NILS HARTMANN
https://nilshartmann.net

GraphQL APIs

A practical introduction with Java

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

Slides (PDF): https://react.schule/api-conf-graphql

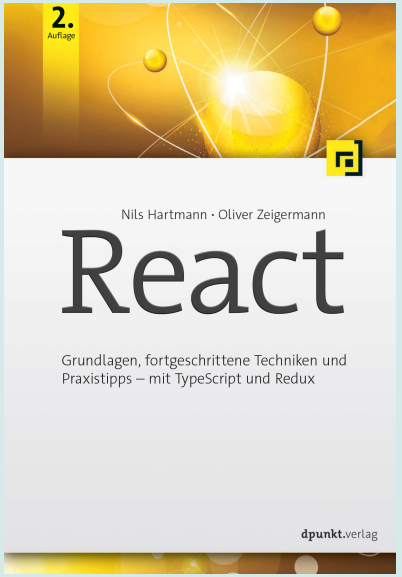
NILS HARTMANN

nils@nilshartmann.net

Freelance developer, architect, trainer from Hamburg

Java
JavaScript, TypeScript
React
GraphQL

Trainings, Workshops,
Coaching



https://reactbuch.de

HTTPS://NILSHARTMANN.NET

...and who are you?

Please introduce yourself:

what is your background?

do you have already graphql experience?

expectations, questions for today?

...

AGENDA

1. **Why GraphQL? Basics and motivation**

2. **GraphQL for Java Applications**
   - Implementing your own API
   - Optimizations
   - Outlook: different frameworks for Java

3. **Q&A, Discussions**

At any time: *your* questions and discussions!

Please don't hesitate to participate via chat or audio!

TODAY...

9.00 – 10.30 Workshop Part I

10.30 – 11.00 Coffee Break

11.00 – 12.30 Workshop Part II

12.30 – 13.30 Lunch

13.30 – 15.00 Workshop Part III

15.00 – 15.30 Coffee Break

15.30 – 17.00 Workshop Part IV

PART 1

GraphQL Basics

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

- https://graphql.org

GraphQL

Specifikation: https://facebook.github.io/graphql/

- Published in 2015 by Facebook
- Since 2018, the GraphQL specification is developed by the GraphQL Foundation

- Spec includes:
  - Query language
  - Schema Definition Language
  - No implementation!
    - Server reference implementation: graphql-js

GraphQL != SQL

- no SQL, no "complete" query language
  - for example: no built-in sorting, no built-in Joins
- no database!
- no framework!

- *GraphQL does not replace your database or backend!*

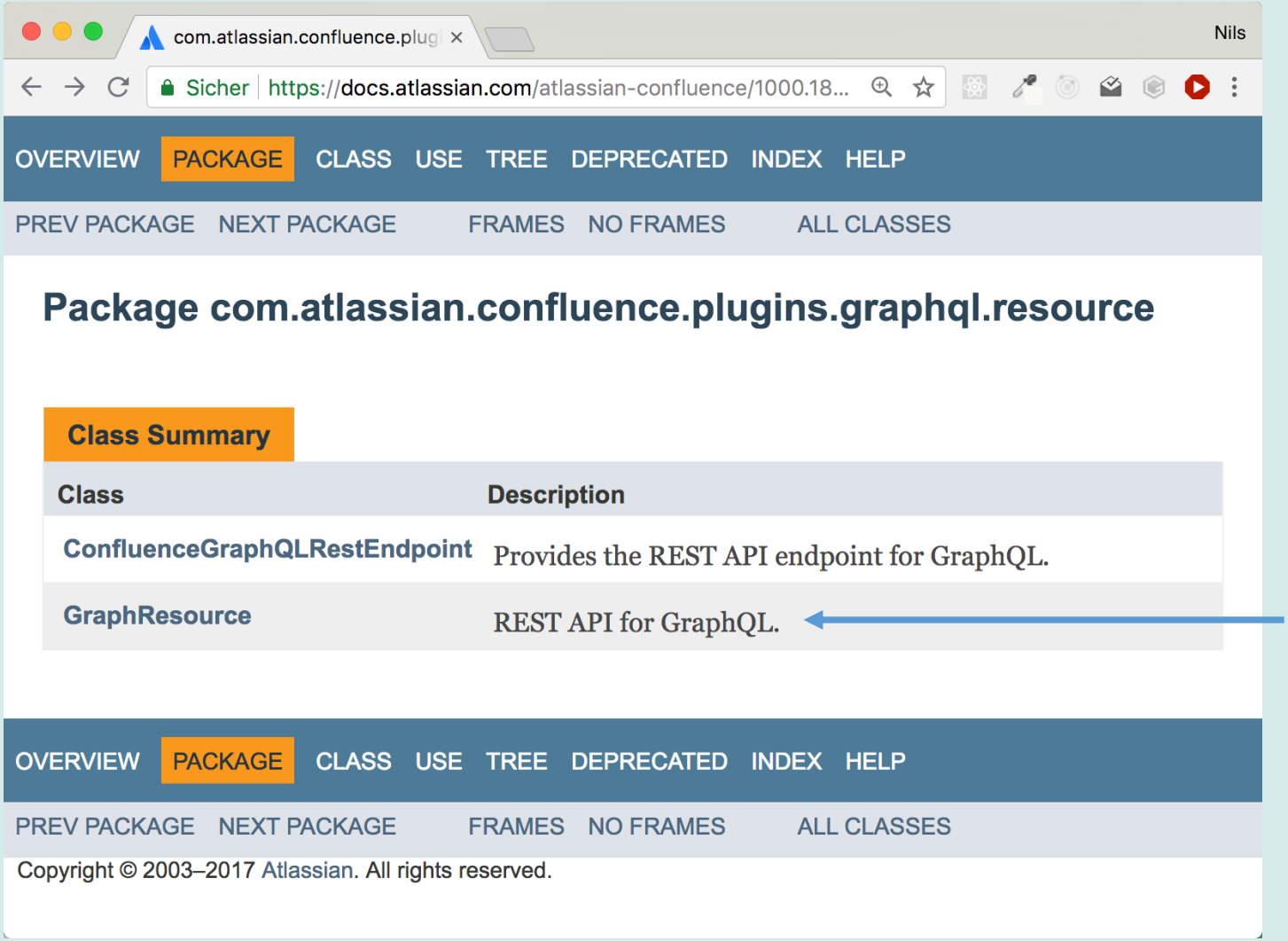
GraphQL != Mainstream

- still "bleeding edge", no mainstreanm
- many experiments, ideas but not that much experience and proven best-practices

- ...but it's used by some more or less big players...

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

GitHub GraphQL API Explorer: https://docs.github.com/en/graphql/overview/explorer



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

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

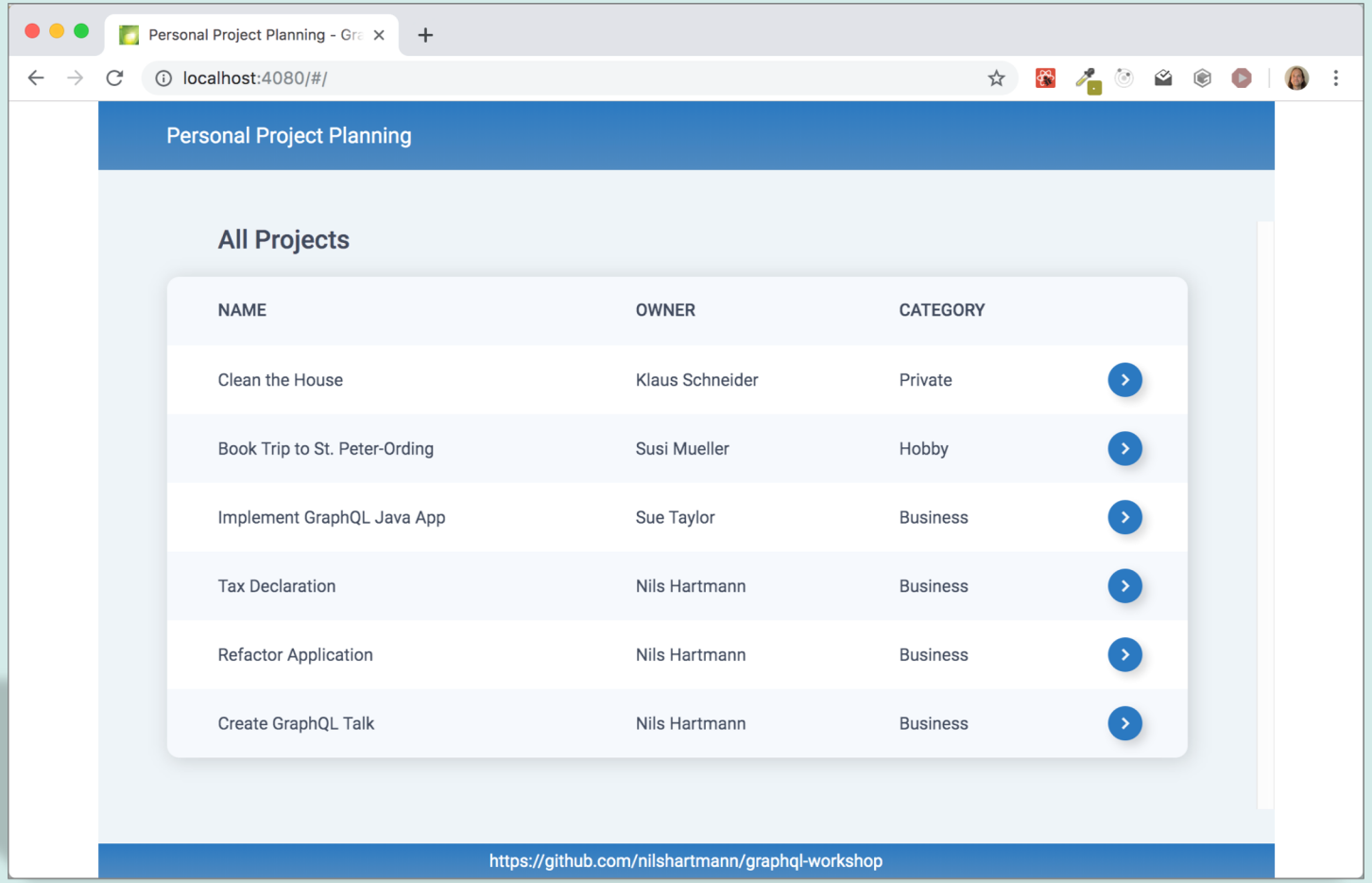
React, Relay and GraphQL: Under the Hood of the Times Website Redesign

A look under the hood.

The New York Times website is changing, and the technology we use to run it is changing too.

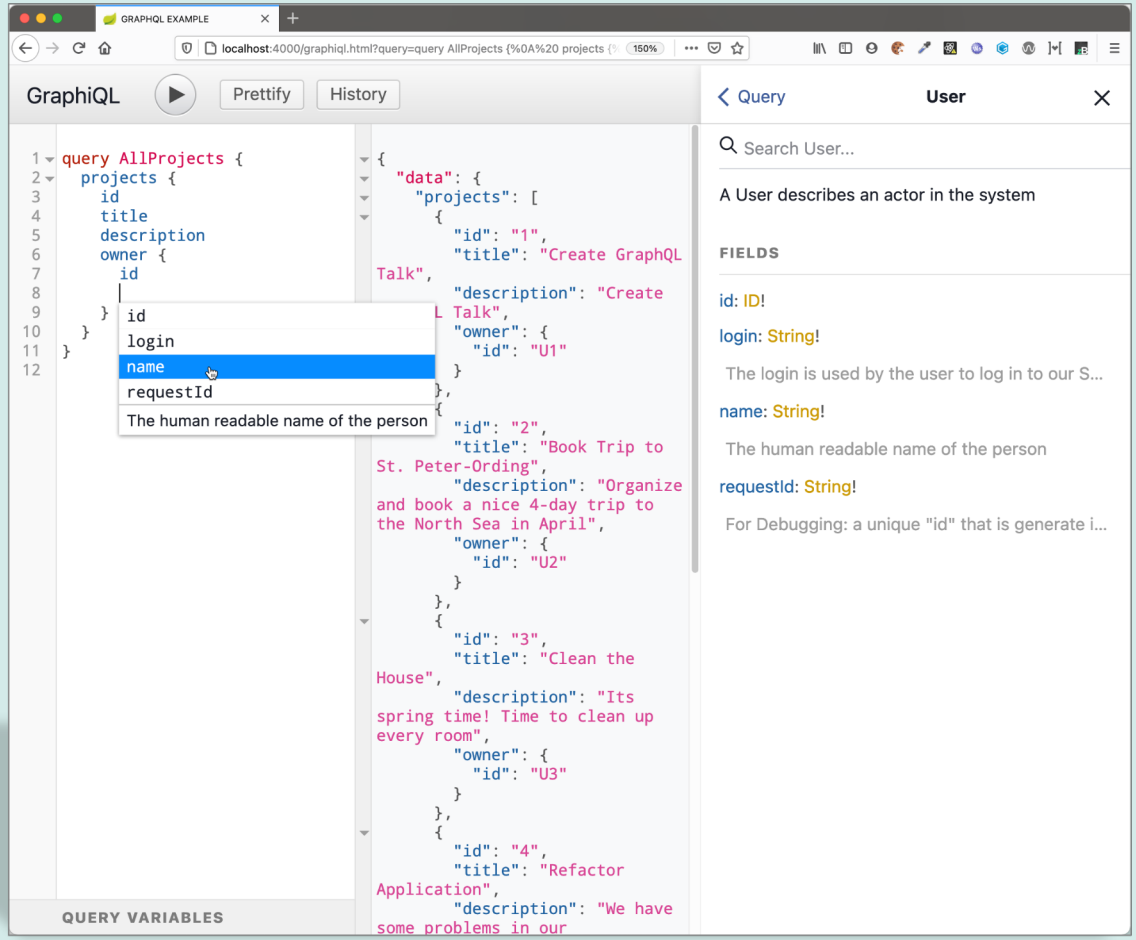
https://open.nytimes.com/react-relay-and-graphql-under-the-hood-of-the-times-website-redesign-22fb62ea9764

NEW YORK TIMES



Example Application

http://localhost:5080

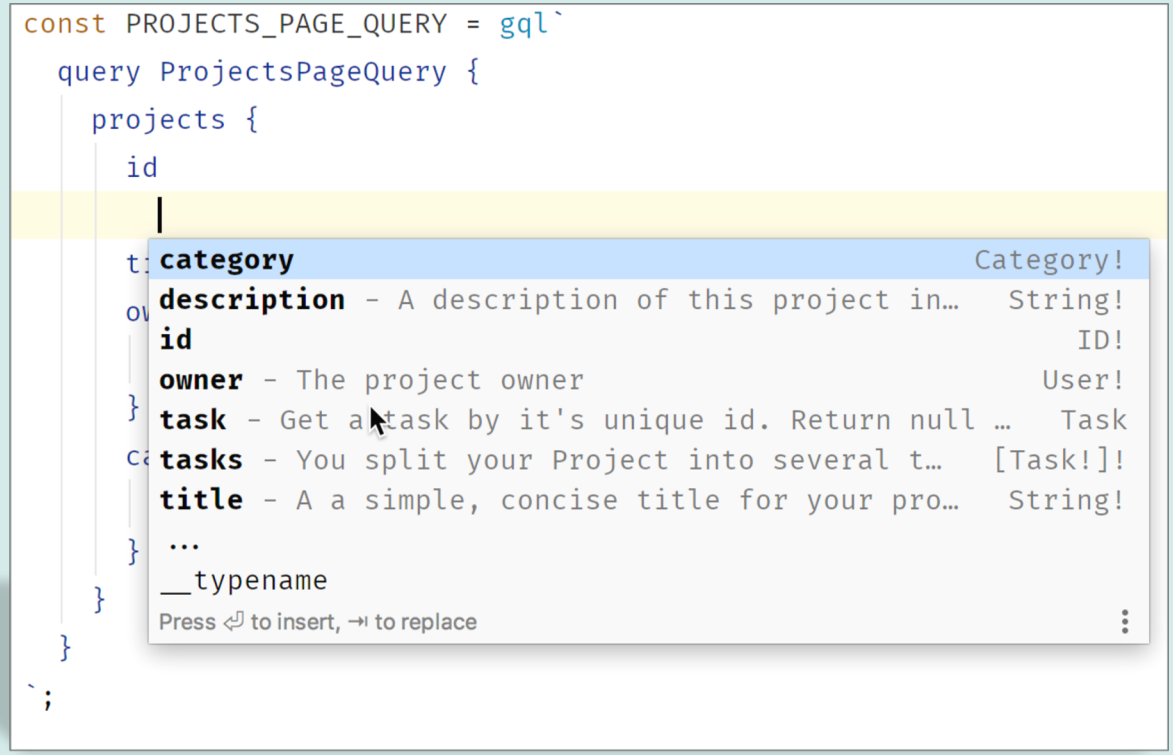


Demo: GraphiQL

https://github.com/graphql/graphiql

http://localhost:4000

RAUS ODER HINTEN



ProjectsPage.tsx

Demo: IDE Support

Beispiel: Intellij IDEA

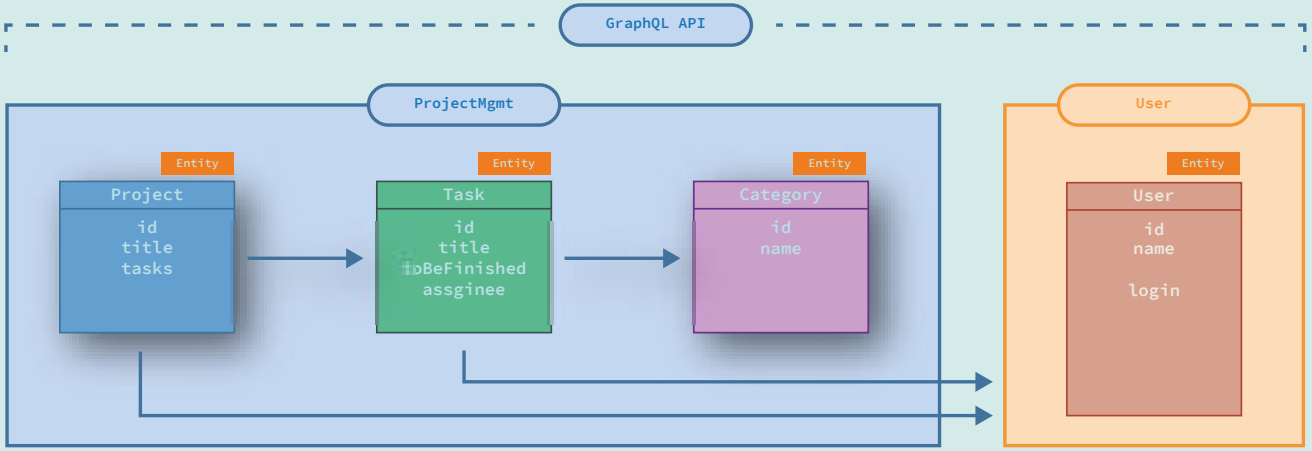
*"GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data"*

- https://graphql.org

GraphQL

TEIL 1: ABFRAGEN UND SCHEMA

"Architecture" of our example application

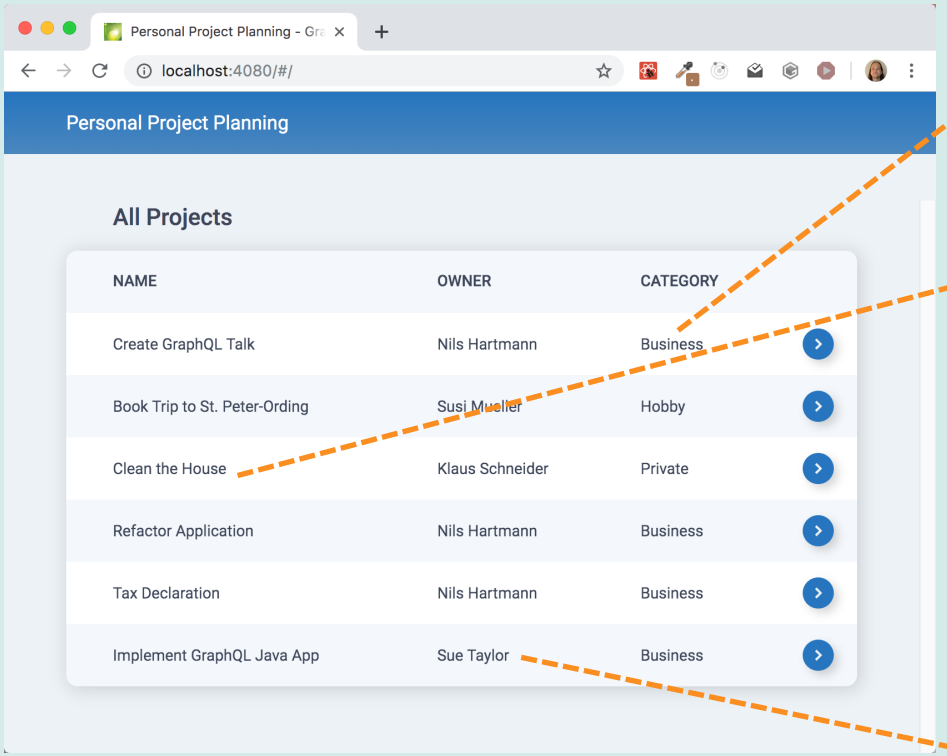


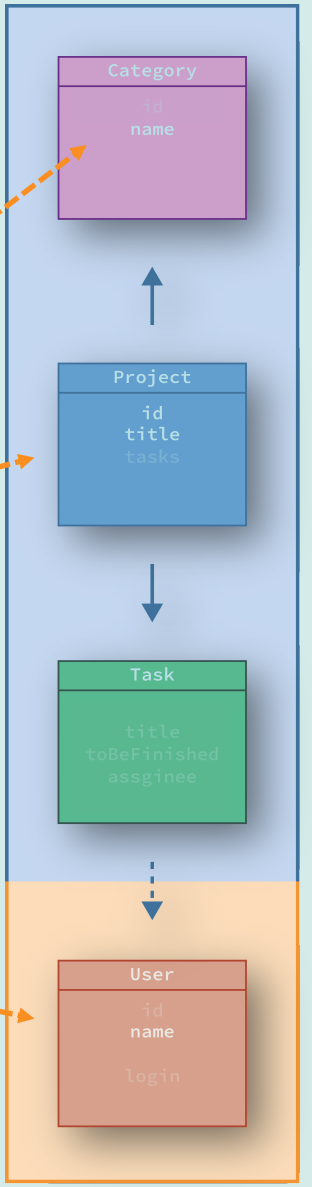
Project Management Application
http://localhost:5000
GraphQL API with graphql-java
Spring Boot, JPA

User (Micro-)Service
http://localhost:5010/users/{id}
REST API
Spring Boot

Use-case specific queries

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

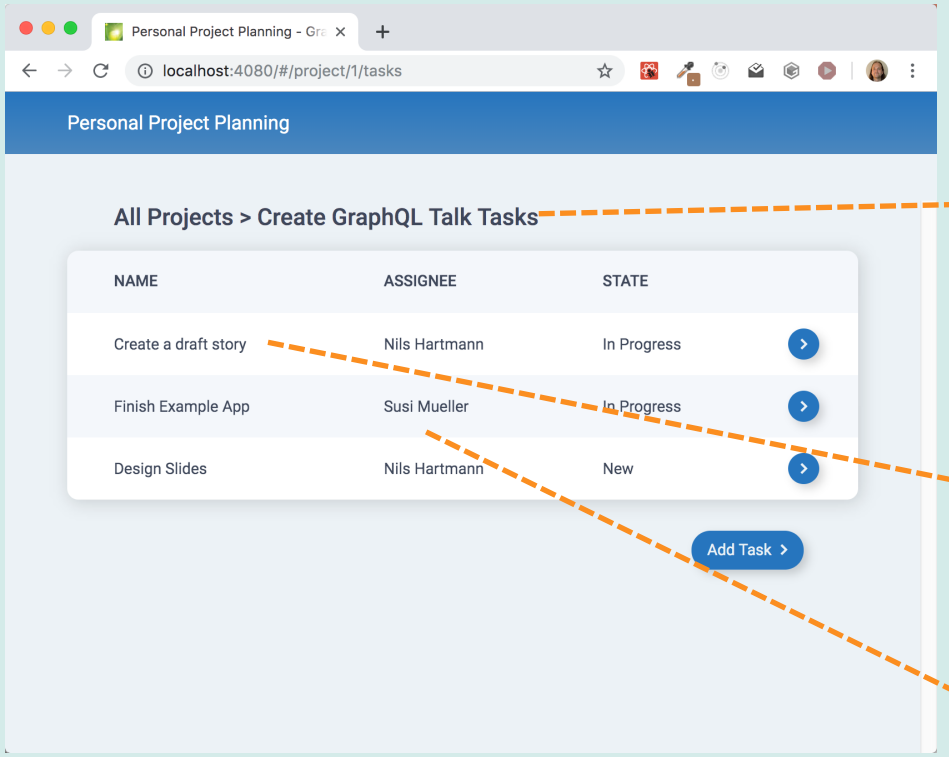


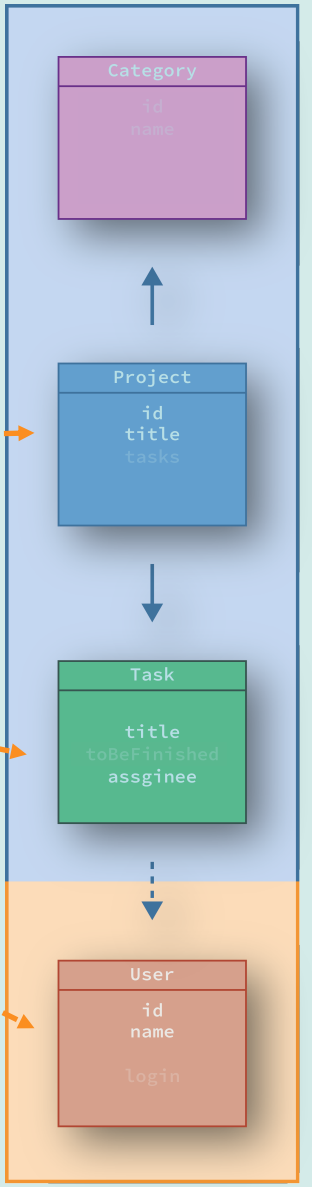


GRAPHQL EINSATZSZENARIEN

Use-case specific queries 2

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

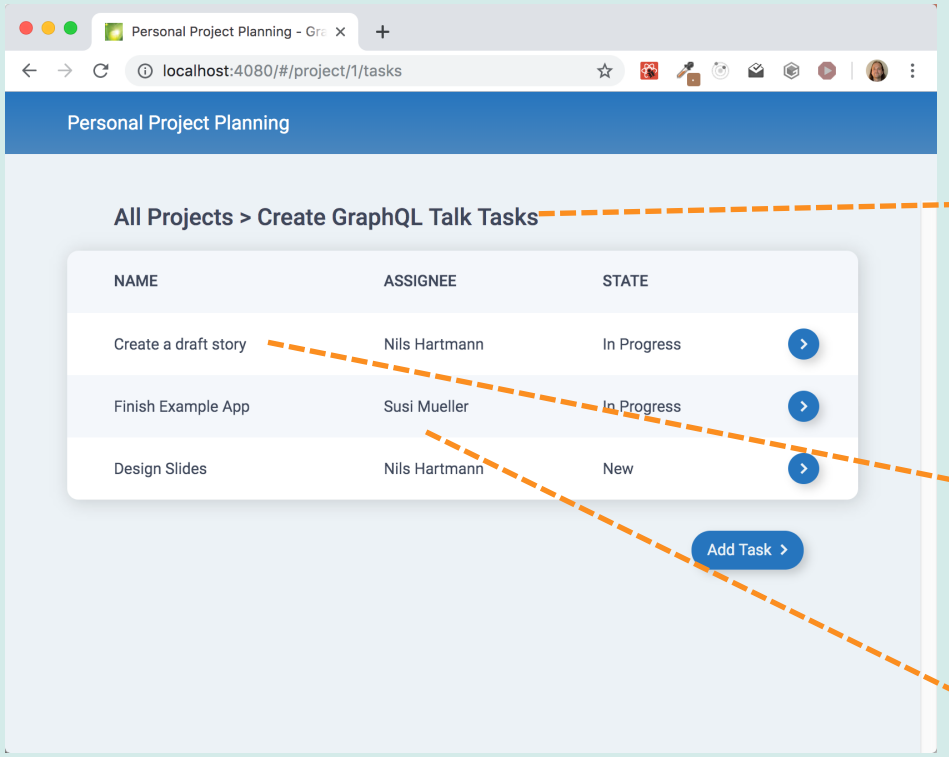


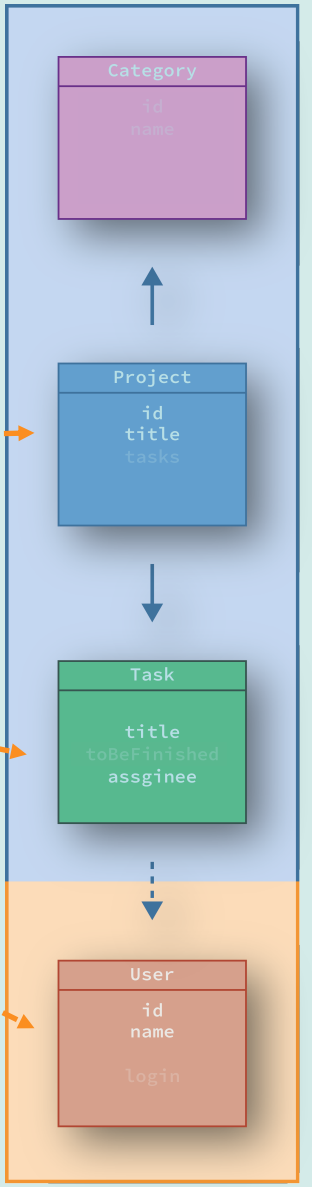


GRAPHQL EINSATZSZENARIEN

Use-case specific queries 2

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





Data is queried and requested, not endpoints 😈

USE CASES

Possible reasons for GraphQL

- Different use-cases with different data
  - Multiple views in the frontend
  - Multiple different clients (public API?)
- API can be enhanced independent of the server (more or less)
  - Client only consumes explizitly queried fields
- Overall view of your domain
- Type-safe API wanted (for example for code generation)

GRAPHQL APIS

GraphQL makes no statement about how our API looks like

👉 *What data we provide is our decision*

👉 ***We decide how the API of our application looks like***

GraphQL is "only" technology

GRAPHQL APIS

GraphQL makes no statement about how our API looks like

👉 *What data we provide is our decision*

👉 ***We decide how the API of our application looks like***

GraphQL is "only" technology

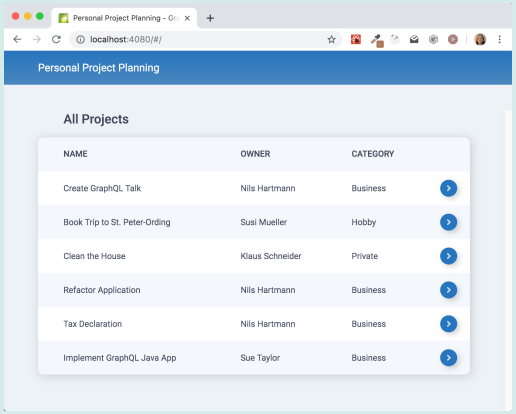
⛔ ~~***"for people who can't make up their minds"***~~

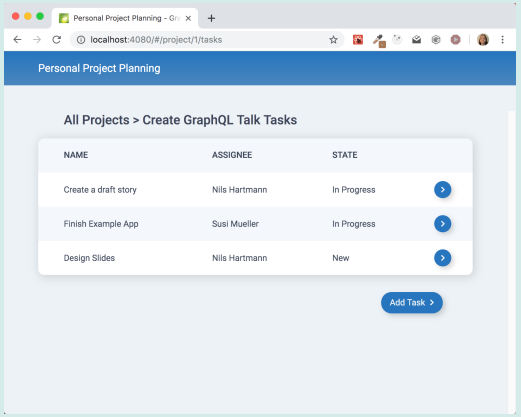
GRAPHQL USE-CASE

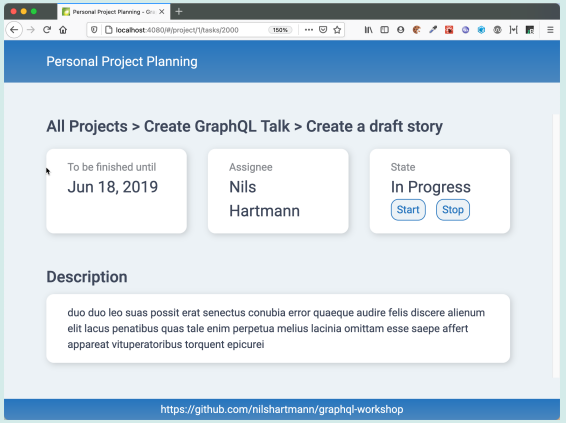
What about REST/HTTP APIs?

🤔 How would a REST/HTTP API look like?

🤔 What would be the pros/cons?

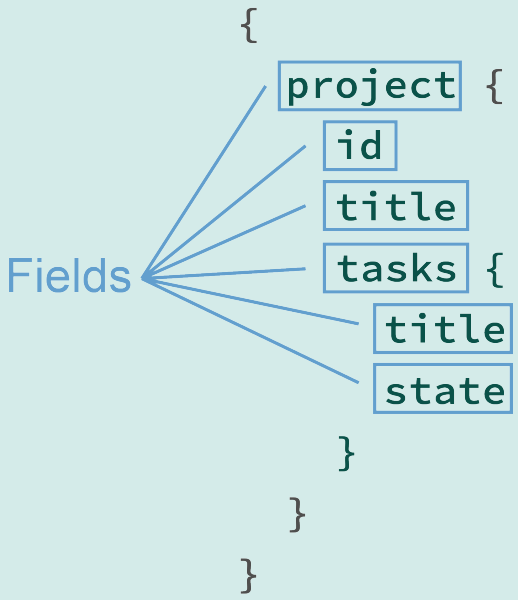






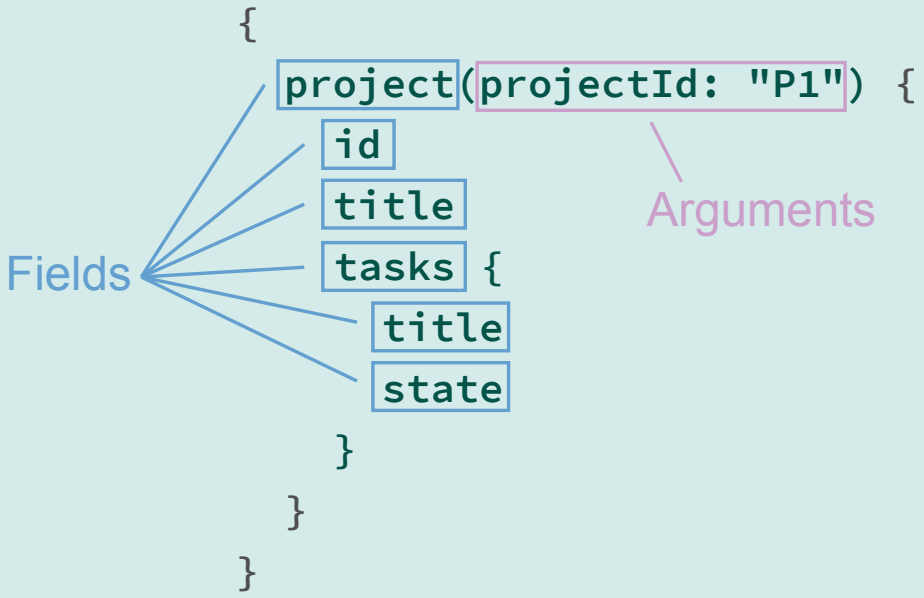
The GraphQL Query Language

QUERY LANGUAGE



- Structured Language to query/request data from your API
- With the language, you select **fields** from object graphs

QUERY LANGUAGE



- Structured Language to query/request data from your API
- With the language, you select **fields** from object graphs
- Fields can have **arguments**

Query Result

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

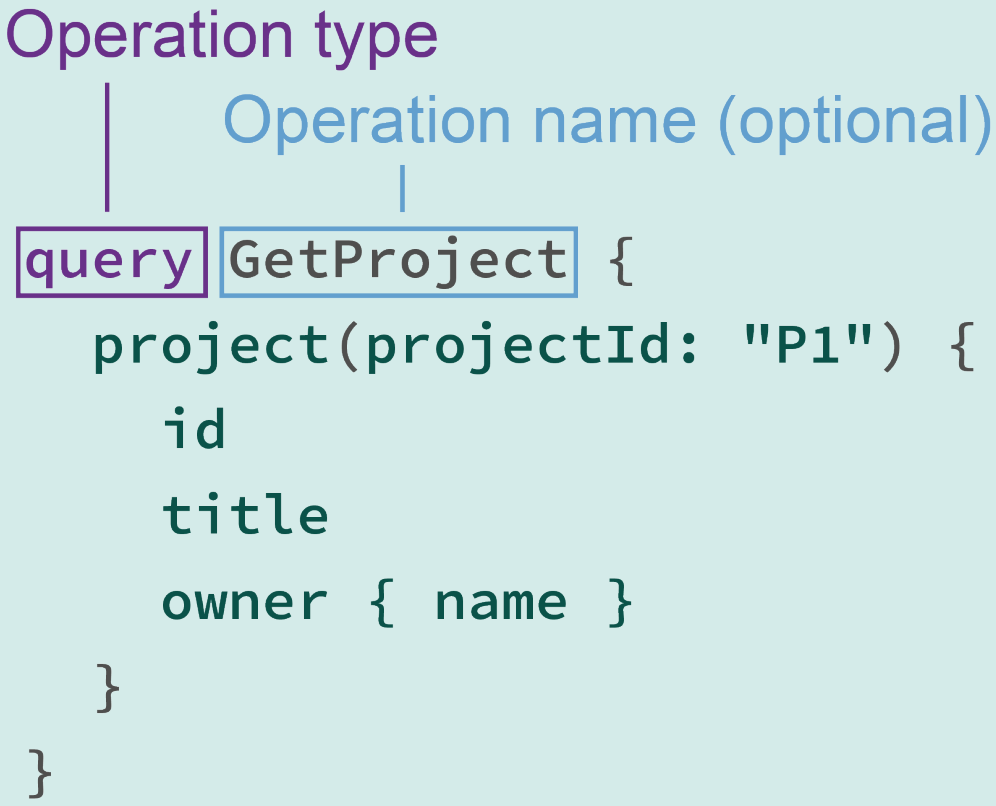
```
"data": {
  "project": {
    "id": "P1"
    "title": "GraphQL Talk"
    "tasks": [
      {
        "state": "IN_PROGRESS",
        "title": "Create Story"
      },
      {
        "state": "NEW",
        "title": "Finish Example"
      }
    ]
  }
}
```

- Identical structure as your query

QUERY LANGUAGE: OPERATIONS

Operation: describe, what the query should do

- query, mutation, subscription



QUERY LANGUAGE: FRAGMENTS

Fragments

- Using fragments you can define reusable subsets of selections
- 👉 GraphiQL!

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

QUERY LANGUAGE: FRAGMENTS

Fragments

- Using fragments you can define reusable subsets of selections

```
fragment UserWithIdAndName on User {
  id
  name
}

query {
  projects {

User! ---------------------------- owner {
        ...UserWithIdAndName
      }

  }
}
```

QUERY LANGUAGE: FRAGMENTS

Fragments

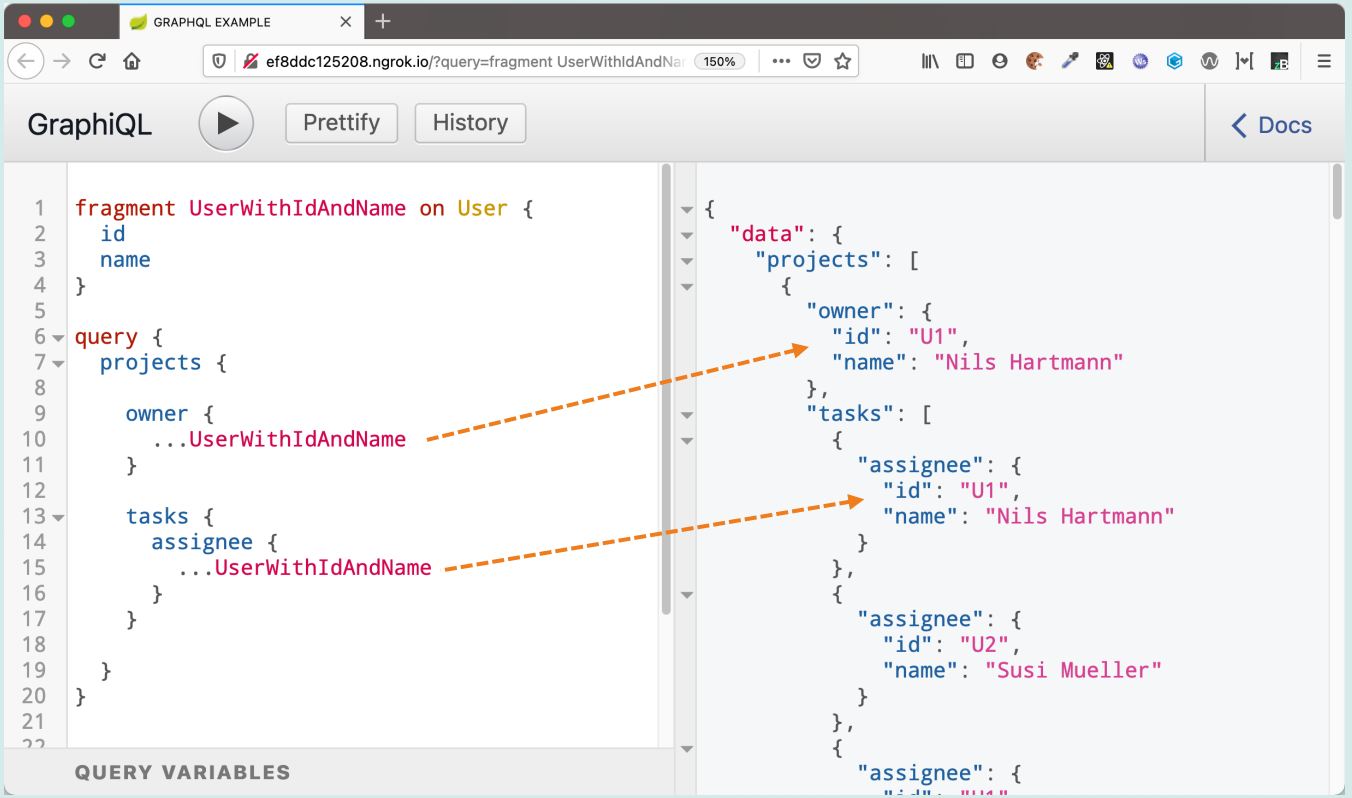
- Using fragments you can define reusable subsets of selections

```
                    fragment UserWithIdAndName on User {
                      id
                      name
                    }

                    query {
                      projects {

User! ------------------------- owner {
                                  ...UserWithIdAndName
                                }

                                tasks {
User! -------------------------   assignee {
                                    ...UserWithIdAndName
                                  }
                                }

                      }
                    }
```

QUERY LANGUAGE: FRAGMENTS

Fragments

- Using fragments you can define reusable subsets of selections



QUERY LANGUAGE: OPERATIONS

Operation: Variables

- Variables can be used to insert placeholders, as in prepared statements
- They're send in a separate object to the server

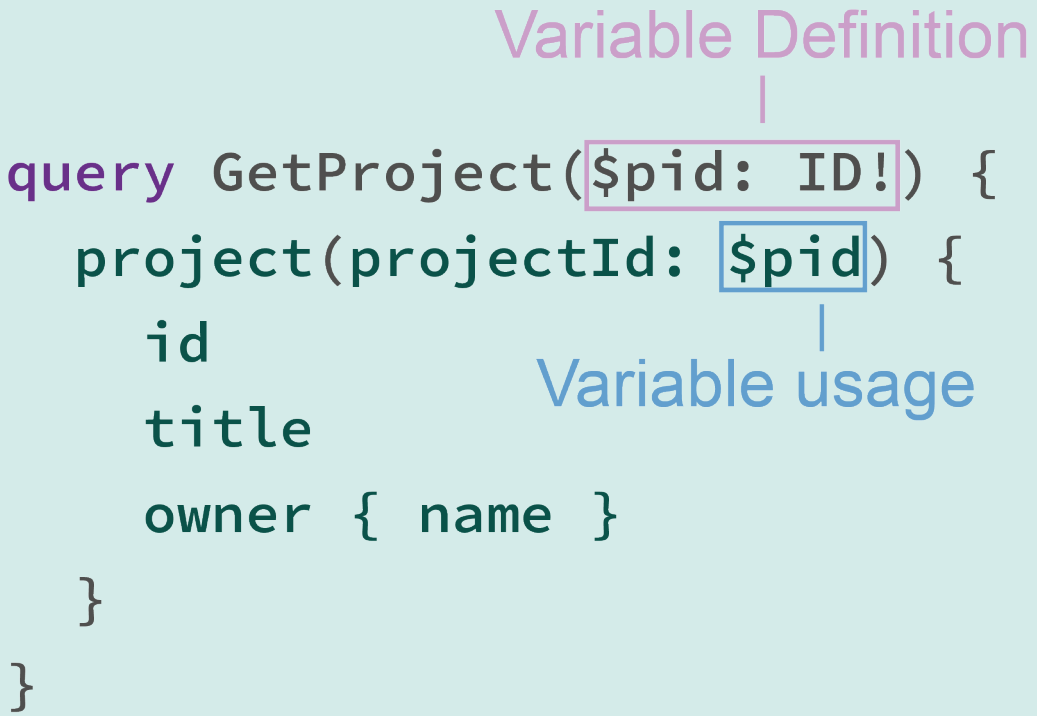
👉 GraphiQL!

QUERY LANGUAGE: OPERATIONS

Operation: Variables

- Variables can be used to insert placeholders, as in prepared statements
- They're send in a separate object to the server

👉 GraphiQL!



QUERY LANGUAGE: MUTATIONS

Mutations

- Mutations can be used to modifiy data
- (would be POST, PUT, PATCH, DELETE in REST)

Operation type

Operation name (optional)

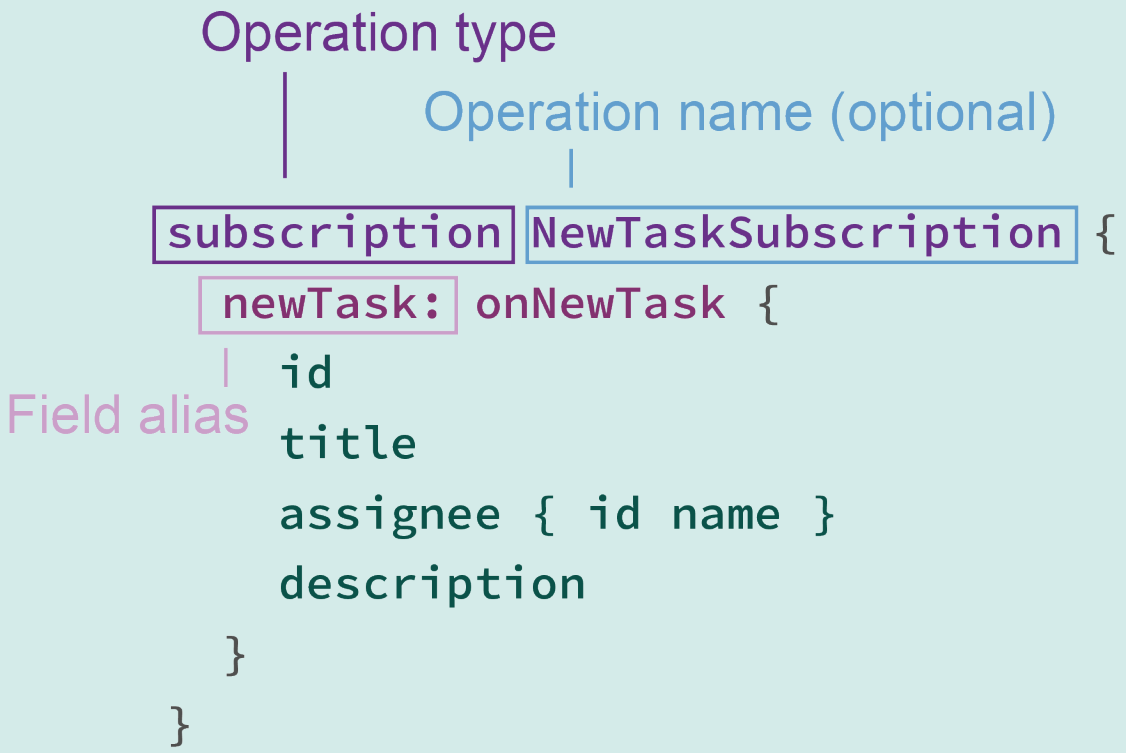
Variable Definition

```
mutation AddTaskMutation($pid: ID!, $input: AddTaskInput!) {
  addTask(projectId: $pid, input: $input) {
    id
    title
    state
  }
}

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

QUERY LANGUAGE: MUTATIONS

Subscription

- Client of your API can subscribe to Server Events, published by the API



EXERCISE: RUNNING QUERIES

Familiarise yourself with GraphiQL and the query language

- *Open GraphiQL on my machine, **I'll copy the URL to the chat***

Explore the API of the project management application

1. Execute a Query, that reads all *projects* and their *owners* (esp. their IDs)

2. Pick one of the returned projects and execute a Mutation, that adds a new *task* to the project
   Can you pass the data for the new task as a Variable to your query?

Please raise your hand in zoom if you're ready. 🙋

EXECUTING QUERIES

Queries can be executed by HTTP requests

- Normaly using HTTP POST (differes enormous from REST...)
- Only one endpoint for all queries

```
$ 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"}
   ]}
}
```

Response from Server

- (JSON-)Map with three defined fields on root-level
- data: your requested data, as seen before

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




}
```

Response from Server

- errors: a list of errors for fields that could not be requested
- still HTTP 200 !

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


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



}
```

Response from Server

- extensions: proprietary data from your GraphQL framework
- Example: logging/debugging informations

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

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

  {"extensions": { . . . }
}
```

PART 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

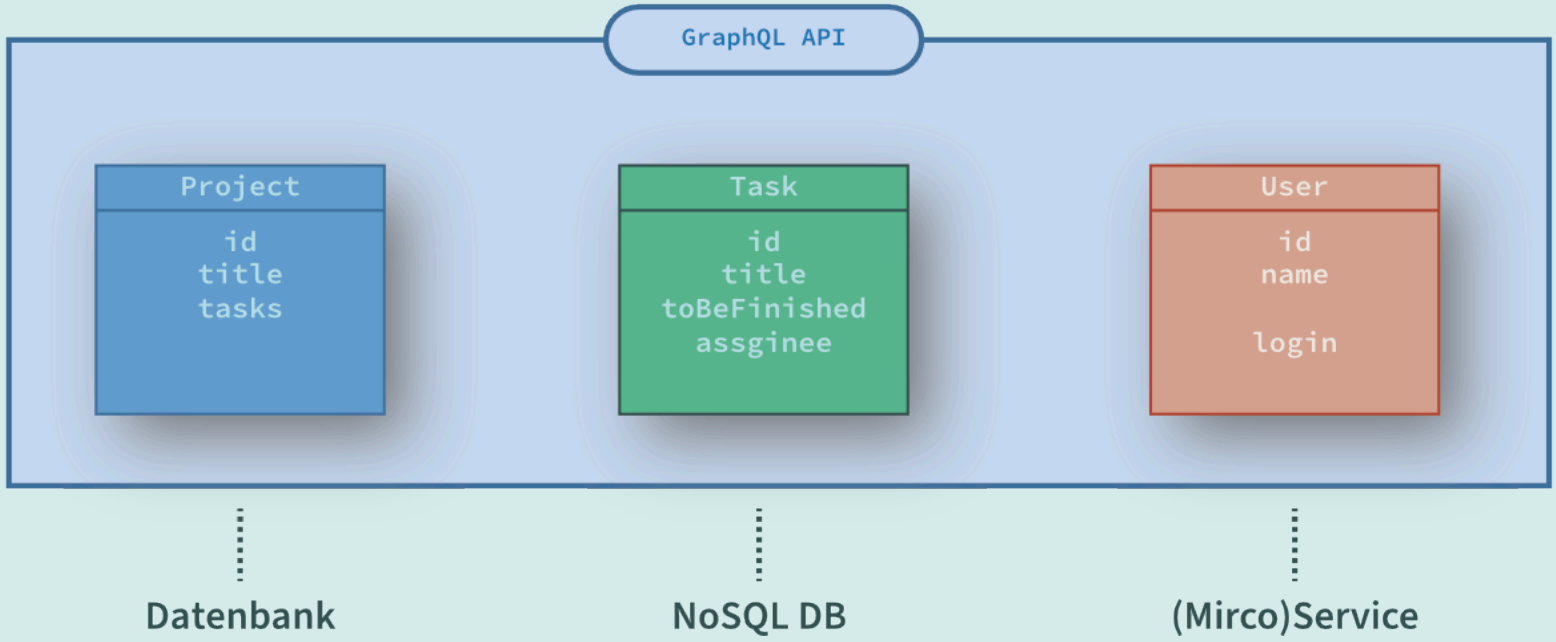
RUNTIME (AKA: YOUR APPLICATION)

GRAPHQL RUNTIME

GraphQL doesn't say anthing about the source of data

👉 *Determining the requested data is our task*

👉 *Data might not only come from (one) database*



GRAPHQL FOR JAVA APPLICATIONS

Option 1: graphql-java

- https://www.graphql-java.com/
- Pure GraphQL implementation, environment independent (no relations to Spring or JEE)
- API is quite low level and sometimes feels "uncommon"

- As this is the base for some other frameworks, we will use this in the following part of this workshop

Option 2: graphql-java-tools

- https://github.com/graphql-java-kickstart/graphql-java-tools
- Again, only GraphQL, not Spring/JEE requirements
- Implement your API using POJOs (might be comparable to Controller classes in Spring/JEE)
- Modular approach, several further projects exists, for example GraphQL Servlets or autoconfiguration for Spring Boot
- In our Workshop, the example app uses the GraphQL Servlet project (https://www.graphql-java-kickstart.com/servlet/)

Option 3: MicroProfile GraphQL

- https://github.com/eclipse/microprofile-graphql
- First version pulished in 2020
- No support for Subscriptions yet
- Schema will be defined by annotations in your java code
- Supported by Wildfly, Quarkus und Open Liberty and others

Optiona 4: Nextflix Domain Graph Service Framework (DGS)

- https://netflix.github.io/dgs/
- Published first in February 2021 (!)
- Based on Spring Boot and graphql-java
- Annotation-based programming model
- Code-Generator for Gradle and Maven creates from your schema definition Java classes
- Supports Subscriptions
- Support for Apollo Federation
  (Popular?) Gateway to combine multiple GraphQL APIs

graphql-java: https://www.graphql-java.com/

- "only" GraphQL framework, no server, no DI, ...
- no relation to JavaEE or Spring
- there are various example and other projects for integrating graphql-java with JavaEE / Spring

Our development tasks

1. Define the Schema of our API (graphql-java)
2. Implement *DataFetchers* that collect the requested data (graphql-java)
3. Expose GraphQL API to an HTTP endpoint (other framework)

GRAPHQL SCHEMA

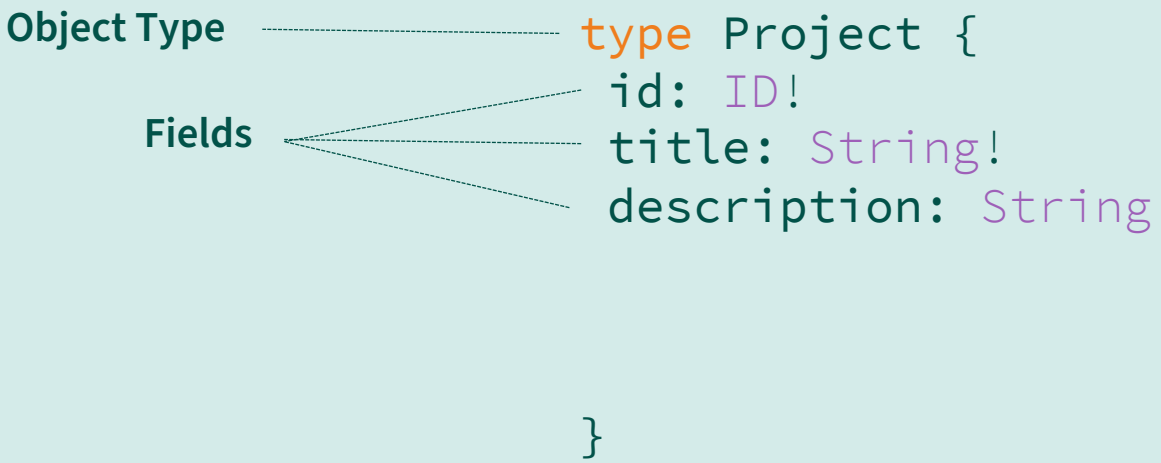
Schema

- A GraphQL API *have to* be defined with a Schema
- Schema describes *Types* and *Fields* of your API
- Only Queries (and Responses) that match your schema are executed (and returned)

- **Schema Definition Language** (SDL)

GRAPHQL SCHEMA

Schema Definition with SDL https://graphql.org/learn/schema/



Schema Definition with SDL

```
type Project {
 id: ID!
 title: String!
 description: String


}
```

Return Type (non-nullable)

Return Type (nullable)

Scalar types:

- **Int**
- **Float**
- **String**
- **Boolean**
- **ID** (Serialized to string, but it's value never get's interpreted somehow)

Schema Definition with SDL

```
type Project {
 id: ID!
 title: String!
 description: String
 owner: User!
}

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

Reference to other type

Schema Definition with SDL

```
type Project {
 id: ID!
 title: String!
 description: String
 owner: User!
 tasks: [Task!]!

}

type User {
 id: ID!
 name: String!
}

type Task {
 id: ID!
}
```

Return Type
List / Array

Schema Definition with SDL

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

type User {
 id: ID!
 name: String!
}

type Task {
 id: ID!
}
```

Arguments and their types

Schema Definition with SDL

```
enum TaskState {
 NEW
 RUNNING
 FINISHED
}
```

Enumeration (similiar to Java)

Schema Definition with SDL

```
enum TaskState {
 NEW
 RUNNING
 FINISHED
}
```

Enumeration

```
input AddTaskInput {
 title: String!
 description: String!
 toBeFinished: String!
 assigneeId: ID!
}
```

Input-Type
(für complex Field arguments)

Root-Types: Entry-Points into the API (Query, Mutation, Subscription)

Root-Type ("Query")

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

Root-Fields

Root-Types: Entry-Points into the 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-Types: Entry-Points into the 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!
}
```

Define Schema using the Schema Definition Language

- Documentation (like Javadoc) can be added with #
- Or blockwise using """, that also allows Markdown for formatting

```
"""
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
}
```

Schema: Instrospection

- Root-Fields "__schema" und "__type" (Example)
- 👉 GraphiQL

```
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.\n\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 EVOLUTION

Only one version! Fields always are queried explicitly

- You can add new fields add anytime without harm

New field

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

}
```

SCHEMA EVOLUTION

Only one version! Fields always are queried explicitly

- You can add new fields add anytime without harm
- You can "deprecate" old fields, that should not be used anylonger

- Usage of fields by clients can be tracked and analyzed

New field

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

directive

Define Schema using SDL

- Create one or more .graphqls-files with your schema definition

```
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!
}
```

Modular schemes

- A scheme can be split across several files
- Types can be *extended* in other files

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

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

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

EXCERCISE - PREPARATION

Preparation

- Before we start with the excercise, we setup the workspace together step-by-step

EXCERCISE - PREPARATION

Setup your Workspace

1. Make sure, you're sources are up-to-date:
   - If not done already, please clone the workspace:

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

   - otherwise please pull latest changes:

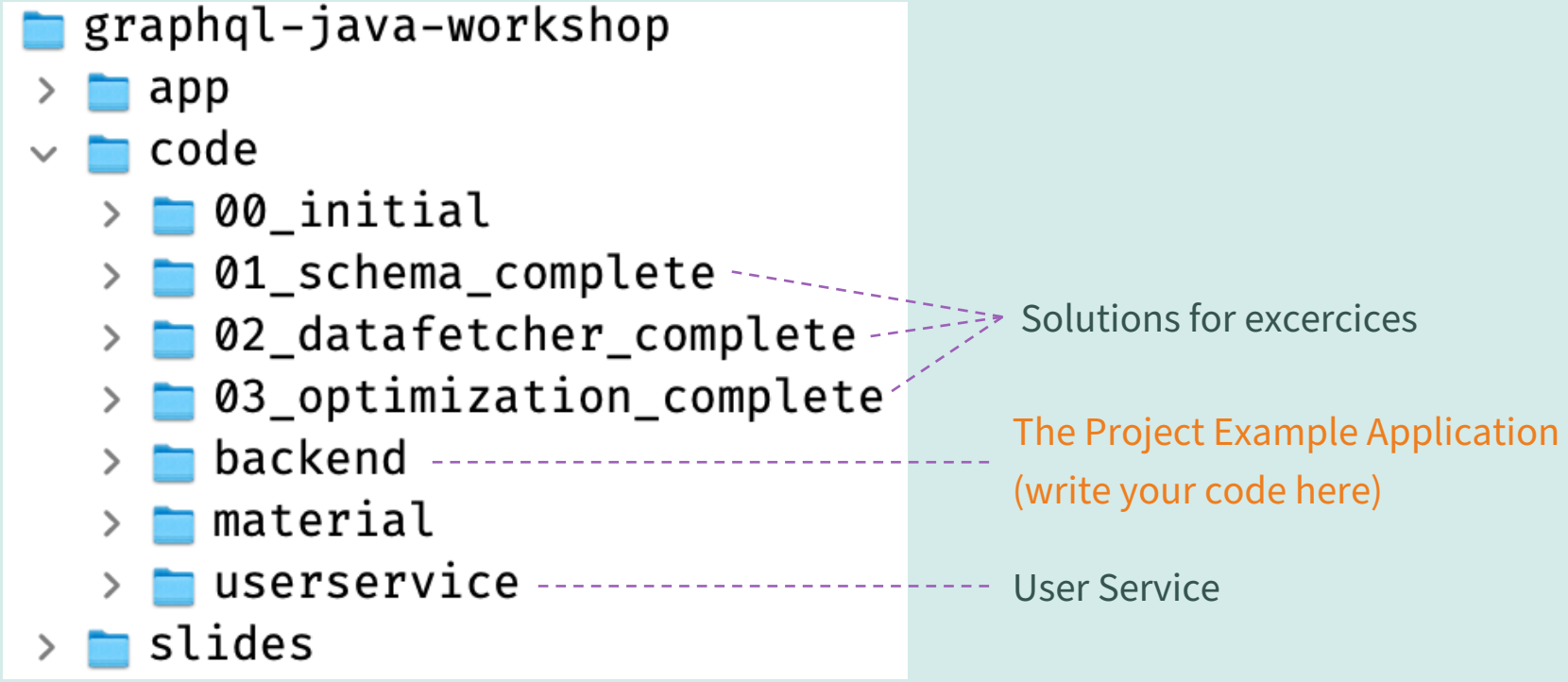
     ```
     git pull
     ```

2. When you're done, please raise your hand in Zoom 🙋

EXCERCISE - PREPARATION

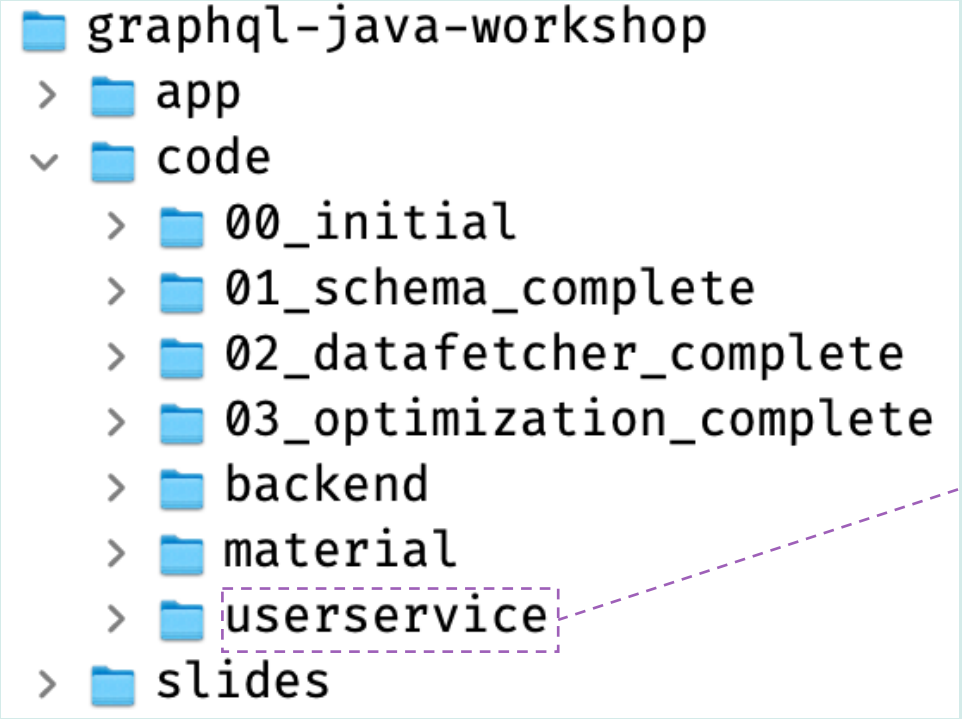
Preparation

The Workshop Repository



EXCERCISE - PREPARATION

Preparation

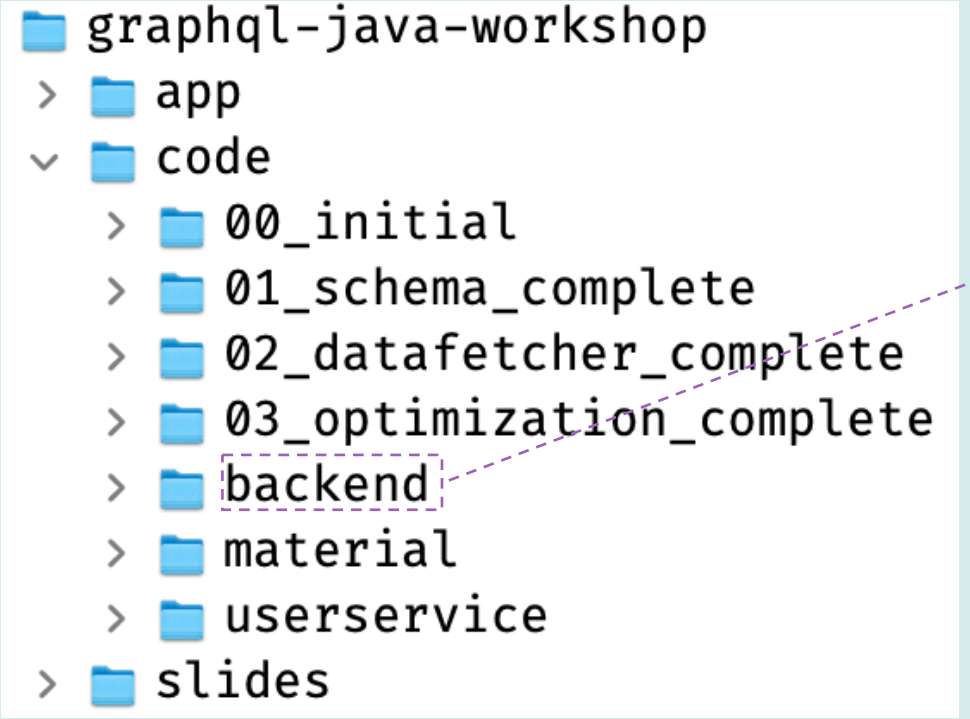


Step 1: Start User Service

1. Open Bash/Shell/Command Line (no IDE needed!)
2. Run:
   ```
   ./gradlew bootRun
   ```
3. Test User Service API: http://localhost:5010/users
4. Raise hand when User Service runs 🙋

EXCERCISE - PREPARATION

Preparation



Step 2: Open and start Example App

1. Open in your IDE
2. Start main class from your IDE:

   `nh.graphql.projectmgmt.ProjectMgmtApplication`

3. Open GraphiQL: http://localhost:5000/graphiql.html
4. Raise hand when GraphiQL runs 🙋

EXCERCISE: DEFINE API SCHEMA

Plugins for you IDE (not needed for today)

GraphQL Plug-in für IDEA "JS GraphQL"

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

GraphQL Plug-in für Eclipse 😢

Anyone aware?

GraphQL Extension for VS Code "Apollo GraphQL for VS Code"

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

EXCERCISE: DEFINE API SCHEMA

Excercise: Complete the schema for the Example Application

1. Open `src/main/resources/projectmgmt.graphqls`
   - You find TODOs in there with further information

2. After changing and saving the file you can inspect your changes in GraphiQL: http://localhost:5000/graphiql.html

   - You can see your changes in the Docs tab on the right side.
   - Running the queries does not work yet!

3. When you're finished, please raise your hand 🙋

   - If you have questions, don't hesitate to ask!
   - You can find a solution in `code/01_schema_complete`

RESOLVING YOUR QUERY: DATA FETCHERS

DATA FETCHERS

DataFetcher (in other GraphQL libs sometimes called **Resolvers**)

- *A **DataFetcher** determines and returns the value for a Field*
  - Required for all fields of your Root-Types (Query, Mutation)
  - For all other fields, Reflection can be used (getter/setter, Maps, ...)

DATA FETCHERS

DataFetcher (in other GraphQL libs sometimes called **Resolvers**)

- *A **DataFetcher** determines and returns the value for a Field*
  - Required for all fields of your Root-Types (Query, Mutation)
  - For all other fields, Reflection can be used (getter/setter, Maps, ...)

- A `DataFetcher` is a functional interface in Java (and can be implemented as a Lambda function)

DATA FETCHERS

DataFetcher (in other GraphQL libs sometimes called **Resolvers**)

- *A **DataFetcher** determines and returns the value for a Field*
  - Required for all fields of your Root-Types (Query, Mutation)
  - For all other fields, Reflection can be used (getter/setter, Maps, ...)

- A `DataFetcher` is a functional interface in Java (and can be implemented as a Lambda function)

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

DATA FETCHERS

DataFetcher (in other GraphQL libs sometimes called **Resolvers**)

- *A **DataFetcher** determines and returns the value for a Field*
  - Required for all fields of your Root-Types (Query, Mutation)
  - For all other fields, Reflection can be used (getter/setter, Maps, ...)

- A `DataFetcher` is a functional interface in Java (and can be implemented as a Lambda function)

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

- Later we will see, how we connect a DataFetcher to our schema

Implementing DataFetchers

- Example: A simple `ping`-Field

  👉 PingFetcher

Schema Definition

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

Implementing DataFetchers

- Example: A simple `ping`-Field

  👉 Try in GraphiQL

Schema Definition

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

Query

```
query { ping }
```

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

Implementing DataFetchers

- Example: A simple `ping`-Field

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";
    }
  };

  // or: as Lambda-Function
  DataFetcher<String> ping = env -> "Hello World";
}
```

The DataFetcherEnvironment

- DataFetcher receiving instances of DataFetcherEnvironment
- This class provides information about the query and it's environment
- It provides the **arguments** for a field
- We will see more use-cases later

```
interface DataFetcherEnvironment {
  <T> T getArgument(String name);
  <T> T getSource();
  <T> T getContext();

  DataFetchingFieldSelectionSet getSelectionSet();

  <K, V> DataLoader<K, V> getDataLoader(String dataLoaderName);
}
```

Implementing DataFetchers: Arguments

👉PingField

Implementing DataFetchers: Arguments

- The **DataFetchingEnvironment** provides the given arguments of a field
- (Complex) input-Types are passed as Map-Instances (more on that later)

Schema Definition

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

Query

```
query {
  ping(msg: "GraphQL")
}
```

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

Implementing DataFetchers: Arguments

- The **DataFetchingEnvironment** provides the given arguments of a field
- (Complex) input-Types are passed as Map-Instances (more on that later)

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: Mutations

- technically the same as queries, but you're allowed to modify data here

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: Subscriptions

- Same as DataFetchers for Query, but must return Reactive Streams Publisher
- Typically used in Web-Clients with WebSockets
- Overall setup and configuration can be tricky

```
type Subscription {
  onNewTask: Task!
}
```

```
import org.reactivestreams.Publisher;


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

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

    return publisher;

  };
}

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

Implementing DataFetchers: Context

- Context can be used to pass informations to data fetchers
- You can define the context object yourself
- Example: Services or Security Informations (current user, roles, ...)

Implementing DataFetchers: Context

- Context can be used to pass informations to data fetchers
- You can define the context object yourself
- Example: Services or Security Informations (current user, roles, ...)

Data Fetcher

```
public class QueryDataFetchers {
  DataFetcher<String> ping = new DataFetcher<>() {
    public Project get(DataFetchingEnvironment env) {
      ProjectMgmtGraphQLContext context = env.getContext();

      ProjectRepository repo = context.getProjectRepository();

      Project project = projectRepository.find(...);
      ...
    }
  };
}
```

Implementing DataFetcher: Return Values

- You can return primitive values, Objects, Lists or Optionals

Schema Definition

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

Project POJO

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

Data Fetcher

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

      return project;
    }
  };
}
```

Connecting Fields to DataFetchers

- The RuntimeWiring connects your DataFetcher with Fields

```
class GraphQLAPIConfiguration {
  RuntimeWiring setupWiring() {

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

    return RuntimeWiring.newRuntimeWiring()




        .build();
  }
}
```

Connecting Fields to DataFetchers

- The RuntimeWiring connects your DataFetcher with Fields

```
type Query {

}
```

```
class GraphQLAPIConfiguration {
  RuntimeWiring setupWiring() {

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

    return RuntimeWiring.newRuntimeWiring()
      .type(newTypeWiring("Query")



      .build();
  }
}
```

Connecting Fields to DataFetchers

- The RuntimeWiring connects your DataFetcher with Fields

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

```
class GraphQLAPIConfiguration {
  RuntimeWiring setupWiring() {

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

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



      .build();
  }
}
```

Connecting Fields to DataFetchers

- The RuntimeWiring connects your DataFetcher with Fields

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

type Project {
  owner: User!
}
```

```
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))


      .build();
  }
}
```

Connecting Fields to DataFetchers

- The RuntimeWiring connects your DataFetcher with Fields

```
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 FOR JAVA APPLICATIONS

EXCERCISE: IMPLEMENT A DATAFETCHER

In case, you look at the example application, some notes...

- Schema is created in GraphQLApiConfiguration.java
- Code there is slightly different to what we've seen on slides, as it uses some Spring code
- Our GraphQL API is exposed as HTTP endpoint using the graphql-java-servlet that is manually registered using Spring API

EXCERCISE: IMPLEMENT A DATAFETCHER

Implement missing Query-DataFetchers for our Application

For the **Query**-type, DataFetchers for fields **projects** and **project** are missing 😢

1. Please implement them in nh.graphql.projectmgmt.graphql.fetcher.**QueryDataFetchers**
   - Add them to RuntimeWiring in **GraphQLApiConfiguration**.setupWiring()
   - In both files you, find TODOs
   - If you haven't finished excercise 1, you can find a complete, working schema file in 01_schema_complete/ src/main/resources, that your can copy into your workspace

2. After implementing your changes, you can now execute queries in GraphiQL
   - Find some example queries on the next page
   - owner and assignee and single task fields cannot be queried yet!

3. When you're finished, please raise hand in Zoom 🙋

EXCERCISE: IMPLEMENT A DATAFETCHER

Implement missing Query-DataFetchers for our Application

After implementing your DataFetcher, the following queries should work:

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

```
query {
  project(id: "1") {
    id title
    tasks {
      id title
    }
}
```

Note: query for owner / assignee or a single project task will NOT work yet.

MORE ON GRAPHQL-JAVA

Implementing DataFetcher: Objekt Graphs

- In **nested** Queries the return value of a DataFetcher is passed as "source" to the next DataFetcher

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

QueryDataFetchers.projectById:
returns an instance of our Project class

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

Implementing DataFetcher: Objekt Graphs

- In **nested** Queries the return value of a DataFetcher is passed as "source" to the next DataFetcher
- If there is no DataFetcher configured for a field, a **PropertyDataFetcher** is used by default
- The PropertyDataFetcher tries to receive values using Reflection
  - For Root-Fields an own DataFetcher is *required*!

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

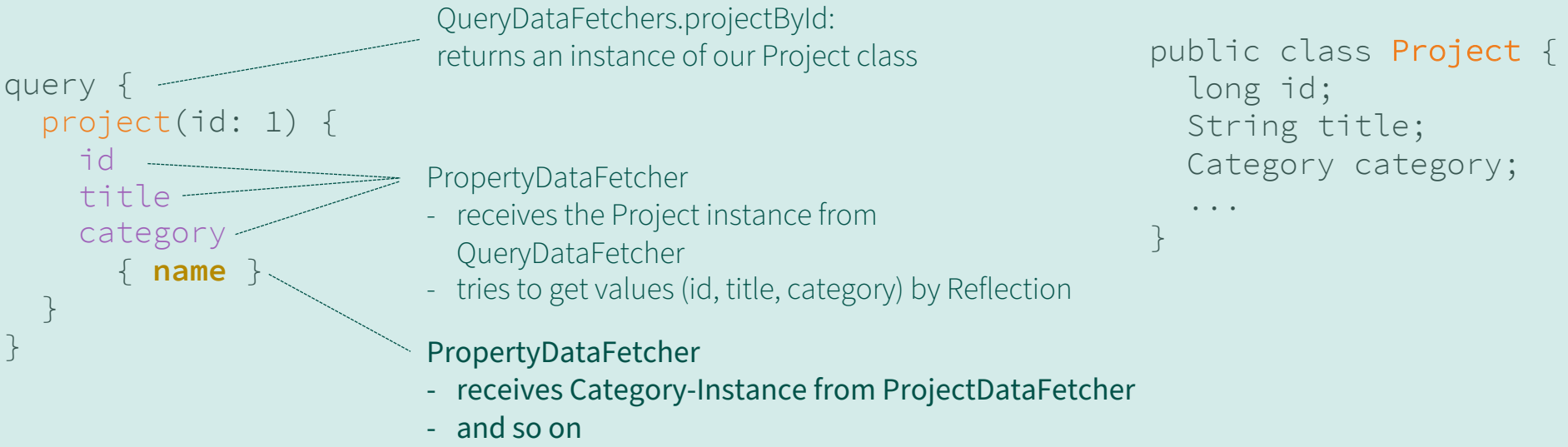
QueryDataFetchers.projectById:
returns an instance of our Project class

PropertyDataFetcher
- receives the Project instance from QueryDataFetcher
- tries to get values (id, title, category) by Reflection

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

Implementing DataFetcher: Objekt Graphs

- In **nested** Queries the return value of a DataFetcher is passed as "source" to the next DataFetcher
- If there is no DataFetcher configured for a field, a **PropertyDataFetcher** is used by default
- The PropertyDataFetcher tries to receive values using Reflection
  - For Root-Fields an own DataFetcher is *required*!



DataFetcher for own Types (not Root Types)

- Default PropertyDataFetcher does not always work
- If there is a "mismatch" between your GraphQL API and Java API
  - Different/missing field names
  - other types

DataFetcher for own Types (not Root Types)

- Default PropertyDataFetcher does not always work
- If there is a "mismatch" between your GraphQL API and Java API
  - Different/missing field names
  - other types
- Example: There is no field "owner" on our Project class
  - only the Id of a User object
  - (the actual user must be read from MicroService)

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

PropertyDataFetcher does not work here 😱

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

DataFetcher for own Types (not Root Types)

- You can write DataFetcher for *all* fields in your GraphQL API
- Non-Root Fetcher works the same, as DataFetchers for Root-Fields
- They receive their parent object as "Source"-Property from the DataFetchingEnvironment

```
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();

      UserService userService =
        env.getContext().getUserService();

      return userService.getUser(ownerId);
    }
  };
}
```

DataFetcher for own Types (not Root Types)

- You have to connect your DataFetchers with their types in RuntimeWiring

```
type Project {
  owner: User!
}
```

```
class GraphQLAPIConfiguration {
  RuntimeWiring setupWiring() {

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

    return RuntimeWiring.newRuntimeWiring()
      .type(newTypeWiring("Query")
      //  ....
      .type(newTypeWiring("Project").
        .dataFetcher("owner", projectFetchers.owner))
      .build();
  }
}
```

EXCERCISE 3: IMPLEMENT DATAFETCHER

Implement DataFetcher for owner and task of the Project type

1. You need to implement DataFetchers
   nh.graphql.projectmgmt.graphql.fetcher.**ProjectDataFetchers**
   - There you find TODOs
   - Register your DataFetchers in **GraphQLApiConfiguration**.setupWiring()!
   - If you haven't finished the previous excercise, you can copy the class QueryDataFetchers from the folder "02_query_datafetcher_complete" into your workspace and also "GraphQLApiConfiguration" from that folder

2. Make sure your DataFetchers work, by executing both queries on the following slide

3. When you're finished, raise your hand 🙋

EXCERCISE 3: IMPLEMENT DATAFETCHER

Implement DataFetcher for owner and task of the Project type

Now both queries should work:

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

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

BACKGROUND: RUNNING QUERIES

BACKGROUND: RUNNING QUERIES

Executable Schema

- Schema und DataFetcher (Wirings) must be connected in a GraphQLSchema
- GraphQL Schema is used to execute Queries

```
class GraphQLApiConfiguration {

  public GraphQLSchema graphqlSchema() {

    File schemaFile = new File("tasks.graphqls");

    RuntimeWiring runtimeWiring = setupWiring();


    SchemaGenerator schemaGenerator = new SchemaGenerator();

    return schemaGenerator.makeExecutableSchema(
      new SchemaParser().parse(schemaFile),
      runtimeWiring
    );
  }
}
```

BACKGROUND: RUNNING QUERIES

Execute Queries in Java

- Result will be returned in nested Map

👉 nh.graphql.projectmgmt.graphql.config.GraphQLApiConfiguration

👉 nh.graphql.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();
```

BACKGROUND: EXPOSING YOUR API

Execute Queries using HTTP endpoint

- Requirement: GraphQL Schema is created (as seen before)

- Option 1: https://github.com/graphql-java/graphql-java-spring
  - REST Controller for Spring (Boot)
  - From same team as graphql-java
  - No support für Subscriptions(?)

- Option 2: https://github.com/graphql-java-kickstart/graphql-java-servlet
  - HTTP Servlet (for Spring or other Servlet Containers)
  - Spring Boot starter available
  - Support for Subscriptions

BACKGROUND: EXPOSING YOUR API

Example: graphql-java-servlet

- Deplyoment of servlet is container/server specific
- Example from our Spring-based application, just to get an idea:

```
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();
  }
}
```

RUNTIME OPTIMIZATIONS

An Example...

DataFetcher

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

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

An Example...

DataFetcher

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

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

Query

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

Question:What happens at runtime? 🤔

An Example...

***One* Database call**
(returns *n* Project-objects)

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

***n* REST-Calls**
(1 for each Project)

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

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

1+n-Problem 😱

Option 1: Async Data Fetcher

- DataFetcher can be run asynchronous in parallel

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

For example: could run in parallel

Option 1: Async Data Fetcher

- DataFetcher can be run asynchronous in parallel

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

Determining all n Tasks could run in parallel, too

Option 1: Async Data Fetcher

- DataFetcher can be run asynchronous in parallel

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

👉*Example: estimation-Field (ProjectDataFetcher)*
👉*Add estimation in GraphQLApiConfiguration*

Option 1: Async Data Fetcher

- DataFetcher can be run asynchronous in parallel
- A DataFetcher can therefore return a CompleteableFuture
  - async convience function can help wrapping a result in a CF

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

Long-running operation

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

Option 1: Async Data Fetcher

***One* Database call**
(returns *n* Project-objects)

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

```
import graphql.schema.AsyncDataFetcher.async;
```

***n* REST-Calls**
(1 for each Project),
but in parallel!

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

- Das not really solve 1+n-problem, but can lead to better performance (also in other use-cases)

Back to our example...

DataFetcher

```
DataFetcher<List<Project>> projectsFetcher = ...;

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

Query

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

Result

```
"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" } }
  ]
}
```

What is still problematic? 🤔

Back to our example...

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 unnecessary redundant loading of data (👉Console) 😱😱

Enhanced Query… …it gets even more evil

DataFetcher

```
DataFetcher<List<Project>> projectsFetcher = ...;

DataFetcher<User> ownerFetcher = async(...);

DataFetcher<User> assigneeFetcher = ...;
```

Query

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

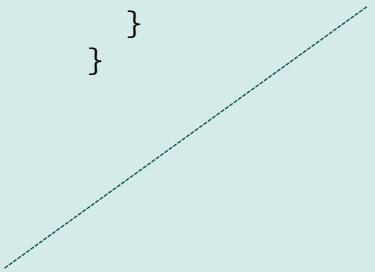
Enhanced Query... ...it gets even more evil

DataFetcher

```
DataFetcher<List<Project>> projectsFetcher = ...;

DataFetcher<User> ownerFetcher = async(...);

DataFetcher<User> assigneeFetcher = env -> userService....;
```

Query

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



Even more Users => even more requests to remote UserService 😱
Might be same users as for project owners => even more unnecessary, redundant requests 😫
-> console!

DataLoader: batching and caching of requests

- Idea taken from JS reference implementation, but common idea in many GraphQL frameworks
- Batches calls to (remote) services
- Can work asynchronous
- All obtained data is cached for the time of one query

Loading Data

- Loading of data is not done directly in DataFetchers anymore but delegated to a DataLoader
- DataLoaders are configured and registered on application startup and can be accessed in all DataFetchers

An Example...

Könnte man auch in Whiteboard Excellidraw erklären

***One* Database call**
(returns *n* Project-objects)

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

***n* Invocations**
(1 for each Project)

```
DataFetcher<User> ownerFetcher =
  env -> {
   String ownerId = (Project)env.getSource()).getOwnerId();

   return userService.getUser(ownerId);

   dataLoder.load(ownerId); // simplified
);
```

***1* Invocation**
for all collected, unique IDs

```
// pseudo code
class UserDataLoader {
  load(List<String> userIds) { ... }
}
```

Adding DataLoader to DataFetcher

- Actual loading of data is moved to a DataLoader (we'll see the implementation later)
- In your DataFetcher, you pass "something" to the DataLoader, that you can later use to load the acutal data (for example the ID of a User Object) in your DataLoader implementation
- GraphQL waits for execution the DataLoader as long as possible, collects all Ids and passes them to your DataLoader implementation

```
public DataFetcher ownerFetcher = new DataFetcher<>() {
  public Object get(DataFetchingEnvironment env) {
      // as seen, no changes here
      Project project = env.getSource();
      String userId = project.getOwnerId();

      // no UserService access anymore, instead delegate to DataLoader

      DataLoader<String, User> dataLoader = env.getDataLoader("userDataLoader");
      return dataLoader.load(userId);
    }
  };
}
```

Loading Data

- The BatchLoader is invoked by GraphQL with an amount of Ids that are collected by an amount of DataFetcher invocations

  For Example: a DataFetcher for User is called 3 times and returns 3 User-Ids . The BatchLoader then is invoked with a List of this three Ids.

Loading Data

- The BatchLoader is invoked by GraphQL with an amount of Ids that are collected by an amount of DataFetcher invocations

  For Example: a DataFetcher for User is called 3 times and returns 3 User-Ids . The BatchLoader then is invoked with a List of this three Ids.

- For each ID, the BatchLoader has to load the appropriate object (for example by invoking a remote service)

```
class ProjectDataLoaders {
  public BatchLoader<String, User> userBatchLoader = new BatchLoader<>() {
    public List<User> load(List<String> keys) {

      // For each key, load the User and return it
      return keys.stream()
              .map(userService::getUser)
              .collect(Collectors.toList()
      );
    }
  }
}
```

Loading Data

- A BatchLoader can be implemented asynchronous, so multiple BatchLoaders can run in parallel
- As with DataFetchers, return a CompletableFuture-Object

```
class ProjectDataLoaders {
  public BatchLoader<String, User> userBatchLoader = new BatchLoader<>() {
    public CompletableFuture<ListUser>> load(List<String> keys) {

      // For each key, load the User and return it
      return CompletableFuture.supplyAsync(() -> keys.stream()
              .map(userService::getUser)
              .collect(Collectors.toList()
      );
    }
  }
}
```

Accessing Context in BatchLoader

- Use BatchLoaderWithContext to receive your GraphQL context object

```
class ProjectDataLoaders {
  public BatchLoaderWithContext<String, User> userBatchLoader = new BatchLoaderWithContext <>() {
    public CompletableFuture<List<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()
      );
    }
  }
}
```

Registering DataLoaders

- DataLoader have to be registered at the DataLoaderRegistry
- In our application we do this when creating the GraphQLContext

```
class ProjectMgmtGraphQLContextBuilder {
  public GraphQLContext build(...) {
    GraphQLContext context = new ...;

    DataLoaderOptions options = DataLoaderOptions.newOptions()
                                .setBatchLoaderContextProvider(() -> context);

    ProjectDataLoaders dataLoaders = new ProjectDataLoaders();

    context.getDataLoaderRegistry().register("userDataLoader",
      DataLoder.newDataLoader(
        dataLoaders.userDataLoader,
        options
      )
    );

    return context;
  }
}
```

Result: Calls to UserService have been reduced

- We "only" have 1+n calls where n is number of unique users in the whole query
- If our endpoint would accept a list of Ids, we even would have 1+1
- 👉 console

```
query {
  owner @useDataLoader {
    id name
  }

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

EXCERCISE: OPTIMIZING REMOTE CALLS

It's up to you: optimize our REST-Calls using DataLoader 🔥

EXCERCISE: OPTIMIZING REMOTE CALLS

It's up to you: optimize our REST-Calls using DataLoader 🔥

1. Please implement a *BatchLoaderWithContext* in nh.graphql.projectmgmt.graphql.fetcher.**ProjectDataLoaders** (see TODOs dort)

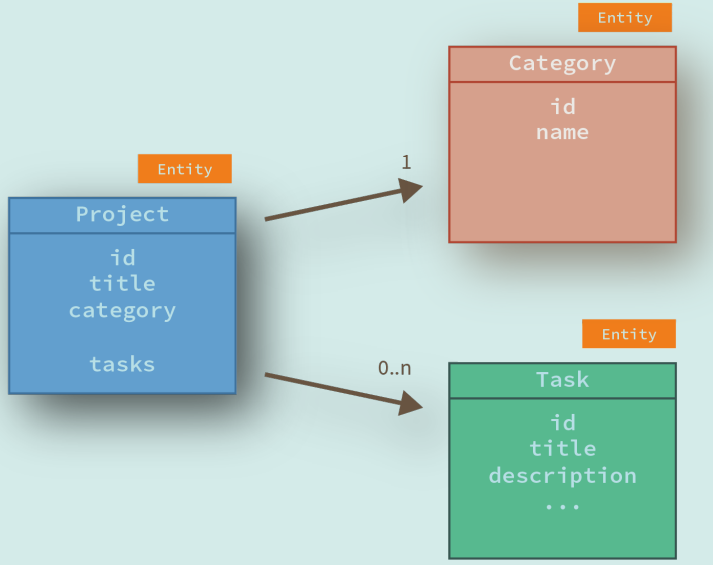
2. In **ProjectMgmtGraphQLContextBuilder.addDataLoaders**(DataLoaderRegistry) instantiate the BatchLoader and register it

3. Use your DataLoader in **ProjectDataFetchers** and **TaskFetchers** to load the User Objects

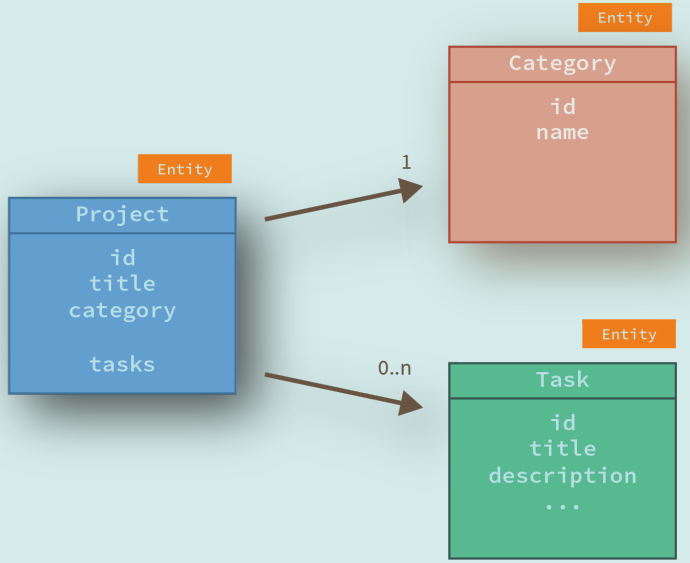
4. Please verify in the application log (or with debugger), that remote requests are reduced when you're finished. The UserService logs every request, so you can see outgoing requests on the console
5. When you're finished, please raise hand 🙋

Optimizing Database Access (opionionated!)

Database Access: Our (JPA-)Model



Accessing the database: An example query...

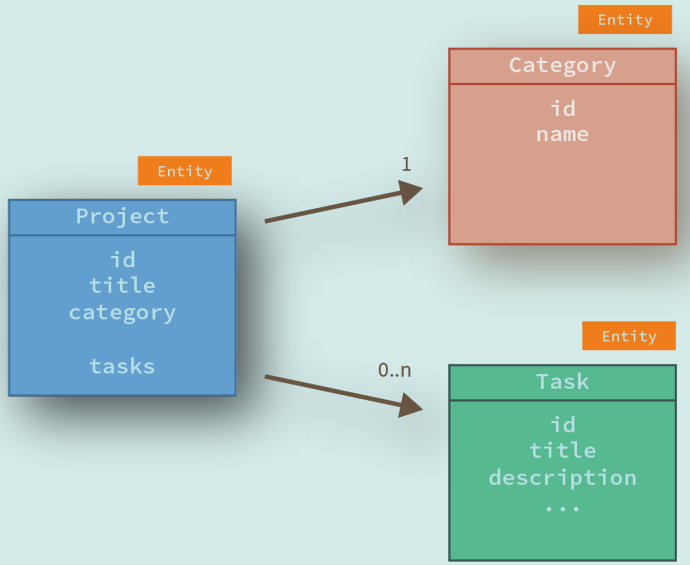



```
query {
  projects {
    title
    category { name }
  }
}
```

Question: How do we map relation Project-Category? Lazy? Eager? 🤔

What would happen in one or the other case?

Accessing the database: Another example query...

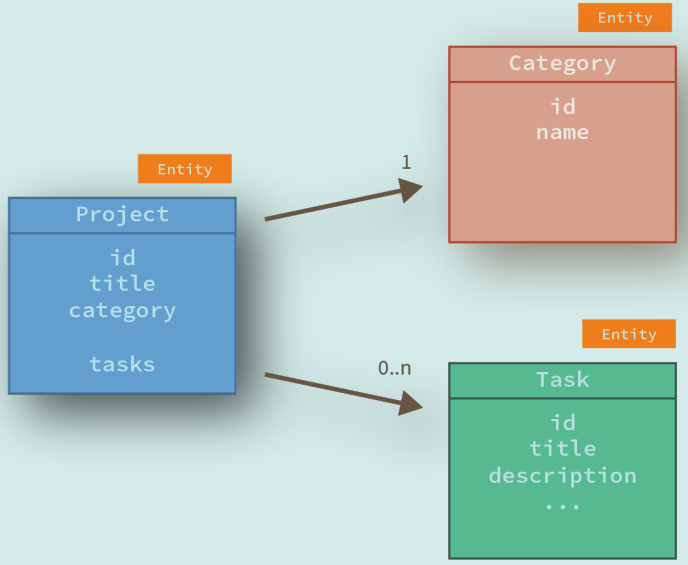



```
query {
  projects {
    title
    tasks { id title }
  }
}
```

Question: How do we map relation Project-Task? Lazy? Eager? 🤔

What would happen in one or the other case?

Accessing the database: An third query...




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

Question: what should we do?! 🤔

DATABASE ACCESS

Accessing the database

We need optimized database SQL queries based on your concrete GraphQL Query!

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

      // 👉 TODO: Optimize based on the actual query
      return projectRepository.findAll();
    }
  };
}
```

What Information do we need here?

SelectionSet: Contains *all* requested fields of a Query

- Access via DataFetchingEnvironment
- We can use the information provided to dynmically construct SQL/JPA/... database queries
- Again: this examples are opionionated, you can use the SelectionSet also in other ways

```
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: Optimice Database Acccess

- Example: 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();
  }
}
```

Modify JPA fetch behaviour based on the actual needed data

SelectionSet: Optimize DB access with JPA FetchGraph

- Example
- 👉 console

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

SUMMARY

Determining data in graphql-java

- For *any* field there is a DataFetcher (Default: PropertyDataFetcher)
- DataFetcher can work asynchronously (by returning a CompletableFuture)
- Using a DataLoader can reduce (remote) calls with caching and batching
- With a SelectionSet per-query optimizations are possible (for example optimized database queries)

OUTLOOK: GRAPHQL-JAVA-TOOLS

graphql-java-tools
https://www.graphql-java-kickstart.com/tools/

- Abstraction, based on graphql-java
- Use POJOs for Data Resolvers

Resolvers with graphql-java-tools

- Resolvers are simple POJOs, Schema defined with SDL
- Checks on application startup if all required resolvers are found

Resolver with graphql-java-tools

- Example: a query resolver
- GraphQLQueryResolver is only a marker interface

```
public class ProjectAppQueryResolver implements GraphQLQueryResolver {

  // Methods on resolver classes must match field names in the schema
  // GraphQL field arguments are passed as method arguments
  public String ping(String msg) {
    return "Hello, " + msg;
  }

  // You can still access graphl-java's DataFetchingEnvironment
  public List<Project> projects(DataFetchingEnvironment env) {
    boolean withCategory = ...;
    boolean withTasks = ...;
    return beerRepository.findAll(withCategory, withTasks);
  }
}
```

Resolver with graphql-java-tools

- Complex arguments (GraphQL input types) are deserialized to POJO instances and passed to your resolver method

```
public 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 with graphql-java-tools

- Resolvers for fields not on root types are typed with their Java types
- The parent object (getSource in graphql-java) will be passed as first method parameter

```
public class ProjectResolver implements GraphQLResolver<Project> {

  public User getOwner(Project parent) {
    return userService.getUser(parent.getOwnerId());
  }

}
```

GRAPHQL-JAVA-TOOLS FOR SPRING BOOT

Spring Boot Starter

- https://github.com/graphql-java-kickstart/graphql-spring-boot

GRAPHQL-JAVA-TOOLS FOR SPRING BOOT

Spring Boot Starter

- https://github.com/graphql-java-kickstart/graphql-spring-boot

- Merges automatically all Schema-Files in classpath (`*.graphqls`)
- Resolver classes that are annotated as Spring Beans (for example `@Component`) are automatically recognized
- Configuration of GraphQL servlet via `application.properties`
- GraphiQL or Playground can be activated and configured by properties

Schema Design

- There is room for improvement in our schema
- We will discuss this only, technically there is nothing new (GraphQL related)

SCHEMA DESIGN

Do you see potential problems with our schema ?

```
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!
}
```

Lists: no filtering, sorting, pagination!

SCHEMA DESIGN

Do you see potential problems with our schema ?

```
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!
}
```

Error Handling: what do we do, when something goes wrong?

SCHEMA DESIGN

Do you see potential problems with our schema ?

```
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!
}
```

Security: is the user allowed to add a new Task?

SCHEMA DESIGN

Do you see potential problems with our schema ?

- How could we improve? Dou you have ideas?

```
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!
}
```

PAGINATION

GraphQL doesn't say anthing about pagination, ordering,...

Two approaches:Page-based or cursor-based

Example: Page-based

```
type Query {
  projects(
    page: Int!,
    pageSize: Int!): ProjectList!
}

type ProjectList {
  page: Int!
  totalElements: Int!
  hasNext: Boolean!
  hasPrev: Boolean!

  projects: [Project!]!
}
```

PAGINIERUNG

GraphQL doesn't say anthing about pagination, ordering,...

Simplified example with Spring Data

```
type Query {
  projects(
    page: Int!,
    pageSize: Int!): ProjectList!
}

type ProjectList {
  page: Int!
  totalElements: Int!
  hasNext: Boolean!
  hasPrev: Boolean!

  projects: [Project!]!
}
```

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;

public class ProjectQueryResolver implements
  GraphQLQueryResolver {

  @Inject
  private ProjectRepository projectRepository;

  public ProjectList projects(int page, int pageSize) {
    Page<Project> page = projectRepository.find(
      PageRequest.of(page, pageSize)
    );

    return new ProjectList(
        page.getNumber(),
        page.getTotalElements(),
        page.hasNext(), page.hasPrevious(),
        page.getContent()
    );
  }
}
```

PAGINIERUNG

GraphQL doesn't say anthing about pagination, ordering,...

Ordering also using own fields or arguments

=> not comparable to out-of-the-box-power of SQL, you have to make sure that your API works as desgined...

```
enum Direction {
  asc, desc
}

type ProjectOrderCriteria {
  field: String!
  direction: Direction!
}

type Query {
  projects(
    page: Int!,
    pageSize: Int!,
    orderBy: ProjectOrderCriteria
  ) : ProjectList!
}
```

```
query {
  projects(page: 1,
           pageSize: 20,
           orderBy(field: "title",  direction: asc)
           ) {
                hasNext
                projects {
                    id
                    title
                }
            }
  }
}
```

GraphQL doesn't say anything about security

SECURITY

GraphQL doesn't say anything about security

Question 1: is your api public or not? Do you want user to be logged in to use your api?

Example with Spring Security: Secure your /graphql endpoint

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

  protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/graphql").authenticated();
  }
}
```

SECURITY

GraphQL doesn't say anything about security

You can secure your business logic as usual

Example: Spring Security (would be similiar with JEE)

```
type Mutation {
  addTask
  (taskInput:
   TaskInput!):
     Task!
}
```

```
public class TaskMutationResolver implements
 GraphQLMutationResolver {

private TaskRepository taskRepository;

  @PreAuthorize(
    "isAuthenticated()"
  )
  public Task addTask(AddTaskInput newTask) {
    Task task = Task.from(newTask);
    taskRepository.save(task);
    return task;
  }
}
```

Schema Directives

Directives are similiar to Annotations in Java

They can be specified in a schema for the Schema itself or with usage in Operations

You could write a directive and process it in your application

```
directive @adminOnly on FIELD_DEFINITION


type Mutation {
  addTask @adminOnly (taskInput: TaskInput!): Task!
}
```

SECURITY

GraphQL doesn't say anything about security

Errors would end up in 'errors'-Objekt

(You could customize the output of the errors object)

ERROR HANDLING

(Technical) Errors end up in the errors-object in your response

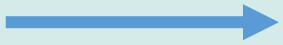
- Business/domain errors could also be part of your schema (own type)

ERROR HANDLING

(Technical) Errors end up in the errors-object in your response

- Business/domain errors could also be part of your schema (own type)
- for example: validation errors

```
type Mutation {
  addTask (taskInput: TaskInput!): Task!
}
```



```
type Mutation {
  addTask(taskInput: TaskInput!): AddTaskResult!
}

type AddTaskResult {
  newTask: Task
  validationErrors: [ValidationError!]
}

type ValidationError {
  field: String!
  msg: String!
}
```

ERROR HANDLING

(Technical) Errors end up in the errors-object in your response

- Business/domain errors could also be part of your schema (own type)
- for example: validation errors
- or as Union Type

```
union AddTaskResult = AddTaskSuccess | AddTaskFailure

type AddTaskSuccess {
  newTask: Task
}

type AddTaskFailure {
  validationErrors: [ValidationError!]
}

type Mutation {
  addTask(taskInput: TaskInput!): AddTaskResult
}
```

```
mutation addTask(taskInput: "...") {
  ...on AddTaskSuccess {
    newTask { id title }
  }

  ...on AddTaskFailure {
    validationErrors: { msg }
  }
}
```

OUTLOOK

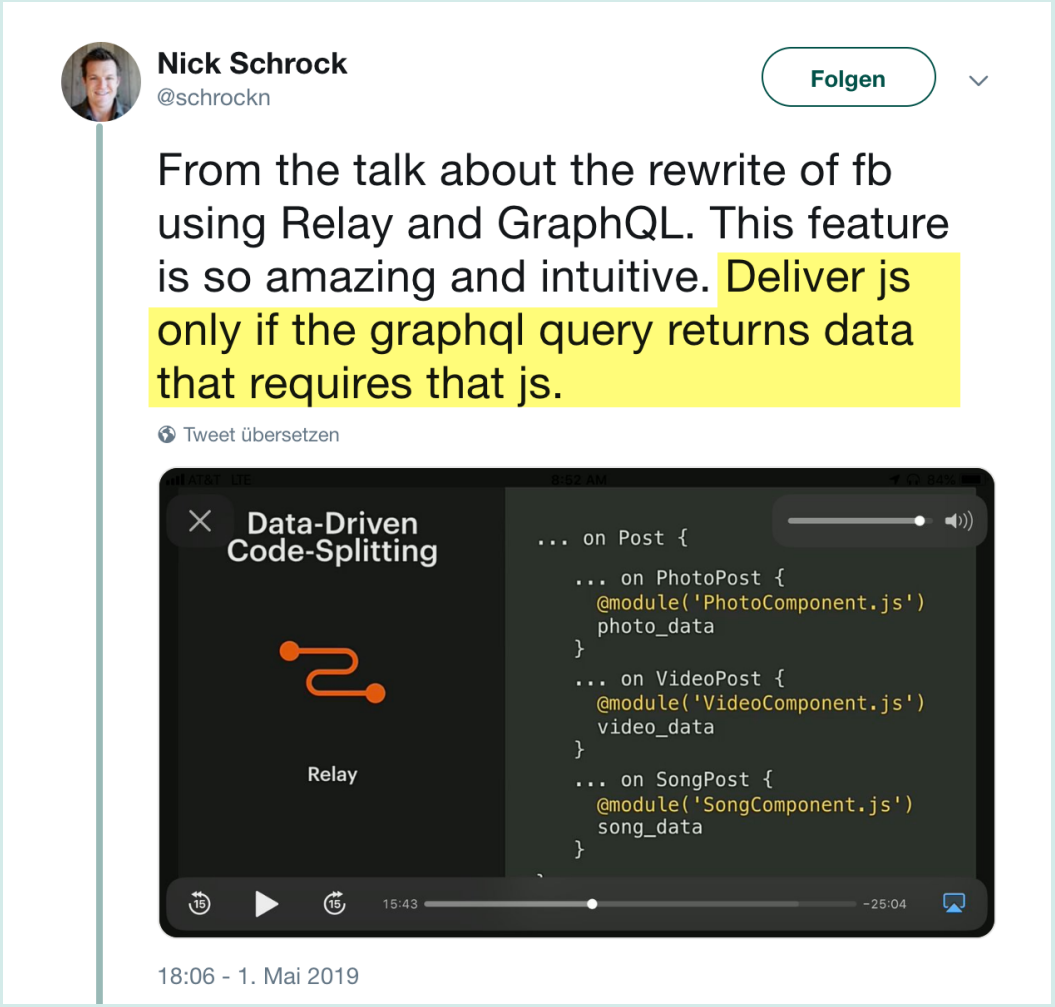
Q & A Discussions

OUTLOOK

- **Do you have questions?**
- **What should we look deeper into?**

https://twitter.com/leeb/status/1123326647552266241

FACEBOOK 5



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

NEXT GEN GRAPHQL?

Thank you!

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

Slides: https://react.schule/api-conf-graphql

Contact: nils@nilshartmann.net