

**NILS HARTMANN**

<https://nilshartmann.net>

**Außenseiter oder Mainstream?**

# **GraphQL**

**Eine Einführung**

Slides (PDF): <https://react.schule/javaland2021-graphql>

# NILS HARTMANN

nils@nilshartmann.net

**Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg**

Java  
JavaScript, TypeScript  
React  
GraphQL

**Trainings & Workshops**

**...auch online bzw. remote!**



<https://reactbuch.de>

**HTTPS://NILSHARTMANN.NET**

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

- <https://graphql.org>

# GraphQL

*Spezifikation: <https://graphql.org/>*

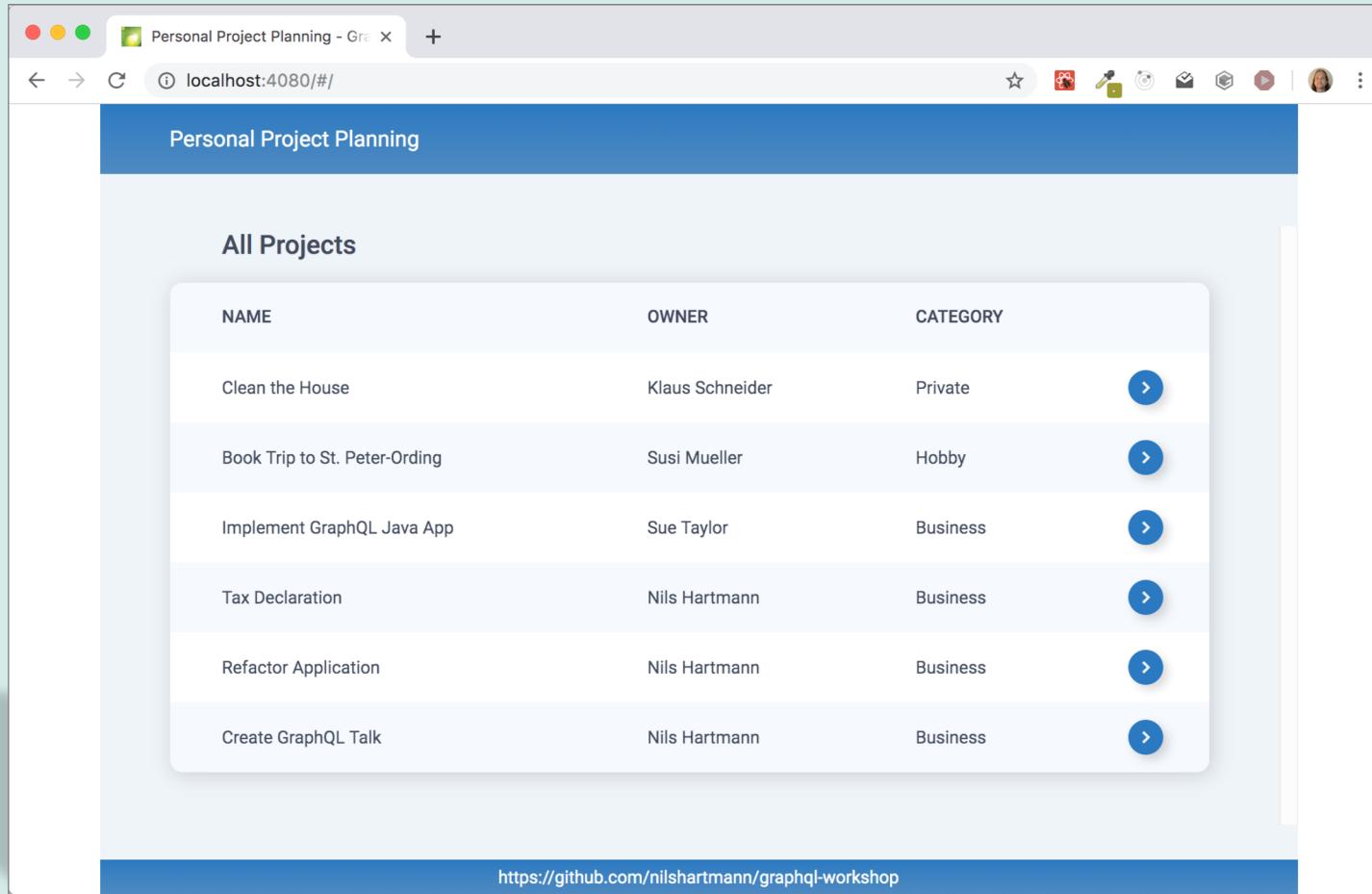
- 2015 von Facebook erstmals veröffentlicht
- Weitere Entwicklung seit 2018 in GraphQL Foundation
- Umfasst:
  - Query Sprache und -Ausführung
  - Schema Definition Language
  - Nicht: Implementierung
    - Referenz-Implementierung: graphql-js

## *GraphQL != Mainstream*

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

## *GraphQL != Mainstream*

- Implementierungen und Einsatz noch "bleeding edge" (?)
- Wenig erprobte Best-Practices (?)
- Talk-Titel-Frage damit beantwortet ✓

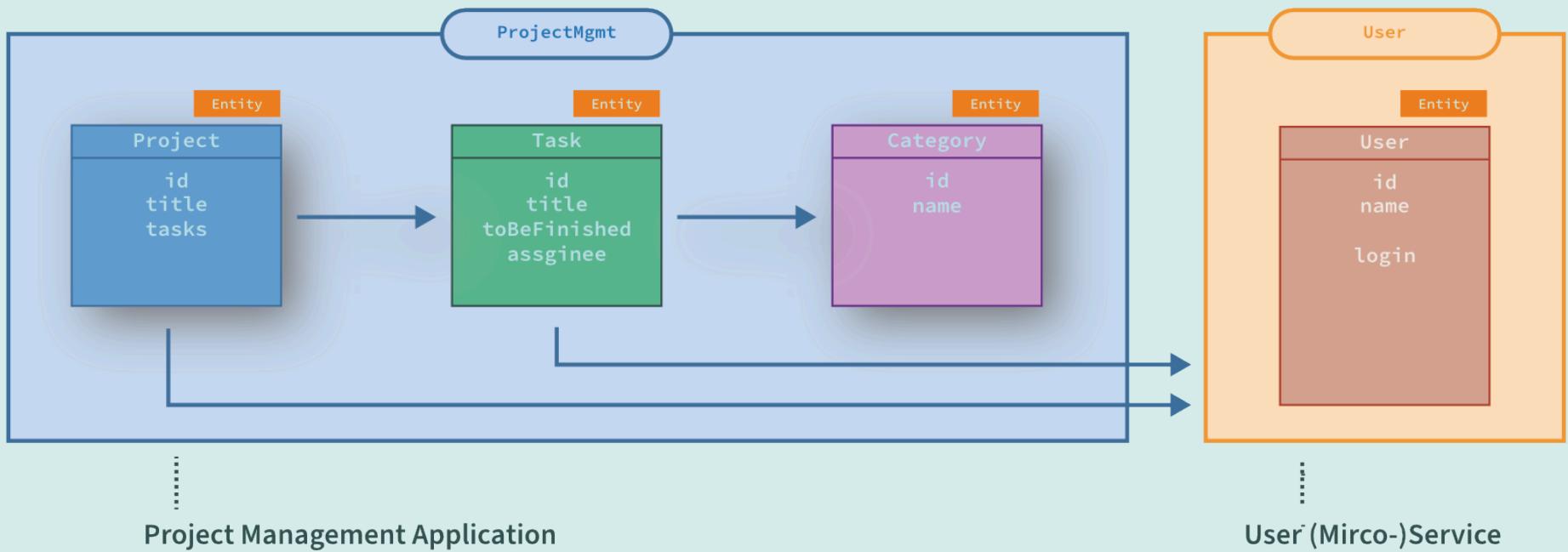


# GraphQL praktisch

Source-Code: <https://github.com/graphql-workshop>

# PROJECT MANAGEMENT APP

## "Architektur"



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

- <https://graphql.org>

# GraphQL

**TEIL 1: ABFRAGEN UND SCHEMA**

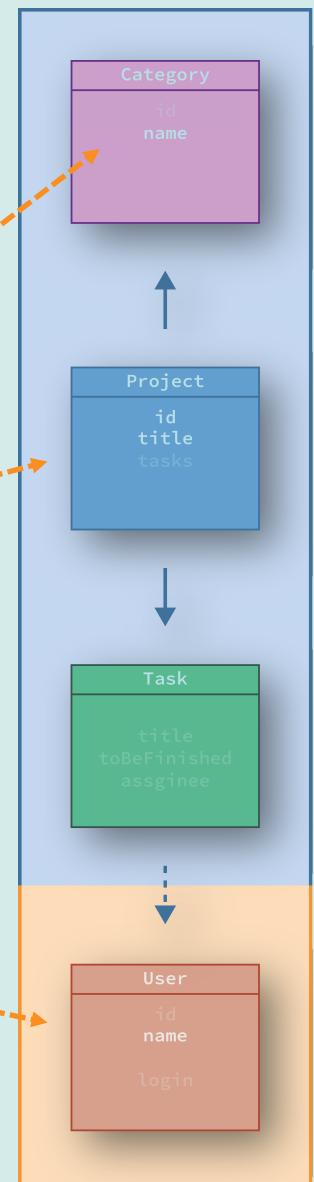
# GRAPHQL EINSATZSzenariEN

## Use-Case spezifische Abfragen – 1

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

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

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



# GRAPHQL EINSATZSzenariEN

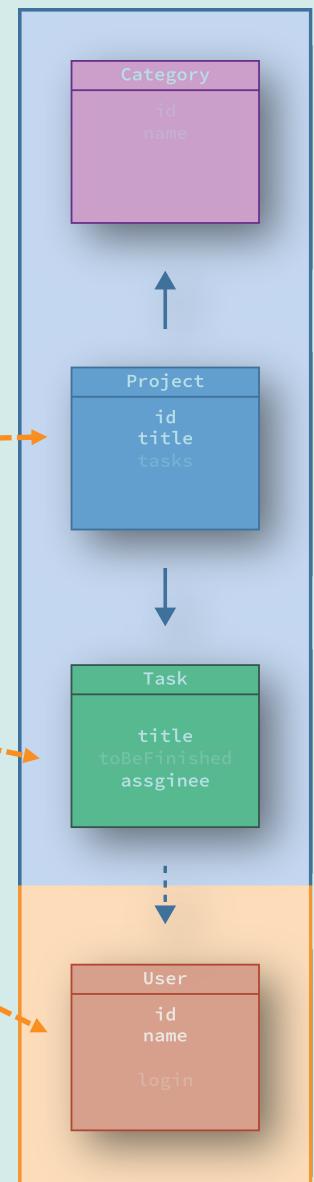
## Use-Case spezifische Abfragen – 2

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

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

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

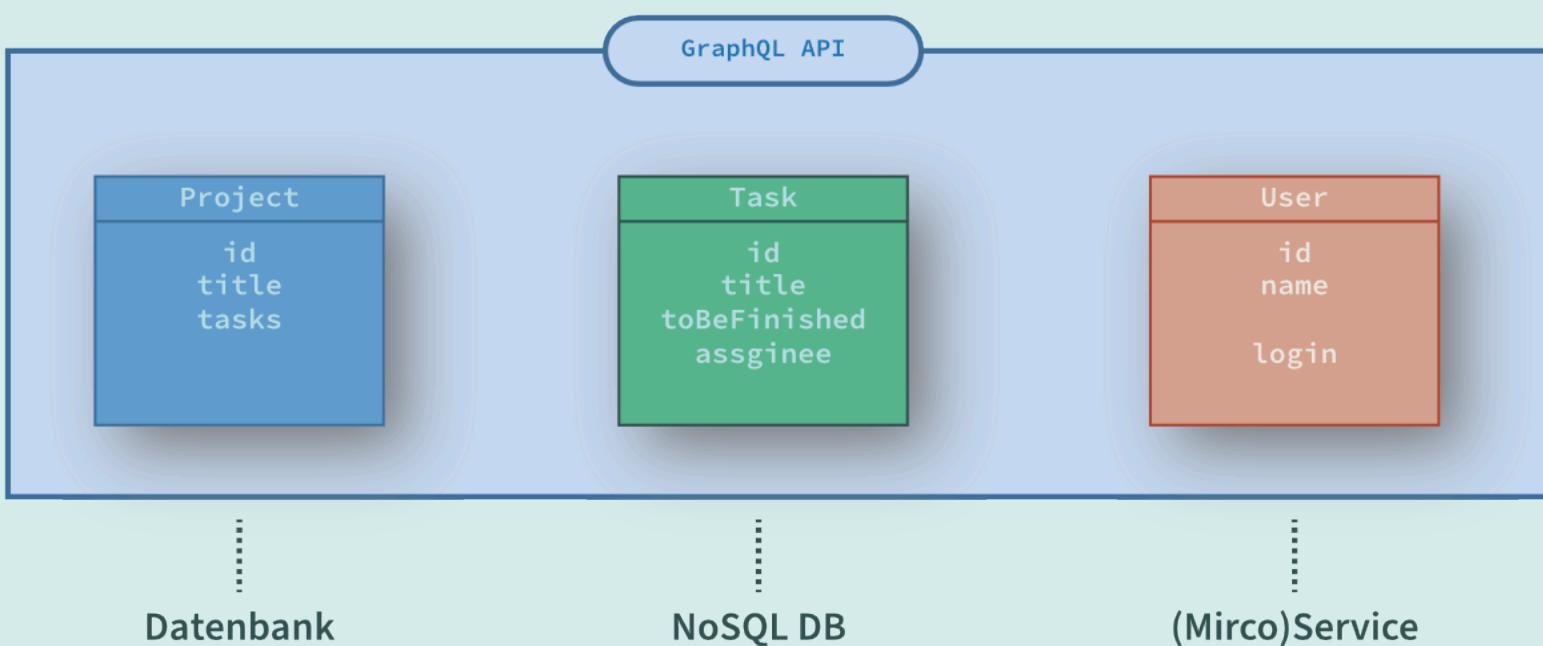
An "Add Task >" button is located at the bottom right of the table. Dashed orange arrows point from the "NAME" column of each row to the "name" field in the GraphQL query on the left.



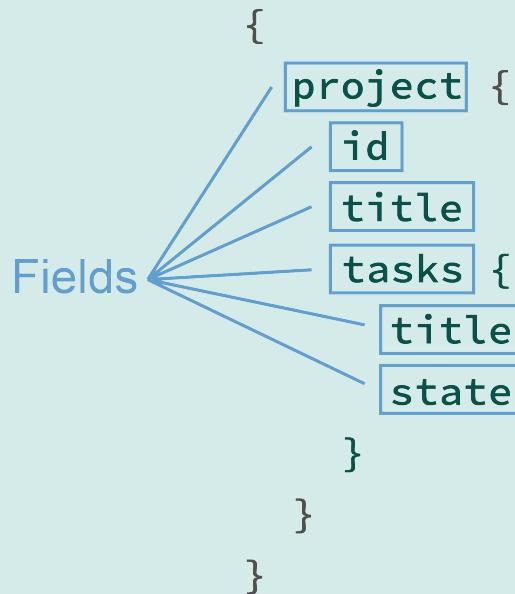
# DATEN QUELLEN

## GraphQL macht keine Aussage, wo die Daten herkommen

- 👉 Ermittlung der Daten ist unsere Aufgabe
- 👉 Müssen nicht aus einer Datenbank kommen
- 👉 Wir bestimmen, wie und welche Daten zur Verfügung gestellt werden!



# QUERY LANGUAGE



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

# QUERY LANGUAGE

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

Fields

Argumente

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

# QUERY LANGUAGE

## Ergebnis

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



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

- Identische Struktur wie bei der Abfrage

# QUERY LANGUAGE: OPERATIONS

**Operation:** beschreibt, was getan werden soll

- query, mutation, subscription

Operation type

Operation name (optional)

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

# QUERY LANGUAGE: 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: $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

- Automatische Benachrichtigung bei neuen Daten

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

# QUERIES AUSFÜHREN

## Queries werden über HTTP ausgeführt

- Üblicherweise per POST
- Ein einzelner Endpoint (üblicherweise /graphql)

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

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

# QUERIES AUSFÜHREN

## Antwort vom Server

- JSON-Objekt
- HTTP Status Codes spielen keine Rolle

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

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

- <https://graphql.org>

# GraphQL (für Java)

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

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Möglichkeit 1: graphql-java

- <https://www.graphql-java.com/>
- Reine GraphQL Implementierung, keine Aussage über Laufzeitumgebung
- API sehr low level und "gewöhnungsbedürftig"
- Grundlage für andere Varianten

## Möglichkeit 2: graphql-java-tools

- <https://github.com/graphql-java-kickstart/graphql-java-tools>
- Reine GraphQL Implementierung, keine Aussage über Laufzeitumgebung
- Implementierung der GraphQL API mit POJOs
- Modular aufgebaut; es existieren z.B. GraphQL Servlets, Auto-Konfiguration für Spring Boot etc.

## Möglichkeit 3: MicroProfile GraphQL

- <https://github.com/eclipse/microprofile-graphql>
- Erst seit Anfang 2020
- Kein Support für Subscriptions
- Schema wird über Annotations definiert
- Support u.a. in Wildfly, Quarkus und Open Liberty

## Möglichkeit 4: Netflix Domain Graph Service Framework (DGS)

- <https://netflix.github.io/dgs/>
- Veröffentlicht Februar 2021
- Basiert auf Spring Boot und graphql-java
- Annotation-basierte DataFetcher
- Code-Generator für Gradle und Maven erzeugt aus Schema Java Klassen
- Support für Subscriptions
- Support für Apollo Federation (Gateway für mehrere GraphQL Services)

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Beispiel: graphql-java

- *Die gezeigten Konzepte sind in GraphQL-Frameworks für andere Programmiersprachen ähnlich!*

# GRAPHQL FÜR JAVA-ANWENDUNGEN

## Schema definieren über Schema-Definition-Language

```
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!  
}
```

## DataFetcher

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

## DataFetcher

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

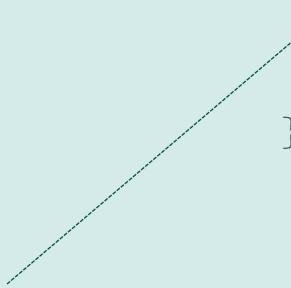
```
interface DataFetcher<T> {  
    T get(DataFetchingEnvironment environment);  
}
```

# DATAFETCHER

## DataFetcher implementieren

- Beispiel: users-Feld

```
public class UserDataFetchers {  
  
    public DataFetcher<List<User>> usersFetcher() {  
        return environment -> userRepository.findAll();  
    }  
  
}  
  
type Query {  
    users: [User!]!  
}  
}
```



# DATAFETCHER

## DataFetcher implementieren: Argumente

- environment gibt Informationen über den Query (z.B. Argumente)

```
public class UserDataFetchers {

    public DataFetcher<List<User>> usersFetcher() {
        return environment -> userRepository.findAll();
    }

    public DataFetcher<User> userFetcher() {
        return environment -> {
            String userId = environment.getArgument("userId");
            return userRepository.getUser(userId);
        };
    }
}

type Query {
    users: [User!]!
    user(userId: ID!): User
}
```

# DATAFETCHER

## DataFetcher implementieren: Mutations

- technisch analog zu Query
- dürfen Daten verändern

```
public DataFetcher<Task> addTaskMutationFetcher() {  
    return environment -> {  
        final Map<String, Object> ri =  
            environment.getArgument("taskInput");  
  
        Task t = new Task();  
        t.setTitle((String)taskInput.get("title"));  
        r.setDescription((String)taskInput.get("description"));  
  
        return taskService.addTask(t);  
    };  
}
```

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

# DATAFETCHER

## DataFetcher implementieren: Subscriptions

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

```
import org.reactivestreams.Publisher;

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

    type Subscription {
        onNewTask: Task!
    }

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

# DATAFETCHER

## DataFetcher implementieren: für eigene Typen

- Felder benötigten eigenen DataFetcher, wenn sie nicht am zuvor zurückgegebenen Objekt enthalten sind
- Beispiel: owner-Feld ist nicht am Project POJO definiert, nur dessen Id (User selbst kommt aus "Microservice")

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

GraphQL API: Owner Object

Java Klasse: "nur" String

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

# DATAFETCHER

## DataFetcher für eigene Typen

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

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

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

### DataLoader: Fasst Aufrufe zusammen und cached Daten

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

## LAUFZEITVERHALTEN: DATALOADER

### DataLoader: Fasst Aufrufe zusammen und cached Daten

- Zusammenfassen von Aufrufen, um unnötige Aufrufe zu vermeiden (Batching)
- Arbeiten asynchron
- Gelesene Daten werden (üblicherweise) für die Dauer eines Requests gecached

## LAUFZEITVERHALTEN: DATALOADER

### DataLoader: Laden von Daten

- Das eigentliche Laden der Daten wird in **BatchLoader** verschoben
- Der BatchLoader wird von GraphQL mit einer *Menge* von IDs aufgerufen, die aus einer *Menge* von DataFetcher-Aufrufen stammen
- Für jede ID muss das gewünschte Objekt zurückgeliefert werden

```
class ProjectDataLoaders {  
    public BatchLoader<String, <Optional<User>> userBatchLoader = new BatchLoader<>() {  
        public CompletableFuture<List<Optional<User>>> load(List<String> keys) {  
            // für jeden Key wird der User geladen und zurückgegeben  
            return users;  
        }  
    }  
}
```

# LAUFZEITVERHALTEN: DATALOADER

## DataLoader: Einbinden im DataFetcher

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

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

## VARIANTE 2: GRAPHQL-JAVA-TOOLS

### Resolver mit graphql-java-tools

- <https://github.com/graphql-java-kickstart/graphql-java-tools>
- Abstraktion, basierend auf graphql-java, arbeitet mit POJOs

## VARIANTE 2: GRAPHQL-JAVA-TOOLS

### Resolver mit graphql-java-tools

- graphql-java-tools verwendet POJOs ("Resolver")
- Argumente der GraphQL API werden direkt übergeben

```
public class ProjectAppQueryResolver implements GraphQLQueryResolver {  
  
    // Zugriff auf DataFetchingEnvironment aus graphql-java möglich  
    public List<Project> user(String userId) {  
        return userRepository.findById(userId);  
    }  
}
```

## VARIANTE 2: GRAPHQL-JAVA-TOOLS

### Beispiel: Mutation

- Auch Argumente sind POJOs

```
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);  
    }  
}
```

## Validierung zur Laufzeit

- Alle Resolver müssen vorhanden sein
  - Return-Types und Methoden-Parameter der Resolver-Funktionen müssen zum Schema passen
- Resolver werden immer mit korrekten Parametern aufgerufen
  - Argumente haben korrekten Typ
  - Argumente sind ggf. nicht null
- Rückgabe-Wert eines Resolvers wird überprüft
  - Client erhält nie ungültige Werte
- Es werden nur Felder herausgegeben, die auch im Schema definiert sind
  - Alle anderen Felder einer Java-Klasse sind "unsichtbar"

## OUT-OF-SCOPE

**GraphQL macht keine Aussage über...**

- Security
- Paginierung, Sortierung

# Zusammenfassung

**GRAPHQL - AUSSENSEITER ODER MAINSTREAM?**

## GraphQL - Zusammenfassung

- **GraphQL != SQL**
  - kein SQL, keine "vollständige" Query-Sprache
    - z.B. keine Sortierung, keine (beliebigen) Joins etc
    - muss man selbst implementieren
  - keine Datenbank!
  - kein Framework!

## GraphQL - Zusammenfassung

- **Interessante, aber noch relativ junge Technologie**

Bricht mit einigen Gewohnheiten aus REST

Erfordert umdenken

REST und GraphQL können zusammen eingesetzt werden

Für APIs, die von Externen verwendet werden sollen, vielleicht noch nicht richtig

## GraphQL - Zusammenfassung

- **Interessante, aber noch relativ junge Technologie**

Bricht mit einigen Gewohnheiten aus REST

Erfordert umdenken

REST und GraphQL können zusammen eingesetzt werden

Für APIs, die von Externen verwendet werden sollen, vielleicht noch nicht richtig

- **Bibliotheken und Frameworks für viele Sprachen**

Prototyp zum Ausprobieren in der Regel schnell gebaut

Für Java mittlerweile vier Bibliotheken, aber: Verbreitung?

## GraphQL - Zusammenfassung

- **Interessante, aber noch relativ junge Technologie**
  - Bricht mit einigen Gewohnheiten aus REST
  - Erfordert umdenken
  - REST und GraphQL können zusammen eingesetzt werden
  - Für APIs, die von Externen verwendet werden sollen, vielleicht noch nicht richtig
- **Bibliotheken und Frameworks für viele Sprachen**
  - Prototyp zum Ausprobieren in der Regel schnell gebaut
  - Für Java mittlerweile vier Bibliotheken, aber: Verbreitung?
- **(noch) kein Mainstream, Empfehlung: ausprobieren, beobachten**



<https://reactbuch.de>

# Vielen Dank!

Beispiel-Code: <https://github.com/nilshartmann/graphql-java-talk>

Slides: <https://react.schule/javaland2021-graphql>

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

**HTTPS://NILSHARTMANN.NET | @NILSHARTMANN**

### Weitere GraphQL Projekte im Java-Umfeld

- **HTTP Endpunkt:** graphql-java-servlet (<https://github.com/graphql-java-kickstart/graphql-java-servlet>)
- **Spring Boot Starter:** <https://github.com/graphql-java-kickstart/graphql-spring-boot>
- **GraphQL Schema mit Java Annotations beschreiben:** <https://github.com/Enigmatis/graphql-java-annotations>

### GraphQL MicroProfile

- **Spezifikation:** <https://github.com/eclipse/microprofile-graphql>
- **Quarkus:** <https://quarkus.io/guides/microprofile-graphql>
- **Open Liberty:** <https://openliberty.io/blog/2020/06/05/graphql-open-liberty-20006.html#GQL>

### GraphQL Code Generator

- **Generator für zahlreiche Sprachen und Bibliotheken:**  
<https://graphql-code-generator.com/>
- **Generator für Queries und Antworten (Java):**  
<https://github.com/adobe/graphql-java-generator>
- **Spring Boot Starter:** <https://github.com/graphql-java-kickstart/graphql-spring-boot>

### GraphQL APIs für bestehende Datenbanken

- **GraphQL als ORM Ersatz (JavaScript, Go):**

<https://prisma.io/>

- **Instant GraphQL Schema für PostgresDB (Node.JS):**

<https://www.graphile.org/postgraphile/>

- **Instant GraphQL Schema für PostgresDB:**

<https://hasura.io/>