PETER WOLF
< Web Developer & Advisor />

GraphQL – CrashCourse

Course Repo
https://github.com/pewobox79/graphql_crash_course_pewo

Source:
https://www.youtube.com/watch?v=xMCnDesBggM&list=PL4cUxeGkcC9gUxtblNUahcsg0WLxmrK_y

fullstack crash course
https://www.youtube.com/watch?v=BcLNfwF04Kw

What is GRAPHQL?

Prerequisits:
Basic NodeJS experience

GraphQL is an alternative to REST API
GraphQL is a kind of layer between your database and the client.

Let´s say, we have an endpoint:

myDomain.com/api/pokemon
myDomain.com/api/pokemon/123

If we request data with REST API concept, the we might face to thing:

1. OVER FETCHING
2. UNDER FETCHING

OVERFETCHING:

If we fetch data of an pokemon, and we want only the Id and name. With the endpoint is much more data incoming which we don´t need.

UNDERFETCHING:

```
{
  "id": "1",
  "title": "Thud",
  "author": {...},
  "price": "10.99",
  "thumbnail_url": "...",
  "video_url": "...",
}
```

let us assume we want to gather an blog article which has an author information included.
We want to provide all articles from that user.
To achieve that, we need to do another query to another endpoint. This will reduce the performance of the application because of multiple request.

How to work with GraphQL data?

For Rest API we usually use POSTMAN to check certain requests.

In GraphQL we will use theGraphiali UI which helps to test the GraphQL Queries.

Schemas and Types

GraphQL servers have a „schema“ that specifies all the fields and their types.
Here a basic schema object.

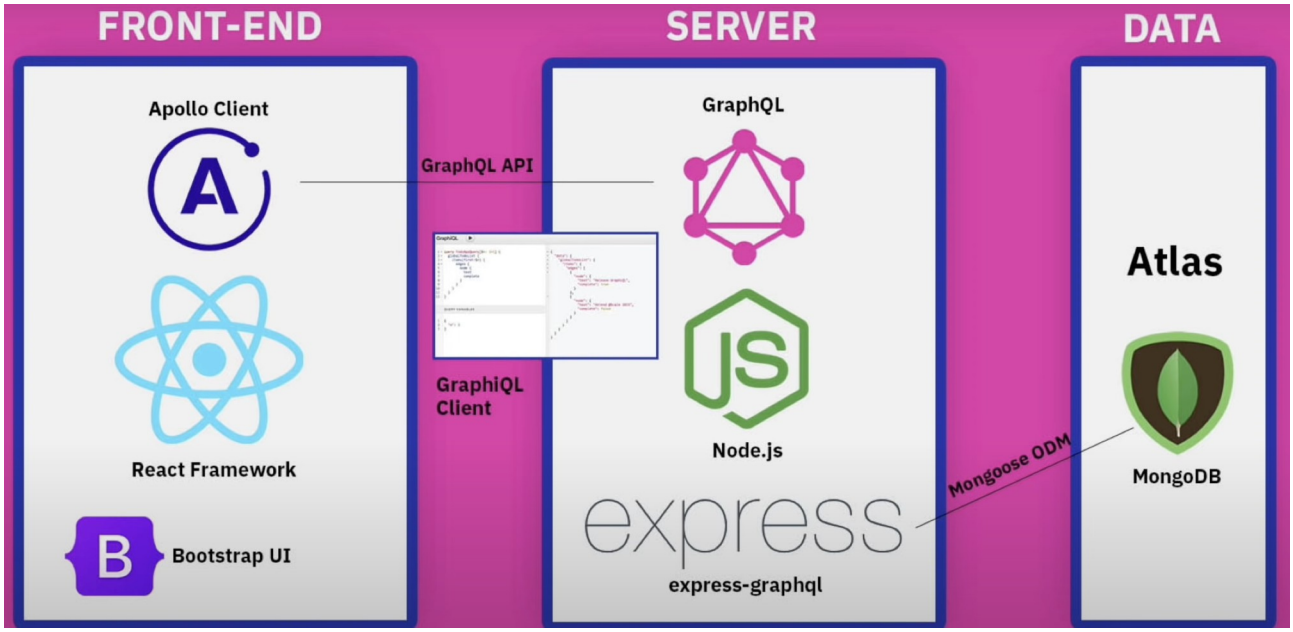
```
type Project {
  name: String!
  description: String!,
  status: String!
}
```

Object Type

Scalar Types includes (the most basic data type that holds only a single atomic value at a time)

- Boolean
- Int
- Float
- String
- ID

Our MERN STACK Project



PETER WOLF
< Web Developer & Advisor />

Let´s start the project

Update Visual Studio Code Extensions:

Add the Extension GraphQL Syntax Highlight to your VSCODE
https://marketplace.visualstudio.com/items?itemName=mquandalle.graphql

Running an Express GraphQL Server

graphql.org/graphql-js/running-an-express-graphql-server/

1. Create a Project folder on your local machine for a NodeJS instance.

2. run terminal command „npm init –type=modal“

3. install needed dependencies (colors is optional)

```
npm i express express-graphql graphql mongoose cors colors
```

4. install dev dependencies

```
npm i -D nodemon dotenv
```

My package.json looks like this

```
package.json   .env
{
  "name": "graphql-crash-course",
  "version": "1.0.0",
  "description": "",
  "main": "server/index.js",
  "type": "module",
  "scripts": {
    "start": "node server/index.js",
    "dev": "nodemon server/index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.19.2",
    "express-graphql": "^0.12.0",
    "graphql": "^15.8.0",
    "mongoose": "^8.2.4"
  },
  "devDependencies": {
    "dotenv": "^16.4.5",
    "nodemon": "^3.1.0"
  }
```

Add Server file structure to Project

root/server/index.js

the index.js file has this content now:

PETER WOLF
< Web Developer & Advisor />

```
index.js   .env
import express from 'express';
import 'dotenv/config';
const port = process.env.PORT || 5001;

const app : Express  = express();


app.listen(port,  hostname: () : void  => {console.log(`listening on ${port}`)})
```

Now we bring in GRAPHQL HTTP

First we need to add the schema folder structure to the server folder

```
server
  schema
    schema.js  30.03.24, 17:01, 0 B
  index.js  30.03.24, 17:01, 286 B Moments ago
```

after that, we add the graphqlHTTP to the server

```
index.js   schema.js   package.json   .env
import express from 'express';
import 'dotenv/config';
const port = process.env.PORT || 5001;
import {graphqlHTTP} from 'express-graphql';
import schema from './graphql/schema';

const app : Express  = express();

//graphql code here
app.use('/graphql', graphqlHTTP( options: {
  schema: schema,
  //we want to use for testing the graphiql ci only in development mode
  graphiql: process.env.NODE_ENV === 'development'
}))


app.listen(port,  hostname: () : void  => {console.log(`listening on ${port}`)})
```

What did we do above:

1. we initialized graphql
2. we defined a schema
3. we added the graphiql UI to be used only in dev mode

Add some Data to the project:

I have a sampledata.js for now added to the project.
Later on we will have a MongoDB database connected.

=> provide a data set by C&P

import Data to the Schema.js and init the graphql dependecies

First we need to import the GraphQLObjectType to define the Types.
For that, we will create a types.js file to keep types separate.

```
import {GraphQLID, GraphQLObjectType, GraphQLString} from "graphql";

1+ usages  new *
export const ClientType : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
  //client type => conventional the type is in uppercase
  //define the  name of the type:
  name: 'Client',
  //define the fields you need
  fields: () : {...}  => ({
    id: {type: GraphQLID},
    name: {type: GraphQLString},
    email: {type: GraphQLString},
    phone: {type: GraphQLString},
  })
})
```

PETER WOLF
< Web Developer & Advisor />

Now we have defined the schema for a client query in schema.js.

Define the Query for calling a single Client

Add the below code to the schema.js file

```
18
19  //Define the query parameters to be able to query the database
20  const Query : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
21    name: 'Query',
22    fields: {
23      client: {
24        type: ClientType,
25        args: {id: {type: GraphQLID}}, //to get a single client
26        resolve(parent : TSource , args : {[p: string]: any} ) : T  {
27          // for now this is the find()method >> later the mongoose query happens here;
28          return clients.find(client : {...}  => client.id === args.id);
29        }
30      }
31    }
32  })
33
34  const schema : GraphQLSchema  =  new GraphQLSchema( config: {
35    query: Query,
36  });
    1+ usages
37  export default schema;
```

PETER WOLF
< Web Developer & Advisor />

Test our Query in Graphiql

lets call the localhost:5001/graphql

This is available only in DEV mode . It is an UI to make query tests.....

In graphQL sieht das dann so aus:

```
{
client(id:"1"){
  name,
  email,
  phone,
  id
}}
```

```
{
  "data": {
    "client": {
      "name": "Tony Stark",
      "email":
"ironman@gmail.com",
      "phone": "343-567-4333",
      "id": "1"
    }
  }
}
```

left is my query – right the output

PETER WOLF
< Web Developer & Advisor />

Query to get ALL Clients

```
//Define the query parameters to be able to query the database
const Query : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
  name: 'Query',
  fields: {
    //get all the clients
    clients: {
      type: new GraphQLList(ClientType),
      resolve() : [{phone: string, name: string,...  {
        return clients;
      }
    },
    //getSingle Client
    client: {
      type: ClientType,
      args: {id: {type: GraphQLID}}, //to get a single client
      resolve(parent : TSource , args : {[p: string]: any} ) : T  {
        // for now this is the find()method >> later the mongoose query happens here;
        return clients.find(client : {...}  => client.id === args.id);
      }
    }
  }
})
```

Make the query for clients

```
1 {
2 clients{
3   id,
4   name
5 }}
```

```
{
  "data": {
    "clients":
[
      {
        "id":
"1",
        "name":
"Tony Stark"
      },
      {
        "id":
"2",
```

PETER WOLF
< Web Developer & Advisor />

Add Query for Projects and Single Project

Define the ProjectType

```
const ProjectType : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
  //define the  name of the type:
  name: 'Project',

  //define the fields you need
  fields: () : {...}  => ({
    id: {type: GraphQLID},
    name: {type: GraphQLString},
    description: {type: GraphQLString},
    status: {type: GraphQLString},
    clientId: {type: GraphQLID},
  })
})
```

Add to the fields object in query the queries for projects and project.

```
fields: {
  //get all the clients
  clients: {...},
  //getSingle Client
  client: {type: ClientType...},
  projects: {
    type: new GraphQLList(ProjectType),
    resolve() : [{clientId: string, name: stri...  {
      return projects;
    },
    //getSingle Project

  },
  project: {
    type:ProjectType,
    args: {id: {type: GraphQLID}}, //to get a single project
    resolve(parent : TSource , args : {[p: string]: any} ) : T  {
      // for now this is the find()method >> later the mongoose query happens here;
      return projects.find(project : {...}  => project.id === args.id);
    }
  }
}
})
```

PETER WOLF
< Web Developer & Advisor />

Update the query to get the Client of the Project Relation.

1. extend the ProjctType with an entity of ClientType. => line 27

```
17  export const ProjectType : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
18    //define the  name of the type:
19    name: 'Project',
20
21    //define the fields you need
22    fields: () :  => ({
23      id: {type: GraphQLID},
24      name: {type: GraphQLString},
25      description: {type: GraphQLString},
26      status: {type: GraphQLString},
27      clientId: {type: GraphQLID},
28      client: {
29        type: ClientType,
30        resolve(parent : TSource , args : {[p: string]: any} ) : T  {
31        // for now this is the find()method >> later the mongoose query happens here;
32        return clients.find(client : {...}  => client.id === parent.clientId);
33      }}
34    })
35  })
```

As you see, the resolve is now in the ProjectType itself. Not in the Query only.
We call in resolve the data list of clients and search for the client.id in that.
The ProjectType resolve has an parent attribute, which provides us the data attributes of ProjectType.
Now we can check for the parent.clientId and find( ) the right entry.

=> check the data set of projects. There is a clientId entity.

Now you can search like that.

```
{
project(id:2){
  name,
  id,
  description,
  client{
    id,
    name
  }
}}
```

```
{
  "data": {
    "project": {
      "name": "Dating App",
      "id": "2",
      "description": "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu.",
      "client": {
        "id": "2",
        "name": "Natasha Romanova"
      }
    }
  }
}
```

PETER WOLF
< Web Developer & Advisor />

Connect GraphQL to a Databasse (mongoDB)

Till now we fetch the data from a static data file.
Now we want to integrate and connect mongoDB (mongoose) to improve the application setup.

1. Create an New Database in MongoDB

2. connect the URI to your Dev Enviroment varibales

```
NODE_ENV = 'development'
PORT=5001
MONGO_URI=mongodb+srv://pewobox79:<password>@wdg013db.l5fesxf.mongodb.net/WDG013DB?retryWrites=true&w=majority&appName=WDG013DB
```

3. create within the server folder a subfolder called /config/ and add the db.js file

4. in the db.js file place this code.

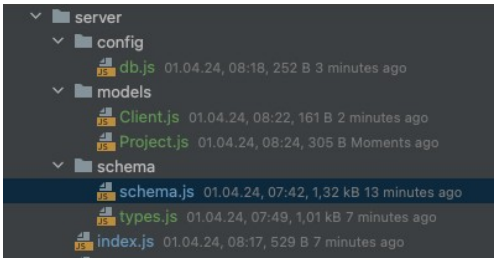
```
import mongoose from "mongoose";
import 'dotenv/config';

const URI = process.env.MONGODB_URI;

1+ usages   new *
export const connectDB =async () : Promise<void> =>{

  const connection :  = await mongoose.connect(URI)
  console.log(`MongoDB connected: ${connection.connection.host}`)
};
```

Keep in mind, that is a basic one. No error handling so far.

5. Place the connect() to the index.js file

Create the MonogoDB Models with mongoose

Add to the server folder this setup



you see now a models folder which contains two Files. .

Client.js
Project.js

Add to Client.js this:

```
import mongoose from "mongoose";

const ClientSchema : Schema<any, Model<...>, {...}, {...}, {...}, {...}, DefaultSchemaOptions, {...}...  = new mongoose.Schema( definition: {
  name: {type: String, required: true},
  email: String,
  phone: String
})

1+ usages   new *
export default mongoose.model( name: 'Client', ClientSchema);
```

PETER WOLF
< Web Developer & Advisor />

Add to Project.js this:

```
import mongoose from "mongoose";

const ProjectSchema : Schema<any, Model<...>, {...}, {...}, {...}, {...}, DefaultSchemaOptions,  = new mongoose.Schema( definition: {
  name: {type: String, required: true},
  description: String,
  status: {
    type: String,
    enum: ['Not Started','In Progress', 'Completed']
  },
  clientId: {type: mongoose.Schema.Types.ObjectId, ref: 'Client'},
})

1 usages  new *
export default mongoose.model( name: 'Project', ProjectSchema);
```

This code provides a clientId which is referencing to the Client Model. This is mongoDB special…

Import Client and Project to the schema.js file of GraphQLObjectType

```
import {clients, projects} from '../sampleData.js';
import {GraphQLID, GraphQLList, GraphQLObjectType, GraphQLSchema, GraphQLString} from "graphql";
import {ClientType, ProjectType} from "./types.js";
import Client from "../models/Client.js";
import Project from "../models/Project.js";

//Define the query parameters to be able to query the database
const Query : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
  name: 'Query',
  fields: {
    //get all the clients
    clients: {
      type: new GraphQLList(ClientType),
      resolve() : Query<  {
        return Client.find();
```

Now you an replace the static file methods with the MongoDB Schemas

```
8   const Query : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
9     name: 'Query',
10    fields: {
11      //get all the clients
12      clients: {
13        type: new GraphQLList(ClientType),
14        resolve() : Query<  {
15          return Client.find();
16        }
17      },
18      //getSingle Client
19      client: {
20        type: ClientType,
21        args: {id: {type: GraphQLID}}, //to get a single client
22        resolve(parent : TSource , args : {[p: string]: any} ) : Query<  {
23          // for now this is the find()method >> later the mongoose query happens here;
24          return Client.find(args.id);
25        }
26      },
27      projects: {
28        type: new GraphQLList(ProjectType),
29        resolve() : Query<  {
30          return Project.find();
31        },
```

```
32        //getSingle Project
33
34      },
35      project: {
36        type:ProjectType,
37        args: {id: {type: GraphQLID}}, //to get a single project
38        resolve(parent : TSource , args : {[p: string]: any} ) : Query<..., ..., unknown, any, "findOne">  {
39          // for now this is the find()method >> later the mongoose query happens here;
40          return Project.findById(args.id);
41        }
42      }
43    }
44  })
```

Add the data Set to the database.

Now that we have our MongoDB database connected with GraphQL, we need to provide the acutall data.

Our next step is to MUTATE the data to the database.

1. add the MUTATION const to the schema.js and extend the schema export…

PETER WOLF
< Web Developer & Advisor />

```
45
46  // Mutations
47  const Mutation : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
48    name: 'Mutation',
49    fields: {
50      addClient:{}
51    }
52  })
53
54  const schema : GraphQLSchema  =  new GraphQLSchema( config: {
55    query: Query,
56    mutation: Mutation,
57  });
    1+ usages   aviareps_pwolf
58  export default schema;
```

2. now create the addClient values and resolver

```
47  const Mutation : GraphQLObjectType<any, any>  = new GraphQLObjectType( config: {
48    name: 'Mutation',
49    fields: {
50      addClient: {
51        type: ClientType,
52        args: {
53          name: {type:GraphQLNonNull(GraphQLString)},
54          email: {type: GraphQLNonNull(GraphQLString)},
55          phone: {type: GraphQLNonNull(GraphQLString)}
56        },
57        resolve(parent : TSource , args : {[p: string]: any} ) {
58          const client : HydratedDocument<InferSchemaType  = new Client( doc: {
59            name: args.name,
60            email: args.email,
61            phone: args.phone
62          });
63
64          return client.save();
65
66        }
67      }
68    }
69  })
```

3. Add in GraphiQL die neuen daten

```
mutation{
  addClient(name:"tony Stark", email:"ironman@gmail.com", phone:"555-555-5555"){
    id,
    name,
    email,
    phone
  }
}
```

```
{
  "data": {
    "addClient": {
      "id": "660b8ce2f809b4eecbdb9c94",
      "name": "tony Stark",
      "email": "ironman@gmail.com",
      "phone": "555-555-5555"
    }
  }
}
```

der Return kommt durch den direkten Aufruf der werte. Die ID wird von mongoDB erstellt.

Add a new Mutation DELETECLIENT

```
  name: 'Mutation',
  fields: {
    addClient: {type: ClientType...},
    deleteClient: {
      type: ClientType,
      args: {id: {type: GraphQLNonNull(GraphQLString)}},
      resolve(parent : TSource , args : {[p: string]: any} ) : Query<..., ..., unknown, any, "findOneAndDelete">  {
        return Client.findByIdAndDelete(args.id);
      }
    }
  }
})
```

Now test this by adding a dummy „test“ contact.

Check with Clients_Query the list of all clients => includs the „test“ cleint as well.

Run the deleteClient( ) on the ID of the „test“

Finally run the clients( ) Query again => „test“ client shoulb be gone.

PETER WOLF
< Web Developer & Advisor />

Add the Mutations for PROJECTS

Now we add the addProject method which contains following code.

```
fields: {
  addClient: {type: ClientType...},
  deleteClient: {type: ClientType...},
  addProject: {
    type: ProjectType,
    args: {
      name: {type: GraphQLNonNull(GraphQLString)},
      description: {type: GraphQLNonNull(GraphQLString)},
      status: {
        type: new GraphQLEnumType( config: {
          name: 'ProjectStatus',
          description: {
            type: GraphQLNonNull(GraphQLString)
          },
          values: {
            "new": {value: 'Not Started'},
            'progress': {value: 'In Progress'},
            'completed': {value: 'Completed'}
          }
        }),
        defaultValue: 'Not Started',
      },
      clientId: {type: GraphQLNonNull(GraphQLID)},
    },
    resolve(parent : TSource , args : {[p: string]: any} ) {
      const project : HydratedDocument<InferSchemaType  = new Project( doc: {
        name: args.name,
        description: args.description,
        status: args.status,
        clientId: args.clientId,
      });

      return project.save();
    }
  }
```

PETER WOLF
< Web Developer & Advisor />

Now we make the mutation query call.

```
mutation{
  addProject(name:"mobile app",
    clientId: "660b8ce2f809b4eecbdb9c94",
    description:"new project description",
    status:new){
    name,
    id
  }
}
```

```
{
  "data": {
    "addProject": {
      "name": "mobile app",
      "id": "660f752db389f8384a469f9d"
    }
  }
}
```

After that, we can check the project with data relation.

```
{
  projects{
    name,
    status,
    client{
      id,
      name
    }
  }
}
```

```
{
  "data": {
    "projects": [
      {
        "name": "mobile app",
        "status": "Not Started",
        "client": {
          "id": "660b8ce2f809b4eecbdb9c94",
          "name": "tony Stark"
        }
      }
    ]
  }
}
```

PETER WOLF
< Web Developer & Advisor />

Add DELETE PROJECT Mutation

```
    deleteProject: {
      type: ProjectType,
      args: {id: {type: GraphQLNonNull(GraphQLID)}},
      resolve(parent : TSource , args : {[p: string]: any} ) : Query<..., ..., unknown, any, "findOneAndDelete">  {
        return Project.findByIdAndDelete(args.id);
      }
    }
  }
})
```

Now we do the Delete Query

```
mutation{
  deleteProject(id: "660f76a388ef92eedffda1f8"){
    name
  }
}
```

```
{
  "data": {
    "deleteProject": {
      "name": "second app"
    }
  }
}
```

finally check the list of projects

it is now without the removed project.

PETER WOLF
< Web Developer & Advisor />

Add Update Method mutation

```
    deleteProject: {type: ProjectType...},
    updateProject: {
      type: ProjectType,
      args: {
        id: {type: GraphQLNonNull(GraphQLID)},
        name: {type: GraphQLString},
        description: {type: GraphQLString},
        status: {
          type: new GraphQLEnumType( config: {
            name: 'ProjectStatusUpdate', //has to be unique value
            description: {
              type: GraphQLNonNull(GraphQLString)
            },
            values: {
              "new": {value: 'Not Started'},
              'progress': {value: 'In Progress'},
              'completed': {value: 'Completed'}
            }
          }),
        },
      },
      resolve(parent : TSource , args : {[p: string]: any} ) : Query<..., ..., unknown, any, "findOneAndUpdate">  {
        return Project.findByIdAndUpdate(args.id,  update: {
          name: args.name,
          description: args.description,
          status: args.status,
        },  options: {new: true});
      }
    }
  }
})
```

Now we can update a project

```
mutation{
  updateProject(id: "660f752db389f8384a469f9d",
    name: "updated name",
    description: "updated discription"){
    name,
    id
  }
}
```

```
{
  "data": {
    "updateProject": {
      "name": "updated name",
      "id": "660f752db389f8384a469f9d"
    }
  }
}
```

DONT FORGET THE APP.USE(CORS()) to add…

FINALLY OUR API IS DONE

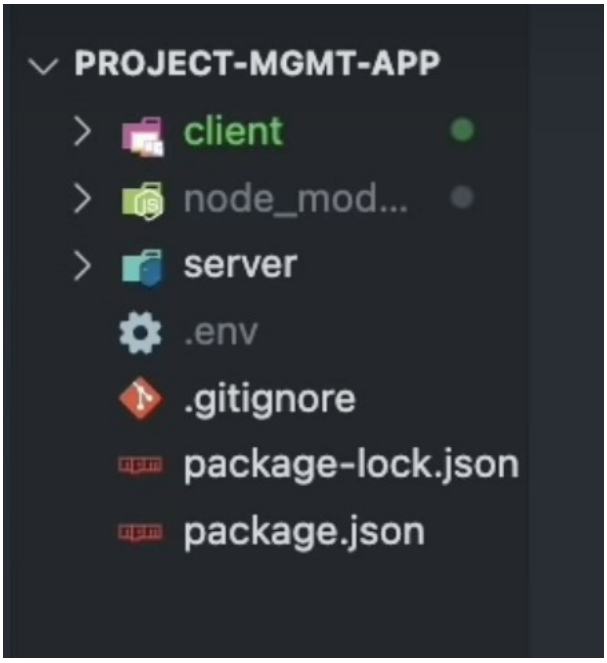
Now lets move to the frontent

Start a new react app

We will run the react app => CLIENT in the main folder of this project

Go to Project folder and run

npm create vite@latest => project name is „client“



Add some needed dependencies

Move to the client folder and run this command

```
npm i @apollo/client graphql react-router-dom react-icons
```

PETER WOLF
< Web Developer & Advisor />

Apolo client and graphql are our main packages to connect the graphql with the frontend!

My actual package.json looks like this

```
{
  "name": "client",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "lint": "eslint . --ext ts,tsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "@apollo/client": "^3.9.10",
    "graphql": "^16.8.1",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-icons": "^5.0.1",
    "react-router-dom": "^6.22.3"
  },
  "devDependencies": {
    "@types/react": "^18.2.66",
    "@types/react-dom": "^18.2.22",
    "@typescript-eslint/eslint-plugin": "^7.2.0",
    "@typescript-eslint/parser": "^7.2.0",
    "@vitejs/plugin-react": "^4.2.1",
    "eslint": "^8.57.0",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.4.6",
    "typescript": "^5.2.2",
    "vite": "^5.2.0"
  }
}
```

PETER WOLF
< Web Developer & Advisor />

Start with some general components

- Header
- Footer
- MainContent

Add the components in the App.js file for now.....

```
graphql-crash-course ~/PhpstormProjects/graphql-crash-co
  client
    node_modules library root
    public
    src
      assets
      components
        Footer.tsx 12.04.24, 13:39, 81 B 2 minutes ago
        Header.tsx 12.04.24, 13:39, 93 B 3 minutes ago
        MainContent.tsx 12.04.24, 13:40, 125 B Moments ago
      App.css 05.04.24, 06:10, 606 B
      App.tsx 12.04.24, 13:40, 303 B Moments ago
      index.css 05.04.24, 06:10, 1,16 kB
      main.tsx 05.04.24, 06:10, 236 B
      vite-env.d.ts 05.04.24, 06:10, 38 B
    .eslintrc.cjs 05.04.24, 06:10, 436 B
    .gitignore 05.04.24, 06:10, 253 B
    index.html 05.04.24, 06:10, 366 B
    package.json 05.04.24, 06:13, 864 B 5 minutes ago
    package-lock.json 05.04.24, 06:13, 126,35 kB
    README.md 05.04.24, 06:10, 1,3 kB
    tsconfig.json 05.04.24, 06:10, 605 B
    tsconfig.node.json 05.04.24, 06:10, 233 B
    vite.config.ts 05.04.24, 06:10, 163 B
  node_modules library root
  server
  .env 01.04.24, 08:15, 171 B
  .gitignore 30.03.24, 17:48, 19 B 7 minutes ago
  package.json 30.03.24, 16:57, 494 B
  package-lock.json 30.03.24, 16:49, 50,92 kB
External Libraries
Scratches and Consoles
```

```
1
2  import './App.css'
3  import {Header} from "./components/Header.tsx";
4  import {Footer} from "./components/Footer.tsx";
5  import MainContent from "./components/MainContent.tsx";
6
   Show usages  aviareps_pwolf *
7  function App() : JSX.Element  {
8
9
10   return (
11       <>
12     <Header/>
13       <MainContent/>
14     <Footer/>
15       </>
16
17   )
18 }
19
   Show usages  aviareps_pwolf
20 export default App
21
```

<MainContent> is a component, which holds children

```
1  import React from "react";
2
   Show usages  new *
3  const MainContent = ({children}: { children: React.ReactNode }) => {
4      return <div style={ {minHeight: "100vh", height: "auto"} }>{ children } </div>
5  }
6
   Show usages  new *
7  export default MainContent
```

That is my starting point!!!

PETER WOLF
< Web Developer & Advisor />

Setup Apolo and graphql for the project.

Now we need to start with the apolo client to provide the function of the module to the application (APP.TSX)

```
import "./App.css"
import {Header} from "./components/Header.tsx";
import {Footer} from "./components/Footer.tsx";
import MainContent from "./components/MainContent.tsx";

import {ApolloClient, ApolloProvider, InMemoryCache} from "@apollo/client";
import Clients from "./components/Clients";

Show usages   aviareps_pwolf *
function App() : JSX.Element  {

    const client : ApolloClient<NormalizedCacheOb...  = new ApolloClient( options: {
        uri: "http://localhost:5001/graphql",
        cache: new InMemoryCache()
    });
    return (
        <ApolloProvider client={ client }>
            <Header/>
            <MainContent>
                <Clients/>
            </MainContent>
            <Footer/>
        </ApolloProvider>

    )
}

Show usages   aviareps_pwolf
export default App
```

We did this:

import Apolo dependecies to manage the Apolo availablilty for the application **inMemoryCache** has the benefit, if there is a reload of the data in the frontend, it shows the new list right away.

PETER WOLF
< Web Developer & Advisor />

The client has two properties:

- uri => source where the server calls are happening in the backend
- cache => enables the cache handling

Fetch clients!

Let us create the Clients component

First we need to import some components from apolo module:

1. gpl => make the actual for the query
2. useQuery => handles the query with loading states for the component

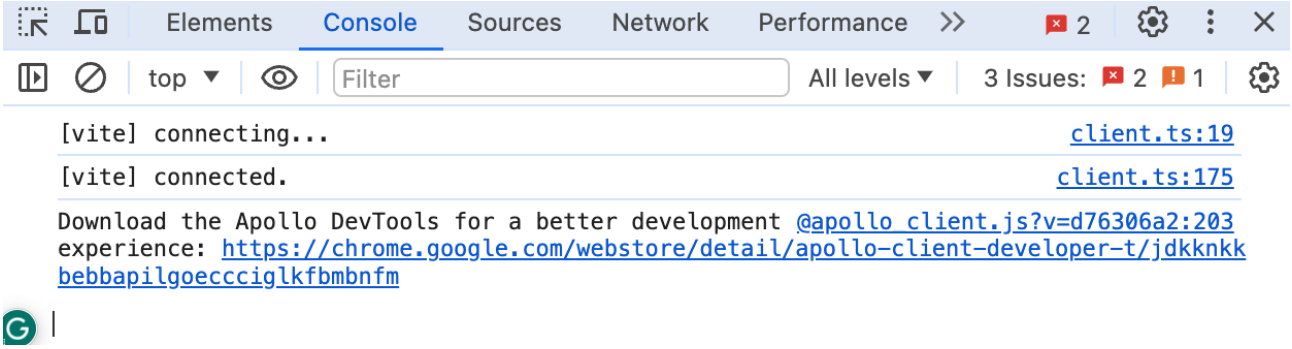
```
1  import {gql, useQuery} from "@apollo/client";
2
3  //the code below is the same as we used to make the query in the graphiql
4  const GET_CLIENTS : DocumentNode  = gql`
5  query getClients {
6  clients{
7  id,
8  name,
9  email,
10 phone}
11 }`
   Show usages  new *
12 const Clients = () : Element  => {
13 const {data, loading : boolean , error : ApolloError | undefined } = useQuery(GET_CLIENTS)
14
15     if (loading) return <div>Loading...</div>;
16     if (error) return <div>Error</div>;
17
18     return <>{!loading && !error &&<div>{ JSON.stringify(data) }</div>}</>
19 }
20
   Show usages  new *
21 export default Clients
```

line 4–11 is based on the graphiql query from graphql
line 13 is response for the state management of the fetch....

PETER WOLF
< Web Developer & Advisor />

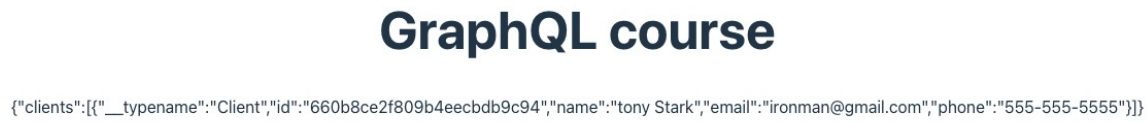
Check the console now….

As you see, this message shows:



unfortunately the DevTools for GraphQL is not really working. Check the reviews…
So we don´t use it.

Visit the page to see the result:



PETER WOLF
< Web Developer & Advisor />

Now we got the data and we can deal with it!

```
Show usages   new *
const Clients = () : Element  => {
    const {data, loading : boolean , error : ApolloError | undefined } = useQuery(GET_CLIENTS)

    if (loading) return <div>Loading...</div>;
    if (error) return <div>Error</div>;

    return <>{ !loading && !error && <div>
        { data?.clients.map((item:{id: string, name: string})  => {
            return <ClientItem
                key={item.id}
                id={item.id}
                name={item.name}/>
        }) }
    </div> }</>
}
```

Clients() › › div › callback for data?.clients.map() › ClientItem#{item.id}

ClientItem.tsx

```
Show usages   new *
const ClientItem=(props:{id: string, name: string})=> {


    return <div style={ {display: "flex", justifyContent: "center", alignItems: "center"} }>
        <h3>{ props.name }</h3>
        <p>delete</p></div>
}

Show usages   new *
export default ClientItem
```

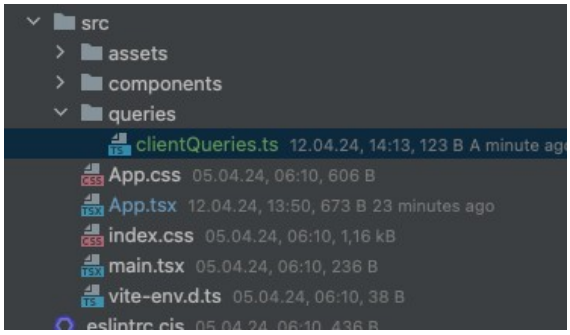
PETER WOLF
< Web Developer & Advisor />

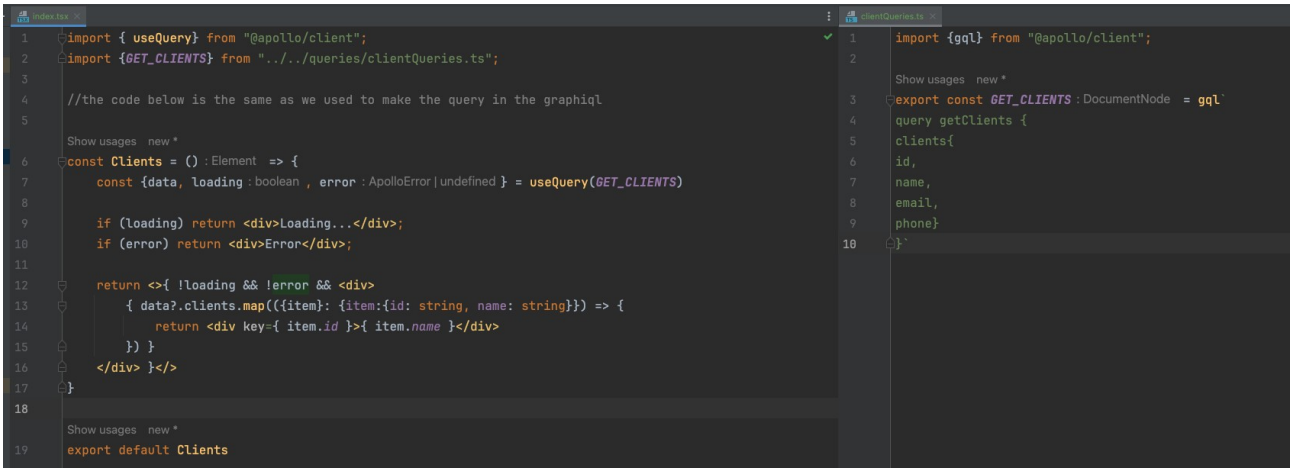
Let us clean up the queries in the component!

Because of a better organisation, we want to put the queries in an seperate file and pick them from there.

Now our project looks like this

folderstructure





```
import { useQuery} from "@apollo/client";
import {GET_CLIENTS} from "../../queries/clientQueries.ts";

//the code below is the same as we used to make the query in the graphiql

Show usages  new *
const Clients = () : Element  => {
    const {data, loading : boolean , error : ApolloError | undefined } = useQuery(GET_CLIENTS)

    if (loading) return <div>Loading...</div>;
    if (error) return <div>Error</div>;

    return <>{ !loading && !error && <div>
        { data?.clients.map(({item}: {item:{id: string, name: string}}) => {
            return <div key={ item.id }>{ item.name }</div>
        }) }
    </div> }</>
}

Show usages  new *
export default Clients
```

```
import {gql} from "@apollo/client";

Show usages  new *
export const GET_CLIENTS : DocumentNode  = gql`
query getClients {
clients{
id,
name,
email,
phone}
}`
```

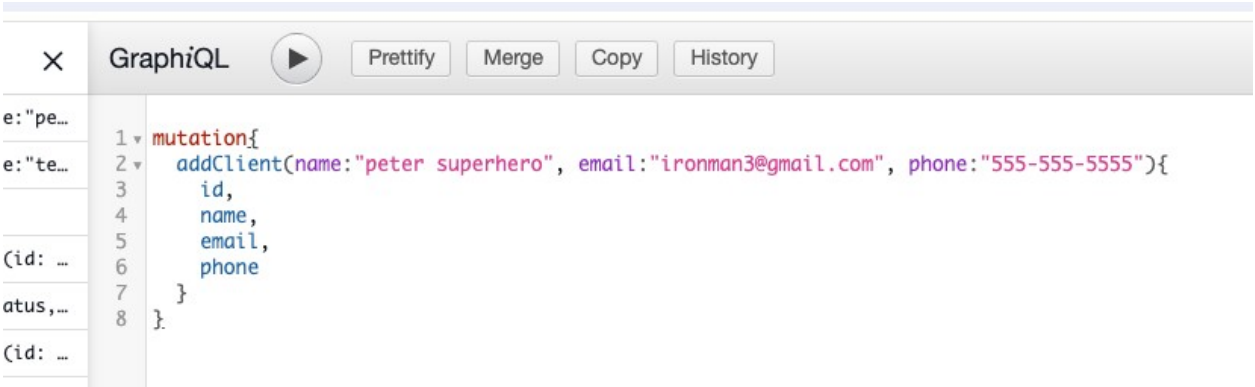
PETER WOLF
< Web Developer & Advisor />

Delete a Client from the frontend

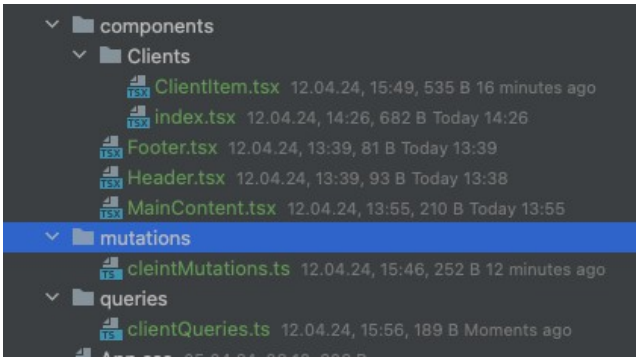
Next we want to be able to delete a client from the frontend

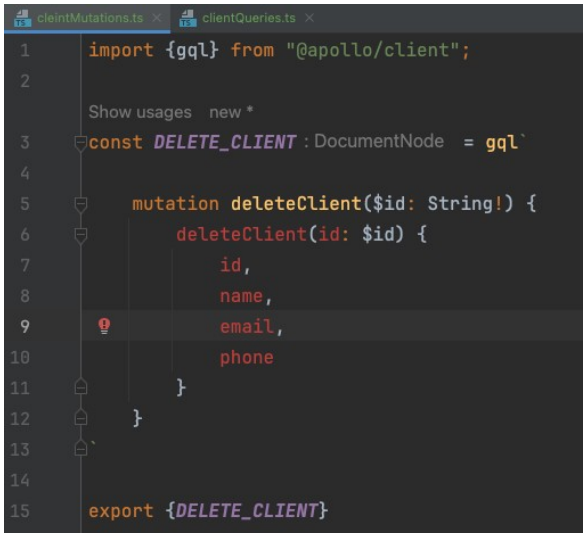
this is a mutation which we will now seperate like queries in our project

before we do that, let us create a new client from the GraphIQL CLI.
(we don´t have a create form yet)



Now we can create the mutations in React Frontend





The error is because of TS Schema…not set up for this exercise

PETER WOLF
< Web Developer & Advisor />

Not refetch after deleteClient

As you can see, the refetching of the new data is not automatically happen.
My Frontend show the old state after I deleted the client.

Fixing this with:
There are two options:

1. Refetching getClients again

2. Updating the cache

Both ways will be shown

1. Refetch the clients

```
const ClientItem=(props:{id: string, name: string})=> {
console.log("props", props)
    const [deleteClient] = useMutation(DELETE_CLIENT, options: {
        variables: {
            id: props.id
        },
        refetchQueries:[{query:GET_CLIENTS}]
    })

    return <div style={ {display: "flex", justifyContent: "center", alignItems: "center"} }>
        <h3>{ props.name }</h3>
        <button onClick={()=>deleteClient()}>delete</button></div>
}

Show usages   new *
export default ClientItem
```

PETER WOLF
< Web Developer & Advisor />

2. Update Cache

```
const ClientItem=(props:{id: string, name: string})=> {
console.log("props", props)
    const [deleteClient] = useMutation(DELETE_CLIENT, options: {
        variables: {
            id: props.id
        },
        update(cache : ApolloCache<any> , {data:{deleteClient}}) : void {
            const {clients} = cache.readQuery( options: {
                query: GET_CLIENTS
            });
            cache.writeQuery( {id, data, ...options}: {
                query: GET_CLIENTS,
                data: {
                    clients: clients.filter(clients => clients.id!== deleteClient.id)
                }
            })
        }
    })
```

TS errors

<u>Additional to the Update Cache</u> option, you might see this error in the console::

```
y({
IENTS,

ients.filter(
lient.id),

MINAL

ing

Download the Apollo DevTools for a better development      bundle.js:87763
experience: https://chrome.google.com/webstore/detail/apollo-client-developer-t/jdkknkkbebbapilgoeccciglkfbmbnfm
▼Cache data may be lost when replacing the  react_devtools_backend.js:4026
clients field of a Query object.

To address this problem (which is not a bug in Apollo Client), define a
custom merge function for the Query.clients field, so InMemoryCache can
safely merge these objects:

  existing: [{"__ref":"Client:629a4978fe9529ef457de1f3"},
{"__ref":"Client:629a49bafe9529ef457de1f5"},
{"__ref":"Client:629b5a56669a5494ca199458"}]
  incoming: [{"__ref":"Client:629a4978fe9529ef457de1f3"},
{"__ref":"Client:629a49bafe9529ef457de1f5"}]

For more information about these options, please refer to the
documentation:

  * Ensuring entity objects have IDs: https://go.apollo.dev/c/generating-unique-identifiers
  * Defining custom merge functions: https://go.apollo.dev/c/merging-non-normalized-objects
```

PETER WOLF
< Web Developer & Advisor />

To solve this, you have to add some code to the App.tsx component

```
5   const cache = new InMemoryCache({
6     typePolicies: {
7       Query: {
8         fields: {
9           clients: {
10            merge(existing, incoming) {
11              return incoming;
12            },
13          },
14          projects: {
15            merge(existing, incoming) {
16              return incoming;
17            },
18          },
19        },
20      },
```

and then update the client const with this code

```
23
24  const client = new ApolloClient({
25    uri: 'http://localhost:5000/graphql',
26    cache,
27  });
```

this should solve the problem

PETER WOLF
< Web Developer & Advisor />

Add a new client form

1. Add a form to the application

ClientItem.tsx | NewClientForm.tsx | App.tsx | schema.js

```
import {useState} from "react";

Show usages  new *
const NewClientForm = () => {

    const [user : {name: string, email: string, ... , setUser : React.Dispatch<React.SetStateA... ] = useState( initialState: {
        name: "",
        email: "",
        phone: ""
    })

Show usages  new *
function handleChange(event:{target:{name: string, value: string}}) : void {
        setUser( value: {...user, [event.target.name]:event.target.value})
}
    Show usages  new *
    function handleSubmit(e: { preventDefault: () => void }) : void  {
        e.preventDefault()
        console.log(user)

    }

    return (<form onSubmit={ handleSubmit }>
        <input type="text" name="name" placeholder={ "name" } onChange={handleChange}/>
        <input type="text" name="email" placeholder={ "email" } onChange={handleChange}/>
        <input type="text" name="phone" placeholder={ "phone" } onChange={handleChange}/>
        <button type="submit">Add Client</button>
    </form>)
}

Show usages  new *
export default NewClientForm
```

PETER WOLF
< Web Developer & Advisor />

2. connect with graphql mutation

first we need to add the mutation

```
cleintMutations.ts   clientQueries.ts

3
4
Show usages   new *
5   const ADD_CLIENT : DocumentNode  =gql`
6       mutation addClient($name: String!, $email: String!, $phone: String!) {
7
8           addClient(name: $name, email: $email, phone: $phone) {
9               id,
10              name,
11              email,
12              phone
13          }
14      }
15  `
16
17
Show usages   new *
18  const DELETE_CLIENT : DocumentNode  = gql`
```

then we import the Add_Client to the form Component

To be able to update the list right away, we need to add as well the GET_CLIENT query to the form component.

PETER WOLF
< Web Developer & Advisor />

Now we need to create the useMutation function in the form

```
ClientItem.tsx   NewClientForm.tsx   App.tsx   schema.js

import {useState} from "react";
import {ADD_CLIENT} from "../mutations/cleintMutations.ts";
import {GET_CLIENTS} from "../queries/clientQueries.ts";
import {useMutation} from "@apollo/client";

Show usages   new *
const NewClientForm = () => {

    const [user : {name: string, email: string, ... , setUser : React.Dispatch<React.SetStateA... ] = useState( initialState: {
        name: "",
        email: "",
        phone: ""
    })

    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    const [addClient]=useMutation(ADD_CLIENT, options: {
        variables:{
            name: user.name,
            email: user.email,
            phone: user.phone
        },
        update(cache : ApolloCache<any> , {data:{addClient}}) : void {
            // eslint-disable-next-line @typescript-eslint/ban-ts-comment
            // @ts-expect-error
            const {clients} = cache.readQuery( options: {
                query: GET_CLIENTS
            });
            cache.writeQuery( {id, data, ...options}: {
                query: GET_CLIENTS,
                data: {
                    clients: [...clients, addClient]
                }
            })
        }
    })
```

PETER WOLF
< Web Developer & Advisor />

3. configure now the submit handler

```
Show usages   new *
function handleSubmit(e: { preventDefault: () => void }) : void  {
    e.preventDefault()
    if (user.name === "", user.email === "", user.phone === "") {
        alert("Please enter values to the fields")
    }

    addClient().then(res : FetchResult<any>  => setResponse(res.data))
    setUser( value: {
        name: "",
        email: "",
        phone: ""
    })
}
```

finally I provide a screen message to inform about the new submission => optional

```
eturn <><form onSubmit={ handleSubmit }>
    <input type="text" name="name" placeholder={ "name" } onChange={ handleChange }/>
    <input type="text" name="email" placeholder={ "email" } onChange={ handleChange }/>
    <input type="text" name="phone" placeholder={ "phone" } onChange={ handleChange }/>
    <button type="submit">Add Client</button>
</form>

{response && <p>new user has been added {response ? response.addClient?.name : "null"}</p
    </>


usages   new *
t default NewClientForm
```

PETER WOLF
< Web Developer & Advisor />

add the mutation to the form

graphql-session > client > src > components > NewClientForm.tsx > NewClientForm

```
import { useMutation } from '@apollo/client'
import {useState} from 'react'
import { ADD_CLIENT } from '../graphql/mutations/clientMutation'
import { GET_CLIENTS } from '../graphql/queries/clientQueries'

const NewClientForm = () => {

    const [user, setUser] = useState({
        name: "",
        phone: "",
        email: ""
    })

    const [addClient] = useMutation(ADD_CLIENT,{
        variables:{
            name: user.name,
            phone: user.phone,
            email: user.email
        },
        refetchQueries:[{query: GET_CLIENTS}]
    })
```

graphql-session > client > src > components > NewClientForm.tsx > NewClientForm > handleSubmit

```
const NewClientForm = () => {



    function handleChange(e:{target:{name: string, value: string}}) {
        setUser({ ...user, [e.target.name]: e.target.value })
    }

    function handleSubmit(e:{preventDefault: ()=>void}){
        e.preventDefault()
        addClient()
    }

    return (
        <form onSubmit={handleSubmit}>
            <input type="text" name="name" placeholder='name' onChange={handleChange} />
            <input type="text" name="email" placeholder='email' onChange={handleChange} />
            <input type="text" name="phone" placeholder='phone' onChange={handleChange} />
            <button type="submit">Add new user</button>
        </form>
    )
}

export default NewClientForm
```