYPESCRIPT

```
{  
  projects {  
    id title  
    owner { name }  
    category { name }  
  }  
}
```

The screenshot shows a web browser window with the title "Personal Project Planning - GraphQL" and the URL "localhost:4080/#/". The main content area is titled "All Projects" and displays a table with six rows of project data. The columns are labeled "NAME", "OWNER", and "CATEGORY". Each row has a blue circular arrow icon on the right side.

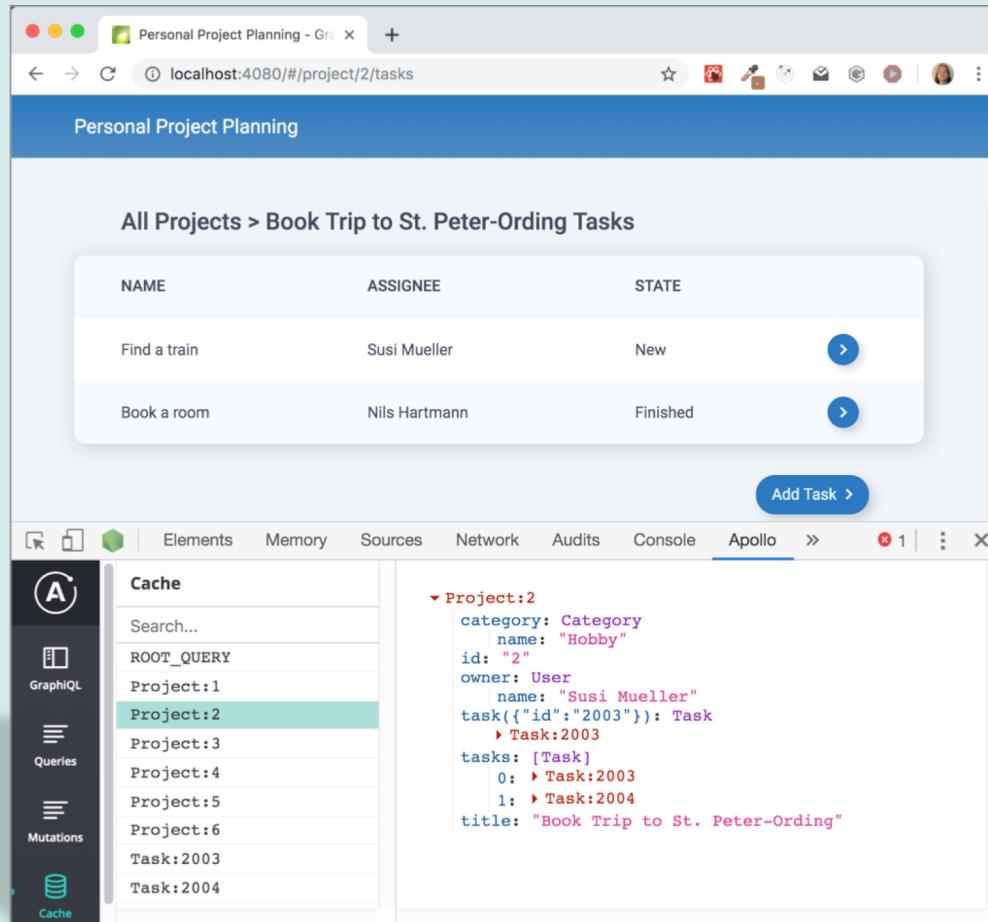
NAME	OWNER	CATEGORY
Create GraphQL Talk	Nils Hartmann	Business
Book Trip to St. Peter-Ording	Susi Mueller	Hobby
Clean the House	Klaus Schneider	Private
Refactor Application	Nils Hartmann	Business
Tax Declaration	Nils Hartmann	Business
Implement GraphQL Java App	Sue Taylor	Business

DIE PROJEKT ÜBERSICHT

```
class ProjectsPageWithoutApollo extends React.Component {  
  state = { projects: [] };  
  
  componentDidMount() {  
    fetch("http://localhost:4000", {  
      method: "POST",  
      body: JSON.stringify({  
        query: `  
          { projects {  
            id title  
            owner { name }  
            category { name }  
          }  
        }`  
      })  
    })  
    .then(res => res.json())  
    .then(({ data }) => this.setState({ projects: data.projects }));  
  }  
  
  render() {  
    return <ProjectsTable projects={this.state.projects} />;  
  }  
}
```

Diese Komponente funktioniert! Aber...
was fehlt? welche Probleme kann es geben?

Demo: Apollo Dev Tools



APOLLO CLIENT

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

- React-Komponenten zum Zugriff auf GraphQL APIs
 - funktioniert mit allen GraphQL Backends
 - es gibt Apollo Componenten auch für Angular, Vue, ...
<https://github.com/apollographql/apollo-client>
- Besonderheit: Anwendungsweiter, globaler **Cache** sorgt für konsistente Darstellung der Daten

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

```
import { gql } from "react-apollo";  
  
const PROJECTS_PAGE_QUERY = gql`  
  query ProjectsPageQuery {  
    projects {  
      id  
      title  
      owner { name }  
  
      catagory { name }  
    }  
  }  
`;
```

Query parsen

SCHRITT 2: QUERIES

Query-Komponente

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

```
import { gql, Query } from "react-apollo";  
  
const PROJECTS_PAGE_QUERY = gql`...`;  
  
function ProjectsPage(props) {  
  return <Query query={PROJECTS_PAGE_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 PROJECTS_PAGE_QUERY = gql`...`;  
  
function ProjectsPage(props) {  
  return <Query query={PROJECTS_PAGE_QUERY}>  
    {({ loading, error, data }) => {  
      ...  
    }}  
  </Query>  
};
```

**Query Ergebnis als
Render-Props**
(wird 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 PROJECTS_PAGE_QUERY = gql`...`;

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

      return <ProjectsTable projects={data.projects} />
    }}
  </Query>
};
```

Ergebnis (samt Fehler)
auswerten

SCHRITT 2: QUERIES

Query-Komponente: Optionen

- Query-Komponente kann weitere Optionen entgegennehmen
- Beispiel: Variablen

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

const QUERY = `const TASK_LIST_PAGE_QUERY = gql`  
  query TaskListPageQuery($projectId: ID!) {  
    project(id: $projectId) { . . . }  
  }  
`;
```

SCHRITT 2: QUERIES

Query-Komponente: Optionen

- Query-Komponente kann weitere Optionen entgegennehmen
- Beispiel: Variablen

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

const QUERY = `const TASK_LIST_PAGE_QUERY = gql`  
  query TaskListPageQuery($projectId: ID!) {  
    project(id: $projectId) { . . . }  
  }  
`;

function ProjectListPage(props) {  
  return <Query query={QUERY} variables={  
    {projectId: props.projectId}}>  
    ...  
  </Query>  
};
```

SCHRITT 2: QUERIES

Query-Komponente: Optionen

- Beispiel: `fetchPolicy`
- Legt fest, wann Requests ausgeführt werden

```
function ProjectListPage(props) {  
  return <Query query={QUERY} variables={  
    {projectId: props.projectId}}  
    fetchPolicy="network-only">  
    ...  
  </Query>  
};
```

- Mögliche Werte:
`cache-first`, `cache-and-network`, `network-only`, `cache-only`, `no-cache`

SCHRITT 2: QUERIES

Alternative: useQuery Hook (Alpha!)

- Support für React Hook API in Alpha Version verfügbar
<https://github.com/apollographql/react-apollo/tree/release-3.0.0/packages/hooks>
- Optionen als 2. Parameter (Variablen etc)

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

const QUERY = gql`...`;

function ProjectsPage(props) {
  const { loading, error, data } = useQuery(QUERY,
    { variables: ... },
  );

  if (loading) { return <h1>Loading...</h1> }
  if (error) { return <h1>Error!</h1> }

  return <ProjectsTable projects={data.projects} />
}
```

Mit TypeScript: Typ-sicherer Zugriff auf Ergebnis

- Wird typisiert mit Query-Resultat
- TS Types für Queries werden mit apollo client:codegen generiert

```
export interface ProjectsPageQuery_projects {  
    __typename: "Project";  
    id: string;  
    /**  
     * A simple, concise title for your project  
     */  
    title: string;  
    /**  
     * The project owner  
     */  
    owner: ProjectsPageQuery_projects_owner;  
    category: ProjectsPageQuery_projects_category;  
}  
  
export interface ProjectsPageQuery {  
    /**  
     * Return an unordered list of all projects  
     */  
    projects: ProjectsPageQuery_projects[];  
}
```

TYPSICHERES GRAPHQL

Mit TypeScript: Typ-sicherer Zugriff auf Ergebnis

- Beispiel Schritt-für-Schritt in code-frontend/workspace/TaskPage.tsx

TYPSICHERES GRAPHQL

Mit TypeScript: Typ-sicherer Zugriff auf Ergebnis

- Wird typisiert mit Query-Resultat

Generierter Type

```
import { gql, Query } from "react-apollo";
import { ProjectsPageQuery } from "./querytypes/...";
```

```
function ProjectsPage(props) {
  return <Query<ProjectsPageQuery> query={...}>
    {({ loading, error, data }) => {
```

OK

```
      const title = data.projects[0].title;
```

Compile-Fehler!

```
      return <ProjectsTable projects={data.projekte} />
    //}
  </Query>
};
```

TYPSICHERES GRAPHQL

Mit TypeScript: Typ-sicherer Zugriff auf Ergebnis

- Wird typisiert mit Query-Resultat und ggf. Variablen

Generierte Types import { gql, Query } from "react-apollo";
 import { TaskPageQuery, TaskPageQueryVariables } from "...";

OK

```
<Query<TaskPageQuery, TaskPageQueryVariables>
  query={...}
  variables={{ projectId:"p1", taskId: "t1"}}>
  >...</Query>
};
```

Compile-Fehler!

```
<Query<TaskPageQuery, TaskPageQueryVariables>
  query={...}
  variables={{ projektId:"p1", taskId: null}}>
  >...</Query>
};
```

Mit TypeScript: Mit Hooks

- Wird typisiert mit Query-Resultat und ggf. Variablen

```
import { TaskPageQuery, TaskPageQueryVariables } from "...";  
  
const { loading, error, data } =  
  useQuery<TaskPageQuery, TaskPageQueryVariables>  
  (TASK_QUERY, { variables: { projectId: "...", taskId: "..." }});
```

Mit TypeScript: Mit Hooks

- Wird typisiert mit Query-Resultat und ggf. Variablen

```
import { TaskPageQuery, TaskPageQueryVariables } from "...";  
  
const { loading, error, data } =  
  useQuery<TaskPageQuery, TaskPageQueryVariables>  
    (TASK_QUERY, { variables: { projectId: "...", taskId: "..." }});
```

Compile-Fehler! const title = data.project.title;
(data might be null)

Mit TypeScript: Mit Hooks

- Wird typisiert mit Query-Resultat und ggf. Variablen

```
import { TaskPageQuery, TaskPageQueryVariables } from "...";  
  
const { loading, error, data } =  
  useQuery<TaskPageQuery, TaskPageQueryVariables>  
    (TASK_QUERY, { variables: { projectId: "...", taskId: "..." }});
```

Compile-Fehler! const title = data.project.title;
(data might be null)

OK const title = data ? data.project.title : null;

Mit TypeScript: Mit Hooks

- Wird typisiert mit Query-Resultat und ggf. Variablen

```
import { TaskPageQuery, TaskPageQueryVariables } from "...";  
  
const { loading, error, data } =  
  useQuery<TaskPageQuery, TaskPageQueryVariables>  
    (TASK_QUERY, { variables: { projectId: "...", taskId: "..." }});
```

Compile-Fehler! const title = data.project.title
(data might be null)

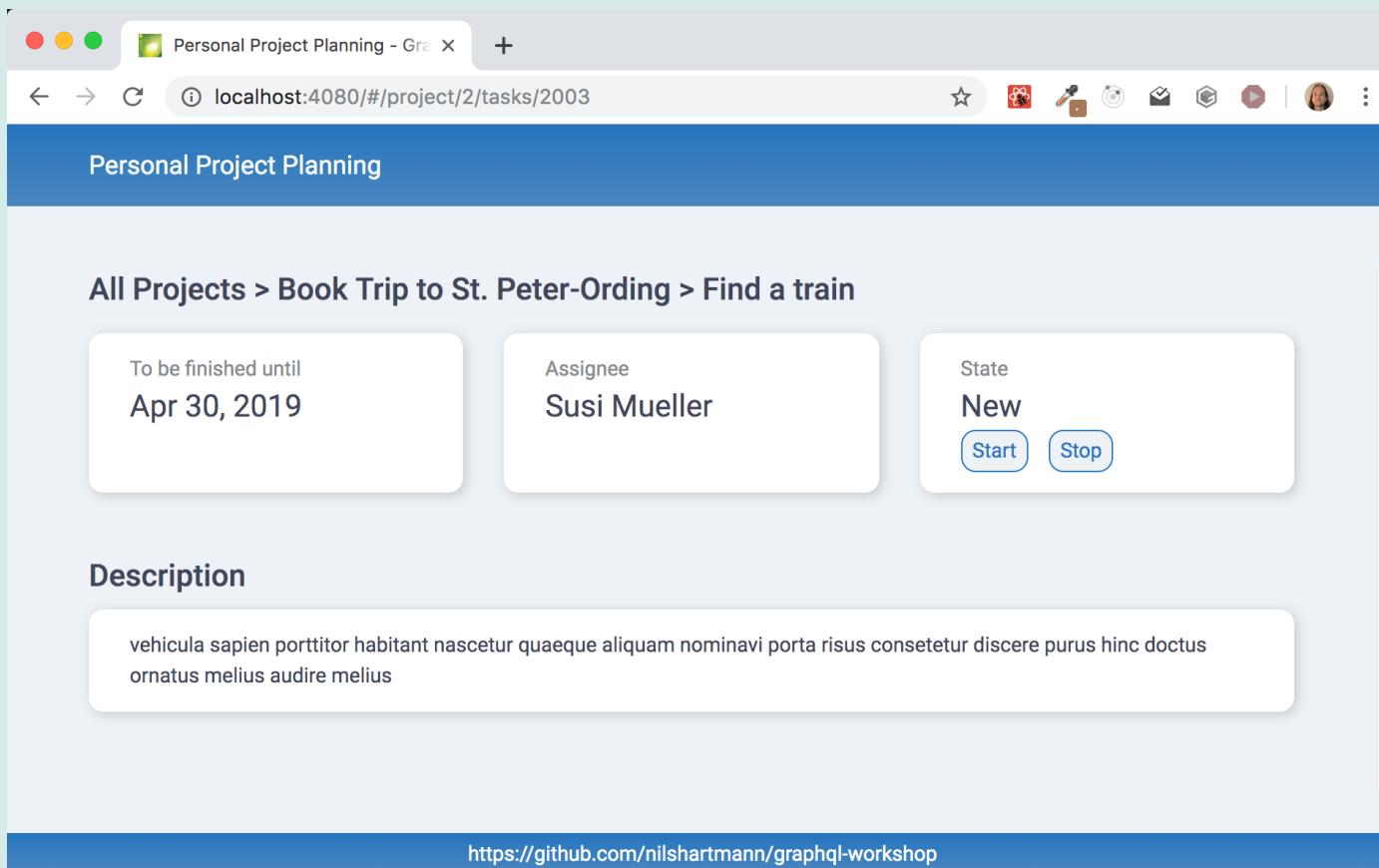
OK const title = data ? data.project.title : null;

Compile-Fehler! const title = data ? data.projekt.title : null;

ÜBUNG 1: EINE GRAPHQL QUERY

Die Task-Ansicht bauen, Teil 1

- Lade die Daten zu dem aktiven Task und übergebe sie an die TaskView-Komponente



TaskPage.tsx

ÜBUNG 1: EINE GRAPHQL QUERY

Vorbereitung (gemeinsam)

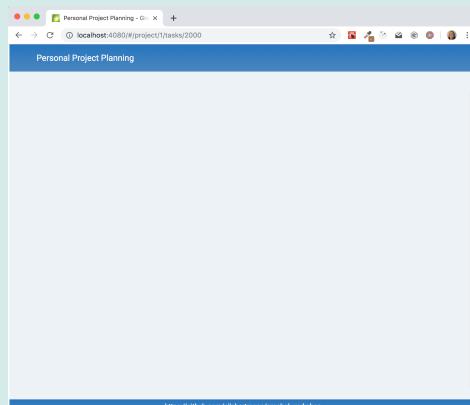
- Läuft dein Server noch? Sehr gut! Sonst bitte starten!
- Falls noch nicht geschehen:
 - `cd code-frontend/workspace`
 - `npm install`
 - `npm start`
- Frontend sollte jetzt unter `http://localhost:4080` automatisch im Browser aufgehen
- Bitte öffne `code-frontend/workspace` in deiner IDE/Editor
- Starte noch zusätzlich `npm run codegen:watch` im workspace-Verzeichnis

ÜBUNG 1: EINE GRAPHQL QUERY

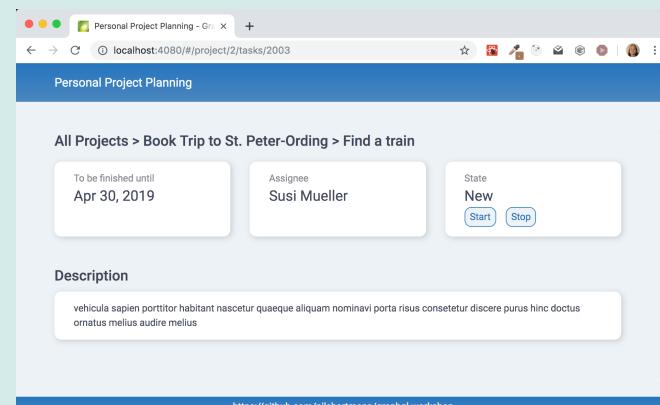
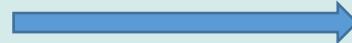
Die Task-Ansicht bauen

In TaskPage.tsx musst Du einen Query definieren, ausführen und das Ergebnis verarbeiten

- In der Datei **TaskPage.tsx** sind TODOs eingetragen
 - Achtung! Nur "ÜBUNG 1", TODO 1-4 machen (Übung 2 ignorieren)
- Wenn Du die Datei speicherst:
 - wird sie automatisch im Browser aktualisiert
 - werden die TypeScript-Definition automatisch generiert



Vorher 😰



Nachher ❤️

MUTATIONS

SCHRITT 3: MUTATIONS

Mutation-Komponente: Führt Mutations aus

- Mutation wird ebenfalls per gql geparsst

```
const ADD_TASK_MUTATION = gql`  
  mutation AddTaskMutation($projectId: ID!, $newTask: AddTaskInput!) {  
    addTask(projectId: $projectId, input: $newTask) {  
      id  
    }  
  }  
`;
```

SCHRITT 3: MUTATIONS

Mutation-Komponente: Führt Mutations aus

- Funktionsweise und API ähnlich wie Query-Komponente

```
import { gql, Mutation } from "react-apollo";  
  
const ADD_TASK_MUTATION = gql`...`;  
  
function AddTaskPage(props) {  
  return <Mutation mutation={ADD_TASK_MUTATION}>  
    </Mutation>  
}
```

SCHRITT 3: MUTATIONS

Mutation-Komponente: Führt Mutations aus

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

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

const ADD_TASK_MUTATION = gql`...`;

function AddTaskPage(props) {
  return <Mutation mutation={ADD_TASK_MUTATION}>
    {addTask => {
      }
    }
  </Mutation>
}
```

SCHRITT 3: MUTATIONS

Mutation-Komponente: Führt Mutations aus

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

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

const ADD_TASK_MUTATION = gql`...`;

function AddTaskPage(props) {
  return <Mutation mutation={ADD_TASK_MUTATION}>
    {addTask => {
      return <AddTaskForm onSave={
        newTask => addTask({
          ...newTask
        })
      } />
    }
  }
</Mutation>
}
```

SCHRITT 3: MUTATIONS

Mutation-Komponente: Führt Mutations aus

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

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

const ADD_TASK_MUTATION = gql`...`;

function AddTaskPage(props) {
  return <Mutation mutation={ADD_TASK_MUTATION}>
    {addTask => {
      return <AddTaskForm onSave={
        newTask => addTask({
          variables: {
            projectId: props.projectId, newTask
          }
        })
      } />
    }
  </Mutation>
}
```

SCHRITT 3: MUTATIONS

Mutation als Hook: useMutation

- useMutation liefert Array zurück:
 - 1. Parameter: Eine Methode, um die Mutation auszuführen
 - 2. Parameter: Objekt mit loading, error-State, wie bei Query

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

function AddTaskPage(props) {
  const [ addTask, { error, loading, data } ] =
    useMutation(ADD_TASK_MUTATION);

  if (error) { return "Mutation failed"; }

  return <AddTaskForm onSave={  
    newTask => addTask({  
      variables: {  
        projectId: props.projectId, newTask  
      })  
    } />  
}
```

SCHRITT 3: MUTATIONS

Mutation als Hook: useMutation

- Zum Zugriff auf das Ergebnis entweder data verwenden (wie gesehen)
- Oder: Promise mit Mutation-Ergebnis

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

function AddTaskPage(props) {
  const [ addTask ] = useMutation(ADD_TASK_MUTATION);

  return <AddTaskForm onSave={  

    newTask => addTask({  

      variables: {  

        projectId: props.projectId, newTask }  

      }).then(data => ...)  

    } />  

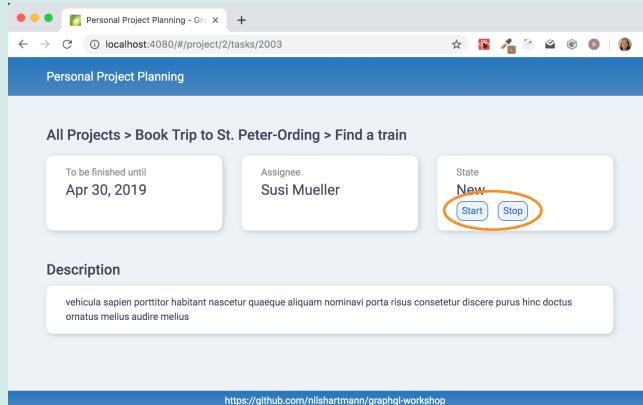
}
```

ÜBUNG 2: EINE GRAPHQL MUTATION

Die Task-Ansicht vervollständigen

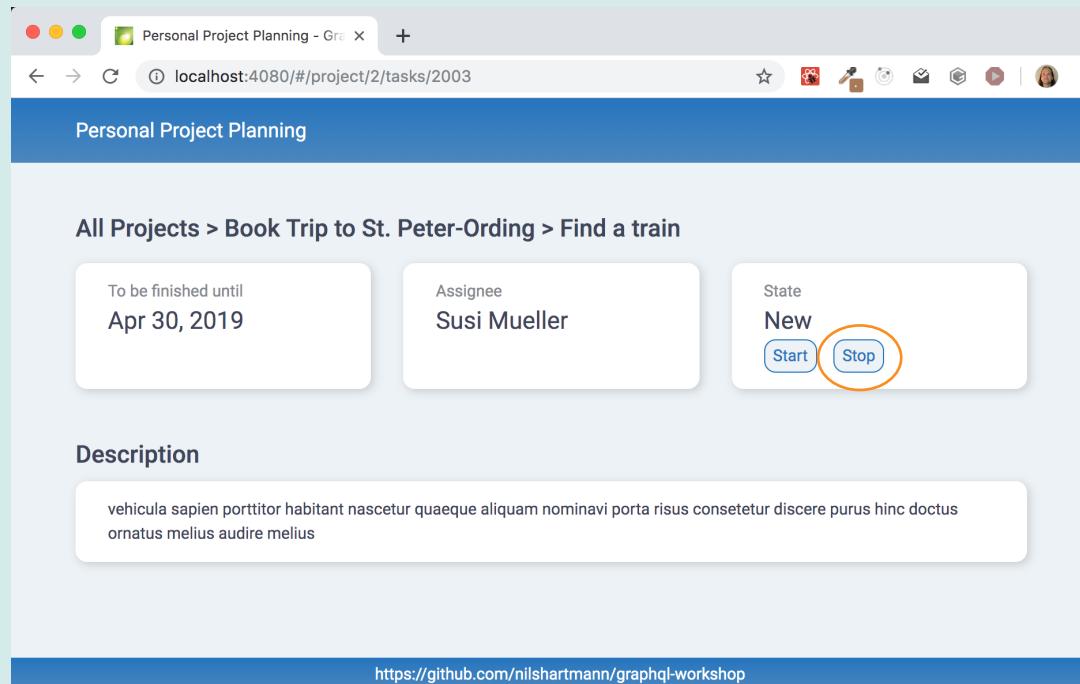
In `TaskPage.tsx` musst Du eine Mutation bauen, um den Zustand des Tasks zu aktualisieren. Der "Start" und "Stop" Knopf sollte danach funktionieren.

- In der Datei `TaskPage.tsx` jetzt die TODOs für "ÜBUNG 2" machen
- Falls Du in Übung 1 nicht fertig geworden ist:
 - kopiere Dateien in `code-frontend/01_useQuery_fertig`
 - in den Ordner `code-frontend/workspace/src/TaskPage`



CACHING

Was passiert wenn wir das Knöpfchen drücken?



CACHING

Der Apollo Cache

- Daten werden gelesen, normalisiert und in Cache geschrieben
- Deswegen immer "id"-Feld mit abfragen (und __typename, das wird aber transparent hinzugefügt)

The screenshot shows the Apollo DevTools interface with the Cache tab selected. On the left sidebar, under the Cache section, there is a list of objects: ROOT_QUERY, Project:1, Project:2, Project:3, Project:4, Project:5, Project:6, Task:2000, Task:2001, Task:2002, Task:2030, Task:2031, Task:2032, Task:2033, and Task:2034. A blue arrow points from the 'Project:1' entry in the sidebar to its expanded object in the main panel. The main panel displays the following JSON data:

```
▼ Project:1
  category: Category
    name: "Business"
  id: "1"
  owner: User
    name: "Nils Hartmann" Falsch! (keine Id abgefragt?)
  tasks: [Task]
    0: ▶ Task:2000
    1: ▶ Task:2001
    2: ▶ Task:2002
    3: ▶ Task:2030
    4: ▶ Task:2031
    5: ▶ Task:2032
    6: ▶ Task:2033
  title: "Create GraphQL Talk"
```

Annotations in orange text are present: 'Falsch! (keine Id abgefragt?)' appears twice next to the 'owner' field and the first task in the 'tasks' array. 'Richtig!' appears once next to the second task in the 'tasks' array.

Der Apollo Cache

- Nach jedem Lesen von Daten wird der Cache aktualisiert
- Alle Komponenten, die daraus Daten beziehen (useQuery, ...) werden aktualisiert
- Deswegen muss der TaskPageQuery auch nicht erneut ausgeführt werden, da die Mutation den aktualisierten Task samt Id zurückbekommt

DATEN AKTUALISIEREN

Cache aktualisieren 1: Nach jedem *Lesen* von Daten wird der Cache aktualisiert

- Erneutes Lesen von Daten eines **Queries**...
 - ...beim erneuten Ausführen eines Queries
 - ...per Zeit-Intervall (Property pollInterval)
 - ...explizit, z.B. nach Interaktion (refetch-Parameter)

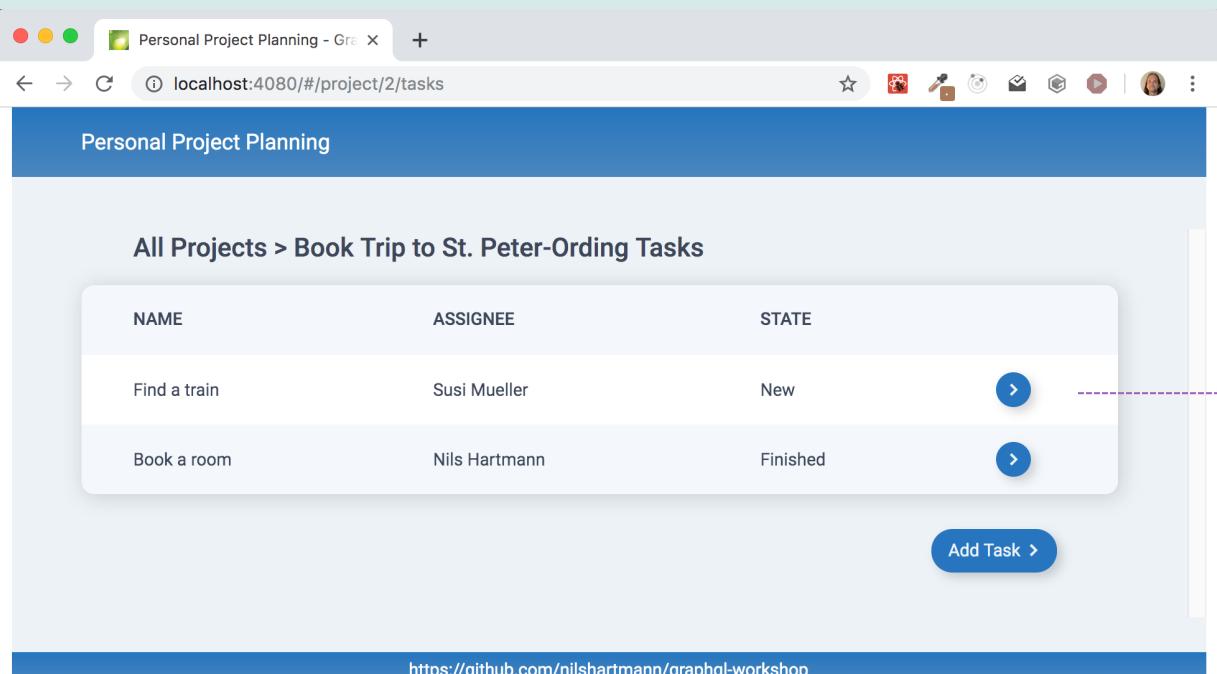
```
const { data, ..., refetch } = useQuery(...);<button onClick={refetch}>Refresh</button>
```
 - durch Subscription (subscribeToMany)
- Erneutes Lesen von Daten einer **Mutation**:
 - ...beim erneuten Ausführen der Mutation
 - ...nach Mutation können Queries erneut ausgeführt werden (refetchQueries)

DATEN AKTUALISIEREN

Cache aktualisieren 2: Cache wird *per API* aktualisiert

SUBSCRIPTIONS

Subscriptions: Per WebSockets kommen Events vom Server



A screenshot of a web browser window titled "Personal Project Planning". The URL in the address bar is "localhost:4080/#/project/2/tasks". The page displays a table of tasks under the heading "All Projects > Book Trip to St. Peter-Ording Tasks". The table has columns: NAME, ASSIGNEE, and STATE. There are two rows: "Find a train" assigned to "Susi Mueller" in the "New" state, and "Book a room" assigned to "Nils Hartmann" in the "Finished" state. Each row has a blue circular button with a right-pointing arrow. A dashed purple line connects the second row to the text "Liste wird automatisch aktualisiert bei neuen Tasks" (List is automatically updated when new tasks are added). At the bottom of the table, there is a blue button labeled "Add Task >". The footer of the browser window shows the URL "https://github.com/nilshartmann/graphql-workshop".

NAME	ASSIGNEE	STATE
Find a train	Susi Mueller	New
Book a room	Nils Hartmann	Finished

TaskListPage.tsx

Liste wird automatisch aktualisiert
bei neuen Tasks

SUBSCRIPTIONS

Subscriptions: Per WebSockets kommen Events vom Server

- Auch die Daten des Events aktualisieren den Cache
- Subscriptions sollen nur gestartet werden, wenn Komponente gemounted ist (um Resourcen Leaks zu vermeiden)
- `subscribeToMore` am Query bietet die Möglichkeit, eine Subscription auszuführen, mit deren Daten die Query aktualisiert wird (Cache)

SUBSCRIPTIONS

Subscriptions: Per WebSockets kommen Events vom Server

- Auch die Daten des Events aktualisieren den Cache
- Subscriptions sollen nur gestartet werden, wenn Komponente gemounted ist (um Resourcen Leaks zu vermeiden)
- `subscribeToMore` am Query bietet die Möglichkeit, eine Subscription auszuführen, mit deren Daten die Query aktualisiert wird (Cache)
- Je nachdem, was die Subscription zurückliefert, muss der Cache manuell gepflegt werden

```
query TaskListPageQuery($projectId: ID!) {  
  project(id: $projectId) {  
    title  
    id  
    tasks {  
      title  
      state  
      ...  
    }  
  }  
}
```

```
subscription OnNewTask($projectId: ID!) {  
  onNewTask(projectId: $projectId) {  
    title  
    state  
    ...  
  }  
}
```

SUBSCRIPTIONS

Subscriptions: Per WebSockets kommen Events vom Server

Starten, sobald
Komponente gemounted
ist...

...bzw sich Properties
verändert haben

```
function TaskListPage(props) {  
  const { loading, error, data, subscribeToMore } = useQuery(...);  
  
  React.useEffect(() => {  
    return subscribeToMore({  
      ...  
    });  
  }, [projectId, subscribeToMore]);  
  
  return ...;  
}
```

SUBSCRIPTIONS

Subscriptions: Per WebSockets kommen Events vom Server

Parameter
Subscription,
Variablen
und optional
updateQuery

```
function TaskListPage(props) {  
  const { loading, error, data, subscribeToMore } = useQuery(...);  
  
  React.useEffect(() => {  
    return subscribeToMore({  
      document: NEW_TASK_SUBSCRIPTION,  
      variables: { projectId },  
      updateQuery: (prev, { subscriptionData }) => {  
        ...  
      }  
    });  
  }, [projectId, subscribeToMore]);  
  
  return ...;  
}
```

SUBSCRIPTIONS

Subscriptions: Per WebSockets kommen Events vom Server

```
function TaskListPage(props) {  
  const { loading, error, data, subscribeToMore } = useQuery(...);  
  
  React.useEffect(() => {  
    return subscribeToMore({  
      document: NEW_TASK_SUBSCRIPTION,  
      variables: { projectId },  
      updateQuery: (prev, { subscriptionData }) => {  
        if (!prev.project) { return prev; }  
  
        return {  
          project: {  
            ...prev.project,  
            tasks: [...prev.project.tasks, subscriptionData.data.newTask]  
          }  
        };  
      }  
    });  
  }, [projectId, subscribeToMore]);  
  
  return ...;  
}
```

Aktualisieren des Caches

Hier: Liste der Tasks
am Projekt aus dem Cache
ergänzen um neuen Task
aus Subscription

FRAGMENTE

Fragmente: Query-Teile wiederverwenden

Identische Teile
(hier: Teile von Task)

```
query ProjectListPageQuery {  
    projects { id tasks { id title assignee { id name } } }  
}  
  
query ProjectPage {  
    project(projectId: "P1") { id title tasks { id title assignee { id name } } }  
}
```

FRAGMENTE

Fragmente: Query-Teile wiederverwenden

Identische Teile
(hier: Teile von Task)

```
query ProjectListPageQuery {  
    projects { id tasks { id title assignee { id name } } }  
}  
  
query ProjectPage {  
    project(projectId: "P1") { id title tasks { id title assignee { id name } } }  
}
```

Fragment Definition

```
fragment TaskFragment on Task {  
    id title assignee { id name }  
}
```

FRAGMENTE

Fragmente: Query-Teile wiederverwenden

Identische Teile
(hier: Teile von Task)

```
query ProjectListPageQuery {  
    projects { id tasks { id title assignee { id name } } }  
  
query ProjectPage {  
    project(projectId: "P1") { id title tasks { id title assignee { id name } } }
```

Fragment Definition

```
fragment TaskFragment on Task {  
    id title assignee { id name }  
}
```

Fragment Verwendung

```
query ProjectListPageQuery {  
    projects { id tasks { ...TaskFragment } }  
  
query ProjectPage {  
    project(projectId: "P1") { id title tasks { ...TaskFragment } }  
}
```

FRAGMENTE

Fragmente: In Apollo mit \${fragment_name} in Query einbinden

```
const TASK_FRAGMENT = gql`fragment TaskFragment on Task
  id title assignee { id name }
}`
```

```
const PROJEKT_LIST_PAGE_QUERY = gql`query ProjectListPageQuery {
  projects { id tasks { ...TaskFragment } }
  ${TASK_FRAGMENT}
}`
```

```
const PROJEKT_PAGE_QUERY = gql`query ProjectListPageQuery {
  projects { id title tasks { ...TaskFragment } }
  ${TASK_FRAGMENT}
}`
```

A background image showing two glasses of beer with thick foam, resting on a light-colored, textured surface that looks like stone or wood grain.

Vielen Dank!

Repository: <https://github.com/nilshartmann/graphql-workshop>

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

Fragen und Kontakt: nils@nilshartmann.net