BEVOR ES LOSGEHT . . .

NPM & Node.js installiert?

Code Editor mit Syntax-Highlighting?

Laptop mit aktuellem Browser?

Projekt herunterladen

npm install

npm start

localhost:3000

localhost:3000/voyager

GraphQL Grundlagen

Dennis Dubbert
@ddubbert

Workshop 1

Workshop 2

01 GRAPHQL GRUNDLAGEN

GraphQL Grundpfeiler
Vergleich zu REST

02 GRAPHQL CLIENT

GraphQL Konzepte
Voyager
Playground
Queries

03 GRAPHQL SERVER

Serverelemente
GraphQL-Yoga
Typdefinitionen
Resolver

04 FRAGEN

Fragen und Diskussion

01 02 03 04

GRAPHQL GRUNDLAGEN

GraphQL Grundpfeiler
Vergleich zu REST

GRAPHQL CLIENT

GraphQL Konzepte
Voyager
Playground
Queries

GRAPHQL SERVER

Serverelemente
GraphQL-Yoga
Typdefinitionen
Resolver

FRAGEN

Fragen und Diskussion

Aufgaben eines Servers

1

Schema

Datenstrukturen festlegen und Schnittstelle bereitstellen

2

Request interpretieren

Anfragen dekonstruieren und benötigte Datenstrukturen / Attribute ermitteln

3

Resolver

Funktionen implementieren, welche angeforderte Daten bereitstellen

4

Response zusammenbauen

Angeforderte Daten zu einem Response zusammensetzen und übermitteln

GraphQL-Yoga

ist eine GraphQL-Server Library für Node.js von Prisma (release v1.0: 01.2018). Sie baut auf verschiedenen erprobten Bibliotheken wie express, apollo-server und graphql.js auf und erleichtert durch eine Abstraktion der Vorgänge den Einstieg in GraphQL, sowie den Aufbau eines GraphQL-Servers.

Aufgaben bei der Verwendung von GraphQL-Yoga

1

Schema

Datenstrukturen festlegen und Schnittstelle bereitstellen

Request[illegible]retieren

Anfrage[illegible]ruieren

Daten[illegible]ibute ermitteln

3

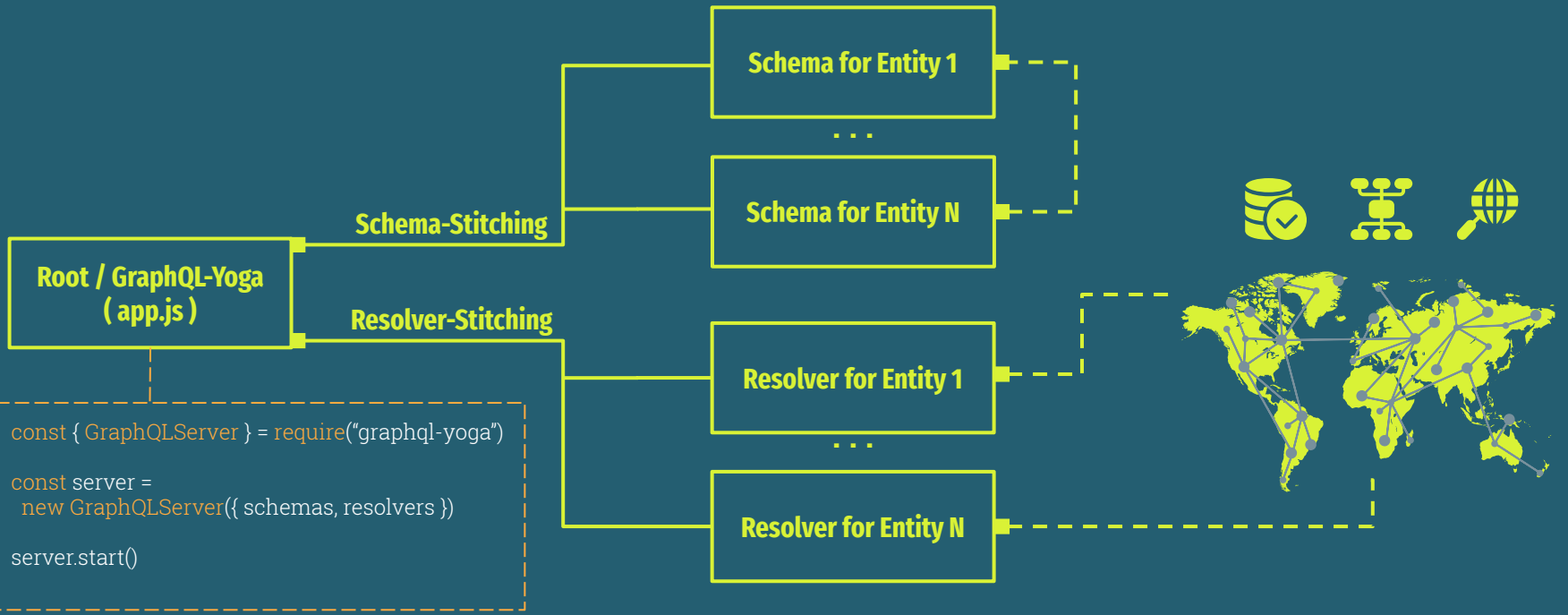
Resolver

Funktionen implementieren, welche angeforderte Daten bereitstellen

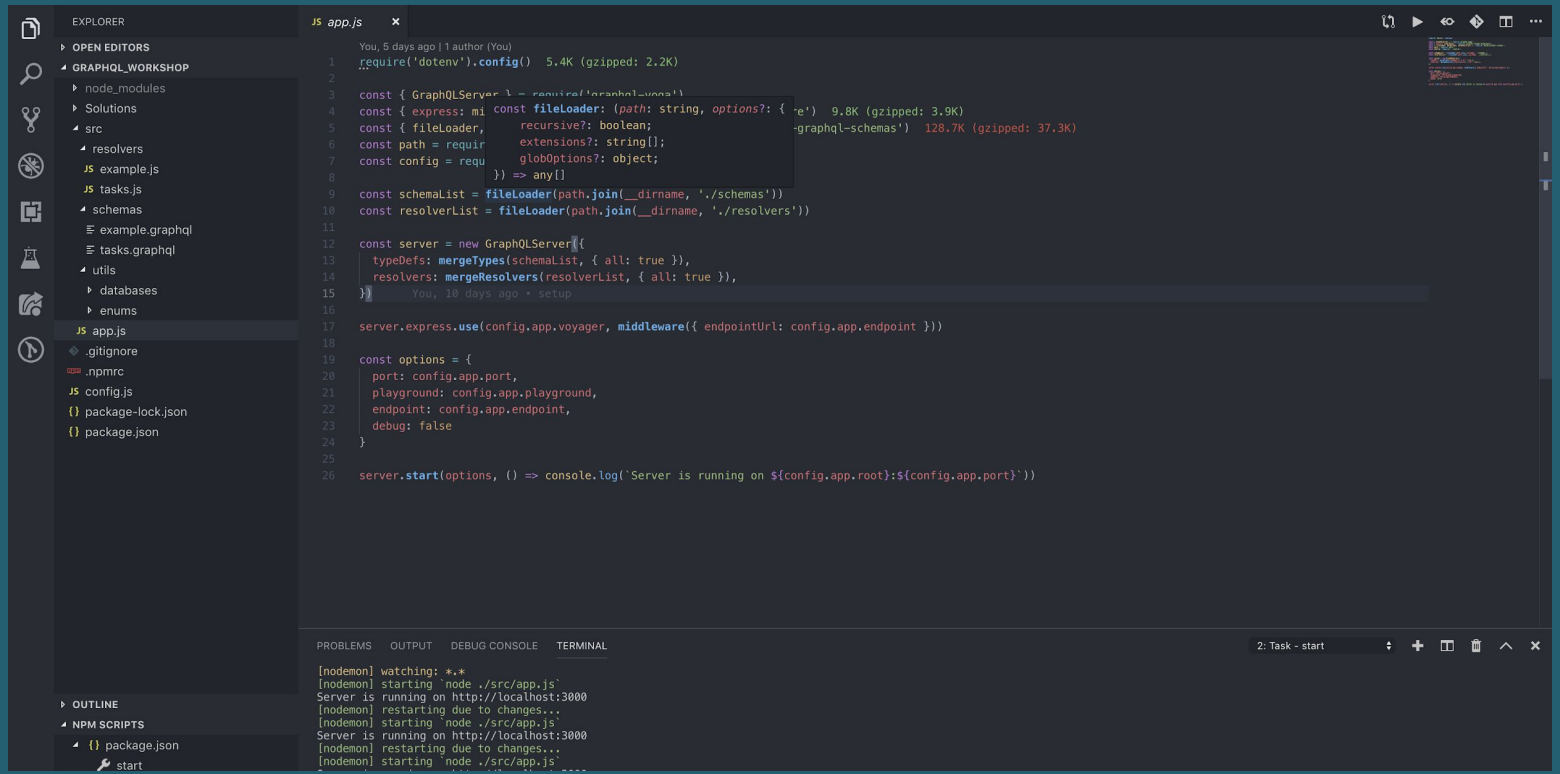
Response[illegible]menbauen

Angef[illegible]ten zu [illegible] zus[illegible]nd übermitteln

Typischer Aufbau eines Servers mit GraphQL-Yoga



Unsere Projektstruktur



Object Types definieren

SIMPLE OBJECT

Definition einer einfachen Entität, welche lediglich aus Skalartypen besteht.

```
type User {
  id: ID!
  username: String!
  email: String!
}
```

Query Types definieren

Queries

definieren den Zugriff auf Daten. Besitzen einen Namen und einen Rückgabewert. Können zudem Argumente entgegennehmen.

```
type Query {

    users : [User!]

    user(id: ID!) : User

}
```

Mutation Types definieren

Mutations

definieren die Erstellung und Bearbeitung von Daten. Besitzen einen Namen und einen Rückgabewert. Nehmen immer Argumente entgegen.

```
type Mutation {

        updateUser(id: ID!, name: String, email: String) : User

}
```

Input Types definieren

Input Types

ermöglichen die Auslagerung von Argumenten in einen eigenen Typen.

```
input UserUpdateInput {
        name: String
        email: String
}
```

```
type Mutation {
        updateUser(id: ID!, input: UserUpdateInput) : User
}
```

Query / Mutation Resolver definieren

```
{
    Query: {
        QueryName: (parent, args, context, info) => {
            const data = // angeforderte Daten besorgen
            return data
        }
    }
}
```

parent

Beinhaltet das Objekt, was von einem vorherigen Resolver zurückgegeben wurde.

(Wird auf Folien 18 + 19 genauer erklärt)

args

Beinhaltet alle, von dem anfragenden Client, übergebenen Argumente.

context

Beinhaltet geteilten internen state, welcher in jedem Resolver erreichbar ist.

info

Beinhaltet genaue Informationen über die Anfrage und dessen momentanen Status.

Query / Mutation Resolver definieren

TypeDef

```
type Query {

    user(id: ID!) : User

}
```

Resolvers

```
const userDB = require("../database/user.db")

module.exports = {
    Query: {
        user: (_parent, args, _context, _info) => {
            const { id } = args // const id = args.id
            return userDB.getUserById(id)
        }
    },
    Mutation: {
        ...
    }
}
```

Server Aufgabe 1

15 Minuten

- In der Datei "src/schemas/tasks.graphql" findet sich der Object-Type “Order”, welcher eine einzelne Bestellung eines Produktes darstellt. Erstellen sie in derselben Datei eine Query namens “orders”. Diese Query soll ein Array aller Bestellungen für einen Produzenten ausgeben oder null, wenn keine passende Bestellung gefunden wurde. Hierzu **muss** der Query die ID des Produzenten als Argument übergeben werden. Nennen sie dieses Argument “producerId”.

- Erstellen sie in der Datei "src/resolver/tasks.js" einen Resolver für die “orders”-Query. In diesem Resolver müssen sie zunächst das Argument entgegen nehmen. Rufen sie anschließend die getOrdersForProducer(producerId)-Funktion des orderDB-Objektes auf, um alle Bestellungen eines Produzenten zu erhalten. Nutzen sie dieses Array als Rückgabewert der Query. Sollte dieses Array jedoch leer sein, so geben sie stattdessen null zurück.

Object Types verschachteln

EXTENDED OBJECT

Definition einer Entität, welche neben einfachen Skalartypen auch andere Objekttypen beinhaltet.

```
type User {
    id: ID!
    username: String!
    email: String!
    company: Company
}

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

Object Types Resolver definieren

```
{
        TypeName: {
                AttributeName: (parent, args, context, info) => {
                        const { ParentAttribute } = parent
                        const data = // angeforderte Daten besorgen
                        return data
                }
        }
}
```

Benötigt wenn

Daten aus dem vorherigen Resolver nicht der Form der Schnittstelle entsprechen und weitere Berechnungen notwendig sind.

(Vorheriger Resolver = Query- oder Objekttyp-Resolver)

Nicht benötigt wenn

Daten aus dem vorherigen Resolver exakt der Schnittstelle entsprechen.

parent

Ausgabe des vorherigen Resolvers.

Wiederverwendung

Bei jedem Zugriff auf ein Attribut eines Typen wird dessen Resolver-Funktion aufgerufen. Durch die Verschachtelung und Wiederverwendung der Typen werden auch deren Resolver wiederverwendet.

Object Types Resolver definieren

TypeDef

```
type User {
        id: ID!
        username: String!
        email: String!
        company: Company
}
```

Database User

```
{
        id: "d467f50a",
        username: "peter-lustig",
        email: "peter@lustig.com",
        company: "371299b7"
}
```

Resolvers

```
module.exports = {
        Query: {
                user: (_parent, args, _context, _info) => {
                        const { id } = args
                        return userDB.getUserById(id)
                }
        },
        User: {
                company: (parent, _args, _context, _info) => {
                        const { company } = parent
                        return companyDB.getCompanyById(company)
                }
        }
}
```

Server Aufgabe 2

15 Minuten

- Da eine GraphQL-Schnittstelle von der Verschachtelung und Wiederverwendung der Objekt-Typen profitiert, sollte auch der Order-Type dies tun. Ändern sie im Schema den Typen des product-Attributes zu Product!, den Typen des producer-Attributes zu Producer! und den Typen des customer-Attributes zu User!.

- Die Datenbankstruktur der Bestellungen bietet lediglich ID's für diese geänderten Attribute, sodass die zuvor erstellte Query zu Fehlern führt. Schreiben sie also für die Eigenschaften product, producer und customer des Order-Typen eigene Attribut-Resolver. Greifen sie in jedem Resolver auf die entsprechenden Attribute des parent-Objektes zu und nutzen sie diese, um die vollständigen Entitäten anzufragen.

(An Produkte gelangen sie mithilfe der productDB.getProductById(productId)-Funktion. Producer und Customer können beide mithilfe der userDB.getUserById(userId)-Funktion angefragt werden)

Enum Types definieren

ENUM

Aufzählungstyp mit endlichen und fest definierten Ausprägungen.

```
enum Day {
        MONDAY
        TUESDAY
        WEDNESDAY
        THURSDAY
        FRIDAY
        SATURDAY
        SUNDAY
    }
```

Enum Types Resolver definieren

```
{
        TypeName: {
                Key_1: “Value_1”,
                Key_2: “Value_2”,
                Key_N: “Value_N”
        }
}
```

Key

Name der Ausprägung, welche im Schema definiert wurde.

Value

Value vom Typ String, welcher dem jeweiligen Key zugeteilt wird und diesen identifiziert.

Enum Types Resolver definieren

TypeDef

```
type User {
        id: ID!
        username: String!
        email: String!
        business_days: [Day!]
}
```

Database User

```
{
        id: "d467f50a",
        username: "peter-lustig",
        email: "peter@lustig.com",
        business_days:
          [ "monday", "tuesday" ]
}
```

Resolvers

```
module.exports = {
        Query: {
                ...
        },
        User: {
                ...
        },
        Day: {
                MONDAY: "monday",
                TUESDAY: "tuesday",
                WEDNESDAY: "wednesday",
                ...
        },
}
```

Interfaces definieren

```
interface User {
    id: ID!
    username: String!
    email: String!
}
```

```
type Producer  implements User {
    id: ID!
    username: String!
    email: String!
    business_days: [Day!]!
    products: [Product!]
    company: Company
}

type Consumer  implements User {
    id: ID!
    username: String!
    email: String!
    purchases: [Product!]
}
```

Union Types definieren

```
union TransferAccount =
    Paypal | Bank
```

```
type Paypal {
    email: String!
}

type Bank {
    account_number: String!
    bank_code: String!
    bank_name: String!
}
```

__resolveType Resolver definieren

```
{
        InterfaceName oder UnionName: {
                __resolveType: (object) => {
                        const typename = // Objekttyp herausfinden
                        return typeName
                }
        }
}
```

Unterscheidung

kann von unterschiedlichen Aspekten abhängig sein. Beispiele sind: vorhandene Attribute oder Attributausprägungen.

TypeName

ist der Name des Typen, wie er im Schema definiert ist, als String.

__resolveType Resolver definieren

Database User

```
{
        id: “d467f50a”,
        username: “peter-lustig”,
        email: “peter@lustig.com”,
        type: “producer”,
        business_days:
            [ “monday”, “tuesday” ],
        ...
}

{
        id: “d467f50a”,
        username: “peter-lustig”,
        email: “peter@lustig.com”,
        type: “consumer”,
        purchases: []
}
```

Resolvers

```
module.exports = {
        Query: {
                ...
        },
        User: {
                __resolveType: (user) => {
                        switch(user.type) {
                                case “producer”: return “Producer”,
                                case “consumer”: return “Consumer”,
                                default: throw Error(“Could not identify”)
                        }
                }
        },
}
```

Interfaces Resolver GraphQL-Yoga

Resolver werden bei GraphQL-Yoga noch **NICHT** von einem Interface an dessen Implementierungen vererbt und müssen somit für jede Sub-Entität selbst implementiert werden !

Resolvers

```
module.exports = {
        User: {
                __resolveType: (user) => {
                        ...
                }
        },
        Producer: {
                ...AlleResolverEinesUsers,
                ...AlleResolverEinesProducers
        },
        Consumer: {
                ...AlleResolverEinesUsers,
                ...AlleResolverEinesConsumers
        },
}
```

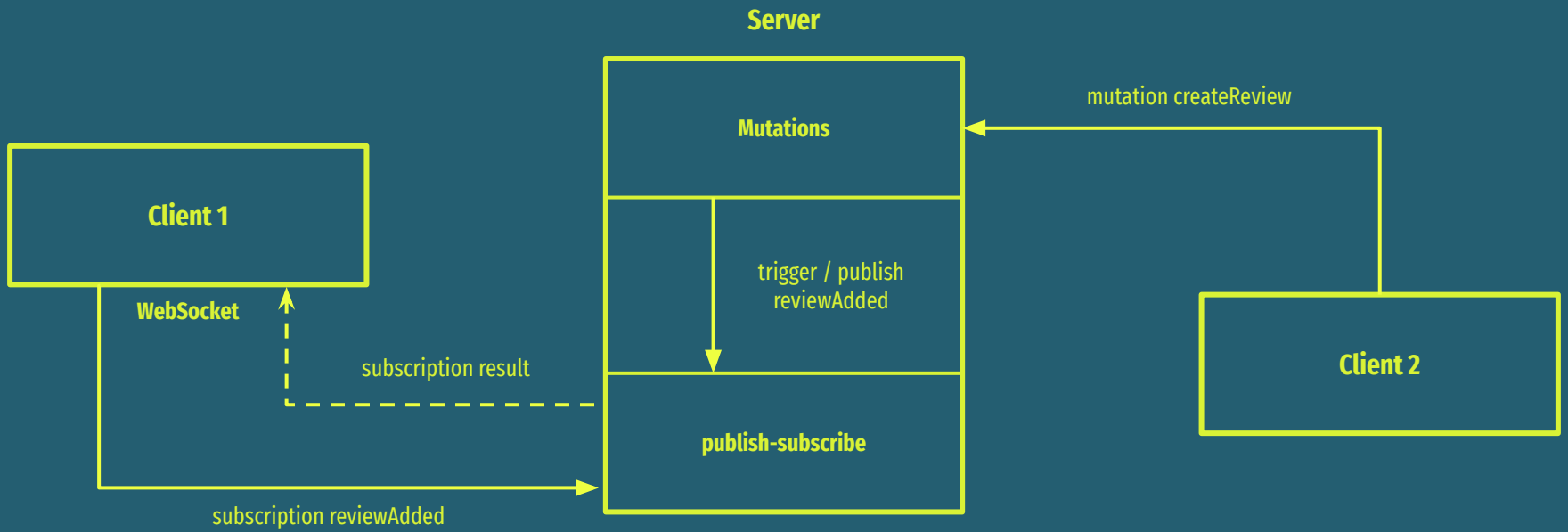
Server Aufgaben 3

15 Minuten

- Da eine Bestellung per Mail (Versand) andere zusätzliche Attribute benötigt, als jene bei einer Abholung (Pickup), bietet sich hier ein Interface an. Ändern sie im Schema den Order-Typen zu einem Interface und erstellen sie hierzu die zwei Sub-Typen MailOrder und PickupOrder. Eine MailOrder benötigt das zusätzliche Pflichtfeld shipping_address vom Typ Address!. Eine PickupOrder benötigt das zusätzliche Pflichtfeld pickup_date vom Typ DateTime!.

- Erstellen sie nun Resolver für die Typen MailOrder und PickupOrder. Da in GraphQL-Yoga keine Vererbung der Resolver stattfindet, müssen sie die zuvor geschriebenen Resolver des Order-Typen auf diese beiden Sub-Entitäten übertragen (copy-paste). Fügen sie nun dem Order-Typen einen Type-Resolver hinzu (__resolveType: (order) => { // TODO }), in welchem sie definieren, wann eine Bestellung eine MailOrder und wann sie eine PickupOrder ist. Hierzu können sie, wie in den Beispielen (siehe Cheatsheet), auf das type-Attribut zugreifen, dessen Ausprägung “mail” oder “pickup” sein kann.

Subscriptions



Subscription Type

```
type Subscription {
        reviewAdded(producerId: ID!) : Review!
}
```

```
subscription reviewSubscription {
        reviewAdded(producerId: "d467f50a") {
                comment
                rating
                creator { username }
                producer { username }
        }
}
```

Struktur und Aufruf im Playground ist identisch zu der einer Query oder Mutation.

Einfache Subscription

```
const { PubSub } = require("graphql-yoga")
const pubsub = new PubSub()

module.exports = {
        Subscription: {
                reviewAdded: {
                        subscribe: (parent, args, context, info) => {
                                return pubsub.asyncIterator("reviewAddedChannel")
                        }
                }
        },
}
```

Auswirkung

es werden alle neuen Nachrichten des spezifizierten Channels / Topics abonniert.

(hier der Channel mit dem Namen "reviewAddedChannel")

Wichtig

die pubsub-Instanz sollte lediglich einmal erstellt und an alle Resolver verteilt werden.

Hierfür bietet sich der context an.

Gefilterte Subscription

```
const { PubSub, withFilter } = require("graphql-yoga")
const pubsub = new PubSub()

module.exports = {
        Subscription: {
                reviewAdded:{
                        subscribe: withFilter(
                                (parent, args, context, info) => { // erste Funktion
                                        return pubsub.asyncIterator("reviewAddedChannel")
                                },
                                (payload, variables) => { // zweite Funktion
                                        return payload.reviewAdded.producerId === variables.producerId
                                }
                        )
                }
        },
}
```

Erste Funktion

dient weiterhin dem Abonnieren des gewünschten Channels / Topics.

Zweite Funktion

ermöglicht das Filtern von Nachrichten innerhalb des angegebenen Channels / Topics.

payload

stellt den Inhalt der Nachricht dar. (Inhalt befindet sich immer in dem Key mit dem selben Namen wie die Subscription)

variables

die Parameter, welche ein Client bei der Subscription übermittelt hat. (Wie args in den anderen Resolvern)

Subscription Anstoßen

```
const { PubSub} = require("graphql-yoga")
const pubsub = new PubSub()

module.exports = {
        Mutation: {
                createReview: (parent, args, context, info) => {
                        const review =  // review Erstellen

                        pubsub.publish("reviewAddedChannel", { reviewAdded: review })

                        return review
                }
        },
}
```

publish

nimmt zwei Argumente entgegen: den Channel-Namen und das Nachrichten-Objekt. Letzteres muss die, von den Subscribern gewünschte, Entität unter einem Key speichern, welcher denselben Namen wie die Subscription trägt.

01 GRAPHQL GRUNDLAGEN

GraphQL Grundpfeiler
Vergleich zu REST

02 GRAPHQL CLIENT

GraphQL Konzepte
Voyager
Playground
Queries

03 GRAPHQL SERVER

Serverelemente
GraphQL-Yoga
Typdefinitionen
Resolver

04 FRAGEN

Fragen und Diskussion

QUELLEN

- Bilder:
  - https://www.pexels.com/
  - https://www.graphql.com/
  - https://github.com/prisma/graphql-yoga
- https://goodapi.co/blog/rest-vs-graphql
- https://www.apollographql.com/docs/apollo-server/

Server Zusatzaufgaben

∞ Minuten

- Eine Bestellung kann per Post verschickt, oder beim Produzenten abgeholt werden. Ergänzen sie im Schema den Order-Typen um das Pflichtfeld “type: OrderType!”, welches Auskunft über die Art der Bestellung liefern soll. Erzeugen sie hierzu den Enum-Type OrderType mit den Ausprägungen “MAIL” und “PICKUP”.

- Erstellen sie einen Resolver für diesen Enum-Typen. Weisen sie der Ausprägung MAIL hierbei den String "mail" und der Ausprägung PICKUP den String "pickup" zu.