Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

"ZPR PWr - Integrated Development ProgramWrocław University of Technology"

Wrocław University of Science and Technology

Politechnika Wrocławska

Advanced Web Technologies

Lab

Topic: GraphQL i.e. the new generation REST API

Prepared by: mgr Piotr Jozwiak

Date: July 2020

Number of hours: 2 hours

Table of Contents

Entry

In this lab, we will try to answer what GraphQL is and look at its most important concepts.

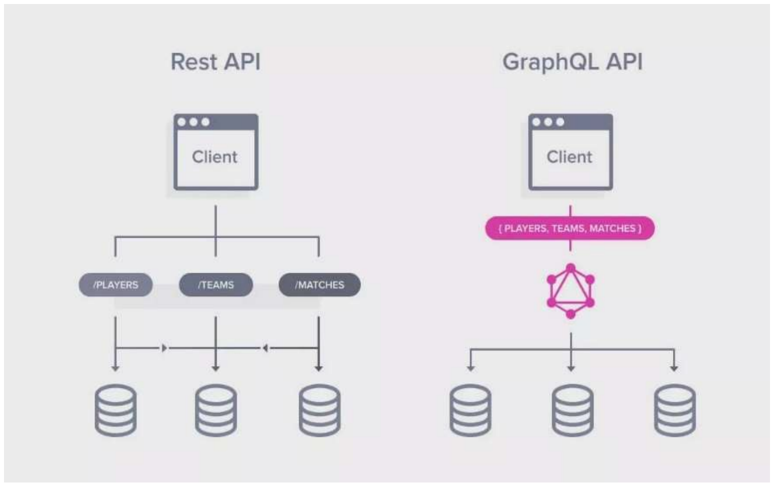
So what is GraphQL? It is a query language used to support the API. It offers much more flexibility compared to the REST API. To put it simply, GraphQL is a data query language.

The reason for its popularization are the limitations of the REST API technology, which hinder the development of applications, mainly cross-platform ones. Let's look at these limitations with an example. Let's imagine that we have a certain REST API server that returns, for example, the following result for a researcher's details query:

```
{
    "id":67,
    "name":"Jan
    Kowalski","title":"Dr.
    Eng.",
    "org":"W8 K7",
    "e-mail":"kan.kowalski@pwr.edu.pl",
    "publications": [/* Here is the table with publication ids */]
```

Let's assume that in the desktop version of the application, all the above data is used. But in the smartphone application we only need {id, name, email}. Naturally, the REST API cannot send us only the information we actually need. So we get more than we need, which is a waste of resources. Of course, one could say that this is a negligible amount. So let's think about what communication would look like if, in addition to the employee's data, we needed to have additional details for each of his publications? In such a situation, we usually have to additionally query the server one by one for each publication separately. And this is no longer a completely negligible operation.

GraphQL comes to the rescue just nowabove situations.A properly constructed graphql query in a single request to the endpoint will receive information about the employee along with the details of all his publications. As if that wasn't enough, graphql also allows extendedfiltering, sorting anddata processing before sending them to the client. This is best illustrated by the diagram below:



How it's working?

The question is how it all works. To answer it, we need to cover three basic GraphQL concepts.

The first is Queries. Communication in GraphQL is based on queries. We define the query using the query keyword and the names of individual resource properties. In response, we receive a JSON object with the same structure as the query, but with filled data. An example GraphQL query for an example with an employee, in which we would also like to receive additional information about his publications, looks like this:

```
{
    employees{
        id
        name
        title
        email
        publications{i
            d
            year
            title
        }
    }
}
```

Another important element are Resolvers. GraphQL cannot deduce itself where the data we are asking for is located. It is the resolvers, i.e. some kind of functions, that indicate data sources for GraphQL. If in a given query we ask for a list of employees, the resolver knows where to get this data.

The last element is Schema. This concept is used to describe the structure and types of data that the server will be able to handle. SDL (Schema Definition Language) is used for description. Schema also allows for good documentation of the server's API, so we don't have to do it manually.

Environment installation

We will use the library to prepare the environment**`graphql-yoga`**running on a simple NodeJs application. This application will be used to prepare a GraphQL server, which can be successfully developed into a full-fledged backend environment.

We will use the npm package manager to prepare the environment. Let's execute the following commands to create a folder for the application:

```
mkdir first-graphql-app
continuedfirst-graphql-app
```

Thenwe generate the package.json file using npm:

```
aslinit
```

All questions can be answeredleaving the default suggestions. We will modify this file later.

Then we install the necessary libraries.At the moment, two are enough for us: graphql-yoga and nodemon. The first one is a library (one of many) supporting the implementation of the graphql server. The second library is used to run NodeJS applications.

```
npm install graphql-yoganodemon
```

Let's create a folder for source files:

```
mkdirsrc
```

Now we can write the rest of the application in our favorite IDE. I recommend, for example, Visual Studio Code.

Before we discuss how to implement the GraphQL environment, let's prepare a simple example to check that the environment is fully functional. First, let's create the src/schema.graphql file. What this file is, we will discuss later, for now let's thoughtlessly enter its content in the form:

```
typeQuery{demo:
  String!
}
```

Next, let's create the main application file src/App.js, the contents of which should look like this:

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

constresolvers=
    {Queries:{
      demo:()=>'Hello, GraphQL is working!',
    },
  }

  constserver=newGraphQLServer({typeD
    efs:'./src/schema.graphql',resolv
    ers,
  });

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

In the example above, we first import the graphql-yoga library. Next, we define an object called resolvers. What it is and what it is responsible for, we will discuss in more detail later in the lab. Then we create the GrapgQLServer server object, to which we pass typeDefs as the path to the previously created schema file and the resolvers object. Finally, we start the server itself.

For all this to work, we still need to define how to run our application. To do this, we need to edit the package.json file, in the scripts section, add the start definition according to the example below:

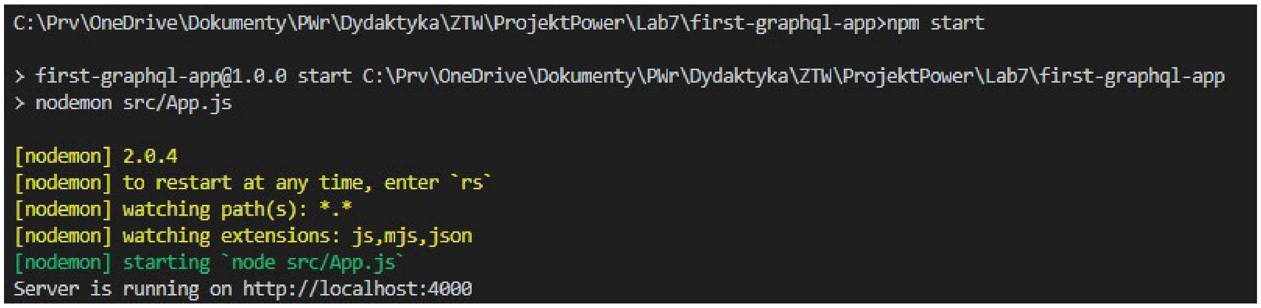
```
{
    //...
    "scripts": {
      "start":" nodemon -e js,graphql src/App.js",

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

Now we can run the application by issuing the following command while inside the folder with the file **`package.json:`**

```
aslstart
```

If everything starts correctly, we will see the appropriate message about the started server:



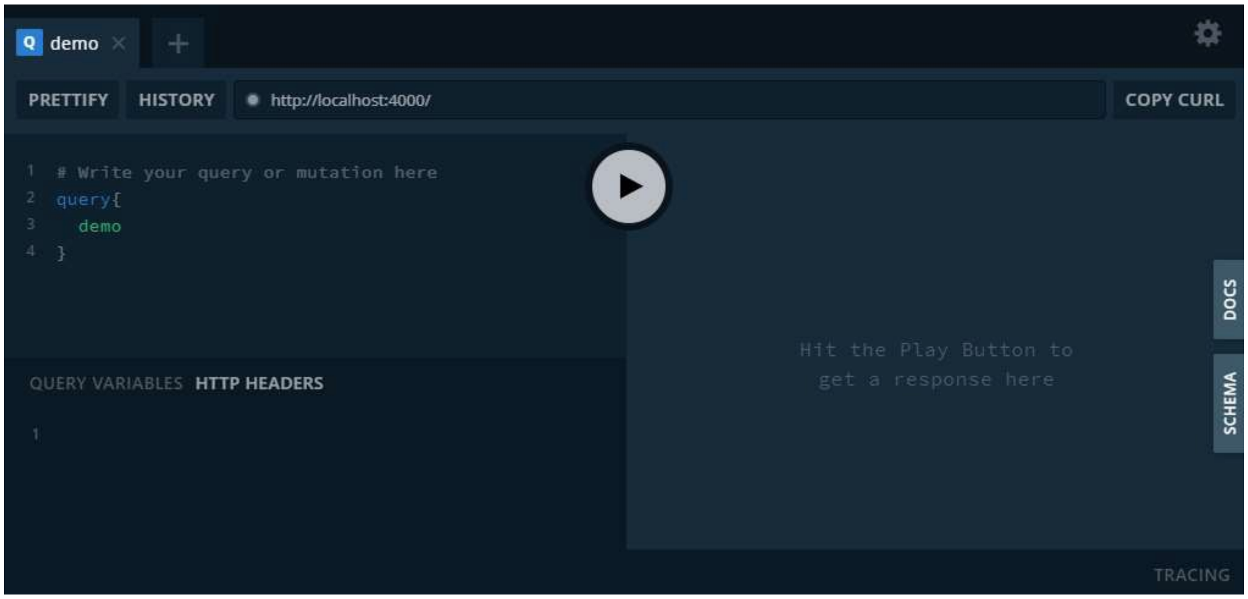
```
C:\Prv\OneDrive\Dokumenty\PWr\Dydaktyka\ZTW\ProjektPower\Lab7\first-graphql-app>npm start

> 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

It's time to check what will run at the given address. So let's go to the browser and run the pagehttp://localhost:4000. The GraphQL playground tool will run at the specified address:



```
# Write your query or mutation here
query{
  demo
}
```

```
Hit the Play Button to
get a response here
```

This is a built-in library functionality that allows you to run graphql queries and view documentation. Queries are entered in the box on the left. We get the results after pressing the Play button on the right. Let's test the operation. Then run the query in the form:

```
query{
    demo
  }
```

We will get a response from the server as a result:

```
{
    "date": {
      "demo":"Hello, GraphQL is working!"
    }
  }
```

A skilled observer will probably notice where the definition of the above result is.

Note that there is a DOCS tab on the right. Let's take a look at its content. This is where the documentation for defined APIs will be displayed.

Schema - API description

At the beginning of working with GraphQL, you should design the schemas that define the server API. SDL (Schema Definition Language) or IDL (Interface Definition Language) is used for this. GraphQL is a strongly typed language that enables interoperability with any programming language.

The basic component of the schema are Object Types, which represent objects and their properties. These are elements such as Query, Mutation and Subscriptions. These three elements form the so-called Root Types in the schema. They are used to download, modify and observe data, respectively.

Schema definitions can be separated into separate files, usually with the *.graphql extension. In our example, one such file has already been created (src/schema.graphql).

Type definition

The most basic componentschemas are type objects that represent a description of the object that can be retrieved from the site. They describe the available fields of this object along with its type. If we were to define the scheme of a researcher, it would look like this:

```
typeEmpleyee{id
    :ID!
    name:String!ti
    tles:String!e-
    mail:String!
    contractor:Boolean!a
    ge:int!
    publications: [articles!]!
}
```

The language itself is very clear, but let's look at the individual elements of the above definition.

- **employees**is an Object Type, comeans that it is a type definition with specified fields.
- **Id**, name, title and moreelement are fields that describe the Employee object. This means that whenever an Employee object is requested, the information specified in that object can be requested for it.
- **String**,**Boolean**, Int, Float are built-in data types.
- **string!**, Int!, in general, an exclamation mark at the end of a data type means that the field is non-nullable. So there will always be some value here.
- **[Article!]!**meansan array of Article objects.

The issue of defining Schema in GraphQL is very extensive. Thus, it goes beyond the scope of a full description here. Therefore, I would like to present only one more issue, which are the arguments that we will use in further examples. A full description of Schema to Type functionality can be found here:https://graphql.org/learn/schema/.

Each field in Schema can define a list of arguments. It's kind of like a function. Arguments are used to pass query parameters. The parameter can be used for filtering, searching or specifying the sort order. Arguments are also used for mutations, whose task is, for example, to add or edit meta-model objects. Let's look at the definition below to get an idea of how arguments work in practice. Suppose we want to define two queries for Employee type. One, returning all researchers and the other, parameterized, which will return a specific employee with a given id:

```
typeQuery{
  employees:
  [Empleyee!]worker(id:ID!):E
}
```

Arguments are defined inside parentheses by specifying their name and type.

Resolvers

A resolver is nothing more than a method that returns a value for a given type or field. The resolver combines the definition of schemas with the method of obtaining data from a source, e.g. a database or a REST service. To understand how Resolvers work, we need to understand how GraphQL queries work. Each query goes through three phases: Parsing, Validation, and Execution.

Parsing is the representation of the query in the form of an AST tree. To see what such a tree looks like, you can use the tool available at:https://astexplorer.net. Just paste the example query and choose graphql as the language.

Validation checks whether the AST matches the schema definition.

The last phase consists in execution by running the necessary methods/resolvers and building the response. Methods are run from the top of the AST tree.

We have already defined such a resolver in our example. Just look at the src/App.js file and find the resolver definition for the query demo. However, to better illustrate how GraphQL works, let's take a look at the complete solution.

graphQL server implementation on the example of TODO list

The following example shows how to implement GraphQL on the example of a simple functionality related to the functionality of the to-do list - commonly called in English: to do.

First, let's define the scheme of a single task to be performed. In our case, it will look like this. As a reminder, we will place the definition of schemas in the schema.graphql file:

```
typeToDoItem{i
    d:ID!
    titles:String!compl
    eted:Boolean!user:U
    ser!
}
```

As you can see in the ToDoItem definition abovealso has a user field of type User. This is an indication of the user to whom the task applies. The user schema definition takes the following form:

```
typeUser{
    id:ID!
    name:String!e-
    mail:String!
    login:String!todos:
    [ToDoItem!]!
}
```

It's worth noting here that the user has a todos field, which is a collection of ToDoItems.

We still need to define the queries that our GraphQL server will handle. We will implement the following four queries:

```
typeQuery{
    todos:
    [ToDoItem!]todo(id:ID!)
    :ToDoItemusers:
    [User!]user(id:ID!):Use
    r
```

In addition to the collection of all users and things to do, we also define a query for a single user and a todo based on their id.

It's time to write resolvers. In our basic example, resolvers will run on arrays in JavaScript. We'll change it to a real data source later. But to make it easier to understand how it works, we will not obfuscate the example with this issue now.

INsrc/App.js file, we define tables with data sources for GraphQL. They take the form:

```
constusersList= [
    {id:1,name:"Jan Konieczny",e-mail:"jan.konieczny@wonet.pl",login:"eternal"},
    {id:2,name:"Anna Wesolowska",e-mail:"anna.w@sad.gov.pl",login:"anna.we
solowska"},
    {id:3,name:"Peter the Brave",e-mail:"piotr.waleczny@gp.pl",login:"valorous"}
];

consttodosList= [


    { id: 1, titles "Repair a car",       completed: false, user_id: 3 },
                  :
    { id: 2, titles "Clean up the       completed: true,  user_id: 3 },
                  :    garage",
    { id: 3, titles "Write an email",    completed: false, user_id: 3 },
                  :
    { id: 4, titles "Pick Up Your        completed: false, user_id: 2 },
];                :    Shoes",
    { id: 5, titles "Send a package",    completed: true,  user_id: 2 },
                  :
    { id: 6, titles "Order a courier",   completed: false, user_id: 3 },
                  :
```

Then we need to write a resolver that will use this data. To do this, we change the resolver definition to the following:

```
constresolvers=
    {Queries:{
        users:()=>usersList,tod
        os:()=>todosList,
    },
  }
```

The above definition shows howthe way GraphQL is supposed to get data for queries users and todos. Let's check if the example works and execute the query shown in the table below. The query result is shown in the right column:

GraphQL query

```
query{
  users{l
    ogin
  }
}
```

Answer

```
{
  "date": {
    "users": [
      {
        "id": "1",
        "login": "necessary"
      },
      {
        "id": "2",
        "login": "anna.wesolowska"
      },
      {
        "id": "3",
        "login": "valorous"
      }
    ]
  }
}
```

As seen in the example above,GraphQL displayed all users. We only asked for id and login fields. So the rest of the fields were not sent at all. We know that the User type also has a ToDoItem collection. So let's add this field to the query:

GraphQL query

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

Answer

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

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

As you can see we got an error message. Why is this happening? Because there is no proper resolver that can read the list of his todos for the user. So let's add a resolver. In the src/App.js file, the resolver definition will now look like this:

```
constresolvers=
    {Queries:{
        users:()=>usersList,tod
        os:()=>todosList,
    },
    user:{
        todos:(parents,args,context,info)=>{
            returntodosList.filters(t=>t.user_id==parents.id);
        }
    }
  }
```

Relative to the previous version, we addedresolver for type User. In this resolver we have implemented a method that returns values for the todos field. Note that this method takes four arguments:

- **`parents`**: is the parent object. In our case, this will be the User object.
- **`args`**: is a list of arguments from the query. In our case, there will be nothing specific there.
- **`context`**: lives as long as the query lives. It is used to transfer information between resolvers.
- **`info`**: provides details about the current query.

The implementation of this method itselfis very simple. From the todosList, we filter all the items whose **`user_id`**is equal toparent.id. Let's run our query again:

InquiryGraphQLOresponse	

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

```
{
  "date": {
    "users": [
      {
        "id": "1",
        "login": "necessary",
        "todos": []
      },
      {
        "id": "2",
        "login": "Anna.wesolowska",
        "todos": [
          {
            "title": "Pick up your
            shoes"
          },
          {
            "title": "Send a package"
          }
        ]
      },
      {
        "id": "3",
        "login": "p.brave",
        "todos": [
          {
            "title": "Fix the car"
          },
          {
            "title": "Clean up the
            garage"
          },
          {
            "title": "Write an email"
          },
          {
            "title": "Order a courier"
          }
        ]
      }
    ]
  }
}
```

Now the query works as it should.Similarly, we need to write a resolver for the user field of type **`ToDoItem`**. It will take the form:

```
 constresolvers= {
  // ...
    ToDoItem:{
           user:(parents,args,context,info)=>{
              returnusersList.find(at=>at.id==parents.user_id);
        }
    },
  // ...
  }
```

OnFinally, it remains to write resolvers for user and todo searches by their ids. At the beginning, let's add appropriate indications for methods in the reslower:

```
constresolvers=
  {Queries:{
    users:()=>usersList,tod
    os:()=>todosList,
    todo:(parents,args,context,info)=>todoById(parents,args,context,info),

    user:(parents,args,context,info)=>userById(parents,args,context,info),
  },
    //...
}
```

They willare two methods todoById and userById, which in our case take the implementation:

```
functionstodoById(parents,args,context,info){ret
    urntodosList.find(t=>t.id==args.id);
}

functionsuserById(parents,args,context,info){ret
    urnusersList.find(at=>at.id==args.id);
}
```

Let's check if everything works and query for the user with id equal to 2 and his list of tasks to be done:

InquiryGraphQLOresponse	

```
query{
  user(id:2){n
    ame todos{
```
	```
	{
	  "date": {
	    "user": {
	      "name": "Anna Wesolaoats",
	```

```
      titles
    }
  }
}
```

```
      "todos": [
        {
          "title": "Pick up your
          shoes"
        },
        {
          "title": "Send a package"
        }
      ]
    }
  }
```

Connecting GraphQL to a REST data source

Finally, we will discuss how to connect GraphQL to a real data source. In our example, it will be a REST service available to developers at:https://jsonplaceholder.typicode.com/users. We will use the list of users from this site to insert it in place of our table.

We will use the axios library to refer to the REST service. We install it with the following command:

```
npm installaxis
```

Thenimport the library in the src/App.js file with the command:

```
constaxis=require("axis")
```

To load the list of users from the REST service, we will write an asynchronous function getRestUsersList(), which will eventually replace the usersLists array. The implementation of this function is very simple and comes down to querying the service and mapping the response to our Schema:

```
async
    functionsgetRestUsersList(){tr
    y{
        constusers=waitaxis.get("https://jsonplaceholder.typicode.com/users
")
        console.log(users);
        returnusers.date.map(({id,name,e-mail,username})=>({id: id,
          name:
          name,email:
          email,login:
          username,
        }))
      }catch(error)
      }
}
```

Then we connect the function to the resolver:

```
constresolvers=
  {Queries:{
    users:async()=>getRestUsersList(),todo
    s:()=>todosList,
    todo:(parents,args,context,info)=>todoById(parents,args,context,info),user:(p
    arents,args,context,info)=>userById(parents,args,context,info),
  },
  //...
```

It's time to test the solution. Let's ask:

InquiryGraphQLOresponse

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

```
{
  "date": {
    "users": [
      {
        "name": "Leanna Graham",
        "id": "1",
        "login": "Brett",
        "todos": []
      },
      {
        "name": "Ervin Howell",
        "id": "2",
        "login": "Antonette",
        "todos": [
          {
            "title": "Pick up your
            shoes"
          },
          {
            "title": "Send a package"
          }
        ]
      },
  # ... the rest of the answer
```

The modification worked correctly. GraphQL reads user data from an external source. Now it becomes clear that any data access implementation can be written here.