

# GRAPHQL WORKSHOP - VORBEREITUNG

W-Lan: `wjax / wjax2019`

## Vorbereitung Klonen und Bauen

1. `git clone https://github.com/nilshartmann/graphql-java-workshop`
2. In `code/userservice`: `./gradlew bootRun`
3. In `code/backend`: `./gradlew build`

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

# GraphQL für Java

**Eine praktische Einführung**

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

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

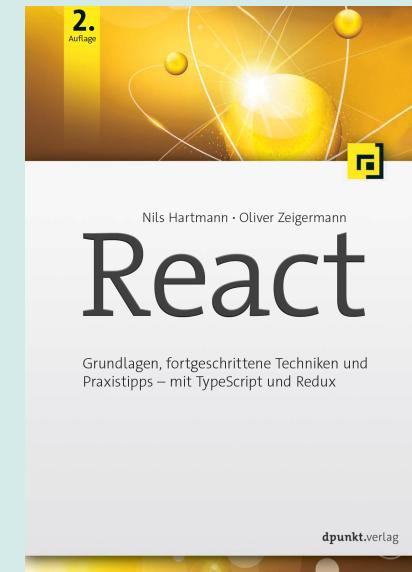
# NILS HARTMANN

nils@nilshartmann.net

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

Java  
JavaScript, TypeScript  
React  
GraphQL

Trainings, Workshops und  
Beratung



**2. Auflage, Dez. 2019**

**HTTPS://NILSHARTMANN.NET**

## AGENDA

### **1. GraphQL Grundlagen: wieso, weshalb, warum**

### **2. GraphQL für Java-Anwendungen**

- API implementieren
- Optimierung
- Alternativen zu graphql-java

### **3. Ausblick, Fragen, Diskussionen**

**Jederzeit: Fragen, und Diskussionen!**

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 der JS-Szene, aber auch außerhalb

# *GraphQL != JavaScript != Java != C#*

- Populär in der JS-Szene, aber auch außerhalb
- "Wie löse ich Problem X in GraphQL" machmal falsche Frage:
  - ist es ein GraphQL Problem
  - oder ein Problem der konkreten Implementierung

## *GraphQL != Mainstream*

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



Folge ich



Announcing GitHub Marketplace and the official releases of GitHub Apps and our GraphQL API

Original (Englisch) übersetzen

# GitHub

## GitHub

GitHub is where people build software. More than 23 million people use GitHub to discover, fork, and contribute to over 64 million projects.

[github.com](https://github.com)

11:46 - 22. Mai 2017

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

GITHUB

Sicher | https://docs.atlassian.com/atlassian-confluence/1000.18...

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

## Package com.atlassian.confluence.plugins.graphql.resource

### Class Summary

Class	Description
ConfluenceGraphQLRestEndpoint	Provides the REST API endpoint for GraphQL.
GraphResource	REST API for GraphQL. ←



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

Copyright © 2003–2017 Atlassian. All rights reserved.

<https://docs.atlassian.com/atlassian-confluence/1000.1829.0/overview-summary.html>

**tom**

@tgvashworth

Folgen



Heh. Twitter GraphQL is quietly serving more than 40 million queries per day. Tiny at Twitter scale but not a bad start.

Original (Englisch) übersetzen

RETWEETS

**93**

GEFÄLLT

**244**

22:59 - 9. Mai 2017

4

93

244

<https://twitter.com/tgvashworth/status/862049341472522240>

TWITTER



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

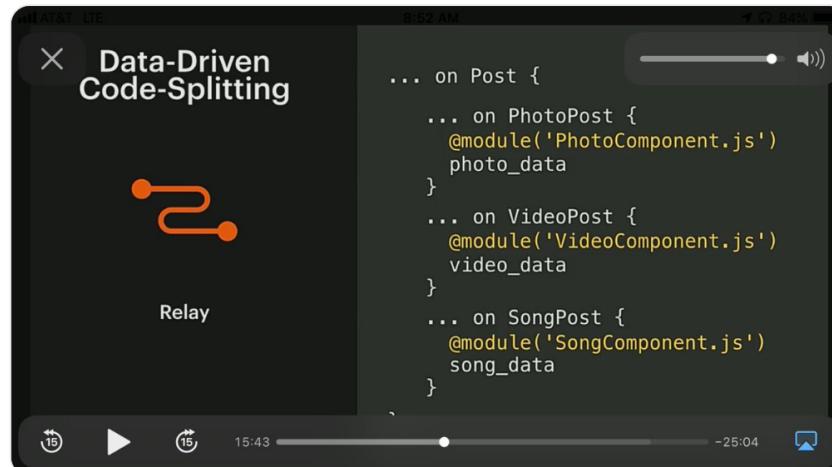


**Nick Schrock**  
@schrockn

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.

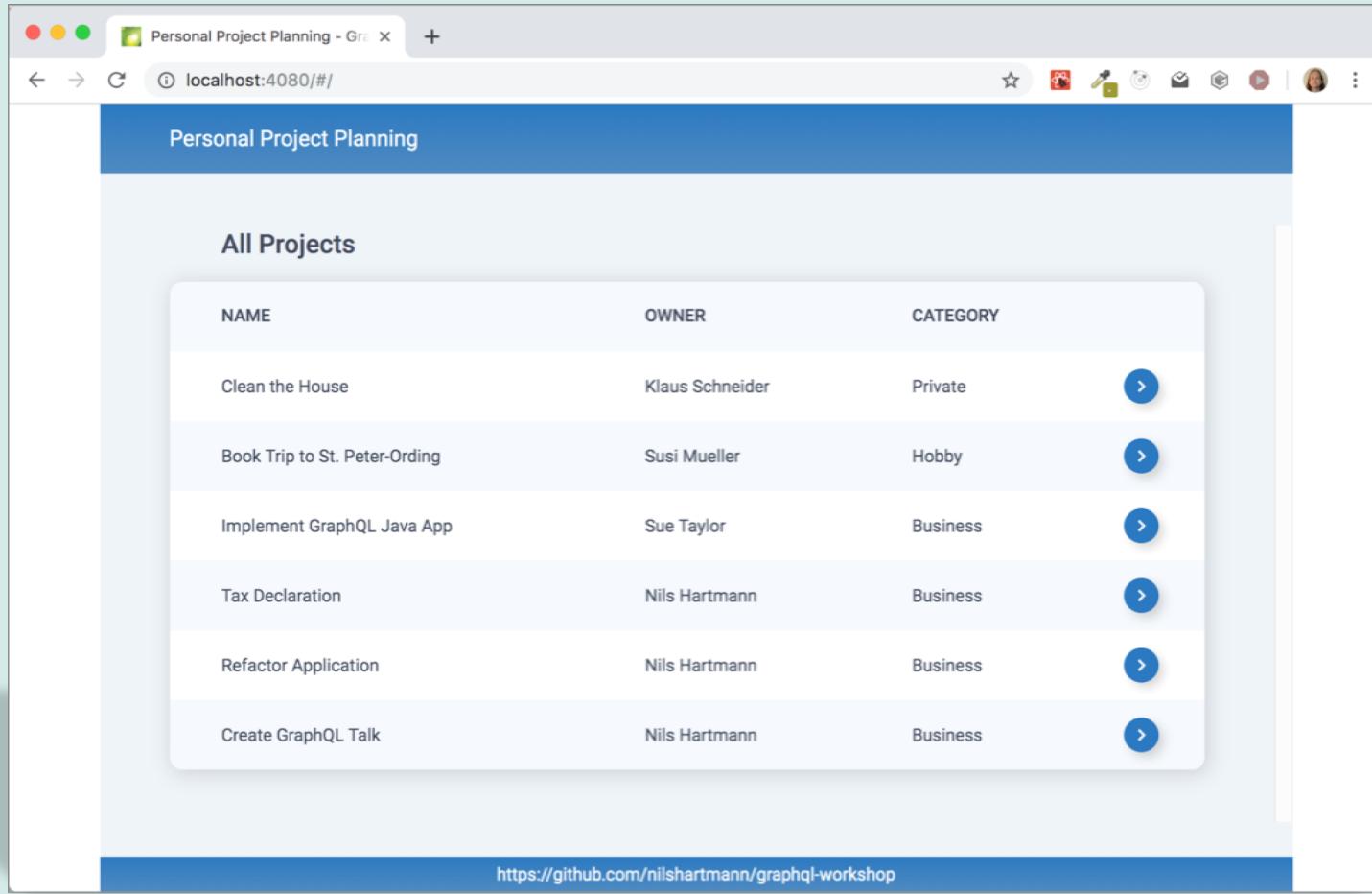
Tweet übersetzen



18:06 - 1. Mai 2019

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

## NEXT GEN GRAPHQL?



# Die Beispiel Anwendung

<http://localhost:5080>

The screenshot shows the Prisma GraphQL Playground interface. On the left, a code editor displays a GraphQL query named 'AllProjectsQuery'. The query retrieves a list of projects, each with an ID, title, description, and owner information. A tooltip for the 'login' field indicates it is used for user authentication. The right side of the interface contains a sidebar with navigation tabs: DOCS, SCHEMA, and MUTATIONS. The SCHEMA tab is active, showing the schema definition for the 'Project' type, which includes fields for id, title, description, owner, and category. Below the schema, sections for QUERIES, MUTATIONS, and SUBSCRIPTIONS are listed. The ARGUMENTS section shows a single argument 'id: ID!'.

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

String! The login is used by the user to log in to our System

project(  
 id: ID!  
)

ping: String!  
users: [User!]!

user(...): User

projects: [Project!]!

project(...): Project

type Project {  
 id: ID!  
 title: String!  
 description: String!  
 owner: User!  
 category: Category!  
 tasks: [Task!]!  
 task(...): Task

onNewTask: Task!  
onTaskChange(...): Task!

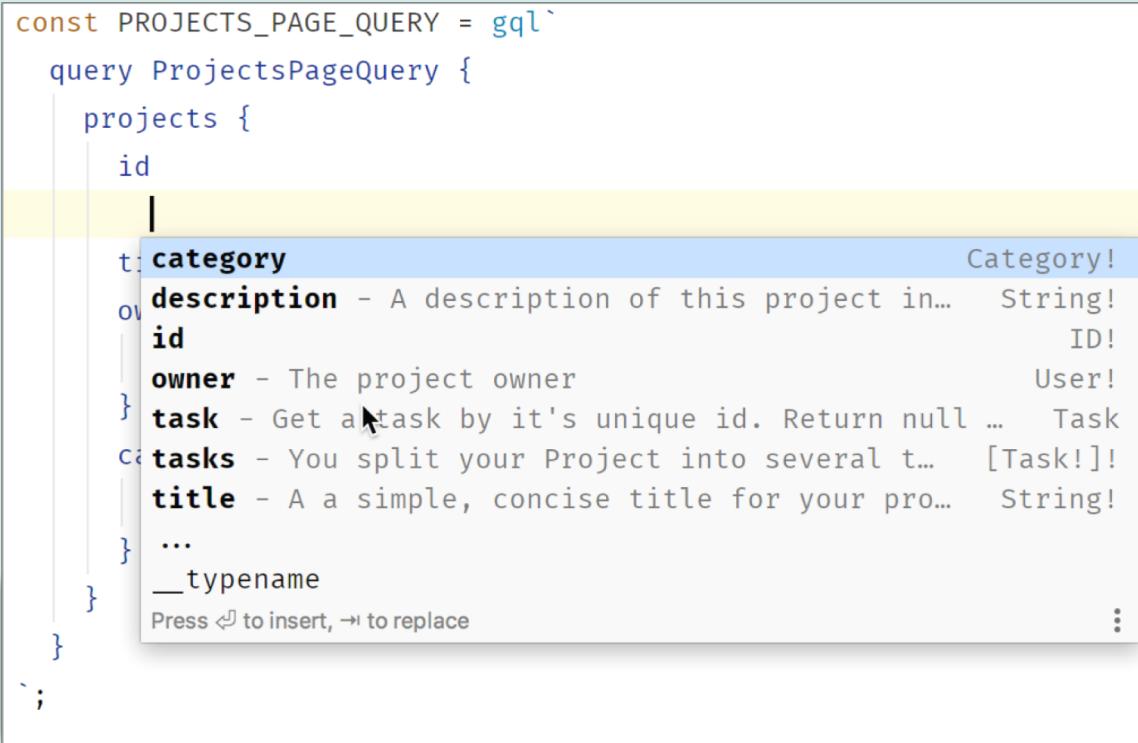
id: ID!

# Demo: Playground

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

<http://localhost:5000>

```
const PROJECTS_PAGE_QUERY = gql`  
query ProjectsPageQuery {  
  projects {  
    id  
    |  
    t: category  
    description - A description of this project in... String!  
    id ID!  
    owner - The project owner User!  
    task - Get a task by it's unique id. Return null ... Task  
    tasks - You split your Project into several t... [Task!]!  
    title - A simple, concise title for your pro... String!  
    ...  
    __typename  
  }  
}  
`;
```

A screenshot of an IDE showing a code completion tooltip for a GraphQL query field. The tooltip lists various fields and their types: category (Category!), description (String!), id (ID!), owner (User!), task (Task), tasks ([Task!]!), title (String!), and \_\_typename. It also includes a note: "Press ⌘ to insert, ⌘ to replace".

The code is part of a file named `ProjectsPage.tsx`, as indicated by the pink text at the bottom right.

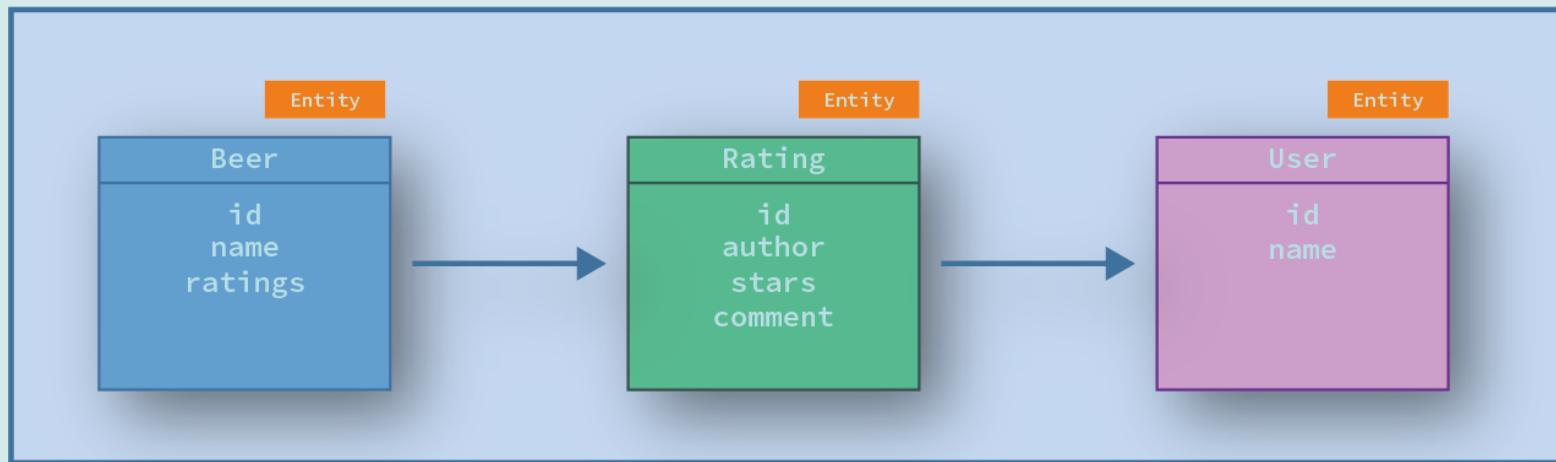
# Demo: IDE Support

Beispiel: IntelliJ IDEA

# **Vergleich mit REST**

# BEERADVISOR DOMAINE

## "Domain-Model"

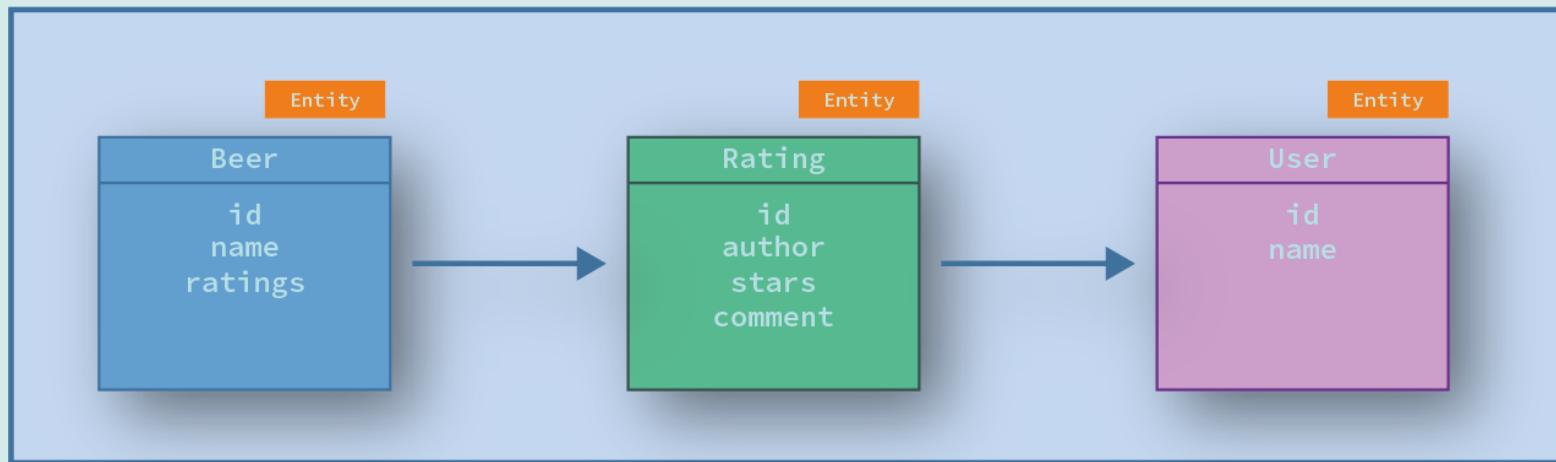


# ABFRAGEN MIT REST

## REST-Zugriff

- Exemplarisch und vereinfacht

GET /beer/1



```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

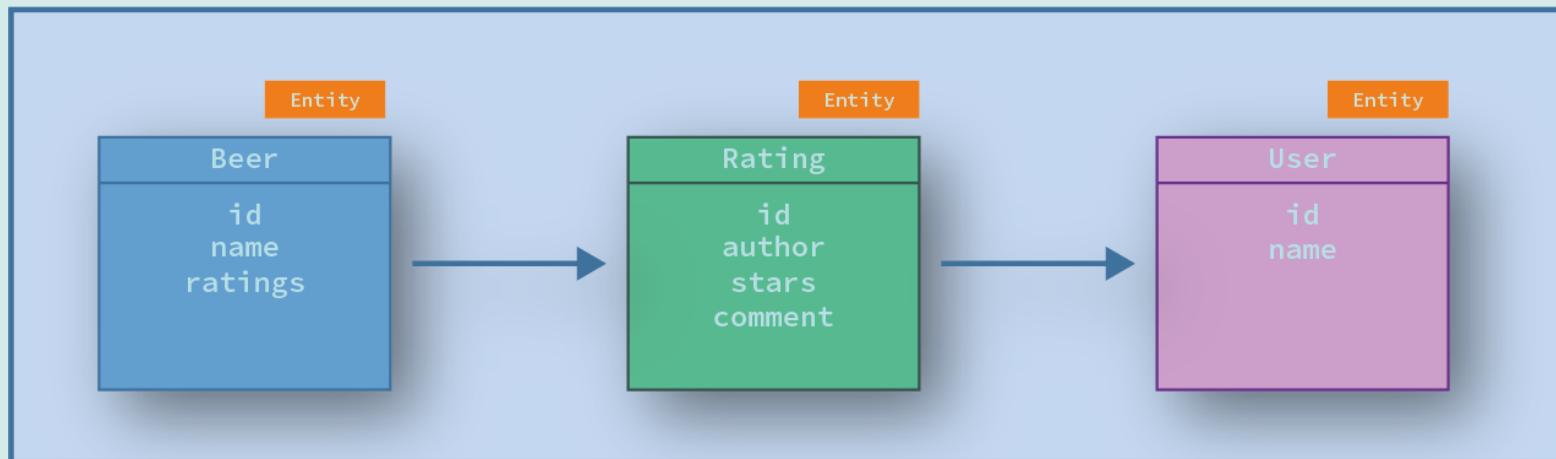
# ABFRAGEN MIT REST

## REST-Zugriff

- Exemplarisch und vereinfacht

GET /beer/1

GET /beer/1/rating/R1



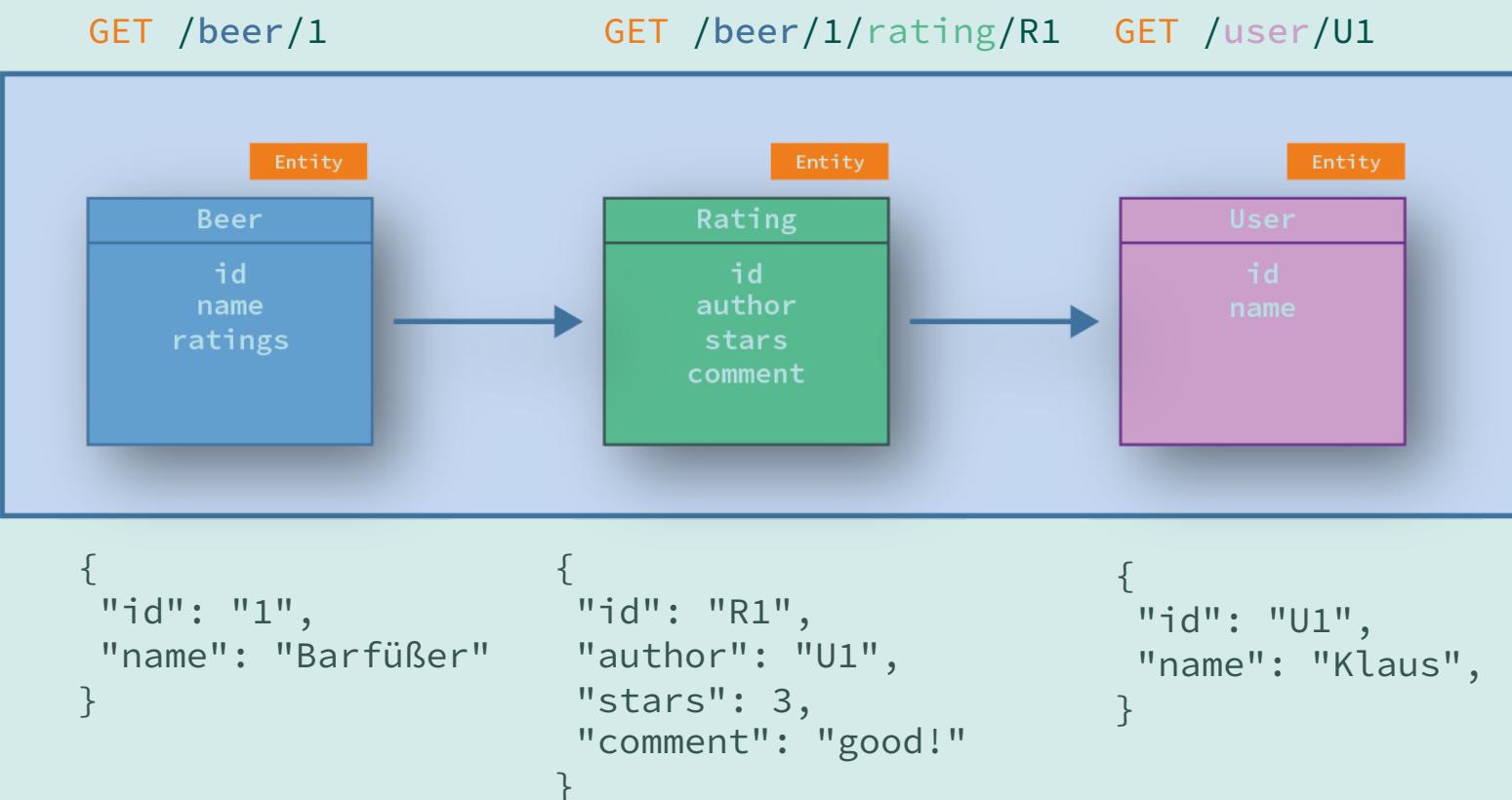
```
{  
  "id": "1",  
  "name": "Barfüßer"  
}
```

```
{  
  "id": "R1",  
  "author": "U1",  
  "stars": 3,  
  "comment": "good!"  
}
```

# ABFRAGEN MIT REST

## REST-Zugriff

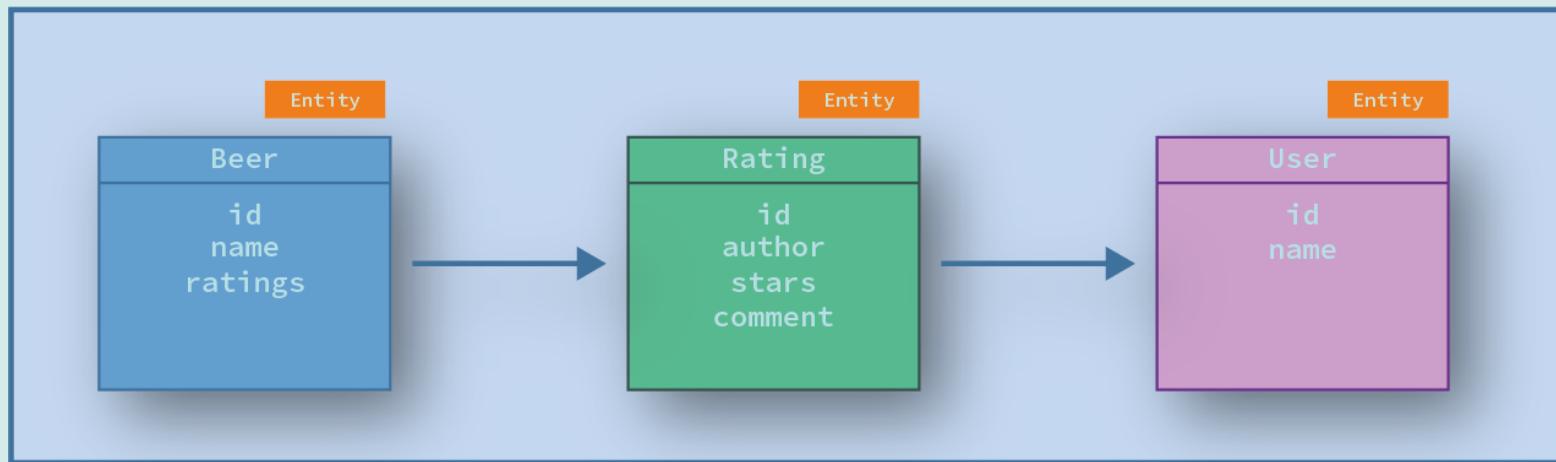
- Pro Entität (Resource) eine Abfrage
- Zurückgeliefert wird immer komplette Resource
- Keine Gesamt-Sicht auf Domaine



# ABFRAGEN MIT GRAPHQL

## GraphQL

```
query { beer
  { name ratings(rid: "R1")
    { stars author { name } }
  }
}
```



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

*"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**

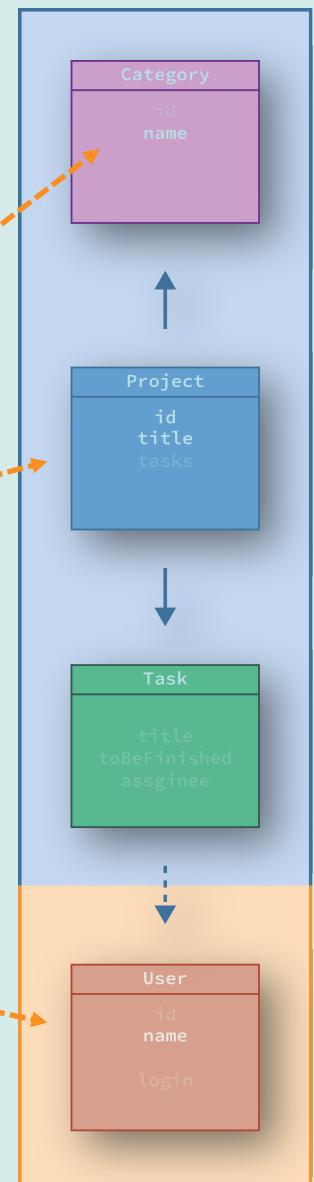
# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 1

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

The screenshot shows a web application titled "Personal Project Planning" with a table titled "All Projects". The table has columns: NAME, OWNER, and CATEGORY. The data is as follows:

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

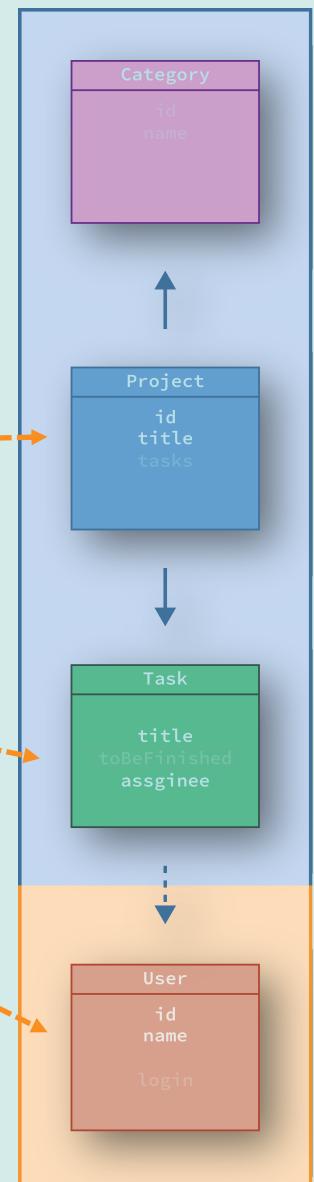
## Use-Case spezifische Abfragen – 2

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

A screenshot of a web browser window titled "Personal Project Planning - GraphQL". The URL is "localhost:4080/#/project/1/tasks". The page displays a table with three rows of task data:

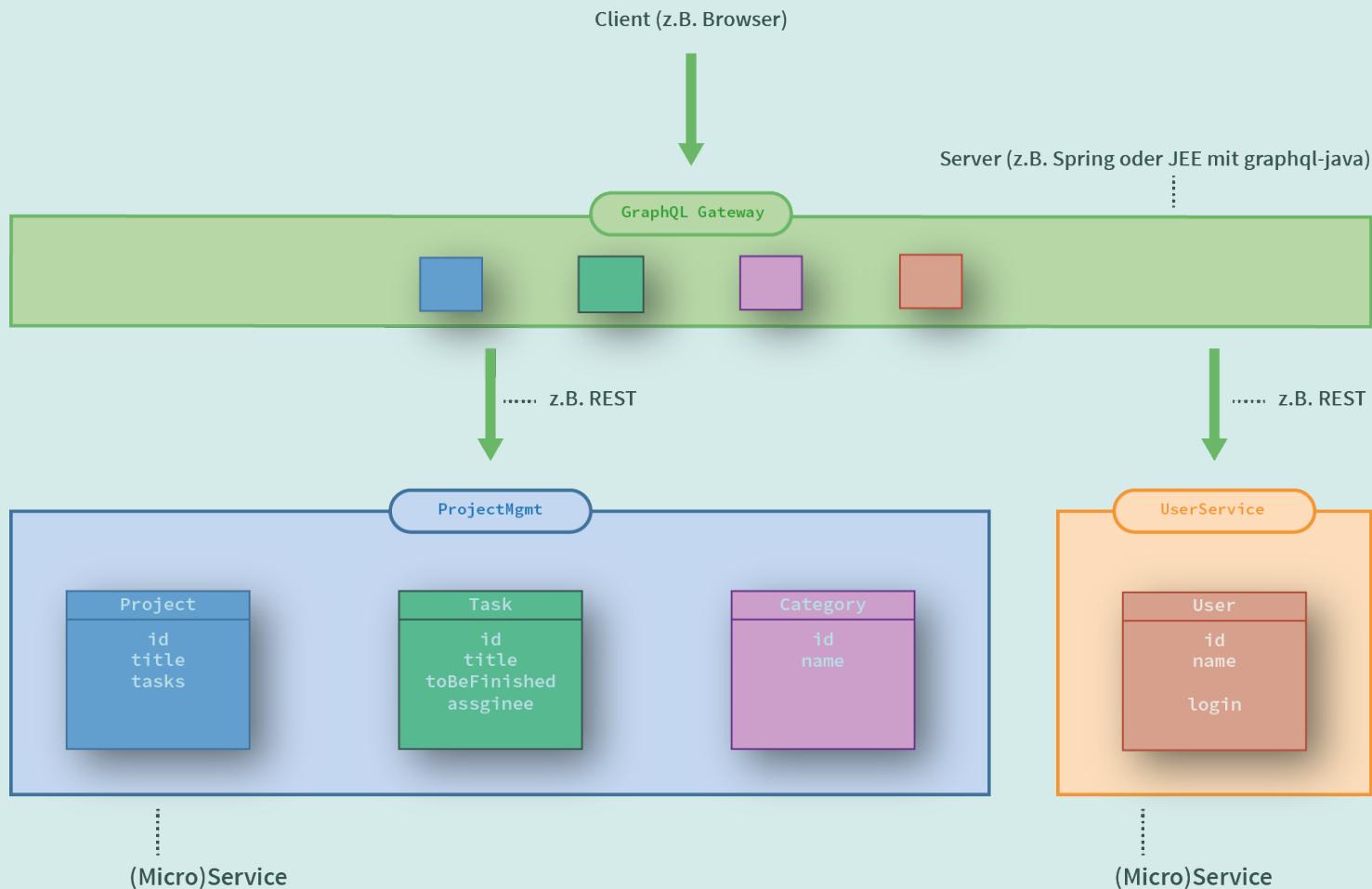
NAME	ASSIGNEE	STATE
Create a draft story	Nils Hartmann	In Progress
Finish Example App	Susi Mueller	In Progress
Design Slides	Nils Hartmann	New

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.



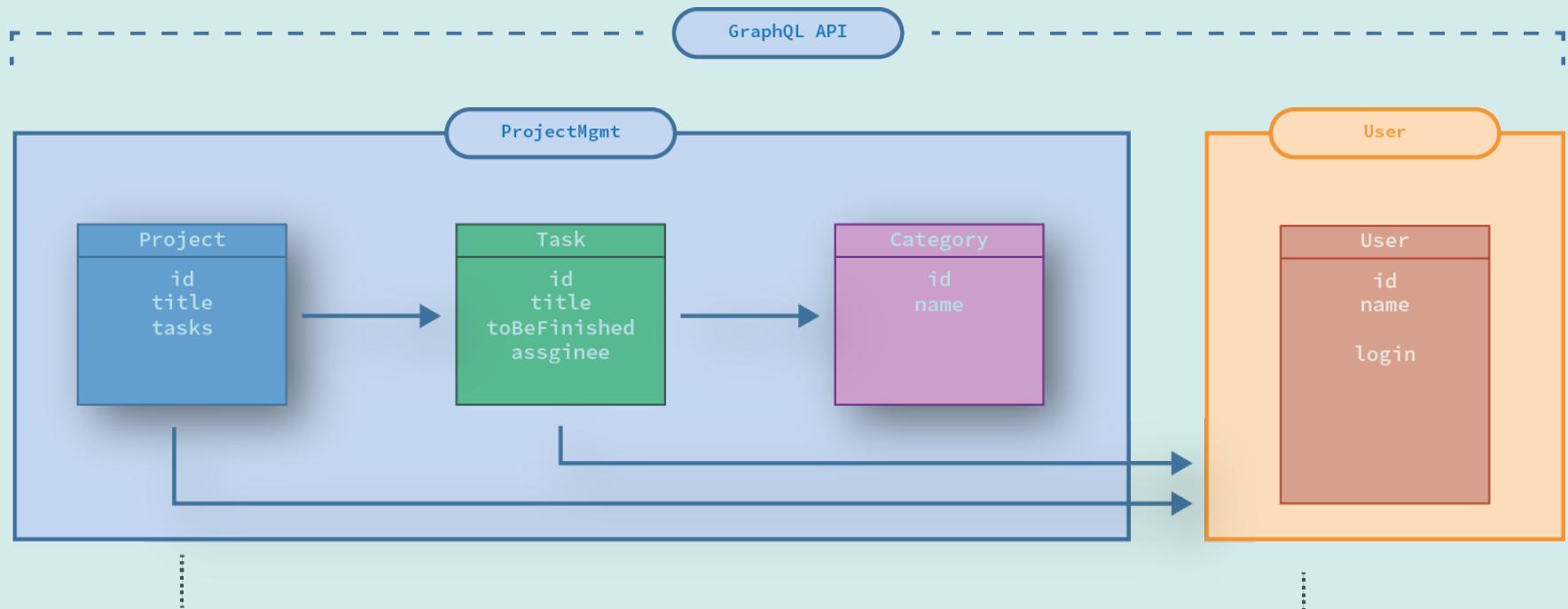
# EINSATZSzenarien

- Gateway für Frontend zu mehreren Backends



# GRAPHQL EINSATZSzenariEN

## "Architektur" Project Beispiel Anwendung



Project Management Application (Spring Boot, graphql-java)

(Micro)Service

REST Endpunkt: <http://localhost:5010/users>  
<http://localhost:5010/users/{id}>

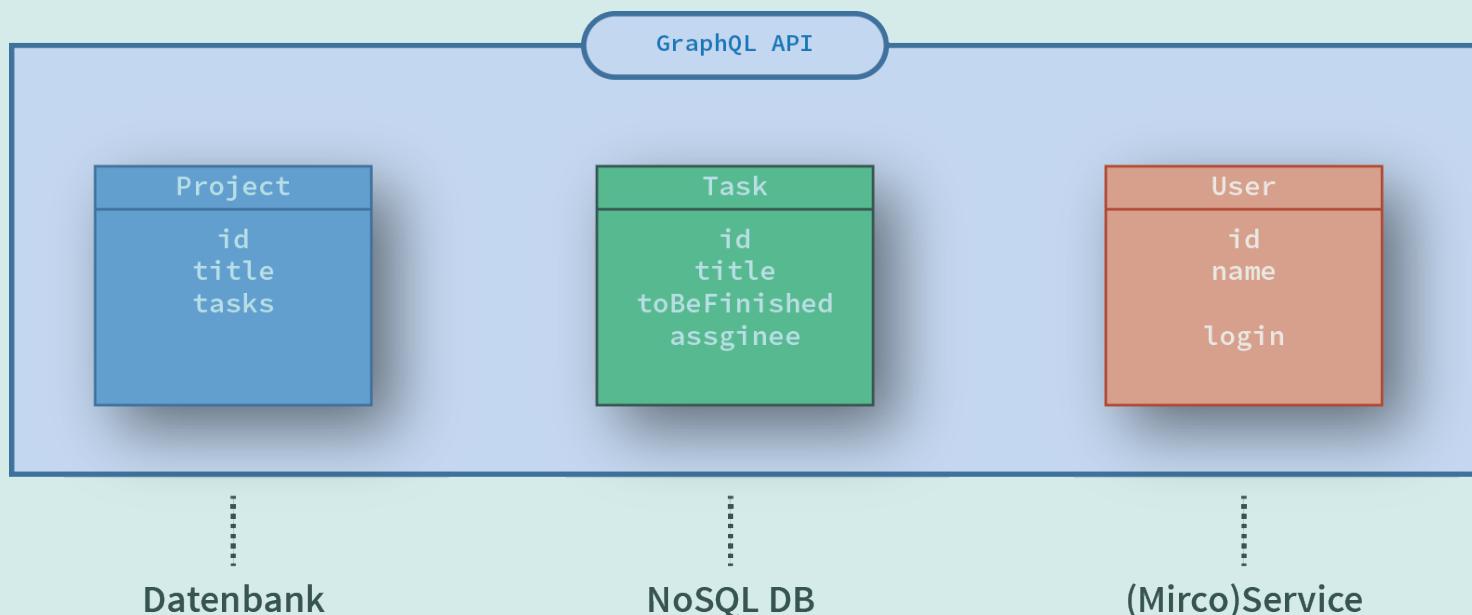
### Gründe für den Einsatz von GraphQL

- Viele unterschiedliche Use-Cases, die unterschiedliche Daten benötigen
  - Unterschiedliche Ansichten im Frontend
  - Unterschiedliche Clients
  - Flexible Architektur im Client
- API kann unabhängig vom Client weiterentwickelt werden
  - Client konsumiert nur explizit abgefragte Daten
- 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

- Variablen werden in einem eigenen Objekt an den Server geschickt
- 👉 Playground!

```
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 (üblicherweise /graphql)

```
$ curl -X POST -H "Content-Type: application/json" \
-d '{"query":"{ projects { title } }}'" \
http://localhost:5000/graphql
```

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

# QUERIES AUSFÜHREN

## Antwort vom Server

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

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

## ÜBUNG: QUERIES AUSFÜHREN

### Mach dich mit dem Playground und der Query-Sprache vertraut

- Öffne den Playground auf meinem Computer (URL steht auf der Tafel)  
[http://SIEHE\\_TAFEL.ngrok.io/](http://SIEHE_TAFEL.ngrok.io/)
1. Mach' dich mit der API des Projektes vertraut
  2. Führe einen Query aus, mit dem Du alle Projekte und alle Benutzer (insb. jeweils deren 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)**

*graphql-java: <https://www.graphql-java.com/>*

- "nur" GraphQL Framework, kein Server, kein DI, o.ä.
- keine Aussage, wie es mit JEE / Spring integriert wird
- gibt aber Beispiele und weitere Projekte dafür

# GRAPHQL SERVER MIT APOLLO

## Aufgaben

1. Schema definieren
2. DataFetchers für das Schema implementieren und konfigurieren
  - Wie/woher kommen die Daten für eine Anfrage
3. GraphQL API per HTTP zur Verfügung stellen

## 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 komplexe 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

Neues Feld -----

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

## 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!]!  
    getProjectById(projectId: ID!): Project  
    project(projectId: ID!): Project @deprecated  
}
```

Direktive

# GRAPHQL FÜR JAVA-ANWENDUNGEN

**Schritt 1: Schema definieren**

**Schritt 2: DataFetchers implementieren**

**Schritt 3: DataFetchers mit Schema verbinden**

**Schritt 4: API per HTTP zur Verfügung stellen**

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Schema definieren über Schema-Definition-Language

- Per .graphqls-Datei (einbinden sehen wir uns später an)

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Project {  
    id: ID!  
    title: String!  
    description: String!  
    owner: User!  
    category: Category!  
    tasks: [Task!]!  
    task(id: ID!): Task  
}  
  
type Category {  
    id: ID!  
    name: String!  
}  
  
enum TaskState {  
    NEW RUNNING FINISHED  
}
```

```
type Task {  
    id: ID!  
    title: String!  
    description: String!  
    state: TaskState!  
    assignee: User!  
    toBeFinishedAt: String!  
}  
  
type Query {  
    users: [User!]!  
    projects: [Project!]!  
    project(id: ID!): Project  
}  
  
input AddTaskInput {  
    title: ID!  
    description: ID!  
    toBeFinishedAt: String  
    assigneeId: ID!  
}  
  
type Mutation {  
    addTask(project: ID!, input: AddTaskInput!):  
        Task!  
    updateTaskState(taskId: ID!, newState: TaskState!):  
        Task!  
}
```

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Schema definieren über Schema-Definition-Language

- Dokumentation kann Zeilenweise mit # hinzugefügt werden
- Oder Blockweise mit """", darin sogar Markdown möglich

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

type Project {
    # The unique ID of this Project
    id: ID!
    ...
}

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

# DAS SCHEMA IN GRAPHQL-JAVA

## Modulare Schema

- Schema kann in mehrere Dateien aufgeteilt werden
- Typen können erweitert werden

```
// queries.graphqls
type Query {
    ping: String
}

// project.graphqls
type Project { ... }

extend type Query {
    project(projectId: ID!): Project
}
```

# ÜBUNGEN - VORBEREITUNG

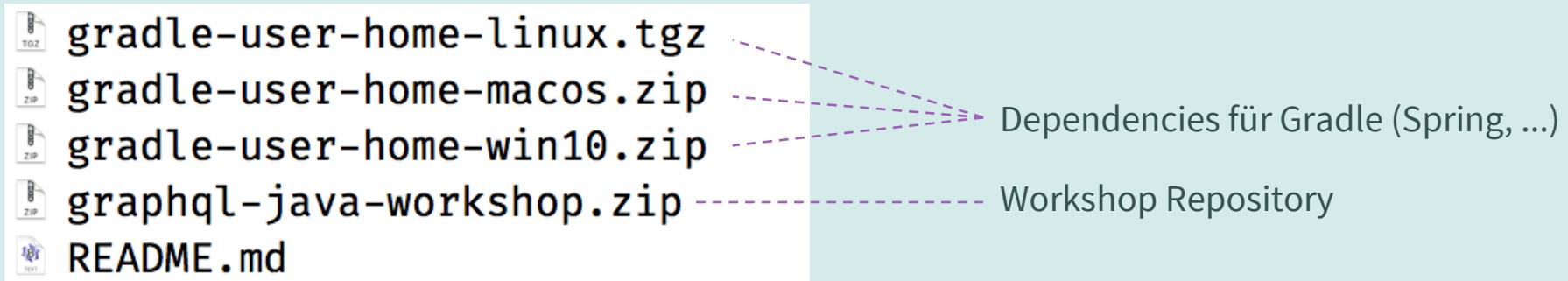
## Option 1 "online": Klonen des Git Repositorys

- `git clone https://github.com/nilshartmann/graphql-java-workshop`

# ÜBUNGEN - VORBEREITUNG

## Option 2 "offline": Der USB-Stick

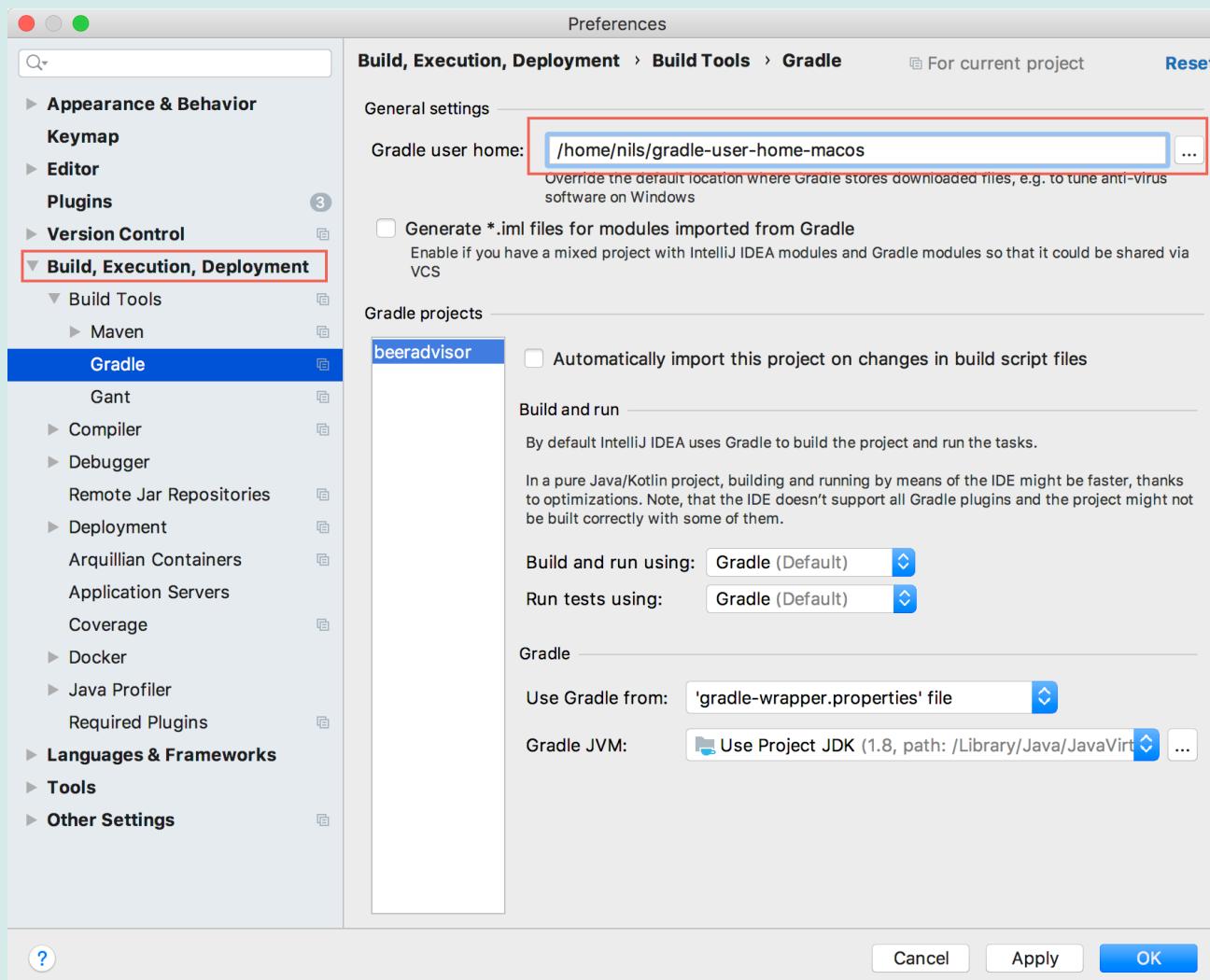
Für den Fall, dass das W-LAN nicht funktioniert, findet ihr alles Notwendige auf dem USB-Stick



1. **graphql-java-workshop.zip** auspacken
2. "Euer" gradle-user-home auspacken
3. Bei Gradle-Aufrufen (gradlew):
  1. Entweder mit -g den Pfad zum ausgepackten Verzeichnis übergeben  
`./gradlew -g /home/nils/gradle-home-macos bootRun`
  2. oder: Umgebungsvariable GRADLE\_USER\_HOME zu dem Pfad setzen

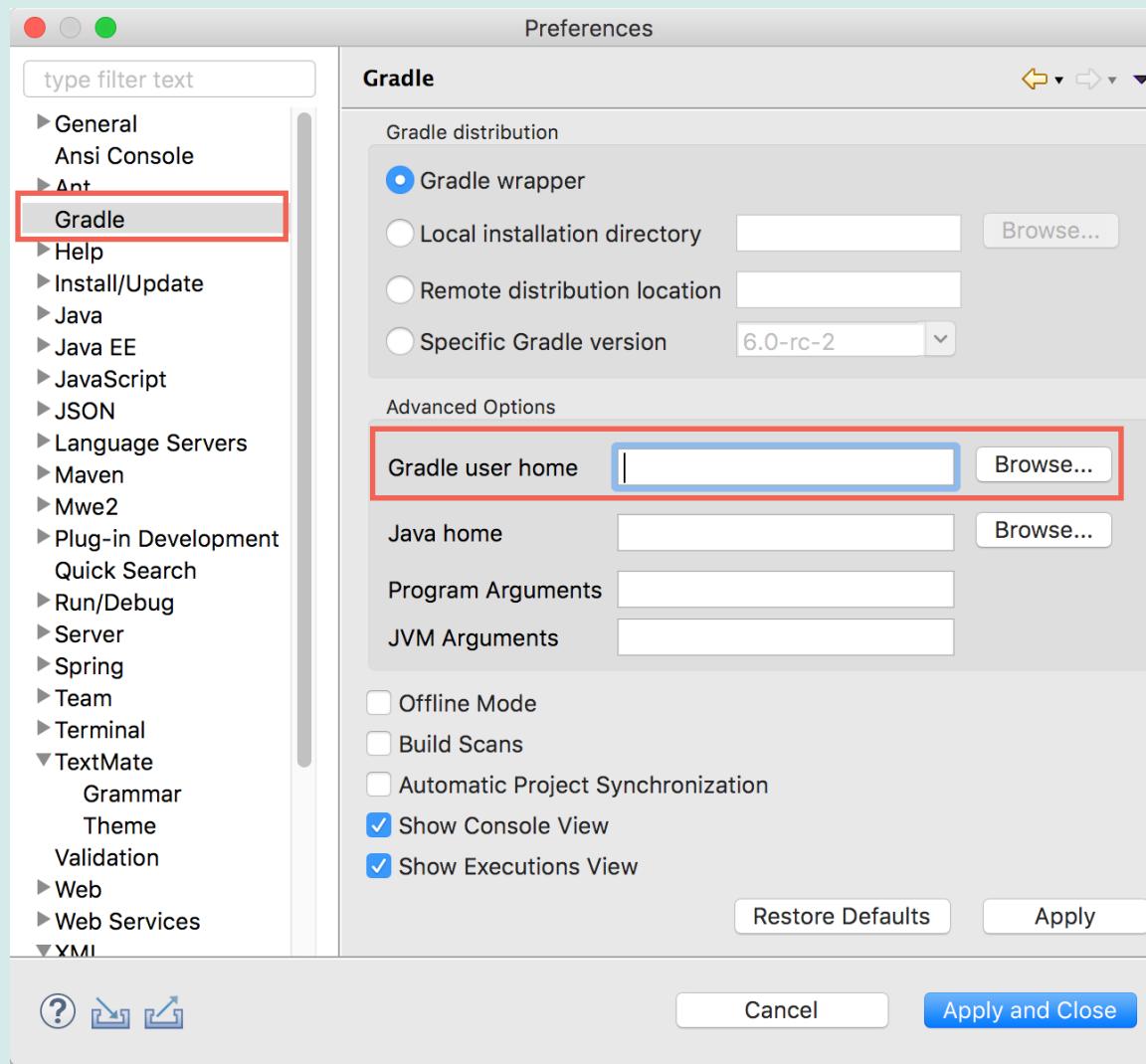
# ÜBUNGEN - VORBEREITUNG

## Option 2 "offline": Gradle User Home setzen - IDEA



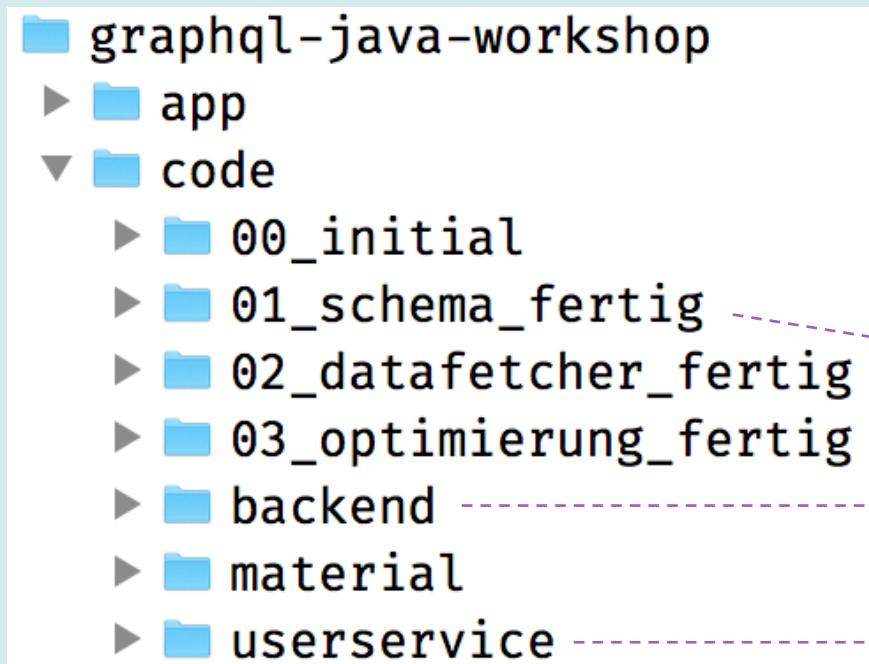
# ÜBUNGEN - VORBEREITUNG

## Option 2 "offline": Gradle User Home setzen - Eclipse



# ÜBUNGEN - VORBEREITUNG

## Vorbereitung



## Das Workshop Repository

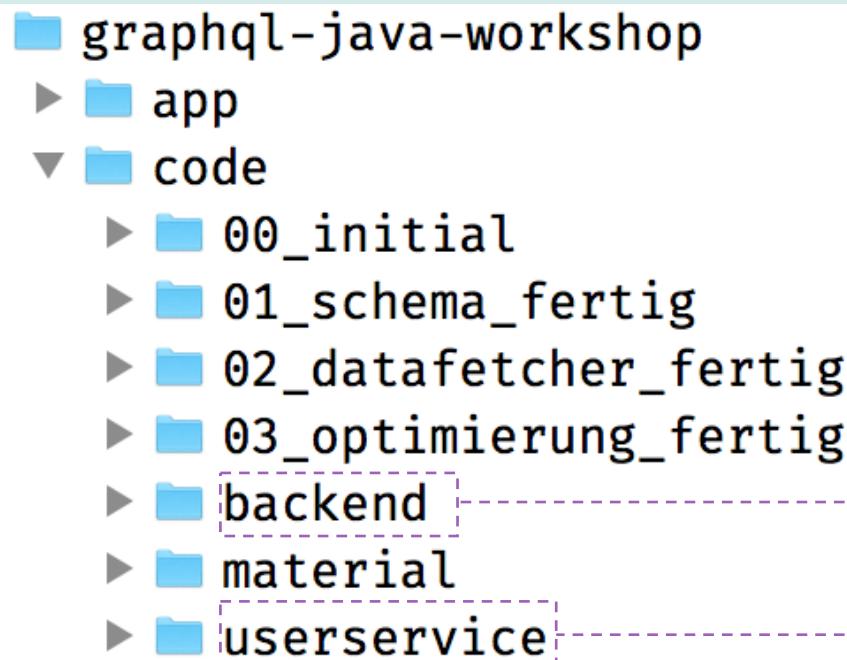
Lösungen für die Übungen

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

Fertiger userservice (nur starten)

## Vorbereitung: Installation und Starten

### Das Workshop Repository



- Beim Ausführen von Gradle bei Bedarf dran denken, das Gradle User Home zu setzen!  
./gradlew -g ... clean bootRun
- In IDE "ProjectMgmtApplication" starten oder per  
"./gradlew clean bootRun"
- hier "./gradlew clean bootRun" ausführen  
(User Service API: <http://localhost:5010/users>)

# ÜBUNG 1: SCHEMA DEFINIEREN

## Vorbereitung (gemeinsam): Starten aller Prozesse

1. In `code/userservice`: `./gradlew clean bootRun`  
Diesen Prozess könnt ihr die ganze Zeit laufen lassen  
Zum Testen, ob Prozess läuft: <http://localhost:5010/users>
2. Gradle-Projekt in `code/backend` in Eurer IDE öffnen:
  - Eclipse: File -> Import -> Gradle -> Existing Gradle Project
  - IDEA: Import Project -> Import from external Model -> Gradle
3. Das Backend starten:
  - Klasse `nh.graphql.projectmgmt.ProjectMgmtApplication` starten
  - Nach dem Ändern&Compilieren der Sourcen sollte Server automatisch neugestartet werden
4. Playground sollte jetzt über <http://localhost:5000> erreichbar sein

# ÜBUNG 1: SCHEMA DEFINIEREN

## GraphQL Plug-in für IDEA "JS GraphQL"

<https://plugins.jetbrains.com/plugin/8097-js-graphql>

## GraphQL Plug-in für Eclipse 😢

Kennt jemand ein Plug-in für Eclipse?

## GraphQL Extension für VS Code "Apollo GraphQL for VS Code"

<https://marketplace.visualstudio.com/items?itemName=apollographql.vscode-apollo>

## ÜBUNG 1: SCHEMA DEFINIEREN

### Übung: 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 `src/main/resources/projectmgmt.graphqls` stehen Todos drin
  - Nach Änderungen am Schema (speichern der Datei) könnt ihr im Playground Eure API-Änderungen sehen
    - <http://localhost:5000/>
    - Schema sollte automatisch aktualisiert werden
    - (Auf "Docs" am rechten Rand klicken)
    - Hinweis: Ausführen der Queries funktioniert noch nicht

## SCHRITT 2: DATA FETCHERS

## DATA FETCHERS

**DataFetcher** (In anderen Implementierungen auch **Resolver** genannt)

- *Ein DataFetcher liefert ein Wert für ein angefragtes Feld*
  - Zwingend erforderlich für Root-Types (Query, Mutation)
  - Default: per Reflection (getter/setter, Maps, ...)

## DATA FETCHERS

**DataFetcher** (In anderen Implementierungen auch **Resolver** genannt)

- *Ein DataFetcher liefert ein Wert für ein angefragtes Feld*
  - Zwingend erforderlich für Root-Types (Query, Mutation)
  - Default: per Reflection (getter/setter, Maps, ...)
- DataFetcher ist funktionales Interface (kann als Lambda implementiert werden):

```
interface DataFetcher<T> {  
    T get(DataFetchingEnvironment environment);  
}
```

## DATA FETCHERS

**DataFetcher** (In anderen Implementierungen auch **Resolver** genannt)

- *Ein DataFetcher liefert ein Wert für ein angefragtes Feld*
  - Zwingend erforderlich für Root-Types (Query, Mutation)
  - Default: per Reflection (getter/setter, Maps, ...)
- DataFetcher ist funktionales Interface (kann als Lambda implementiert werden):

```
interface DataFetcher<T> {  
    T get(DataFetchingEnvironment environment);  
}
```
- DataFetcher werden später mit dem Schema "verbunden"

# DATAFETCHER

## DataFetcher implementieren

- Beispiel: ping-Feld

Schema Definition

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

Query

```
query { ping }
```

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

# DATAFETCHER

## DataFetcher implementieren

- Beispiel: ping-Feld

Schema Definition

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

Query

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

Data Fetcher

```
public class QueryDataFetchers {  
    DataFetcher<String> ping = new DataFetcher<>() {  
        public String get(DataFetchingEnvironment env) {  
            return "Hello, World";  
        }  
    };  
  
    // Alternativ: als Lambda  
    DataFetcher<String> ping = env -> "Hello World";  
}
```

# DATAFETCHER

## Das DataFetcherEnvironment

- Fetcher-Methoden wird eine Instanz von DataFetcherEnvironment übergeben
- Das DFE enthält Informationen über den aktuellen Query und die Umgebung
- Wir werden noch mehrere Anwendungsfälle dafür sehen

```
interface DataFetcherEnvironment {  
    <T> T getArgument(String name);  
    <T> T getSource();  
    <T> T getContext();  
  
    DataFetchingFieldSelectionSet getSelectionSet();  
  
    <K, V> DataLoader<K, V> getDataLoader(String dataLoaderName);  
}
```

# DATAFETCHER

## DataFetcher implementieren: Argumente

- Über das **DataFetchingEnvironment** kann auf Argumente zugegriffen werden
- Es werden nur gültige Werte übergeben (gemäß Schema)
- input-Types werden als Map übergeben

Schema Definition

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

Query

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

# DATAFETCHER

## DataFetcher implementieren: Argumente

- Über das **DataFetchingEnvironment** kann auf Argumente zugegriffen werden
- Es werden nur gültige Werte übergeben (gemäß Schema)
- input-Types werden als Map übergeben

Schema Definition

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

Query

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

Data Fetcher

```
public class QueryDataFetchers {  
    DataFetcher<String> ping = new DataFetcher<>() {  
        public String get(DataFetchingEnvironment env) {  
            String msg = env.getArgument("msg");  
  
            if (msg == null) { msg = "World"; }  
            return "Hello, " + msg;  
        }  
    };  
}
```

# DATAFETCHER

## DataFetcher implementieren: Context

- Context ist ein beliebiges Objekt, das allen DataFetchers übergeben wird
- Kann zum Beispiel Zugriff auf Services, Security etc ermöglichen
- (Alternative wäre mit Dependency Injection zu arbeiten)

Data Fetcher

```
public class QueryDataFetchers {  
    DataFetcher<String> ping = new DataFetcher<>() {  
        public Project get(DataFetchingEnvironment env) {  
            ProjectRepository projectRepository = env.getContext();  
  
            Project project = projectRepository.find(...);  
            ...  
        }  
    };  
}
```

# DATAFETCHER

## DataFetcher implementieren: Rückgabewerte

- Rückgabewert können neben primitiven Typen auch Objekte oder Listen sein

Schema Definition

```
type Query {  
    project(id: ID!): Project  
}
```

Project POJO

```
public class Project {  
    long id;  
    String title;  
    ...  
}
```

Data Fetcher

```
public class QueryDataFetchers {  
    DataFetcher<Project> projectById = new DataFetcher<>() {  
        public String get(DataFetchingEnvironment env) {  
            String projectId = env.getArgument("id");  
            ProjectRepository repository = ...; // aus Context oder DI  
            Optional<Project> project =  
                repository.get(projectId);  
  
            return project;  
        }  
    };
```

## DataFetcher implementieren: Objekt Graphen

- Bei **geschachtelten** Abfragen wird der Rückgabewert des vorherigen Fetchers an den nächsten Übergeben
- Sofern kein DataFetcher für ein Feld registriert ist, wird per Default der **PropertyDataFetcher** verwendet (Registrierung sehen wir uns später an)
- Der PropertyDataFetcher ermittelt Daten aus POJOs

# DATAFETCHER

## DataFetcher implementieren: Objekt Graphen

- Bei **geschachtelten** Abfragen wird der Rückgabewert des vorherigen Fetchers an den nächsten Übergeben
- Sofern kein DataFetcher für ein Feld registriert ist, wird per Default der **PropertyDataFetcher** verwendet (Registrierung sehen wir uns später an)
- Der PropertyDataFetcher ermittelt Daten aus POJOs

```
query {  
  project(id: 1) {  
    id  
    title  
    category  
      { name }  
  }  
}
```

# DATAFETCHER

## DataFetcher implementieren: Objekt Graphen

- Bei **geschachtelten** Abfragen wird der Rückgabewert des vorherigen Fetchers an den nächsten übergeben
- Sofern kein DataFetcher für ein Feld registriert ist, wird per Default der **PropertyDataFetcher** verwendet (Registrierung sehen wir uns später an)
- Der PropertyDataFetcher ermittelt Daten aus POJOs

```
query {  
  project(id: 1) {  
    id  
    title  
    category  
      { name }  
  }  
}
```

QueryDataFetchers.projectById  
(liefert Project-Instanz zurück)

```
public class Project {  
  long id;  
  String title;  
  Category category;  
  ...  
}
```

# DATAFETCHER

## DataFetcher implementieren: Objekt Graphen

- Bei **geschachtelten** Abfragen wird der Rückgabewert des vorherigen Fetchers an den nächsten übergeben
- Sofern kein DataFetcher für ein Feld registriert ist, wird per Default der **PropertyDataFetcher** verwendet (Registrierung sehen wir uns später an)
- Der PropertyDataFetcher ermittelt Daten aus POJOs

```
query {  
  project(id: 1) {  
    id  
    title  
    category  
    { name }  
  }  
}
```

QueryDataFetchers.projectById  
(liefert Project-Instanz zurück)

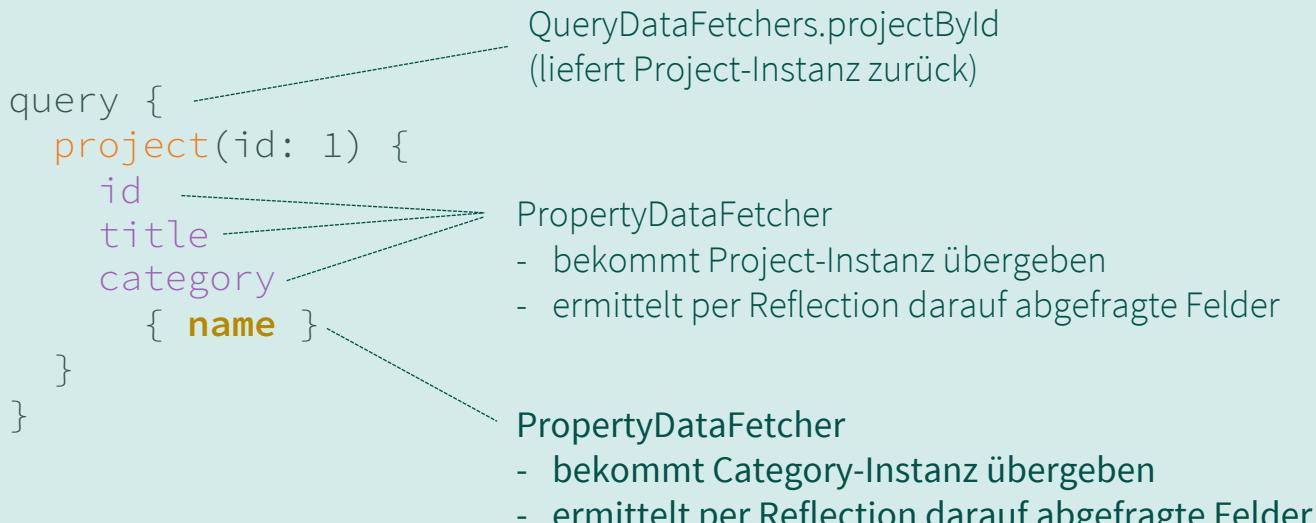
PropertyDataFetcher  
- bekommt Project-Instanz übergeben  
- ermittelt darauf per Reflection abgefragte Felder

```
public class Project {  
  long id;  
  String title;  
  Category category;  
  ...  
}
```

# DATAFETCHER

## DataFetcher implementieren: Objekt Graphen

- Bei **geschachtelten** Abfragen wird der Rückgabewert des vorherigen Fetchers an den nächsten übergeben
- Sofern kein DataFetcher für ein Feld registriert ist, wird per Default der **PropertyDataFetcher** verwendet (Registrierung sehen wir uns später an)
- Der PropertyDataFetcher ermittelt Daten aus POJOs



```
public class Project {  
  long id;  
  String title;  
  Category category;  
  ...  
}  
  
public class Category {  
  long id;  
  String name;  
  ...  
}
```

# DATAFETCHER

## DataFetcher für eigene Typen

- PropertyDataFetcher hilft nicht immer
- Beispiel: owner-Feld ist nicht am Project POJO definiert, nur dessen Id  
(User kommt aus Mircoservice)

```
query {  
  project(id: 1) {  
    id  
    title  
    category  
    { name }  
    owner {  
      name  
    }  
  }  
}
```

PropertyDataFetcher greift hier nicht

```
public class Project {  
  long id;  
  String title;  
  Category category;  
  String ownerId;  
  ...  
}
```

# DATAFETCHER

## DataFetcher für eigene Typen

- Für eigene Typen bzw. deren Felder können ebenfalls DataFetcher definiert werden
- Funktionieren wie gesehen, nur dass Parent-Objekt ("Source") übergeben wird

```
query {  
  project(id: 1) {  
    id  
    title  
    category  
    { name }  
    owner {  
      name  
    }  
  }  
}
```

```
public class ProjectDataFetchers {  
  DataFetcher<User> owner = new DataFetcher<>() {  
    public String get(DataFetchingEnvironment env) {  
      Project parent = env.getSource();  
      String ownerId = parent.getOwnerId();  
  
      return userService.getUser(ownerId);  
    }  
  };  
}
```

# DATAFETCHER

## DataFetcher implementieren: Mutations

- technisch analog zu allen anderen DataFetchers, dürfen aber Daten verändern

Schema Definition

```
input AddTaskInput {  
    title: String!  
    description: String!  
}  
  
type Mutation {  
    addTask(projectId: ID!, input: AddTaskInput!): Task!  
}
```

Data Fetcher

```
public class MutationDataFetchers {  
    DataFetcher<Task> addTask = new DataFetcher<>() {  
        public Task get(DataFetchingEnvironment env) {  
            String projectId = env.getArgument("projectId");  
            Map input = env.getArgument("input");  
            String title = input.get("title");  
            String description = input.get("description");  
  
            return taskService.newTask(projectId, title, description);  
        }  
    };  
}
```

# DATAFETCHER

## DataFetcher implementieren: Subscriptions

- Müssen Reactive Streams Publisher zurückliefern
- Beim Lesen über HTTP üblicherweise über Websockets

```
import org.reactivestreams.Publisher;

public DataFetcher<Publisher<Task>> onNewTask() {

    return environment -> {
        Publisher<Task> publisher = getTaskPublisher();
        return publisher;
    };
}

// Publisher: nh.graphql.tasks.domain.TaskPublisher
```

type Subscription {  
 onNewTask: Task!  
}

# RUNTIME WIRING

## Verbinden von Feldern mit DataFetcher

- Im RuntimeWiring werden DataFetcher einzelnen Feldern zugewiesen

```
class GraphQLAPIConfiguration {  
    RuntimeWiring setupWiring() {  
  
        QueryFetchers queryFetchers = ...;  
        MutationFetchers mutationFetchers = ...;  
        ProjectFetchers projectFetchers = ...;  
  
        return RuntimeWiring.newRuntimeWiring()  
            .build();  
    }  
}
```

# RUNTIME WIRING

## Verbinden von Feldern mit DataFetcher

- Im RuntimeWiring werden DataFetcher einzelnen Feldern zugewiesen

```
class GraphQLAPIConfiguration {  
    RuntimeWiring setupWiring() {  
  
        QueryFetchers queryFetchers = ....;  
        MutationFetchers mutationFetchers = ....;  
        ProjectFetchers projectFetchers = ....;  
  
        type Query {  
            return RuntimeWiring.newRuntimeWiring()  
                .type(newTypeWiring("Query")  
  
                .build();  
        }  
    }  
}
```

# RUNTIME WIRING

## Verbinden von Feldern mit DataFetcher

- Im RuntimeWiring werden DataFetcher einzelnen Feldern zugewiesen

```
class GraphQLAPIConfiguration {
    RuntimeWiring setupWiring() {

        QueryFetchers queryFetchers = ...;
        MutationFetchers mutationFetchers = ...;
        ProjectFetchers projectFetchers = ...;

        type Query {
            ping: String!
            project(id: ID!): Project
        }
    }

    return RuntimeWiring.newRuntimeWiring()
        .type(newTypeWiring("Query")
            .dataFetcher("ping", queryFetchers.ping)
            .dataFetcher("project", queryFetchers.projectById))

        .build();
    }
}
```

# RUNTIME WIRING

## Verbinden von Feldern mit DataFetcher

- Im RuntimeWiring werden DataFetcher einzelnen Feldern zugewiesen

```
class GraphQLAPIConfiguration {
    RuntimeWiring setupWiring() {

        QueryFetchers queryFetchers = ...;
        MutationFetchers mutationFetchers = ...;
        ProjectFetchers projectFetchers = ...;

        type Query {
            ping: String!
            project(id: ID!): Project
        }

        type Project {
            owner: User!
        }

        return RuntimeWiring.newRuntimeWiring()
            .type(newTypeWiring("Query")
                .dataFetcher("ping", queryFetchers.ping)
                .dataFetcher("project", queryFetchers.projectById))
            .type(newTypeWiring("Project").
                .dataFetcher("owner", projectFetchers.owner))

            .build();
    }
}
```

# RUNTIME WIRING

## Verbinden von Feldern mit DataFetcher

- Im RuntimeWiring werden DataFetcher einzelnen Feldern zugewiesen

```
type Query {  
  ping: String!  
  project(id: ID!): Project  
}  
  
type Project {  
  owner: User!  
}  
  
type Mutation {  
  addTask(projectId:...): Task  
}
```

```
class GraphQLAPIConfiguration {  
  RuntimeWiring setupWiring() {  
  
    QueryFetchers queryFetchers = ...;  
    MutationFetchers mutationFetchers = ...;  
    ProjectFetchers projectFetchers = ...;  
  
    return RuntimeWiring.newRuntimeWiring()  
      .type(newTypeWiring("Query")  
        .dataFetcher("ping", queryFetchers.ping)  
        .dataFetcher("project", queryFetchers.projectById))  
      .type(newTypeWiring("Project").  
        .dataFetcher("owner", projectFetchers.owner))  
      .type(newTypeWiring("Mutation").  
        .dataFetcher("addTask", mutationFetchers.addTask))  
      .build();  
  }  
}
```

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Ausführbares Schema erzeugen

- Statisches Schema und DataFetcher (Wirings) werden verknüpft
- Einstiegspunkt zum Ausführen von Queries

```
class GraphQLApiConfiguration {  
  
    public GraphQLSchema graphqlSchema() {  
  
        // Schritt 1: Schema-Beschreibung aus Datei einlesen  
        File schemaFile = new File("tasks.graphqls");  
  
        // Schritt 2+3: DataFetcher & RuntimeWiring (wie zuvor gesehen)  
        RuntimeWiring runtimeWiring = setupWiring();  
  
        SchemaGenerator schemaGenerator = new SchemaGenerator();  
  
        return schemaGenerator.makeExecutableSchema(  
            new SchemaParser().parse(schemaFile),  
            runtimeWiring  
        );  
    }  
}
```

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Queries per API ausführen

- Ergebnis wird in verschachtelter Map zurückgeliefert

- 👉 nhgraphql.projectmgmt.graphql.config.GraphQLApiConfiguration
- 👉 nhgraphql.projectmgmt.QueryExecutionTest

```
GraphQLSchema schema = ...;

GraphQL graphQL = GraphQL.newGraphQL(schema).build();

ExecutionInput executionInput =
    ExecutionInput.newExecutionInput
        .query("query { project { title tasks { title } } }")
        .context(new ProjectMgmtGraphQLContext())
        .build();

Map<String, Object> result = graphQL.execute(executionInput).toSpecification();
```

## Queries ausführen (per HTTP)

- Voraussetzung: GraphQL Schema ist erzeugt
- Variante 1: <https://github.com/graphql-java/graphql-java-spring>
  - REST Controller für Spring (Boot)
  - Stammt aus graphql-java Projektfamilie
  - Kein Support für Subscriptions zurzeit
- Variante 2: <https://github.com/graphql-java-kickstart/graphql-java-servlet>
  - HTTP Servlet (für Spring bzw Servlet Container)
  - Auch als Starter für Spring Boot verfügbar

## graphql-java-servlet

- Deployment Server-spezifisch
- Beispiel aus unserer Anwendung (Spring-basiert)

```
import javax.servlet.annotation.WebServlet;

import graphql.schema.GraphQLSchema;
import graphql.servlet.GraphQLHttpServlet;
import graphql.servlet.config.GraphQLConfiguration;

@WebServlet(urlPatterns = { "/graphql" }, loadOnStartup = 1)
public class GraphQLServlet extends GraphQLHttpServlet {
    @Autowired
    private GraphQLSchema schema;

    protected GraphQLConfiguration getConfiguration() {
        return GraphQLConfiguration
            .with(schema)
            .build();
    }
}
```

## Schema in der Beispiel-Anwendung

- Das Schema wird in GraphQLApiConfiguration.java erzeugt
  - Dort etwas anders, als vorhin gezeigt, weil mit Spring Hilfsmitteln
  - Konzept ist aber identisch, es wird ein GraphQLSchema erzeugt
- 
- Die GraphQL API wird über das graphql-java-servlet zur Verfügung gestellt

## ÜBUNG 2: DATAFETCHER IMPLEMENTIEREN

### Implementiere fehlende DataFetcher für unsere Anwendung

- Am **Query**: Felder `projects` und `project`
- Am **Project**: Felder `owner` und `task`

Die Änderungen müssen vorgenommen werden in:

- `nh.graphql.projectmgmt.graphql.fetcher.QueryDataFetchers`
- `nh.graphql.projectmgmt.graphql.fetcher.ProjectDataFetchers`
- `GraphQLApiConfiguration.setupWiring()`

dort sind entsprechende TODOs eingetragen.

- (Falls Du mit Übung 1 nicht fertig geworden bist, kannst Du die fertige Schema-Datei aus `01_schema_fertig` in deinen Workspace kopieren nach `src/main/resources`)

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

- Funktionierende Queries zum Testen auf der nächsten Slide
- Die Unit-Tests in `PAllProjectMgmtApplicationTests` sollten funktionieren (@Ignore entfernen)

Hinweis: manchmal funktioniert automatisches Reload nicht richtig (dubiose Reflection-Fehler zur Laufzeit), dann Server manuell neu starten

## ÜBUNG 2: RESOLVER IMPLEMENTIEREN

**Implementiere fehlende DataFetcher für unsere Anwendung**

Nach dem Implementieren sollten folgende Queries funktionieren:

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

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

Außerdem die JUnit-Tests in der Klasse ProjectMgmtApplicationTests!

# LAUFZEITVERHALTEN

# LAUFZEITVERHALTEN

## Ein Beispiel...

### DataFetcher

```
DataFetcher<List<Project>> projectsFetcher =  
    env -> projectRepository.getAll();
```

### Query

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

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

# LAUFZEITVERHALTEN

## Ein Beispiel...

**EIN** Datenbankzugriff  
(liefert  $n$  Projekte)

**$n$**  REST-Aufrufe  
(1x je Projekt)

```
-----  
DataFetcher<List<Project>> projectsFetcher =  
    env -> projectRepository.getAll();
```

```
-----  
DataFetcher<User> ownerFetcher =  
    env -> userService.getUser(  
        ((Project)env.getSource()).getOwnerId()  
    );
```

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

**$1+n$ -Problem** 😱

## Option 1: Asynchroner Data Fetcher

- DataFetcher können CompletableFuture-Objekte zurückgeben
  - async-Funktion dafür Convience
- Fetcher werden dann parallel ausgeführt

👉 Beispiel: estimation-Feld (*ProjectDataFetcher*)

👉 GraphQLApiConfiguration estimation-Feld hinzufügen

## Option 1: Asynchroner Data Fetcher

**EIN** Datenbankzugriff  
(liefert  $n$  Projekte)

**$n$**  REST-Aufrufe  
(1x je Projekt),  
**aber parallel!**

```
DataFetcher<List<Project>> projectsFetcher =  
    env -> projectRepository.getAll();  
  
import graphql.schema.AsyncDataFetcher.async;  
  
DataFetcher<User> ownerFetcher =  
    async(env -> userService.getUser(  
        ((Project)env.getSource()).getOwnerId()  
    ));
```

- Löst nicht das 1+n-Problem, aber kann Performance-Vorteil bedeuten

# LAUFZEITVERHALTEN

## Unser Query...

### DataFetcher

```
DataFetcher<List<Project>> projectsFetcher = ...;  
DataFetcher<User> ownerFetcher = async(...);
```

### Query

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

### Ergebnis

```
"data": {  
  "projects": [  
    { "owner": { "id": "U1", "name": "Nils Hartmann" } },  
    { "owner": { "id": "U2", "name": "Susi Mueller" } },  
    { "owner": { "id": "U3", "name": "Klaus Schneider" } }  
    { "owner": { "id": "U1", "name": "Nils Hartmann" } },  
    { "owner": { "id": "U4", "name": "Sue Taylor" } },  
    { "owner": { "id": "U1", "name": "Nils Hartmann" } }  
  ]  
}
```

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

# LAUFZEITVERHALTEN

## Unser Query...

### DataFetcher

```
DataFetcher<List<Project>> projectsFetcher = ...;  
DataFetcher<User> ownerFetcher = async(...);
```

### Query

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

### Ergebnis

```
"data": {  
  "projects": [  
    { "owner": { "id": "U1", "name": "Nils Hartmann" } },  
    { "owner": { "id": "U2", "name": "Susi Mueller" } },  
    { "owner": { "id": "U3", "name": "Klaus Schneider" } }  
    { "owner": { "id": "U1", "name": "Nils Hartmann" } },  
    { "owner": { "id": "U4", "name": "Sue Taylor" } },  
    { "owner": { "id": "U1", "name": "Nils Hartmann" } }  
  ]  
}
```

*n+1-Problem plus unnötiges, doppeltes Laden von Daten (-> Console)* 🤦‍♂️🤦‍♂️

# LAUFZEITVERHALTEN

## Unser Query... ...nächste Eskalation

### DataFetcher

```
DataFetcher<List<Project>> projectsFetcher = ...;  
  
DataFetcher<User> ownerFetcher = async(...);  
  
DataFetcher<User> assigneeFetcher = ...;
```

### Query

```
query {  
  projects {  
    owner {  
      id name  
    }  
    tasks { assignee { id name }  
  }  
}
```

# LAUFZEITVERHALTEN

## Unser Query... ...nächste Eskalation

### DataFetcher

```
DataFetcher<List<Project>> projectsFetcher = ...;  
  
DataFetcher<User> ownerFetcher = async(...);  
  
DataFetcher<User> assigneeFetcher = env -> userService....;
```

### Query

```
query {  
  projects {  
    owner {  
      id name  
    }  
    tasks { assignee { id name }  
  }  
}
```

Noch mehr User => noch mehr Zugriffe aus UserService 😱

Potentiell dieselben User wie beim Owner => noch mehr überflüssige Zugriffe 😫

Zwei DataFetcher => Lösung nicht "lokal" in einem DF machbar 😱

-> console!

### DataLoader: Fasst Aufrufe zusammen und cached Daten

- Konzept kommt ursprünglich aus der JavaScript Implementierung
- Zusammenfassen von Aufrufen, um unnötige Aufrufe zu vermeiden (Batching)
- Funktioniert ebenfalls asynchron
- Gelesene Daten werden (üblicherweise) für die Dauer eines Requests gecached

## LAUFZEITVERHALTEN: DATALOADER

### DataLoader: Fasst Aufrufe zusammen und cached Daten

- Konzept kommt ursprünglich aus der JavaScript Implementierung
- Zusammenfassen von Aufrufen, um unnötige Aufrufe zu vermeiden (Batching)
- Funktioniert ebenfalls asynchron
- Gelesene Daten werden (üblicherweise) für die Dauer eines Requests gecached

### Laden von Daten

- Das eigentliche Laden der Daten wird in **BatchLoader** verschoben
- BatchLoader werden bei der Konfiguration des Schemas erzeugt und können in den DataFetcher abgefragt werden

# LAUFZEITVERHALTEN: DATALOADER

## DataLoader 1: Laden von Daten

- Der BatchLoader wird von GraphQL mit einer *Menge* von IDs aufgerufen, die aus einer *Menge* von DataFetcher-Aufrufen stammen
- Für jede ID muss das gewünschte Objekt zurückgeliefert werden
- Es werden nur eindeutige IDs an den BatchLoader übergeben (keine doppelten)

```
class ProjectDataLoaders {  
    public BatchLoader<String, <Optional<User>> userBatchLoader = new BatchLoader<>() {  
        public CompletableFuture<List<Optional<User>>> load(List<String> keys) {  
  
            // Für jeden Key wird der User geladen und zurückgegeben  
            return CompletableFuture.supplyAsync(() -> keys.stream()  
                .map(userService::getUser)  
                .collect(Collectors.toList())  
            );  
        }  
    }  
}
```

# LAUFZEITVERHALTEN: DATALOADER

## DataLoader 1: Laden von Daten

- Variante: BatchLoaderWithContext bekommt den GraphQL Context übergeben

```
class ProjectDataLoaders {  
    public BatchLoaderWithContext<String, <Optional<User>>> userBatchLoader = new BatchLoaderWithContext <>() {  
        public CompletableFuture<List<Optional<User>>> load(List<String> keys, BatchLoaderEnvironment env) {  
            ProjectMgmtGraphQLContext context = env.getContext();  
            UserService userService = context.getUserService();  
  
            // Für jeden Key wird der User geladen und zurückgegeben  
            return CompletableFuture.supplyAsync(() -> keys.stream()  
                .map(userService::getUser)  
                .collect(Collectors.toList())  
            );  
        }  
    }  
}
```

## DataLoader 2: Einbinden im DataFetcher

- Im DataFetcher wird das Laden an den BatchLoader delegiert
- GraphQL wartet mit dem Aufruf des BatchLoaders so lange wie möglich
- Alle bis dahin abgefragten Ids werden gesammelt und zusammen an den BatchLoader übergeben

```
public DataFetcher ownerFetcher = new DataFetcher<>() {  
    public Object get(DataFetchingEnvironment env) {  
        // wie bisher  
        Project project = env.getSource();  
        String userId = project.getOwnerId();  
  
        // kein UserService-Zugriff mehr, sondern DataLoader verwenden  
  
        DataLoader<String, User> dataLoader = env.getDataLoader("userDataLoader");  
        return dataLoader.load(userId);  
    }  
};  
}  
  
// Analog für assignee-Fetcher im Project
```

# LAUFZEITVERHALTEN: DATALOADER

## DataLoader 3: Registrieren

- Die DataLoader werden in einer DataLoaderRegistry registriert
- Die DataLoaderRegistry kann beim Erzeugen des GraphQL Contexts befüllt werden

```
class ProjectMgmtGraphQLContextBuilder {  
    public GraphQLContext build(...) {  
        GraphQLContext context = new ...;  
  
        ProjectDataLoaders dataLoaders = new ProjectDataLoaders();  
        context.getDataLoaderRegistry().register("userDataLoader", dataLoaders.userDataLoader);  
  
        return context;  
    }  
}
```

## LAUFZEITVERHALTEN: DATALOADER

### Ergebnis: Aufrufe wurden reduziert

- Wir haben "nur" noch 1+Anzahl-eindeutiger-User-Zugriffe über ganzen Query
- ➡ console

```
query {  
    owner @useDataLoader {  
        id name  
    }  
  
    projects {  
        tasks {  
            assignee @useDataLoader {  
                id name  
            }  
        }  
    }  
}
```

## LAUFZEITVERHALTEN: DATALOADER

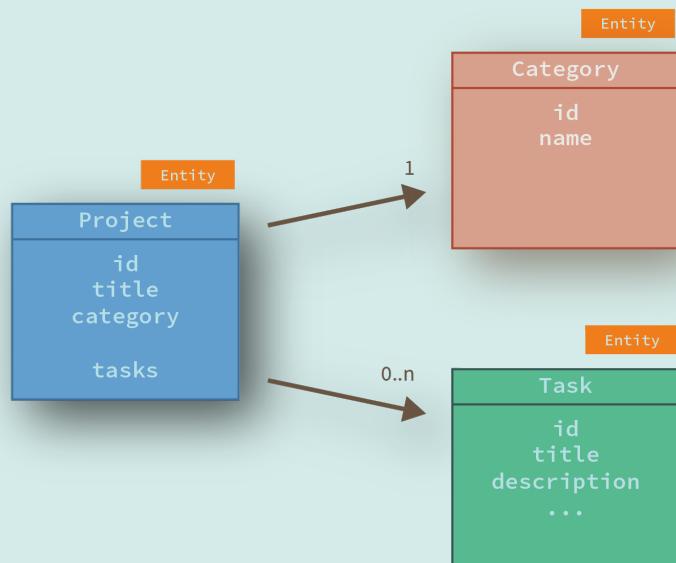
### Ergebnis: Aufrufe wurden reduziert

- Wir haben "nur" noch 1+Anzahl-eindeutiger-User-Zugriffe über ganzen Query
- ➡ console
- Weitere mögliche Verbesserungen:
  - Caching über Query-Grenze hinaus (je nach Domaine)
  - Sofern remote API das unterstützt, könnten Zugriffe noch weiter reduziert werden:

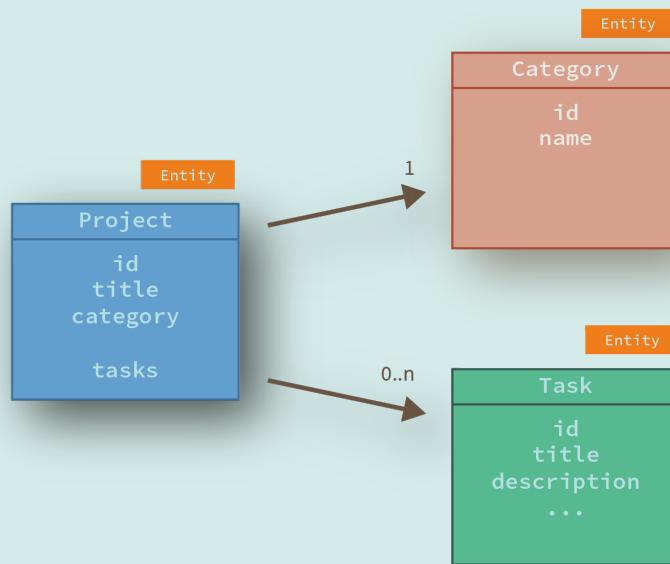
```
public BatchLoader<String, User> userBatchLoader = new BatchLoader<>() {  
    public CompletableFuture<ListUser>> load(List<String> keys) {  
        return CompletableFuture.supplyAsync(() -> userService.loadAll(keys))  
    }  
}
```

# LAUFZEITVERHALTEN

## Datenbankzugriffe: Unser (JPA-)Model



## Datenbankzugriffe: Ein Query



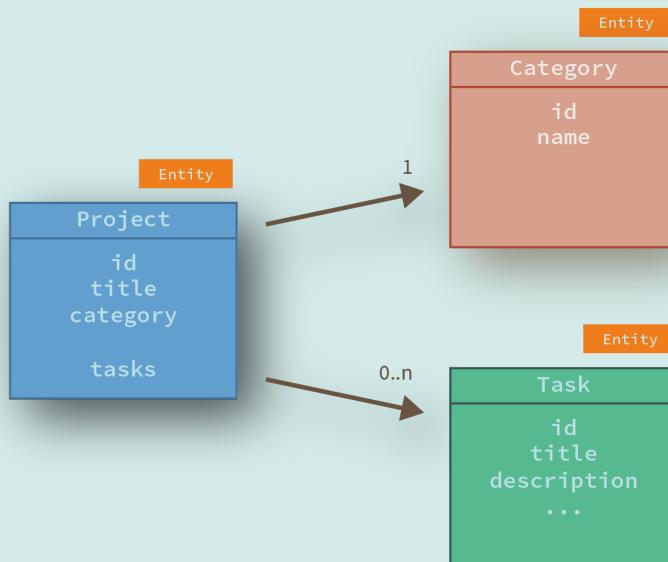
```
query {
    projects {
        title
        category { name }
    }
}
```

*Frage: Wie mappen wir die Beziehungen Project-Category? Lazy? Eager? 🤔*

*Was passiert im einen bzw. anderen Fall?*

# LAUFZEITVERHALTEN

## Datenbankzugriffe: Ein Query

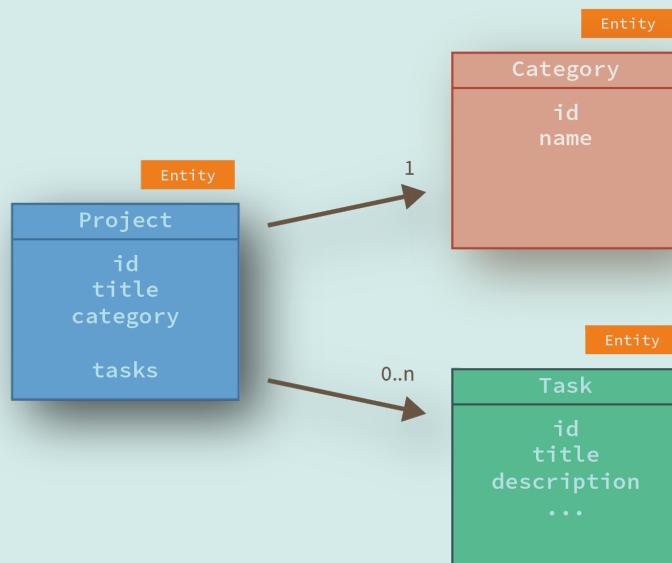


```
query {
    projects {
        title
        tasks { id title }
    }
}
```

*Frage: Wie mappen wir die Beziehungen Project-Task? Lazy? Eager? 🤔*

*Was passiert im einen bzw. anderen Fall?*

## Datenbankzugriffe: Ein Query



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

*Frage: Wie mappen wir unsere Beziehungen denn nun? 🤔*

## Datenbankzugriffe: Ein Query

*Abhängig vom konkreten GraphQL Query benötigen wir optimale Zugriffe!*

```
// Bestehender DataFetcher
public DataFetcher<List<Project>> projectsFetcher = new DataFetcher<>() {
    public Iterable<Project> get(DataFetchingEnvironment env) {

        // TODO: Optimieren je nach Query
        return projectRepository.findAll();
    }
};
```

*Welche Informationen benötigen wir dafür?*

# LAUFZEITVERHALTEN: DATENBANKZUGRIFFE

## SelectionSet: Enthält *alle* abgefragten Felder des Queries

- Zugriff über DataFetchingEnvironment
- Abhängig von abgefragten Feldern können wir JPA/SQL/... Queries zusammenbauen bzw optimieren

```
public DataFetcher<Iterable<Project>> projects = new DataFetcher<>() {  
    public Iterable<Project> get(DataFetchingEnvironment env) throws Exception {  
  
        boolean withCategory = env.getSelectionSet().contains("category");  
        boolean withTasks = env.getSelectionSet().contains("tasks");  
  
        return projectRepository.findAll(withCategory, withTasks);  
    }  
};
```

## SelectionSet: Optimierter DB-Zugriff

- Beispiel mit JPA FetchGraph

```
public class ProjectRepository {  
    public List<Project> findAll(boolean withCategory, boolean withTasks) {  
  
        EntityGraph<Project> entityGraph = entityManager.createEntityGraph(Project.class);  
        if (withCategory) {  
            entityGraph.addSubgraph("category");  
        }  
        if (withTasks) {  
            entityGraph.addSubgraph("tasks");  
        }  
  
        TypedQuery<Project> query = em.createQuery("SELECT p FROM Project p", Project.class);  
        query.setHint("javax.persistence.fetchgraph", entityGraph);  
  
        return query.getResultList();  
    }  
}
```

## SelectionSet: Optimierter DB-Zugriff

- Beispiel mit JPA FetchGraph
- 👉 console

```
query {
    projects @useEntityGraph {
        title
        tasks { id title }
        category { name }
    }
}
```

### Ermittlung der Daten in GraphQL

- Es gibt für *jedes* Feld einen DataFetcher (Default: PropertyDataFetcher)
- DataFetcher können asynchron arbeiten (CompletableFuture zurückliefern)
- Mit DataLoader können Aufrufe durch Caching und Batching eingespart werden
- Über SelectionSet können Datenbank-Abfragen optimiert werden

## ÜBUNG 3: DATAFETCHING OPTIMIEREN

**Optimiere REST- und Datenbankzugriffe**

## ÜBUNG 3: DATAFETCHING OPTIMIEREN

### Optimiere REST- und Datenbankzugriffe

#### Schritt 1: REST-Zugriffe mit DataLoader optimieren

- In `nh.graphql.projectmgmt.graphql.fetcher.ProjectDataLoaders` musst Du einen `BatchLoaderWithContext` implementieren (s. TODOs dort)
- In `ProjectMgmtGraphQLContextBuilder.addDataLoaders(DataLoaderRegistry)` musst Du den BatchLoader an der DataLoaderRegistry hinzufügen
- In `ProjectDataFetchers` und `TaskFetchers` kannst Du deinen DataLoader verwenden, um die User-Objekte zu laden
- Überprüfe im Logging, ob die Zugriffe auf den User-Service reduziert werden, dazu am Besten in deinem DataLoader entsprechende Log-Ausgaben einbauen. Auch die UserService-Application loggt alle Anfragen auf ihrer Konsole

## ÜBUNG 3: DATAFETCHING OPTIMIEREN

### Optimiere REST- und Datenbankzugriffe

#### Schritt 2: Datenbank-Zugriffe optimieren

Vorbereitung: Kopiere `code/material/ProjectRepository.java` in deinen Workspace und ersetze die bestehende Klasse `nh.graphql.projectmgmt.domain.ProjectRepository`

- In Übung 1 haben wir DataFetchers für 'project' und 'projects' am Query implementiert.
- In diesen DataFetchern sollst Du nun jeweils prüfen ob im Query das 'tasks' und/oder 'category'-Feld enthalten ist und die Informationen an die Aufrufe im ProjectRepository übergeben.  
Falls deine DataFetcher nicht funktionieren, kannst Du dir den fertigen Stand aus    kopieren.
- Die jeweils von Hibernate generierten SQL-Aufrufe kannst Du im Logging auf der Konsole kontrollieren

*graphql-java-tools*

*<https://www.graphql-java-kickstart.com/tools/>*

- Abstraktion, basierend auf graphql-java, arbeitet mit POJOs

## Resolver mit graphql-java-tools

- Resolver sind POJOs, Schema weiterhin per SDL
- Beim Starten wird überprüft, ob für alle Felder ein Resolver vorhanden ist

## Resolver mit graphql-java-tools

- Resolver sind POJOs, Schema weiterhin per SDL
- Beim Starten wird überprüft, ob für alle Felder ein Resolver vorhanden ist

```
public class ProjectAppQueryResolver implements GraphQLQueryResolver {  
  
    // Methoden an Resolver-Klasse entsprechen Feld-Namen im Objekt  
    // Argumente werden als Java-Parameter übergeben  
    public String ping(String msg) {  
        return "Hello, " + msg;  
    }  
  
    // Zugriff auf DataFetchingEnvironment aus graphql-java möglich  
    public List<Project> projects(DataFetchingEnvironment env) {  
        boolean withCategory = ...;  
        boolean withTasks = ...;  
        return beerRepository.findAll(withCategory, withTasks);  
    }  
}
```

## Resolver mit graphql-java-tools

- Komplexe Argumente (input types) werden ebenfalls als Argument übergeben

```
class AddTaskInput {  
    private String title;  
    private String description;  
    ...  
}  
  
public class ProjectAppMutationResolver implements GraphQLMutationResolver {  
  
    public Task addTask(String projectId, AddTaskInput input) {  
        return taskService.createTask(projectId, input);  
    }  
}
```

## Resolver mit graphql-java-tools

- Resolver können für eigene Typen gebaut werden
- Parent-Objekt wird als Parameter an die Resolver-Methode übergeben

```
public class ProjectResolver implements GraphQLResolver<Project> {  
  
    public User getOwner(Project parent) {  
        return userService.getUser(parent.getOwnerId());  
    }  
  
}
```

# GRAPHQL-JAVA-TOOLS FÜR SPRING BOOT

## Spring Boot Starter

- <https://github.com/graphql-java-kickstart/graphql-spring-boot>
- Basiert auf Resolvern (aus graphql-java-tools)

# GRAPHQL-JAVA-TOOLS FÜR SPRING BOOT

## Spring Boot Starter

- <https://github.com/graphql-java-kickstart/graphql-spring-boot>
- Basiert auf Resolvern (aus graphql-java-tools)
- Mergt alle Schema-Dateien im Klassenpfad zusammen (\*.graphqls)
- Resolver werden als Beans annotiert (zB. @Component) und automatisch dem Schema hinzugefügt
- Servlet-Konfiguration erfolgt per application.properties
- GraphiQL (API Explorer) kann ebenfalls per Konfiguration aktiviert werden

## WEITERE THEMEN

- **Habt ihr Fragen? Was sollen wir uns noch anschauen?**

# SCHEMA DESIGN

## Unser Schema

- Welche Probleme könnten wir hiermit haben?

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Project {  
    id: ID!  
    title: String!  
    description: String!  
    owner: User!  
    category: Category!  
    tasks: [Task!]!  
    task(id: ID!): Task  
}  
  
type Category {  
    id: ID!  
    name: String!  
}  
  
enum TaskState {  
    NEW RUNNING FINISHED  
}  
  
type Task {  
    id: ID!  
    title: String!  
    description: String!  
    state: TaskState!  
    assignee: User!  
    toBeFinishedAt: String!  
}  
  
type Query {  
    users: [User!]!  
    projects: [Project!]!  
    project(id: ID!): Project  
}  
  
input AddTaskInput {  
    title: ID!  
    description: ID!  
    toBeFinishedAt: String  
    assigneeId: ID!  
}  
  
type Mutation {  
    addTask(project: ID!, input: AddTaskInput!):  
        Task!  
    updateTaskState(taskId: ID!, newState: TaskState!):  
        Task!  
}
```

# SCHEMA DESIGN

## Unser Schema

- Welche Probleme könnten wir hiermit haben?

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Project {  
    id: ID!  
    title: String!  
    description: String!  
    owner: User!  
    category: Category!  
    tasks: [Task!]!  
    task(id: ID!): Task  
}  
  
type Category {  
    id: ID!  
    name: String!  
}  
  
enum TaskState {  
    NEW RUNNING FINISHED  
}  
  
type Task {  
    id: ID!  
    title: String!  
    description: String!  
    state: TaskState!  
    assignee: User!  
    toBeFinishedAt: String!  
}  
  
type Query {  
    users: [User!]!  
    projects: [Project!]!      Paginierung?  
    project(id: ID!): Project  
}  
  
input AddTaskInput {  
    title: ID!  
    description: ID!  
    toBeFinishedAt: String  
    assigneeId: ID!  
}  
  
type Mutation {  
    addTask(project: ID!, input: AddTaskInput!):  
        Task!  
    updateTaskState(taskId: ID!, newState: TaskState!):  
        Task!  
}
```

# SCHEMA DESIGN

## Unser Schema

- Welche Probleme könnten wir hiermit haben?

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Project {  
    id: ID!  
    title: String!  
    description: String!  
    owner: User!  
    category: Category!  
    tasks: [Task!]!  
    task(id: ID!): Task  
}  
  
type Category {  
    id: ID!  
    name: String!  
}  
  
enum TaskState {  
    NEW RUNNING FINISHED  
}  
  
type Task {  
    id: ID!  
    title: String!  
    description: String!  
    state: TaskState!  
    assignee: User!  
    toBeFinishedAt: String!  
}  
  
type Query {  
    users: [User!]!  
    projects: [Project!]!  
    project(id: ID!): Project  
} Abwärtskompatibilität?  
  
input AddTaskInput {  
    title: ID!  
    description: ID!  
    toBeFinishedAt: String  
    assigneeId: ID!  
}  
  
type Mutation {  
    addTask(project: ID!, input: AddTaskInput!):  
        Task!  
    updateTaskState(taskId: ID!, newState: TaskState!):  
        Task!  
}
```

# SCHEMA DESIGN

## Unser Schema

- Welche Probleme könnten wir hiermit haben?

```
type User {  
    id: ID!  
    login: String!  
    name: String!  
}  
  
type Project {  
    id: ID!  
    title: String!  
    description: String!  
    owner: User!  
    category: Category!  
    tasks: [Task!]!  
    task(id: ID!): Task  
}  
  
type Category {  
    id: ID!  
    name: String!  
}  
  
enum TaskState {  
    NEW RUNNING FINISHED  
}  
  
type Task {  
    id: ID!  
    title: String!  
    description: String!  
    state: TaskState!  
    assignee: User!  
    toBeFinishedAt: String!  
}  
  
type Query {  
    users: [User!]!  
    projects: [Project!]!  
    project(id: ID!): Project  
}  
  
input AddTaskInput {  
    title: ID!  
    description: ID!  
    toBeFinishedAt: String!  
    assigneeId: ID!  
}  
  
type Mutation {  
    addTask(project: ID!, input: AddTaskInput!): Task!  
    updateTaskState(taskId: ID!, newState: TaskState!): Task!  
}
```

Fehlerbehandlung?

# SCHEMA DESIGN

Was können wir verbessern, um die Probleme zu lösen?

👉 Welche Ideen habt ihr?

```
type User {  
  id: ID!  
  login: String!  
  name: String!  
}  
  
type Project {  
  id: ID!  
  title: String!  
  description: String!  
  owner: User!  
  category: Category!  
  tasks: [Task!]!  
  task(id: ID!): Task  
}  
  
type Category {  
  id: ID!  
  name: String!  
}  
  
enum TaskState {  
  NEW RUNNING FINISHED  
}  
  
type Task {  
  id: ID!  
  title: String!  
  description: String!  
  state: TaskState!  
  assignee: User!  
  toBeFinishedAt: String!  
}  
  
type Query {  
  users: [User!]!  
  projects: [Project!]!  
  project(id: ID!): Project  
}  
  
input AddTaskInput {  
  title: ID!  
  description: ID!  
  toBeFinishedAt: String  
  assigneeId: ID!  
}  
  
type Mutation {  
  addTask(project: ID!, input: AddTaskInput!):  
    Task!  
  updateTaskState(taskId: ID!, newState: TaskState!):  
    Task!  
}
```



# Vielen Dank!

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

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

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