

NILS HARTMANN

<https://nilshartmann.net>

Apollo GraphQL

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

Slides (PDF): <https://react.schule/apollo-workshop>

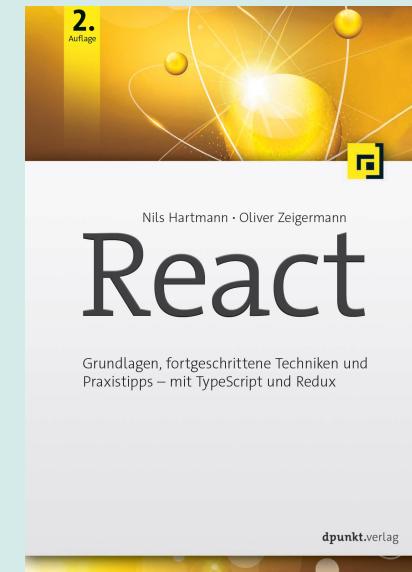
NILS HARTMANN

nils@nilshartmann.net

Freiberuflicher Software-Architekt und -Entwickler

Java
JavaScript, TypeScript
React
GraphQL

Trainings, Workshops,
Coaching



<https://reactbuch.de>

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

...und ihr?

Bitte stellt euch kur vor:

was ist Euer technischer Hintergrund?

habt ihr bereits graphql-Erfahrung?

Erwartungen, Fragen für heute?

ZEITPLAN...

19. Mai, 9:00 – 13:00

20. Mai, 9:00 – 13:00

GraphQL Server mit Apollo GraphQL

25. Mai, 9:00 – 13:00

GraphQL Clients mit Apollo (Beispiel: React)

Zwischendurch: Pausen ☕ 😴

Jederzeit:

Fragen, Diskussionen, Meinungen

Beteiligt Euch per Chat oder Audio

AGENDA

1. GraphQL Grundlagen und Abfragesprache

2. Backend mit Apollo Server:

- Schema
- Resolver

3. Zugriff auf Datenbank und REST-APIs

4. Offene Fragen, Q&A, 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/>

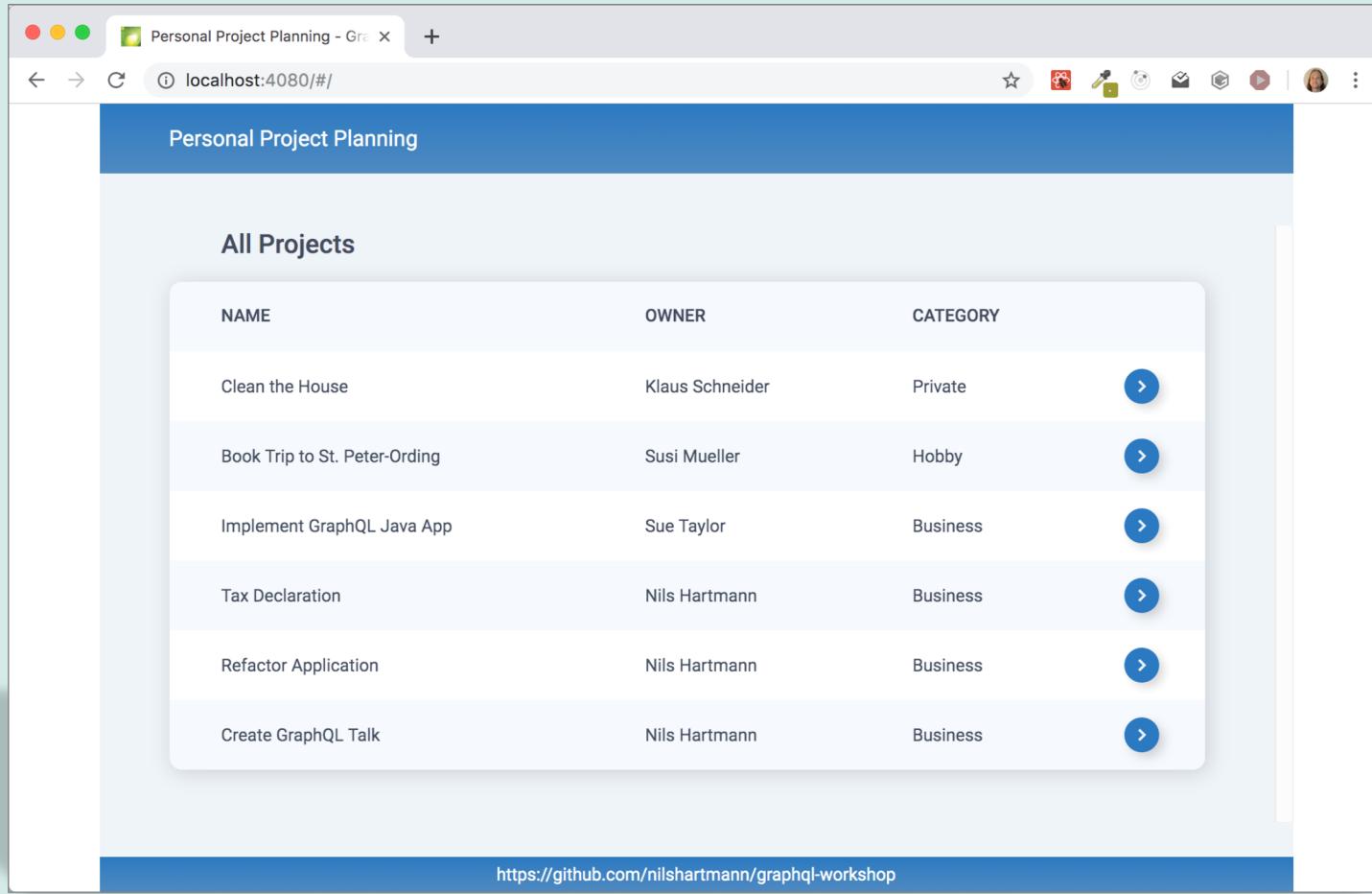
- 2015 von Facebook erstmals veröffentlicht
- Weitere Entwicklung seit 2018 in GraphQL Foundation
- Umfasst:
 - Query Sprache und -Ausführung
 - Schema Definition Language
 - Nicht: Implementierung
 - Referenz-Implementierung: graphql-js

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 != Apollo

- Apollo ist "nur" ein Anbieter von GraphQL-Lösungen



Die Beispiel Anwendung

<http://localhost:4080>

The screenshot shows the GraphQL Playground interface running at localhost:4000. On the left, a query editor window displays the following GraphQL code:

```
query AllProjectsQuery {
  projects {
    id
    title
    description
    owner {
      name
    }
  }
}
```

The cursor is hovering over the word "login" in the "owner" field of the first project. A tooltip provides a detailed description: "String! The login is used by the user to log in to our System". Below the query editor, there are sections for "QUERY VARIABLES" and "HTTP PREVIEW".

The right side of the interface is a documentation sidebar with tabs for "DOCS", "SCHEMA", and "GRAPHQL". The "SCHEMA" tab is active, showing the Prisma schema definition:

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

type User {
  id: ID!
  name: String!
  ping: String!
  users: [User!]!
  user(...): User!
  projects: [Project!]!
  project(...): Project!
}

type Task {
  id: ID!
  title: String!
  description: String!
  owner: User!
  onNewTask: Task!
  onTaskChange(...): Task!
}
```

The "DOCS" tab contains links to "QUERIES", "MUTATIONS", "SUBSCRIPTIONS", and "ARGUMENTS". The "TYPE DETAILS" section provides a brief overview of the "Project" type.

Demo: Playground

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

<http://localhost:4000>

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

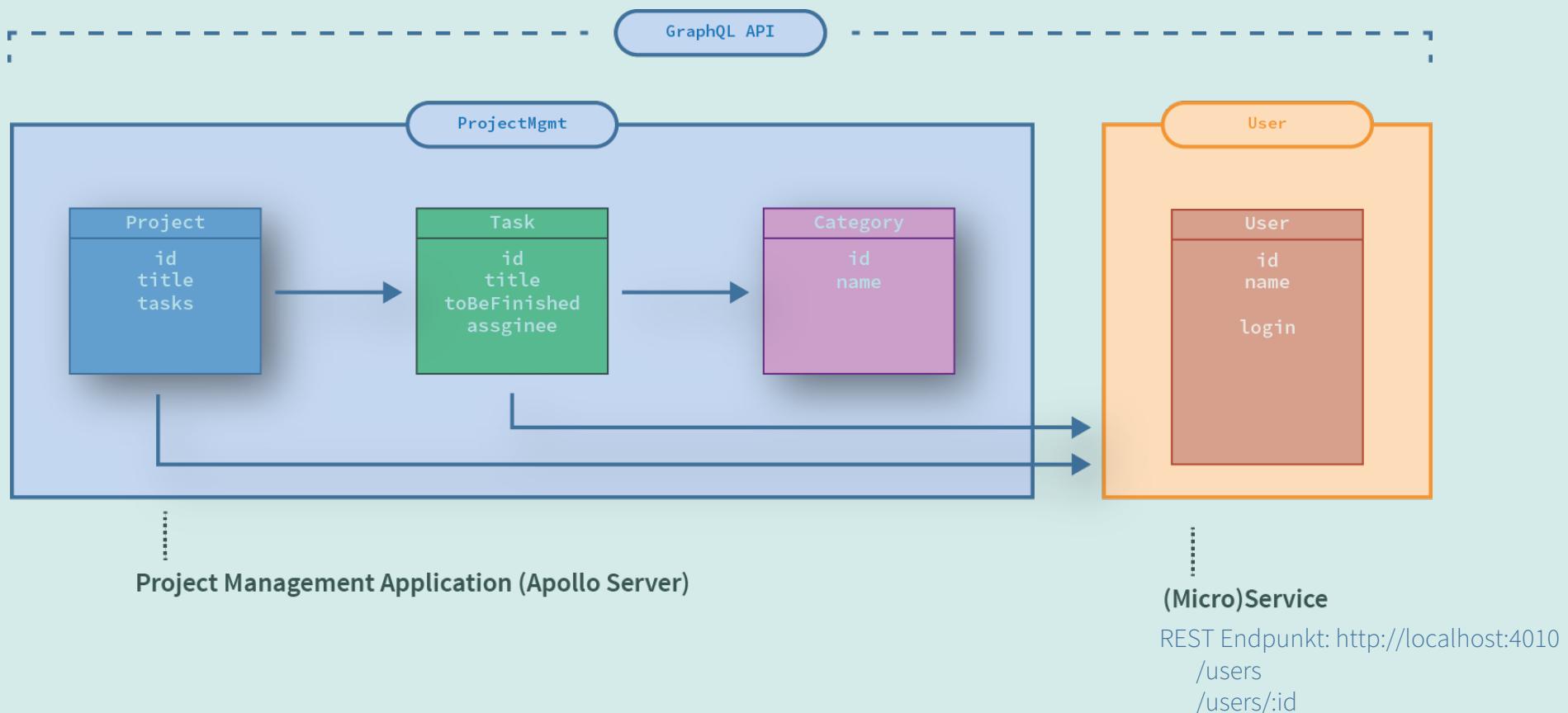
- <https://graphql.org>

GraphQL

TEIL 1: ABFRAGEN UND SCHEMA

GRAPHQL EINSATZSzenariEN

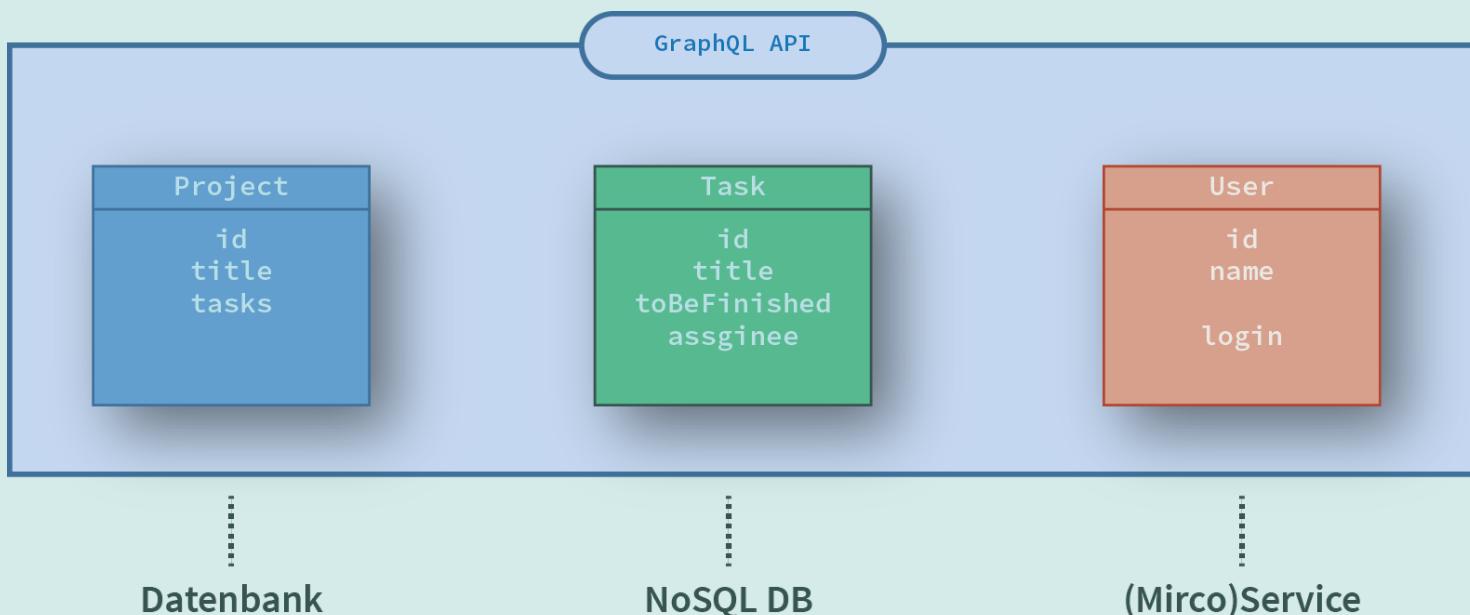
"Architektur" Project Beispiel Anwendung



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*

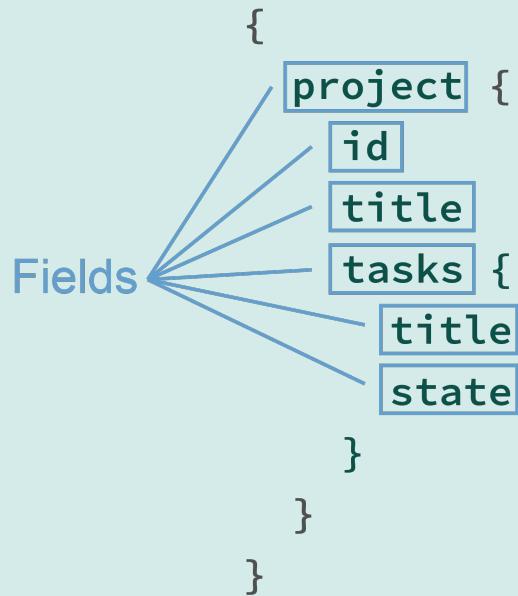


GraphQL ist nur ein kleiner (?) Teil im Stack

- Spec definiert nur die Sprache
- *Apollo unterstützt bei der Implementierung*
- *aber: viele, viele Fragen/Entscheidungen sind Projekt-abhängig*
- *Das hat auch Konsequenzen für diese Schulung!*

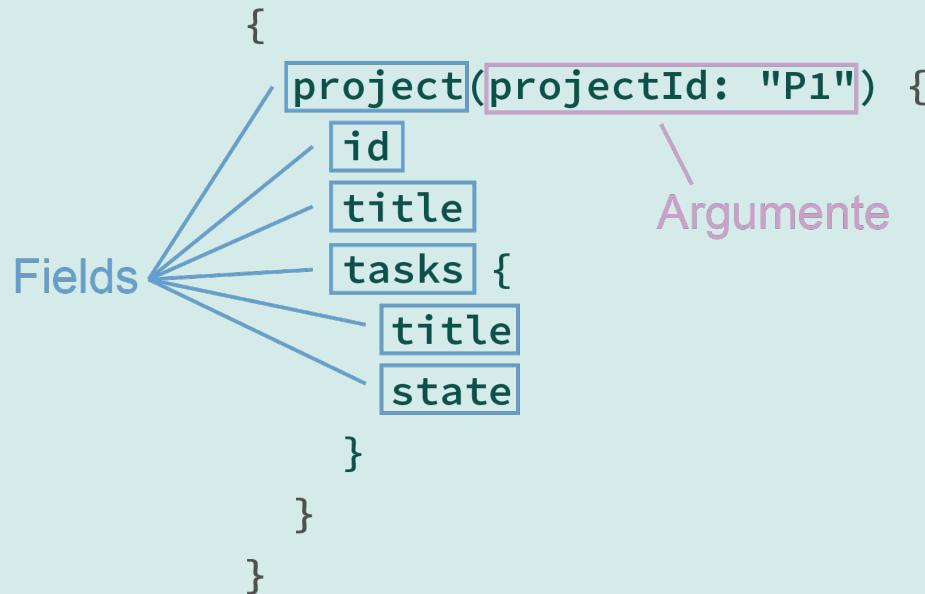
Die GraphQL Query Sprache

QUERY LANGUAGE



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

QUERY LANGUAGE



- 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: FRAGMENTS

Fragmente: Wiederverwendbare Sub-Selections

-  GraphiQL!

```
fragment UserWithIdAndName on User {  
    id  
    name  
}
```

QUERY LANGUAGE: FRAGMENTS

Fragmente: Wiederverwendbare Sub-Selections

```
fragment UserWithIdAndName on User {  
    id  
    name  
}  
  
query {  
    projects {  
  
User! ----- owner {  
        ...UserWithIdAndName  
    }  
  
    }  
}
```

QUERY LANGUAGE: FRAGMENTS

Fragmente: Wiederverwendbare Sub-Selections

```
fragment UserWithIdAndName on User {  
    id  
    name  
}  
  
query {  
    projects {  
  
User! ----- owner {  
        ...UserWithIdAndName  
    }  
  
    tasks {  
        assignee {  
            ...UserWithIdAndName  
        }  
    }  
}
```

QUERY LANGUAGE: FRAGMENTS

Fragmente: Wiederverwendbare Sub-Selections

The screenshot shows a GraphQL playground interface with the title "GRAPHQL EXAMPLE". The query editor contains the following code:

```
1 fragment UserWithIdAndName on User {  
2   id  
3   name  
4 }  
5  
6 query {  
7   projects {  
8     owner {  
9       ...UserWithIdAndName  
10      }  
11    }  
12  
13   tasks {  
14     assignee {  
15       ...UserWithIdAndName  
16     }  
17   }  
18 }  
19  
20 }
```

The results pane displays the JSON response from the GraphQL endpoint `ef8ddc125208.ngrok.io/?query=fragment UserWithIdAndName`. The response includes a "data" field with "projects" and "tasks" arrays. Each item in these arrays has an "owner" or "assignee" field, respectively, which is a reference to the `UserWithIdAndName` fragment. Dashed orange arrows point from the fragment reference in the query to the corresponding object in the response.

```
{  
  "data": {  
    "projects": [  
      {  
        "owner": {  
          "id": "U1",  
          "name": "Nils Hartmann"  
        }  
      },  
      {  
        "tasks": [  
          {  
            "assignee": {  
              "id": "U1",  
              "name": "Nils Hartmann"  
            }  
          },  
          {  
            "assignee": {  
              "id": "U2",  
              "name": "Susi Mueller"  
            }  
          },  
          {  
            "assignee": {  
              "id": "U3",  
              "name": "Hans Müller"  
            }  
          }  
        ]  
      }  
    ]  
  }  
}
```

QUERY LANGUAGE: OPERATIONS

Operation: Variablen

- Variablen können in einem Query für Platzhalter verwendet werden
- Ähnlich Prepared Statement in SQL
- Variablen werden in einem separaten Objekt an den Server geschickt

👉 GraphiQL!

QUERY LANGUAGE: OPERATIONS

Operation: Variablen

- Variablen können in einem Query für Platzhalter verwendet werden
 - Ähnlich Prepared Statement in SQL
 - Variablen werden in einem separaten Objekt an den Server geschickt
- 👉 GraphiQL!

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

Variable usage

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",
  assgineeId: "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
  }
}
```

ÜBUNG: QUERIES AUSFÜHREN

Mach dich mit dem Playground und der Query-Sprache vertraut

- Öffne den Playground auf meinem Computer (**URL im Chat**)

Mach' dich mit der API der Projektverwaltung App vertraut

1. Führe einen Query aus, mit dem Du *alle* Projekte und *alle* Benutzer (insb. jeweils die IDs) erhältst
2. Such dir eins der bestehenden Projekte aus und führe eine Mutation aus, mit der Du eine neue Aufgabe einem bestehenden Projekt hinzufügst

Bitte hebe deine Hand in Teams, wenn Du fertig bist 

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

- (JSON-)Map mit drei Feldern auf Root-Ebene
- data: wie gesehen

```
{  
  "data": {"projects": [ . . . ] },  
  
}
```

EXECUTING QUERIES

Antwort vom Server

- errors: Fehler, die während der Verarbeitung aufgetreten sind
- trotzdem in der Regel HTTP 200 !

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

QUERIES AUSFÜHREN

Antwort vom Server

- extensions: proprietäre Erweiterungen von GraphQL Frameworks
- Beispiel: Trace-Informationen von Apollo

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

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>

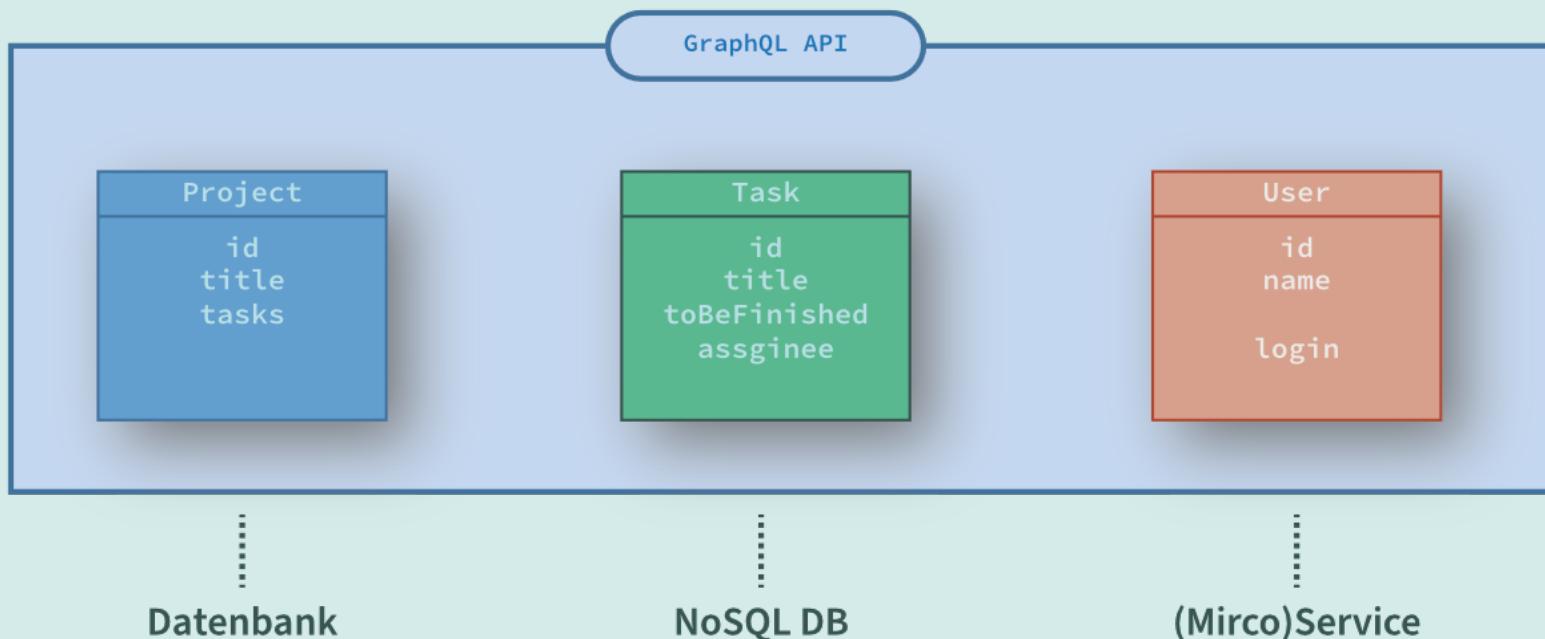
GraphQL Server

TEIL 2: RUNTIME-UMGEBUNG (AKA: EURE ANWENDUNG)

GRAPHQL RUNTIME

GraphQL macht keine Aussage, wo unsere Daten herkommen

- 👉 Ermittlung der angefragten Daten ist unsere Aufgabe
- 👉 Daten können aus unterschiedlichen Quellen kommen



*"A community building flexible open source **tools for**
GraphQL."*

- <https://github.com/apollographql>

Apollo

2019

"Apollo is the industry-standard GraphQL implementation, providing the data graph layer that connects modern apps to the cloud."

- <https://www.apollographql.com/>

Apollo

2021

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
 - "Federation": Zusammenführen verschiedener GraphQL APIs

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)

GRAPHQL SCHEMA

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)

Root-Type
("Query")

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

Root-Fields

Root-Type
("Mutation")

```
type Mutation {  
    addTask(newTask: AddTaskInput): Task!  
}
```

Input Type

Root-Type
("Subscription")

```
type Subscription {  
    onNewTask: Task!  
    onTaskChange(projectId: ID!): Task!  
}
```

Root-Types und Types: Es gibt nur "Types"

- Root-Types und fachliche Types sind sehr ähnlich:
 - Root-Types haben standardisierte Namen
 - Root-Types kommen nur auf Root-Ebene vor (logisch...)
 - Fachliche Typen gehen überall, nur nicht auf Root-Ebene
- Ansonsten aber identisch:
 - ihr fragt daraus Felder ab
 - werden wir auch bei der Implementierung sehen

GRAPHQL SCHEMA

Type-Definition in Apollo Server über Schema-Definition-Language

- Dokumentation kann mit Markdown-Code versehen werden
- """ Multi-Line, " Single-Line

```
// schema.js

module.exports = gql` 
  """
    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
  }
`;
```

GRAPHQL SCHEMA

Schema: Instrospection

- Root-Fields "__schema" und "__type" (Beispiel)
- Kann/sollte in Produktion ausgeschaltet werden
- ➡️ Playground

```
query {  
  __type(name: "Project") {  
    name  
    kind  
    description  
    fields {  
      name description  
      type { ofType { name } }  
    }  
  }  
}
```

```
{  
  "data": {  
    "__type": {  
      "name": "Project",  
      "kind": "OBJECT",  
      "description": "A **Project** is the central entity in our system.\nIt is owned by a **User**\nand have 0..n **Tasks** assigned to it.  
Projects can be grouped by a **Category**\nto make management easier.",  
      "fields": [  
        {  
          "name": "id",  
          "description": null,  
          "type": {  
            "ofType": {  
              "name": "ID"  
            }  
          }  
        },  
        ...  
      ]  
    }  
  }  
}
```

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 Definition
(schema.js)

```
const { gql } = require("apollo-server");

module.exports = gql`  
  type Project {  
    id: ID!  
    title: String!  
    description: String!  
    tasks: [Task!]!  
  }  
  
  type Task { . . . }  
  
  type Query {  
    projects: [Project!]!  
    project(projectId: ID!): Project  
  }  
`;
```

Root-Fields (erforderlich)

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 = gql`type Project { ... } `;
```

```
// query.js
module.exports = gql`type Query { ... } `;
```

```
// server.js
const projectTypes = require("./projects");
const queryTypes = require("./query");

const server = new ApolloServer({
  typeDefs: [projectTypes, queryTypes],
  ...
});
```

VORBEREITUNG ÜBUNGEN

Vorbereitung

Vor der Übung richten wir gemeinsam den Workspace ein

VORBEREITUNG ÜBUNGEN

Schritt 1: Klonen

1. Bitte klonen zunächst das Repository
 - If not done already, please clone the workspace:

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

2. Wenn ihr fertig seid, bitte Hand heben in Teams 

VORBEREITUNG ÜBUNGEN

Das Repository

GRAPHQL-APOLLO-WORKSHOP

- > app
- ✓ code-backend
 - > 01_schema
 - > 02_resolver
 - > 03_context
 - > 04_union
 - > userservice
 - > workspace
- ✓ slides
 - ≡ graphql-apollo-workshop.pptx

Das Workshop Repository

Lösungen für die Übungen

Fertiger userservice (nur starten)

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

Slides

VORBEREITUNG ÜBUNGEN

Das Repository

GRAPHQL-APOLLO-WORKSHOP

- > app
- ✓ code-backend
 - > 01_schema
 - > 02_resolver
 - > 03_context
 - > 04_union
 - > userservice
 - > workspace
- ✓ slides
 - ≡ graphql-apollo-workshop.pptx

Das Workshop Repository

"npm install" und "npm start"

"npm install" und "npm start"

Schritt 2: Installieren der npm-Pakete

- Vorbereitung (gemeinsam): Starten aller Prozesse
 1. In "code-backend/userservice": "npm install"
 2. In "code-backend/userservice": "npm start"
 3. In "code-backend/workspace": "npm install"
 4. In "code-backend/userservice": "npm install"
 5. Playground sollte jetzt über <http://localhost:4000> erreichbar sein
 6. Wenn Playground läuft, bitte Hand heben in Teams 

Schritt 3: Die Übungen

- 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_...
 - Hier könnt ihr bei Problemen nachsehen (oder jederzeit Fragen natürlich)
- Der "UserService" kann die ganze Zeit durchlaufen

ÜBUNG 1: SCHEMA DEFINIEREN

Vervollständige das Schema der Beispiel-Anwendung

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
 3. Es muss eine Mutation beschrieben 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, sonst neuladen
 - (Auf "Docs" am rechten Rand klicken)
 - Hinweis: Ausführen der Queries funktioniert noch nicht

RESOLVER

RESOLVER

Resolver-Funktion

Ein Resolver liefert einen Wert für ein angefragtes Feld in einer Query

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

Resolver-Funktionen

👉 ping-resolver und Resolver-Map

BEISPIEL

1. ping-Feld mit String und Argument
2. ping-Feld mit Objekt als Rückgabe
3. ping-Feld mit uppercaseMessage

RESOLVER

Resolver-Funktionen

Schema Definition

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

Query

```
query { ping }
```



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

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

RESOLVER

Beispiel: Root-Resolver mit Argumenten

Schema Definition

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

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

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: "GraphQL") → "data": {  
    "ping": "Hello, GraphQL"  
  }  
}
```

```
const resolvers = {  
  Query: {  
    ping: (_, { msg }) => `Hello, ${msg}`  
  }  
}
```

Resolver mit Argumenten

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

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) {  
            return db.getTasks(project._taskId)  
        }  
    }  
}
```

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}) {  
            return db.updateTaskState(taskId, newState);  
        }  
    }  
}
```

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

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  
});
```

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`
- Mutation: `updateTaskState`
- Am Task: Felder `tasks` und `task`

Die Änderungen müssen in `query.js`, `mutation.js` und `project.js` vorgenommen werden

- dort sind entsprechende TODOs eingetragen
- nach den Änderungen in `query.js` bitte in `server.js` den Project-Resolver hinzufügen
- (Falls Du mit Übung 1 nicht fertig geworden bist, einfach die `schema.js`-Datei 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

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  
    }  
  }  
}
```

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
 - Gucken wir uns später an!

👉 Beispiel: auth-Context

token aus Header

auth-Funktion aufrufen

User als currentUser setzen

in Ping-Resolver verwenden, um User auszugeben

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
(server.js)

```
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
(server.js)

```
const context = (req) => (  
  { user: req.headers.user }  
) ;
```

Server-Konfiguration

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

SCHRITT 2: RESOLVER

Resolver: Der Context

- Der Context wird jedem Resolver als 3. Parameter übergeben

Context Definition
(server.js)

```
const context = (req) => (
  { user: req.headers.user }
);
```

Resolver mit Context

```
const resolvers = {
  Query: {
    ping(_, _args, { user }) {
      return `Hello, ${user}`
    }
  }
}
```

ÜBUNG 3: AUTHENTIFIZIERUNG UND CONTEXT

Schritt 1: Context erzeugen

- Implementiere in **server.js** die Funktion zum Erzeugen des Contexts
- Der Context soll den http-Header "token" auslesen und den Wert an unseren Fake-Authentification-Service übergeben, und den gefundenen User als "currentUser" in den Context setzen (wenn kein User gefunden wird, kein Context)
- **(siehe Todos in server.js)**
- Einen http-Header kannst Du zum Testen über den Playground setzen:

```
QUERY VARIABLES HTTP HEADERS (1)  
1 {  
2   "token": "U1"  
3 }
```



ÜBUNG 3: AUTHENTIFIZIERUNG UND CONTEXT

Schritt 2: Context verwenden

Die addTask-Mutation soll sicherstellen:

- nur ein "eingeloggter" Benutzer darf Tasks erstellen (context.currentUser !== null)
- der assignee des Tasks muss dem aktuellen Benutzer entsprechen
- **(siehe Todos in mutation.js)**

Fehlerbehandlung

- Unsere addTask-Mutation kann eine Reihe von Fehlern erzeugen
- Welche Möglichkeiten haben wir, damit umzugehen? 🤔

UNION TYPES

Union Types: Menge von möglichen Typen

👉 Beispiel: 30_error

UNION TYPES

Union Types: Menge von möglichen Typen

Schema

```
type AddTaskSuccess {  
    newTask: Task!  
}  
  
type AddTaskFailure {  
    errorCode: Int!  
    errorMessage: String!  
}  
  
union AddTaskResponse = AddTaskSuccess | AddTaskFailure  
  
mutation {  
    addTask(...): AddTaskResponse  
}
```

UNION TYPES

Union Types: Menge von möglichen Typen

Schema

```
union AddTaskResponse = AddTaskSuccess | AddTaskFailure
```

Mutation

```
mutation {
  addTask(...) {
    ...on AddTaskSuccess {
      newTask { id }
    }

    ...on AddTaskFailure {
      errorCode
    }
  }
}
```

RESOLVER FÜR UNION TYPES

Resolver

- Der addTask Resolver liefert nun zwei Typen zurück:

```
addTask(_, {projectId, input}, {currentUser}) {
  if (!currentUser) {
    return {
      errorCode: 666,
      errorMessage: "No User!"
    }
  }

  const task = await db.addTask(...);

  return { newTask: task };
}
```

AddTaskFailure

AddTaskSuccess

RESOLVER FÜR UNION TYPES

Resolver

- Der addTask Resolver liefert nun zwei Typen zurück
- Apollo muss wissen, welcher Typ es ist – dafür eigener Resolver!
- Das zu prüfende Objekt wird an `__resolveType` übergeben
- Die Funktion liefert den Typ-Namen zurück (oder null)
- Der Resolver wird wie gewohnt in ResolverMap eingetragen

TaskResponse-Resolver

```
const TaskResponse = {
  __resolveType(obj) {
    if ("errorCode" in obj) {
      return "AddTaskFailure";
    } else if ("newTask" in obj) {
      return "AddTaskSuccess";
    }
    return null;
  }
}
```

RESOLVER FÜR UNION TYPES

Resolver

- Der addTask Resolver liefert nun zwei Typen zurück
- Apollo muss wissen, welcher Typ es ist – dafür eigener Resolver!
- Das zu prüfende Objekt wird an `__resolveType` übergeben
- Die Funktion liefert den Typ-Namen zurück (oder null)
- Der Resolver wird wie gewohnt in ResolverMap eingetragen

```
const resolvers = {
  // wie bekannt
  AddTaskResponse: require("./add-task-response.js")
}
```

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

Server-Konfiguration

ÜBUNG 4: EIN UNION TYPE

Die addTask-Mutation soll einen Fehler zurückgeben, anstatt Errors zu werfen

Schritt 1: Definiere den union-Type (schema.js)

- Füge AddTaskSuccess, AddTaskFailure und AddTaskResponse hinzu
 - AddTaskResponse soll ein Pflicht-Feld haben (newTask)
 - AddTaskFailure soll errorCode (Int) und errorMessage (String) haben

Schritt 2: Modifiziere den addTask-Resolver (mutation.js)

- Liefer die neuen Typen zurück (nicht verkünsteln, was die Fehlermeldungen angeht 😊)

Schritt 3: Implementiere den AddTaskResponse Resolver

- Siehe TODO in resolvers/add-task-response.js

ZUGRIFF AUF DATEN-QUELLEN

• Datenquellen sind verschiedene Formen von Speichermedien, die für die Verarbeitung von Daten vorgesehen sind.

• Es gibt verschiedene Arten von Datenquellen:

- Dateien

- Datenbanken

- Webdienste

- Echtzeitdaten

- Big Data

- Machine Learning

- Deep Learning

- Natural Language Processing

- Computer Vision

- Robotics

- Internet of Things

- Blockchains

- Edge Computing

- Cloud Computing

- Edge Computing

Datenquellen

- Wir können beliebige Datenquellen (DB, REST, ...) verwenden
 - Weder GraphQL noch Apollo machen da eine Aussage zu
 - Ihr könnt also auch Eure Lieblingslibraries verwenden
-
- Aber es gibt ein paar Klippen!

ZUGRIFF AUF DATEN-QUELLEN

Datenzugriff ist potentiell teuer!

- Wir können Felder im Schema mit cacheControl markieren
- Damit können wir steuern, wie lange ein Feld "gültig" ist, der entsprechende Resolver wird dann nicht aufgerufen
- Wenn es mehrere Felder in einem Query gibt, die mit cacheControl markiert sind, entscheidet die kürzeste Cache-Dauer

👉 Beispiel: Cache-Control am ping-Feld

Tracing

- Apollo kann die Ausführungszeit Eurer Resolver messen
- Daten stehen im Playground zur Verfügung
- Tracing muss eingeschaltet werden

Server-Konfiguration

```
const server = new ApolloServer({  
    // ...  
    tracing: true  
});
```

ZUGRIFF AUF DATEN-QUELLEN

Laufzeitverhalten

Query

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

Frage: Was passiert beim Ausführen dieses Queries? 🤔

ZUGRIFF AUF DATEN-QUELLEN

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? 🤔

ZUGRIFF AUF DATEN-QUELLEN

Laufzeitverhalten

EIN Datenbankzugriff
(liefert n Projekte)

n REST-Aufrufe
(1x je Project)

```
const resolvers = {
  Query: {
    projects() => {
      return db.getAllProjects();
    }
  },
  Project: {
    owner(project) => {
      return userService.getUser(project._ownerId);
    }
  }
}

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

$1+n$ -Problem 😱

DataSources

DataSources können für den Zugriff auf externe Systeme verwendet werden

- Können Daten cachen!
- Zum Beispiel REST-Service, Datenbank etc
- Fertige Implementierung für REST-Services
- Community-Implementierungen u.a. für SQL

DATASOURCES FÜR DATENBANKEN

DataSources: Zugriff auf Datenbank

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 von Apollo: Zugriff auf REST deutlich öfter als auf Datenbank
 - Weil per GraphQL Microservices "aggregiert" werden

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 von Apollo: Zugriff auf REST deutlich öfter als auf Datenbank
 - Weil per GraphQL Microservices "aggregiert" werden
- Alternative zu DataSource für DBs: prisma (<https://www.prisma.io/graphql>)

DIE REST DATASOURCE

Die REST Data Source <https://www.apollographql.com/docs/apollo-server/features/data-sources/>

- Fester Bestandteil von Apollo (apollo-datasource-rest)

👉 Demo: Data Source (40_...)

👉 Demo Cache-Header (in userservice anpassen)

DIE REST DATASOURCE

Die REST Data Source <https://www.apollographql.com/docs/apollo-server/features/data-sources/>

- Fester Bestandteil von Apollo (apollo-datasource-rest)

👉 Demo: Data Source (40_...)

👉 Demo Cache-Header (in userservice anpassen)

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();  
  
  }  
}
```

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)

```
}
```

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)

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

DIE REST DATASOURCE

Bereistellen von DataSources

- DataSourcen werden pro Query-Ausführung erzeugt
- Über dataSources-Konfiguration am Server bekannt gemacht
- Stehen dann automatisch über Kontext zur Verfügung

Context Definition

```
const dataSources = (req) => (
  { projectDataSource: new ProjectDataSource(),
    userDataSource: new UserDataSource(),
  }
);
```

DataSources

DATASOURCES

Bereitstellen von DataSources

- DataSourcen werden pro Query-Ausführung erzeugt
- Über dataSources-Konfiguration am Server bekannt gemacht
- Stehen dann automatisch über Kontext zur Verfügung

DataSources

```
const dataSources = (req) => (
  { projectDataSource: new ProjectDataSource(),
    userDataSource: new UserDataSource(),
  }
);
```

Server-Konfiguration

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

DATASOURCES

Zugriff auf DataSources

- Die DataSources stehen unter dem Key "dataSources" in den Resolvern über Kontext zur Verfügung

Resolver

```
const Query = {
  projects(_, __, { dataSources }) {
    return dataSources.projectDataSource.getProjects()
  }
}
```

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);
    }
  }
}
```

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

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)

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 4: DATA SOURCE VERWENDEN

Die REST DataSource verwenden, die Implementierung ist fertig

- Das Modul befindet sich in datasources/UserRESTDataSource.js, die Klasse heißt ebenfalls UserRESTDataSource
- Passe die Konfiguration in server.js an und füge eine Instanz der UserRESTDataSource bei jedem Request hinzu
- Verwende die DataSource in query.js, project.js und task.js
- Führe einen Query aus, der alle Projekte, mit ihren Owners und allen ihren Tasks samt ihren Assginees liest
 - Wie viele Aufrufe werden durchgeführt?
 - Siehe Konsole vom userservice und Apollo Server



Vielen Dank!

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

Fragen und Kontakt: nils@nilshartmann.net

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