FACHHOCHSCHULE KUFSTEIN

STUDIENGANG: WEB BUSINNES & TECHNOLOGY

SEMINARARBEIT

WAS IST GRAPHQL UND WIE KÖNNEN APIS MIT GRAPHQL IN PHP/LARAVEL ERSTELLT WERDEN?

Vorgelegt von: Stuefer Lukas

Leitung: Stefan Huber

Inhaltsverzeichnis

1 Einleitung

Im Rahmen meines Studienganges Web Business & Technology an der Fachhochschule Kufstein besuche ich das Fach Serverseitige Software Entwicklung. Diese Arbeit trägt zur Vollständigkeit dieses Faches bei. Die Arbeit beschäftigt sich mit der Fragestellung was GraphQL ist und wie dieses Angewendet werden kann. Weiters wird aufgezeigt welche Unterschiede zwischen GraphQL und REST bestehen. Für das fertiggestellte Tutorial wird der Code auf Github (https://github.com/Luggist/graphQl) zur Verfügung gestellt.

2 GraphQL

GraphQL ist eine Abfragesprache und serverseitige „Runtime“ für APIs, die den Kunden nur diejenigen Daten zur Verfügung stellt, die sie wirklich brauchen (https://graphql.org, 2020).

GraphQL wurde entwickelt um API´s schneller, entwicklungsfreundlich und flexibler zu machen. Mit dieser Alternative zu REST können Entwickler einfacher Anfragen strukturieren und mit einem einzigen API-Aufruf Daten aus mehreren Quellen gleichzeitig abrufen.Außerdem können API-Maintainer mit GraphQL flexibel Felder hinzufügen oder entfernen, ohne dass dies bestehende Anfragen beeinträchtigt. Entwickler können APIs mit ihrer bevorzugten Methode erstellen, und die GraphQL-Spezifikation stellt sicher, dass die API für den Kunden auf vorhersehbare Weise funktioniert. (https://www.redhat.com/, 2020) Weiters ist zu erwähnen, dass GraphQL von Facebook entwickelt wurde.

2.1 Was ist eine API

API ist die Abkürzung von Application Programming Interface und bezeichnet somit eine Programmierschnittstelle. Diese Schnittstelle dient dazu Informationen zwischen einer Anwendung und einzelnen Programmteilen auszutauschen. Die Übergabe der Dateien erfolgt durch eine zuvor definierte Syntax.

2.2 Abfragesprache

Das Konzept von GraphQL ist eine Query Language den Programmen einen unkomplizierten Zugriff auf eine API ermöglichen soll. Während grundsätzlich andere Schnittstellen nur sehr strikte Abfragen ermöglichen, sticht GraphQl durch ein hohes Maß an Flexibilität hervor. Das zeigt sich darin, dass keine Limitierung für die Anzahl der Abgefragten Ressourcen gibt, und dass gezielt definiert werden kann, welche Datenfelder abgefragt werden sollen. GraphQL erlaubt dabei sowohl lesende als auch schreibende bzw. verändernde Abfragen. (https://www.ionos.at/, 2019)

2.3 Typsystem

GraphQL arbeitet mit einem eigenen Typsystem, das erlaubt API´s durch Datentypen zu beschreiben. Diese Datenstrukturen schaffen den eigentlichen Rahmen für die Abfragen. Jeder Typ besteht aus Feldern, die wiederum Typangaben enthalten. Dieses individuelle geschaffene System dient dann als Orientierungspunkt um Anfragen zu validieren und fehlerhafte Abfragen ablehnen zu können. (https://www.ionos.at/, 2019)

2.4 Laufzeitumgebung

GraphQL liefert verschiede Server-Laufzeitumgebungen zu der Ausführung der Query´s mit. Aus diesem Grund stehen mehrere Bibliotheken für viele Programmiersprachen wie PHP, JavaScript, Java, Go und noch mehr bereit. (https://www.ionos.at/, 2019)

2.5 Hierarchische Struktur

Datenbestände, die durch die API abrufbar sind, haben eine hierarchische Struktur. Es lassen sich automatisch Beziehungen zwischen einzelnen Projekten erzeugen und auf deren Basis auch Komplexe Anfragen in einem einzigen Request formulieren. Diese Struktur ist vor allem für graphenorientierte Datenbanken und Benutzer-Interfaces, welche ebenfalls hierarchisch aufgebaut sind, geeignet. (https://www.ionos.at/, 2019)

2.6 Starke Typisierung

Jede Ebene einer Abfrage entspricht einem bestimmten Typ. Wobei jeder Typ ein Set aus verfügbaren Feldern beschreibt. Dieses Typsystem, kann nicht wie bereits erwähnt, automatisch feststellen ob eine Abfrage korrekt oder fehlerhaft war, sondern ebenfalls, bereits während der Entwicklung bzw. vor dem Abschicken einer Abfrage, die Fehlermeldung ausspielen. (https://www.ionos.at/, 2019)

2.7 Flexibilität

GraphQL ermöglicht einfache flexible Abfragen zu erstellen und zu starten. Weiterhin vereinfacht GraphQL die Entwicklung und Anpassung der Schnittstellen, wobei das Entwicklerteam völlig unabhängig der anderen Teams ist. Weiters lassen sich sämtliche Änderungen und Erweiterungen der API´s ohne Versionierung durchführen. (https://www.ionos.at/, 2019)

3 GraphQL vs REST

Eine REST-API ist ein Architekturkonzept für netzwerkbasierte Software. GraphQL hingegen ist eine Abfragesprache, eine Spezifikation und ein Satz von Werkzeugen, der über einen einzigen Endpunkt unter Verwendung von HTTP arbeitet. Darüber hinaus wurde REST in den letzten Jahren zur Erstellung weiterer APIs verwendet, während der Schwerpunkt von GraphQL auf der Optimierung von Leistung und Flexibilität lag. Im unteren werden beide gegenübergestellt. (Russell, 2019)

3.1 REST-API

REST-API steht für „Representational State Transfer - Application Programming Interface“. Sie macht den Austausch von Informationen möglich, wenn diese sich auf unterschiedlichen Systemen befinden. Dank REST-API ist es möglich, Informationen und Aufgaben auf verschiedene Server zu verteilen und mit Hilfe eines HTTP-Requests anzufordern. Der HTTP-Request setzt sich aus dem Endpoint und den entsprechenden Parametern zusammen (https://de.ryte.com/, kein Datum).

3.2 Vorteile von GraphQL über REST

Wenn eine REST-API zum Abrufen von Informationen verwendet wird, erhält man immer einen vollständigen Datensatz zurück. Wenn man beispielsweise Informationen von zwei Objekten anfordern möchten, müssen zwei REST-API-Anforderungen durchgeführt werden. Der Vorteil der REST-APIs liegt in ihrer Einfachheit – Es gibt einen Endpunkt, der eine Aufgabe erledigt, so dass sie leicht zu verstehen und zu manipulieren ist. Mit anderen Worten: Wenn man einen X-Endpunkt hat, stellt dieser X-Daten bereit. Umgekehrt könnte man, wenn Informationen von einem bestimmten Endpunkt sammeln wollten, die Felder, die die REST-API zurückgibt, nicht einschränken; Man erhält immer einen vollständigen Datensatz. Dieses Phänomen wird als Over Fetching bezeichnet. GraphQL verwendet seine Abfragesprache, um die Anfrage genau auf das zuzuschneiden was benötigt wird. Von mehreren Objekten bis hinunter zu bestimmte Felder innerhalb jeder Entität. GraphQL würde den Endpunkt X nehmen, und damit einen Bestimmten Zweck erfüllen, der ihm jedoch zuerst gesagt werden muss. Im Wesentlichen ist GraphQL leistungsfähig, wenn man erst einmal weiß, wie man es benutzt. Da diese nur die Daten abrufen, die man benötigt, begrenzt dieser den Umfang der erforderlichen Verarbeitung. (Russell, 2019) (https://www.howtographql.com, kein Datum)

3.3 Vorteile von REST über GraphQL

REST ist zum Industriestandard für Unternehmen geworden, die APIs einsetzen. REST-Endpunkte sind ausgereift und gibt es schon seit einiger Zeit. Sogar "Spät-OEMs" haben jetzt REST-Endpunkte. Ebenso sind API-Analysen für REST aufgrund der begrenzten Anzahl von Werkzeugen für GraphQL einfacher zu erhalten. Es wird versprochen, dass es in naher Zukunft mehr Einblicke geben wird, sobald Tools, die GraphQL unterstützen, in größerem Umfang verfügbar sind. Hinzu kommt, dass im Gegensatz zu GraphQL, sich REST sich leichter lernen lässt. (Russell, 2019)

4 Tutorial

In diesem Tutorial wird erklärt, wie man eine API von GraphQL mithilfe von PHP und dem Framework Laravel erstellt. Dieses Tutorial baut auf der Anleitung von dem Blog Pusher auf. (https://blog.pusher.com, kein Datum) Diese Einführung benötigt Kenntnisse in den Bereichen Laravel und GraphQL

4.1 Start

Zuerst benötigen wir ein Startprojekt. Dies wird mit dem Befehl

```
PS C:\Users\Lukas\desktop> laravel new tutorial
```

erstellt. Um GraphQL in der Laravel-Anwendung zu verwenden, werden wir ein Paket namens rebing/laravel-graphql verwenden.

```
PS C:\Users\Lukas\desktop\tutorial-graphql> composer require rebing/graphql-laravel
```

Sobald die Installation abgeschlossen ist, wird folgender Befehl ausgeführt. Hiermit werden zwei neue Dateien erzeugt, welche uns erlaubt unsere Anwendung zu testen.

```
PS C:\Users\Lukas\desktop\tutorial-graphql> php artisan vendor:publish --provider=Rebing\GraphQL\GraphQLServiceProvider
```

4.2 Erstelle Task Model

Um das Task Model und deren Migrationsdatein zu erstellen, führen wir folgenden Befehl in unserer Datei aus:

```
PS C:\Users\Lukas\desktop\tutorial-graphql> php artisan make:model Task -m
```

Sobald dies erledigt ist, öffnen wir das Migrationfile und ändern die Up-Methode wie unten angeführt ab.

```php
public function up()
{
    Schema::create( table: 'tasks', function (Blueprint $table) {
        $table->increments( column: 'id');
        $table->string( column: 'title');
        $table->boolean( column: 'is_completed')->default( value: 0);
        $table->timestamps();
    });
}
```

Wir fügen der Aufgabentabelle grundlegende Spalten hinzu. Wir setzen die Spalte is_completed standardmäßig auf false.

Als nächstes müssen wir die Migrationen durchführen. Doch bevor wir das tun können, müssen wir unsere Datenbank einrichten. Für die Zwecke dieses Tutorials werden wir SQLite verwenden. Erstellen wir also eine database.sqlite-Datei innerhalb des Datenbankverzeichnisses. Zuletzt aktualisieren wir die .env-Datei wie unten beschrieben:

```
DB_CONNECTION=sqlite
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=C:\Users\Lukas\Desktop\tutorial-graphql\database\database.sqlite
DB_USERNAME=root
DB_PASSWORD=
```

Nun starten wir die Migration:

```
C:\Users\Lukas\Desktop\tutorial-graphql>php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table (0.06 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table (0.04 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated:  2019_08_19_000000_create_failed_jobs_table (0.02 seconds)
Migrating: 2020_07_31_093325_create_tasks_table
Migrated:  2020_07_31_093325_create_tasks_table (0.02 seconds)
```

4.3 Database seeder

Um einige Daten zur Verfügung zu haben, mit denen wir arbeiten können, erstellen wir einen Datenbank-Seeder, um die Aufgabentabelle mit einigen Daten zu füllen. Führen wir den folgenden Befehl aus:

```
C:\Users\Lukas\Desktop\tutorial-graphql> php artisan make:seeder TasksTableSeeder
Seeder created successfully.
```

Wir erstellen dann innerhalb des Verzeichnisses database/factories eine neue Datei TaskFactory.php und fügen den untenstehenden Code darin ein:

```php
<?php

/** @var \Illuminate\Database\Eloquent\Factory $factory */

use Faker\Generator as Faker;

$factory->define( class: App\Task::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
    ];
});
```

Wir geben nur den Wert der Titelspalte an, da wir die is_completed-Spalte bereits standardmäßig als falsch definiert haben.

Als nächstes aktualisieren wir die run-Methode von database/seeds/TasksTableSeeder.php wie unten beschrieben:

```
public function run()
{
    factory( class: App\Task::class,  amount: 5)->create();
}
```

Nun könnnen wir den database seeder starten:

```
C:\Users\Lukas\Desktop\tutorial-graphql>php artisan db:seed --class=TasksTableSeeder
Database seeding completed successfully.
```

4.4 Definieren von Schemas

Nachdem die Einrichtung abgeschlossen ist, sollten wir mit der Definition der Schemas für unsere Aufgabenliste beginnen. Schemas beschreiben, wie Daten geformt werden und welche Daten auf dem Server abgefragt werden können. Es gibt zwei Arten von Schemas: Abfrage und Mutation. Unsere Schemas können innerhalb von config/graphql.php definiert werden.

4.5 Erstellung der GraphQL query

Unsere To-Do-Liste wird nur eine einzige Anfrage enthalten, nämlich alle Aufgaben, die dem GraphQL-Server hinzugefügt wurden, abzurufen. Bevor wir eine Abfrage erstellen können, müssen wir zunächst einen Typ definieren. Wir definieren also einen Task-Typ. Dazu erstellen wir ein neues GraphQL-Verzeichnis innerhalb des app-Verzeichnisses. Erstellen wir innerhalb des GraphQL-Verzeichnisses ein neues Verzeichnis Type. Wir geben dem Typ einen Namen und eine Beschreibung. Außerdem definieren wir die Felder (id, title, is_completed), die der Aufgabentyp haben wird. Dann erstellen wir im Verzeichnis app/graphQL/Type eine neue Datei names TaskType.php und fügen folgenden Code hinzu:

```php
<?php

namespace App\GraphQL\Type;

use GraphQL\Type\Definition\Type;
use Folklore\GraphQL\Support\Type as GraphQLType;

class TaskType extends GraphQLType
{
    protected $attributes = [
        'name' => 'Task',
        'description' => 'A task'
    ];

    public function fields()
    {
        return [
            'id' => [
                'type' => Type::nonNull(Type::int()),
                'description' => 'The id of a task'
            ],
            'title' => [
                'type' => Type::string(),
                'description' => 'The title of a task'
            ],
            'is_completed' => [
                'type' => Type::boolean(),
                'description' => 'The status of a task'
            ],
        ];
    }
}
```

Nun lassen wir uns die Abfrage erstellen. Erstellen wir ein neues Query-Verzeichnis innerhalb von app/GraphQL, sowie innerhalb des Query-Verzeichnisses eine neue Datei Tasks-Query.php und geben wir den folgenden Code darin ein:

```php
'schemas' => [
    'default' => [
        'query' => [
            'tasks' => \App\GraphQL\Query\TasksQuery::class,
        ],
        // ...
    ]
],

'types' => [
    'Task' => \App\GraphQL\Type\TaskType::class,
],
```

Wir geben der Abfrage einen Namen von Aufgaben, dann definieren wir den Abfragetyp (das ist der oben definierte Aufgabentyp). Da die Abfrage alle erstellten Aufgaben abrufen soll, verwenden wir den Typ listOf für den Aufgabentyp, der eine Reihe von Aufgaben zurückgibt. Schließlich definieren wir eine Auflösungsmethode, die das tatsächliche Abrufen der Aufgaben durchführt. Wir geben einfach alle Tasks aus der Datenbank zurück. Wenn unser Typ und unsere Abfrage definiert sind, fügen wir sie zu config/graphql.php hinzu:

```
'schemas' => [
    'default' => [
        // ...
        'mutation' => [
            'newTask' => \App\GraphQL\Mutation\NewTaskMutation::class,
            'updateTaskStatus' => \App\GraphQL\Mutation\UpdateTaskStatusMutation::class,
        ]
    ]
],


'types' => [
    'Task' => \App\GraphQL\Type\TaskType::class,
],
```

4.6 Creating GraphQL mutations

Mutationen werden verwendet, um Schreiboperationen durchzuführen. Wir werden zwei Mutationen erstellen, die dazu verwendet werden, eine neue Aufgabe hinzuzufügen bzw. den Status einer Aufgabe zu aktualisieren. Wir erstellen ein neues Mutationsverzeichnis innerhalb von app/GraphQL, sowie innerhalb von der Mutation eine neue Datei NewTaskMutation.php und geben den folgenden Code darin ein:

```php
<?php /** @noinspection ALL */

namespace App\GraphQL\Mutation;
use GraphQL\Type\Definition\Type;
use Rebing\GraphQL\Support\Facades\GraphQL;
use Rebing\GraphQL\Support\Mutation;
use App\User;
class NewUserMutation extends Mutation
{
    protected $attributes = [
        'name' => 'NewUser'
    ];
    public function type()
    {
        return GraphQL::type( name: 'users');
    }
    public function args()
    {
        return [
            'name' => [
                'name' => 'name',
                'type' => Type::nonNull(Type::string())
            ],
            'email' => [
                'name' => 'email',
                'type' => Type::nonNull(Type::string())
            ],
            'password' => [
                'name' => 'password',
                'type' => Type::nonNull(Type::string())
            ]
        ];
    }
    public function resolve($root, $args) {
        $args['password'] = bcrypt($args['password']);
        $user = User::create($args);
        if (!$user) {
            return null;
        }
        return $user;
    }
}
```

Wir geben der Mutation den Namen newTask, dann definieren wir den Typ, den diese Mutation zurückgeben wird. Wir definieren eine args-Methode, die ein Array von Argumenten zurückgibt, die die Mutation akzeptieren kann. Die Mutation akzeptiert nur ein Argument, nämlich den Titel einer Aufgabe. Zuletzt definieren wir eine Auflösungsmethode, die das tatsächliche Einfügen der neuen Aufgabe in die Datenbank durchführt und die neu erstellte Aufgabe zurückgibt.

Was ist, wenn wir die Daten validieren wollen, bevor wir eine neue Aufgabe hinzufügen? Nun, es ist möglich, Validierungsregeln zu einer Mutation hinzuzufügen. Das laravel-graphql-Paket verwendet den Laravel Validator, um die Validierung gegen die Args durchzuführen. Es gibt zwei Möglichkeiten, Mutationen eine Validierung hinzuzufügen: Wir können eine Regelmethode definieren und ein Array mit den Regeln für jedes Argument zurückgeben, oder wir definieren die Regeln direkt beim Definieren eines Arguments. Wir werden uns für die zweite Möglichkeit entscheiden.

Aktualisieren wir die args-Methode wie unten beschrieben.:

```php
public function args()
{
    return [
        'title' => [
            'name' => 'title',
            'type' => Type::nonNull(Type::string()),
            'rules' => ['required'],
        ]
    ];
}
```

Wir fügen eine Regel hinzu, die das Titelargument erforderlich macht.

Lassen uns nun die Mutation erstellen, um den Status einer Aufgabe zu aktualisieren. Erstellen wir eine neue Datei UpdateTaskStatusMutation.php innerhalb von app/GraphQL/Mutation und fügen den folgenden Code darin ein:

```php
<?php /** @noinspection ALL */

namespace App\GraphQL\Mutation;
use GraphQL;
use GraphQL\Type\Definition\Type;
use Rebing\GraphQL\Support\Mutation;
use App\Task;

class UpdateTaskStatusMutation extends Mutation
{
    protected $attributes = [
        'name' => 'updateTaskStatus'
    ];

    public function type()
    {
        return GraphQL::type('Task');
    }

    public function args()
    {
        return [
            'id' => [
                'name' => 'id',
                'type' => Type::nonNull(Type::int()),
                'rules' => ['required'],
            ],
            'status' => [
                'name' => 'status',
                'type' => Type::nonNull(Type::boolean()),
                'rules' => ['required'],
            ]
        ];
    }

    public function resolve($root, $args)
    {
        $task = Task::find($args['id']);

        if (!$task) {
            return null;
        }

        $task->is_completed = $args['status'];
        $task->save();

        return $task;
    }
}
```

Dies ist der NewTaskMutation recht ähnlich. Sie akzeptiert zwei Argumente: die ID der zu aktualisierenden Aufgabe und den Status der Aufgabe. Die Auflösungsmethode holt den Task mit der gelieferten ID aus der Datenbank. Wenn es keine Aufgabe mit der angegebenen ID gibt, geben wir einfach null zurück. Andernfalls aktualisieren wir die Aufgabe mit dem angegebenen Status und verbleiben in der Datenbank. Schließlich geben wir die Aufgabe zurück, um sie zu aktualisieren.

Um die Mutationen abzuschließen, fügen wir sie der Datei config/graphql.php:date der Aufgabe hinzu.

```
'schemas' => [
    'default' => [
        // ...
        'mutation' => [
            'newTask' => \App\GraphQL\Mutation\NewTaskMutation::class,
            'updateTaskStatus' => \App\GraphQL\Mutation\UpdateTaskStatusMutation::class,
        ]
    ]
],
```

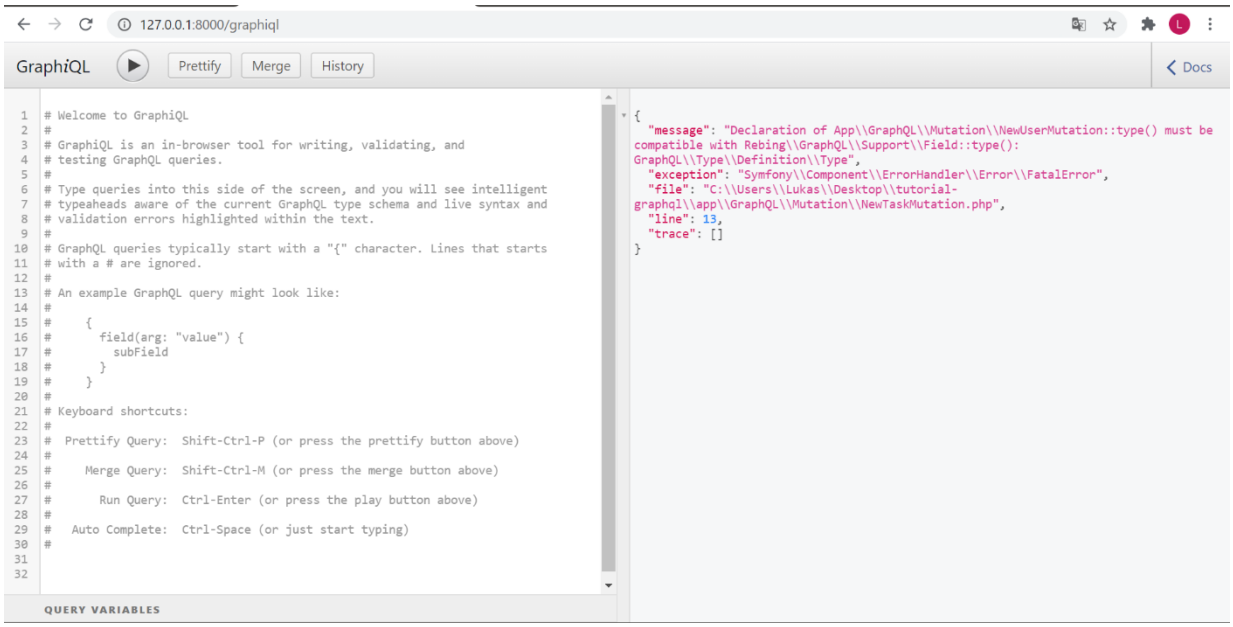
4.7 Testing mit GraphiQL

Bislang haben wir einen GraphQL-Server mit einer Abfrage und zwei Mutationen erstellt. Sehen wir uns nun an, wie wir den Server mit GraphiQL austesten. Erinnern wir uns, dass bei der Veröffentlichung der laravel-graphql-Paketdateien auch eine resources/views/vendor/graphql/graphiql.php erstellt wurde. Dies wird es uns ermöglichen, unseren GraphQL-Server innerhalb unserer Laravel-Anwendung zu testen.

Starten wir die Anwendung mit dem folgenden Befehl:

```
C:\Users\Lukas\Desktop\tutorial-graphql>php artisan serve
Laravel development server started: http://127.0.0.1:8000
```

Dadurch wird die Anwendung auf http://127.0.0.1:8000 zum Laufen gebracht, dann besuchen wir http://127.0.0.1:8000/graphiql in dem Browser. GraphiQL sollte nun wie in der Abbildung unten laufen:



(Leider kam hier ein Fehler, den ich nicht beheben konnte)

5 Fazit

Ziel dieser Arbeit war das Kennenlernen der GraphQL API. Zusammenfassend lässt sich sagen das GraphQL einige Vorteile gegenüber der REST Schnittstelle hat, obwohl erstere ein wenig komplizierter zu lernen ist. Da ich diese API noch nie verwendet hatte war es interessant und aufschlussreich.

6 Literaturverzeichnis

https://blog.pusher.com. (kein Datum). Von https://blog.pusher.com/graphql-laravel/ abgerufen

https://de.ryte.com/. (kein Datum). Von https://de.ryte.com/wiki/REST-API abgerufen

https://graphql.org. (2020). Von https://graphql.org abgerufen

https://www.howtographql.com. (kein Datum). Von https://www.howtographql.com/basics/1-graphql-is-the-better-rest/ abgerufen

https://www.ionos.at/. (03. 09 2019). Von https://www.ionos.at/digitalguide/websites/web-entwicklung/graphql/ abgerufen

https://www.redhat.com/. (2020). Von https://www.redhat.com/de/topics/api/what-is-graphql abgerufen

Russell, D. (21. 11 2019). https://www.rubrik.com. Von https://www.rubrik.com/blog/graphql-vs-rest-apis/#:~:text=The%20Core%20Difference%20Between%20REST,a%20single%20endpoint%20using%20HTTP. abgerufen