Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”

Politechnika Wrocławska

Wydział Informatyki i Zarządzania

Zaawansowane Technologie Webowe

Laboratorium

Temat:	GraphQL czyli REST API nowej generacji
Opracował:	mgr inż. Piotr Jóźwiak
Data:	lipiec 2020
Liczba godzin:	2 godziny

Table of Contents

Wstęp

W tym laboratorium postaramy się odpowiedzieć czym jest **GraphQL** oraz przyjrzymy się jego najważniejszym koncepcjom.

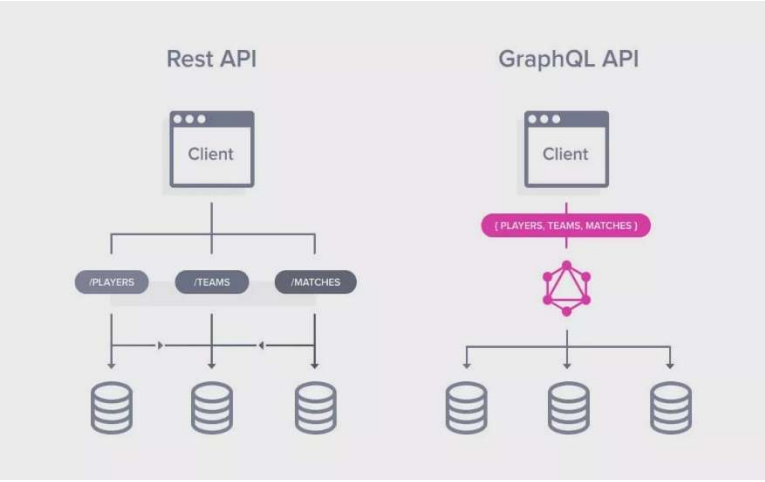
Zatem czym jest **GraphQL**? Jest to ***query language*** wykorzystywany do obsługi API. Oferuje on dużo większą elastyczność w porównaniu do **REST API**. W dużym uproszczeniu można powiedzieć, że GraphQL jest językiem odpytywania o dane.

Przyczyną jego popularyzacji są ograniczenia technologii REST API, które utrudniają rozwój aplikacji, głównie tych wieloplatformowych. Przyjrzyjmy się tym ograniczeniom na przykładzie. Wyobraźmy sobie, że mamy pewien serwer REST API, który zwraca np. poniższy wynik dla zapytania o szczegóły pracownika naukowego:

```
{
    "id": 67,
    "name": "Jan Kowalski",
    "title":"dr inż.",
    "org": "W8 K7",
    "email": "kan.kowalski@pwr.edu.pl",
    "publications": [/* Tutaj tablica z id publikacjami*/]
}
```

Załóżmy, że w wersji desktopowej aplikacji, wszystkie powyższe dane są wykorzystywane. Ale już w aplikacji na smartphone potrzebne są nam tylko {id, name, email}. Naturalnie REST API nie ma możliwości przesłania nam tylko tych informacji, które w rzeczywistości będziemy potrzebować. Zatem otrzymamy więcej niż potrzeba, co jest marnotrawieniem zasobów. Oczywiście można by powiedzieć, że to pomijalna ilość. Zastanówmy się zatem, jak wyglądałaby komunikacja, gdyby oprócz danych pracownika potrzeba by było mieć dodatkowe szczegóły dla każdej jego publikacji? W takiej sytuacji zazwyczaj musimy dodatkowo odpytać serwer pojedynczo o każdą publikację oddzielnie. A to już nie jest całkiem pomijalna operacja.

GraphQL przychodzi z pomocą właśnie w powyższych sytuacjach. Odpowiednio zbudowane zapytanie graphql w pojedynczym żądaniu do endpoint'a otrzyma informacje o pracowniku wraz z doczytanymi szczegółami wszystkich jego publikacji. Jakby tego było mało, to graphql umożliwia także rozbudowane filtrowanie, sortowanie i przetwarzanie danych przed ich wysłaniem do klienta. Najlepiej obrazuje to poniższy diagram:



Jak to działa?

Powstaje pytanie jak to wszystko działa. Aby odpowiedzieć na nie, musimy omówić trzy podstawowe koncepcje GraphQL.

Pierwszą z nich są **Queries**. Komunikacja w GraphQL opiera się o zapytania. Zapytanie definiujemy z pomocą słowa kluczowego query oraz nazwy poszczególnych właściwości zasobów. W odpowiedzi otrzymujemy obiekt JSON o takiej samej strukturze jak zapytanie, ale z wypełnionymi danymi. Przykładowe zapytanie GraphQL dla przykładu z pracownikiem, w którym chcielibyśmy także otrzymać dodatkowe informacje o jego publikacjach wygląda następująco:

```
{
    Employee {
        id
        name
        title
        email
        publications {
            id
            year
            title
        }
    }
}
```

Kolejnym ważnym elementem są **Resolvery**. GraphQL nie potrafi sam wydedukować, gdzie znajdują się dane, o które pytamy. To właśnie resolvery, czyli pewnego rodzaju funkcje wskazują źródła danych dla GraphQL. Jeśli w danym zapytaniu pytamy o listę pracowników, to resolver wie skąd wziąć te dane.

Ostatnim elementem jest **Schema**. Koncept ten służy do opisu struktury oraz typów danych, które serwer będzie w stanie obsłużyć. Do opisu służy język SDL (Schema Definition Language). Schema, umożliwia także dobre udokumentowanie API serwera, dlatego nie musimy tego wykonywać ręcznie.

Instalacja środowiska

Do przygotowania środowiska wykorzystamy bibliotekę **`graphql-yoga`** uruchomioną na prostej aplikacji **NodeJs**. Aplikacja ta posłuży nam do przygotowania serwera GraphQL, który z powodzeniem może zostać rozwinięty do pełnoprawnego środowiska backendowego.

Do przygotowanie środowiska skorzystamy z managera pakietów **`npm`**. Wykonajmy poniższe polecenia celem utworzenia folderu na aplikację:

```
mkdir first-graphql-app
cd first-graphql-app
```

Następnie generujemy plik **`package.json`** z pomocą **`npm`**:

```
npm init
```

Na wszystkie pytania można odpowiedzieć pozostawiając domyślne propozycje. Ten plik będziemy jeszcze modyfikować później.

Następnie instalujemy niezbędne biblioteki. Na ten moment wystarczą nam dwie: **`graphql-yoga`** oraz **`nodemon`**. Pierwsza z nich jest biblioteką (jedną z wielu) wspomagającą implementowanie serwera *graphql*. Druga biblioteka służy do uruchamiania aplikacji **NodeJS**.

```
npm install graphql-yoga nodemon
```

Utwórzmy jeszcze folder na pliki źródłowe:

```
mkdir src
```

Teraz możemy już resztę aplikacji pisać w ulubionym środowisku IDE. Polecam np. Visual Studio Code.

Zanim przejdziemy do omawiania sposobu implementowania środowiska GraphQL przygotujmy prosty przykład, aby sprawdzić, że środowisko jest w pełni funkcjonalne. W pierwszej kolejności utwórzmy plik **`src/schema.graphql`**. Czym ten plik jest, omówimy w dalszej kolejności, na tą chwilę bezrefleksyjnie wprowadźmy jego zawartość w postaci:

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

Następnie utwórzmy główny plik aplikacji **`src/App.js`**, którego zawartość powinna wyglądać następująco:

```
const { GraphQLServer } = require('graphql-yoga');

const resolvers = {
    Query: {
      demo: () => 'Witaj, GraphQL działa!',
    },
  }

  const server = new GraphQLServer({
    typeDefs: './src/schema.graphql',
    resolvers,
  });

  server.start(() => console.log(`Server is running on http://localhost:4000`));
```

W przedstawionym powyżej przykładzie na początku importujemy bibliotekę **graphql-yoga**. W dalszej kolejności definiujemy obiekt nazwany **resolvers**. Czym on jest i za co odpowiada będziemy dokładniej omawiać w dalszej części laboratorium. Następnie tworzymy obiekt serwera **GrapgQLServer**, któremu przekazujemy **typeDefs** jako ścieżka do wcześniej utworzonego pliku z schemą oraz obiekt **resolvers**. Na koniec startujemy sam serwer.

Aby to wszystko mogło działać musimy jeszcze zdefiniować jak uruchomić naszą aplikację. W tym celu musimy wyedytować plik **package.json**, w którym w sekcji **scripts** należy dodać definicję **start** według poniższego przykładu:

```
{
    //...
    "scripts": {
      "start": " nodemon -e js,graphql src/App.js",

      "test": "echo \"Error: no test specified\" && exit 1"
    },
    //...
}
```

Teraz możemy już uruchomić aplikację wydając poniższe polecenie będąc wewnątrz folderu z plikiem **package.json**:

```
npm start
```

Jeśli wszystko uruchomi się poprawnie ukaże się nam odpowiedni komunikat o wystartowanym serwerze:

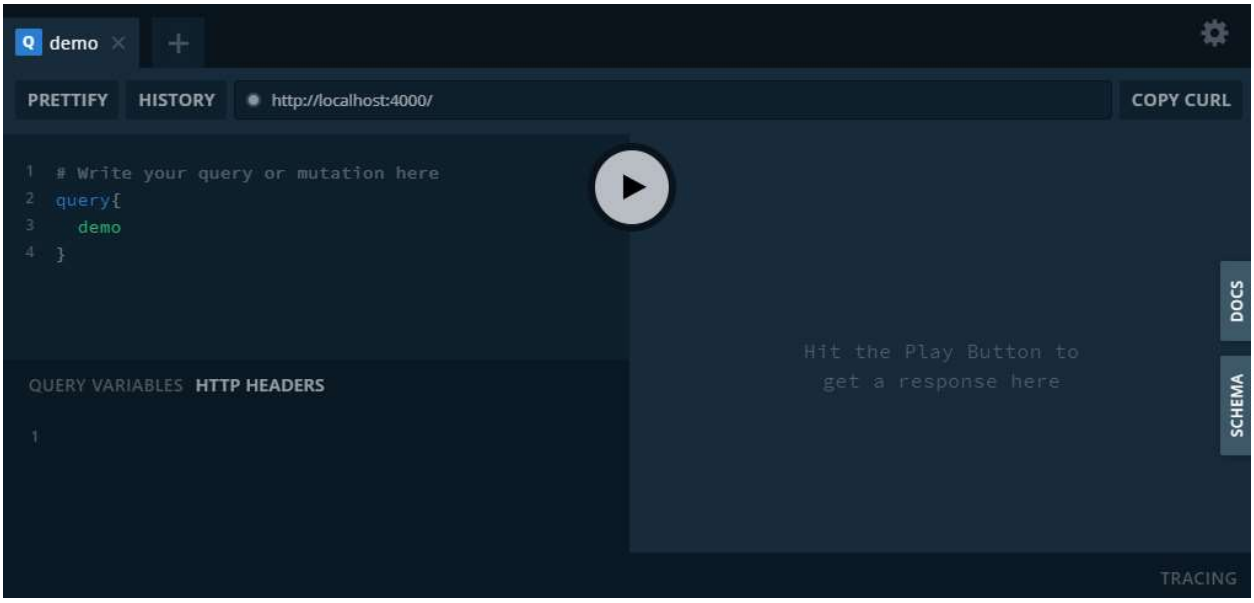
```
C:\Prv\OneDrive\Dokumenty\PWr\Dydaktyka\ZTW\ProjektPower\Lab7\first-graphql-app>npm start

> first-graphql-app@1.0.0 start C:\Prv\OneDrive\Dokumenty\PWr\Dydaktyka\ZTW\ProjektPower\Lab7\first-graphql-app
> nodemon src/App.js

[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/App.js`
Server is running on http://localhost:4000
```

GraphQL Playground

Pora sprawdzić co się uruchomi pod wskazanym adresem. Zatem przejdźmy do przeglądarki i by uruchomić stronę http://localhost:4000. Pod wskazanym adresem uruchomi się narzędzie GraphQL playground:



Jest to wbudowana w bibliotekę funkcjonalność umożliwiająca uruchamianie zapytań graphql oraz przeglądanie dokumentacji. Zapytania wprowadza się w pole z lewej strony. Wyniki otrzymujemy po wciśnięciu przycisku Play po prawej stronie. Przetestujmy działanie. Następnie należyuruchomić zapytanie w postaci:

```
query{
    demo
  }
```

Otrzymamy w wyniku odpowiedź z serwera:

```
{
    "data": {
      "demo": "Witaj, GraphQL działa!"
    }
  }
```

Wprawny obserwator zapewne zauważy, gdzie jest definicja powyższego wyniku.

Zwróćmy uwagę, że po prawej stronie jest zakładka DOCS. Przyjrzyjmy się jej zawartości. Jest to miejsce, w którym będzie wyświetlana dokumentacja dla zdefiniowanych API.

Schema – opis API

Na początku pracy z GraphQL należy zaprojektować schematy, które definiują API serwera. Służy do tego język SDL (*Schema Definition Language*) lub IDL (Interface Definition Language). GraphQL jest językiem silnie typowanym umożliwiającym współpracę w połączeniu z dowolnym językiem programowania.

Podstawowym komponentem schematu są **Object Types**, reprezentujące obiekty oraz jego właściwości. Są to elementy takie jak **`Query`**, **`Mutation`** oraz **`Subscriptions`**. Te trzy elementy tworzą tzw. **`Root Types`** w schemacie. Służą one do odpowiednio: pobierania, modyfikowania oraz obserwowania danych.

Definicję schematów można wydzielić do oddzielnych plików, zazwyczaj z rozszerzeniem **`*.graphql`**. W naszym przykładzie jeden taki plik został już utworzony (**`src/schema.graphql`**).

Definicja typu

Najbardziej podstawowym komponentem schemy są obiekty **`type`**, które reprezentują opis obiektu, który można pobrać z serwisu. Opisują one dostępne pola tego obiektu wraz z jego typem. Gdybyśmy mieli zdefiniować schemat pracownika naukowego to wyglądałby on tak:

```
type Empleyee {
    id: ID!
    name: String!
    title: String!
    email: String!
    contractor: Boolean!
    age: Int!
    publications: [Article!]!
}
```

Sam język jest bardzo czytelny, jednak przyjrzyjmy się poszczególnym elementom powyższej definicji.

- **`Employee`** jest Object Type'em, co oznacza, że jest definicją typu posiadającą wyszczególnione pola.
- **`Id`**, **`name`**, **`title`** oraz kolejne element są polami opisującymi obiekt **`Employee`**. Oznacz to tyle, że za każdym razem gdy zażądany zostanie obiekt **`Employee`**, to dla niego można zażądać wyszczególnionych w tym obiekcie informacji.
- **`String`**, **`Boolean`**, **`Int`**, **`Float`** są wbudowanymi typami danych.
- **`String!`**, **`Int!`**, w ogólności znak wykrzyknika na końcu typu danych oznacza, że pole jest nie nullowe (non-nullable). Zatem zawsze będzie tutaj jakaś wartość.
- **`[Article!]!`** oznacza tablicę obiektów typu **`Article`**.

Zagadnienie definiowania Schemy w GraphQL jest bardzo obszerne. Tym samym wykracza poza możliwości pełnego opisu w tym miejscu. Dlatego chciałbym przedstawić jeszcze tylko jedno zagadnienie jakim są argumenty, które będziemy wykorzystywać w dalszych przykładach. Pełen opis funkcjonalności Schematów do typów można odnaleźć tutaj: https://graphql.org/learn/schema/.

Każde pole w Schemie może definiować listę argumentów. Przypomina to w pewnym sensie funkcję. Argumenty są wykorzystywane do przekazania parametrów query. Parametr może służyć do filtrowania, szukania czy określenia kolejności sortowania. Argumenty wykorzystywane są także do mutacji, których zadaniem jest np. dodanie czy edycja obiektów meta-modelowych. Przyjrzyjmy się poniższej definicji, aby przybliżyć działanie argumentów w praktyce. Załóżmy, że chcemy dla typu **`Employee`** zdefiniować dwa zapytania. Jedno zwracając wszystkich pracowników naukowych oraz drugie, parametryzowane, które zwróci konkretnego pracownika o zadanym **`id`**:

```
type Query {
  employees: [Empleyee!]
  employee(id: ID!): Empleyee
```

```
}
```

Argumenty definiuje się wewnątrz nawiasów okrągłych, podając ich nazwę oraz typ.

Resolvery

Resolver jest niczym innym jak metodą zwracającą wartość dla danego typu lub pola. Resolver pełni funkcję łączenia definicji schematów ze sposobem pozyskania danych ze źródła, np. bazy danych czy seriwsu REST. Aby dobrze zrozumieć jak działają Resolvery musimy zrozumieć jak działają zapytania GraphQL. Każde zapytanie przechodzi przez trzy fazy: **Parsowanie**, **Walidacja** oraz **Wykonanie**.

Parsowanie jest przedstawieniem zapytania w formie drzewa AST. Aby zobaczyć jak wygląda takie drzewo można skorzystać z narzędzia dostępnego pod adresem: https://astexplorer.net. Wystarczy wkleić przykładowe zapytanie oraz wybrać graphql jako język.

Walidacja sprawdza czy drzewo AST jest zgodne z definicją schematu.

Ostatnia faza polega na wykonaniu, poprzez uruchomienie niezbędnych metod/resolverów i zbudowania odpowiedzi. Metody uruchamiane są od szczytu drzewa AST.

Taki resolver już definiowaliśmy w naszym przykładzie. Wystarczy zerknąć do pliku **`src/App.js`** i odszukać definicji resolvera dla query demo. Jednakże, aby lepiej zobrazować działanie GraphQL przyjrzyjmy się kompletnemu rozwiązaniu.

Implementacja serwera graphQL na przykładzie listy TODO

Poniższy przykład przedstawia sposób implementacji GraphQL na przykładzie prostej funkcjonalności związanej z funkcjonalnością listy spraw do wykonania – powszechnie nazywanych z języka angielskiego: *to do*.

W pierwszej kolejności zdefiniujmy schemat pojedynczego zadania do wykonania. W naszym przypadku będzie wyglądał w następujący sposób. Dla przypomnienia definicję schematów będziemy umieszczać w pliku **`schema.graphql`**:

```
type ToDoItem{
    id: ID!
    title: String!
    completed: Boolean!
    user: User!
}
```

Jak widać w powyższej definicji **`ToDoItem`** posiada także pole **`user`** typu **`User`**. Jest to wskazanie na użytkownika, którego dotyczy dane zadanie. Definicja schematu użytkownika przyjmuje postać:

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

```
    login: String!
    todos: [ToDoItem!]!
}
```

Warto tutaj zauważyć, że użytkownik posiada pole **todos**, które jest kolekcją **ToDoItems**.

Musimy jeszcze zdefiniować zapytania jakie nasz serwer GraphQL będzie obsługiwać. Zaimplementujemy cztery poniższe zapytania:

```
type Query {
    todos: [ToDoItem!]
    todo(id: ID!): ToDoItem
    users: [User!]
    user(id: ID!): User
}
```

Oprócz kolekcji wszystkich użytkowników oraz rzeczy do wykonania definiujemy także zapytanie o pojedynczego użytkownika oraz **todo** na podstawie ich **id**.

Pora na napisanie resolwerów. W naszym podstawowym przykładzie resolwery będą działały na podstawie tablic w JavaScript. Później zmienimy je na prawdziwe źródło danych. Ale, aby łatwiej zrozumieć, jak to działa nie będziemy teraz zaciemniać przykładu tym zagadnieniem.

W pliku **src/App.js** definiujemy tablice ze źródłami danych dla GraphQL. Przyjmują one postać:

```
const usersList = [
    { id: 1, name: "Jan Konieczny", email: "jan.konieczny@wonet.pl", login: "jkon
ieczny" },
    { id: 2, name: "Anna Wesołowska", email: "anna.w@sad.gov.pl", login: "anna.we
solowska" },
    { id: 3, name: "Piotr Waleczny", email: "piotr.waleczny@gp.pl", login: "p.wal
eczny" }
];

const todosList = [
    { id: 1, title: "Naprawić samochód", completed: false, user_id: 3 },
    { id: 2, title: "Posprzątać garaż",  completed: true,  user_id: 3 },
    { id: 3, title: "Napisać e-mail",    completed: false, user_id: 3 },
    { id: 4, title: "Odebrać buty",      completed: false, user_id: 2 },
    { id: 5, title: "Wysłać paczkę",     completed: true,  user_id: 2 },
    { id: 6, title: "Zamówic kuriera",   completed: false, user_id: 3 },
];
```

Następnie musimy napisać resolwer, który skorzysta z tych danych. W tym celu zmieniamy definicję resolevera na poniższą:

```
const resolvers = {
    Query: {
        users: () => usersList,
        todos: () => todosList,
    },
  }
```

Powyższa definicja wskazuje w jaki sposób GraphQL ma uzyskać dane dla zapytań **users** oraz **todos**. Sprawdźmy, czy przykład działa i wykonajmy zapytanie widoczne w poniższej tabeli. Wynik zapytania widoczny jest w prawej kolumnie:

Zapytanie GraphQL

```
query{
  users{
    id
    login
  }
}
```

Odpowiedź

```
{
  "data": {
    "users": [
      {
        "id": "1",
        "login": "jkonieczny"
      },
      {
        "id": "2",
        "login": "anna.wesolowska"
      },
      {
        "id": "3",
        "login": "p.waleczny"
      }
    ]
  }
}
```

Jak widać na powyższym przykładzie, GraphQL wyświetlił wszystkich użytkowników. Pytaliśmy tylko o pola **id** oraz **login**. Zatem reszta pól nie została w ogóle przesłana. Wiemy, że typ **User** posiada także kolekcję **ToDoItem**. Zatem dodajmy do zapytania to pole:

Zapytanie GraphQL

```
query{
  users{
        id
    login
    todos {
      title
    }
  }
```

Odpowiedź

```
{
  "data": {
    "users": null
  },
  "errors": [
    {
      "message": "Cannot return null for non-
nullable field User.todos.",
```

```
}
```

```
      "locations": [
        {
          "line": 5,
          "column": 5
        }
      ],
      "path": [
        "users",
        0,
        "todos"
      ]
    }
  ]
}
```

Jak widać otrzymaliśmy komunikat o błędzie. Dlaczego tak się dzieje? Ponieważ nie został napisany odpowiedni resolver, który dla użytkownika potrafi wczytać listę jego **todos**. Zatem dopiszmy taki resolwer. W pliku **src/App.js** definicja resolwerów będzie wyglądała teraz następująco:

```
const resolvers = {
    Query: {
        users: () => usersList,
        todos: () => todosList,
    },
    User:{
        todos: (parent, args, context, info) => {
            return todosList.filter(t => t.user_id == parent.id);
        }
    }
  }
```

Względem poprzedniej wersji dodaliśmy resolver dla typu **User**. W tym resolwerze zaimplementowaliśmy metodę zwracającą wartości dla pola **todos**. Zwróć uwagę, że metoda ta przyjmuje cztery argumenty:

- **parent**: jest obiektem nadrzędnym. W naszym przypadku będzie to obiekt User.
- **args**: to lista argumentów z zapytania. W naszym przypadku nie będzie tam nic konkretnego.
- **context**: żyje tak długo jak żyje zapytanie. Służy do przekazywania informacji pomiędzy resolverami.
- **info**: dostarcza szczegółów o aktualnym zapytaniu.

Sama implementacji tej metody jest bardzo prosta. Z listy todosList filtrujemy wszystkie elementy, których **user_id** jest równe **parent.id**. Wykonajmy nasze zapytanie ponownie:

Zapytanie GraphQL

```
query{
  users{
        id
    login
    todos {
      title
    }
  }
}
```

Odpowiedź

```
{
  "data": {
    "users": [
      {
        "id": "1",
        "login": "jkonieczny",
        "todos": []
      },
      {
        "id": "2",
        "login": "anna.wesolowska",
        "todos": [
          {
            "title": "Odebrać buty"
          },
          {
            "title": "Wysłać paczkę"
          }
        ]
      },
      {
        "id": "3",
        "login": "p.waleczny",
        "todos": [
          {
            "title": "Naprawić samochód"
          },
          {
            "title": "Posprzątać garaż"
          },
          {
            "title": "Napisać e-mail"
          },
          {
            "title": "Zamówic kuriera"
          }
        ]
      }
    ]
  }
}
```

Teraz już zapytanie działa tak jak powinno. Analogicznie musimy napisać resolver dla pola **user** z typu **ToDoItem**. Przyjmie on postać:

```
const resolvers = {
  // ...
    ToDoItem:{
        user: (parent, args, context, info) => {
            return usersList.find(u => u.id == parent.user_id);
        }
    },
  // ...
  }
```

Na koniec pozostaje napisać resolwery dla wyszukiwania użytkownika oraz **todo** po ich identyfikatorach. Na początek dopiszmy odpowiednie wskazania na metody w reslowerze:

```
const resolvers = {
  Query: {
    users: () => usersList,
    todos: () => todosList,
    todo: (parent, args, context, info) => todoById(parent, args, context, info),
    user: (parent, args, context, info) => userById(parent, args, context, info),
  },
    //...
}
```

Będą to dwie metody **todoById** oraz **userById**, które w naszym przypadku przyjmują implementację:

```
function todoById(parent, args, context, info){
    return todosList.find(t => t.id == args.id);
}

function userById(parent, args, context, info){
    return usersList.find(u => u.id == args.id);
}
```

Sprawdźmy, czy wszystko działa i wykonajmy zapytanie o użytkownika o id równym 2 wraz z jego listą zadań do wykonania:

Zapytanie GraphQL	Odpowiedź

`query{`
`  user(id: 2){`
`  name`
`  todos{`
	`{`
	`  "data": {`
	`  "user": {`
	`  "name": "Anna Wesołowska",`

```
      title
    }
  }
}
```

```
      "todos": [
        {
          "title": "Odebrać buty"
        },
        {
          "title": "Wysłać paczkę"
        }
      ]
    }
  }
}
```

Podłączenie GraphQL do źródła danych REST

Na koniec omówimy sobie jak podłączyć GraphQL do prawdziwego źródła danych. W naszym przykładzie będzie to serwis REST dostępny dla deweloperów pod adresem: https://jsonplaceholder.typicode.com/users. Wykorzystamy listę użytkowników z tego serwisu, aby wstawić ją zamiast naszej tablicy.

Do odwoływania się do serwisu REST skorzystamy z biblioteki axios. Instalujemy ją poniższym poleceniem:

```
npm install axios
```

Następnie importujemy bibliotekę w pliku **src/App.js** poleceniem:

```
const axios = require("axios")
```

Aby wczytać listę użytkowników z serwisu REST napiszemy asynchroniczną funkcję **getRestUsersList()**, która docelowo zastąpi tablicę **usersLists**. Implementacja tej funkcji jest bardzo prosta i sprowadza się do odpytania serwisu i zmapowania odpowiedzi do naszej Schemy:

```
async function getRestUsersList(){
    try {
        const users = await axios.get("https://jsonplaceholder.typicode.com/users
")
        console.log(users);
        return users.data.map(({ id, name, email, username }) => ({
          id: id,
          name: name,
          email: email,
          login: username,
        }))
      } catch (error) {
        throw error
```

```
      }
}
```

Następnie podłączamy funkcję do resolvera:

```
const resolvers = {
  Query: {
    users: async () => getRestUsersList(),
    todos: () => todosList,
    todo: (parent, args, context, info) => todoById(parent, args, context, info),
    user: (parent, args, context, info) => userById(parent, args, context, info),
  },
  //...
```

Pora przetestować rozwiązanie. Zadajmy zapytanie:

Zapytanie GraphQL

```
query{
  users{
    name
    id
        login
    todos{
      title
    }
  }
}
```

Odpowiedź

```
{
  "data": {
    "users": [
      {
        "name": "Leanne Graham",
        "id": "1",
        "login": "Bret",
        "todos": []
      },
      {
        "name": "Ervin Howell",
        "id": "2",
        "login": "Antonette",
        "todos": [
          {
            "title": "Odebrać buty"
          },
          {
            "title": "Wysłać paczkę"
          }
        ]
      },
  # ... reszta odpowiedzi
```

Modyfikacja poprawnie zadziałała. GraphQL wczytuje dane użytkowników z zewnętrznego źródła. Teraz staje się już jasne, że w tym miejscu można napisać dowolną implementację dostępu do danych.