1. Use **npm init** to initialize package.json
2. Install apollo-server to create a graphql server:

**npm install apollo-server**

1. Install graphql to write graphql schema and types:  
   **npm install graphql**
2. We can install nodemon while working on our server so that we don’t have to repeatedly stop-start our server after making any changes.

**npm install nodemon**

**Note:** setup the package.json such a way that on npm start it runs the root server file.

"scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "start": "nodemon index.js"

  },

Here, we have created index.js as the entry point of our server.

1. We can now import ApolloServer class and pass typeDefs and resolvers as constructor parameter to it.

const { ApolloServer } = require('apollo-server');

const server = new ApolloServer({

typeDefs, resolvers

});

Here, typeDefs refers to all the types of definition that we have defined. Resolvers are nothing but a function which tries to resolve the type. One of the ways to resolve types is by making an API call.

1. We can now use the server variable to listen to our server at a particular port.

//default port configured in apollo-server is 4000

server.listen().then((url) => {

console.log(`Your URL is running on port: ${url.port}`)

})

**Note:** This will fail for now since we haven’t created any typeDefs and resolvers.

1. We can now create a ~/schema/type-defs.js file where we can define the types.

const { gql } = require('apollo-server');

const typeDefs = gql`

type User {

id: ID!

name: String!

username: String!

age: Int!

nationality: String!

}

type Query {

users: [User!]!

}

`;

module.exports = {

typeDefs

};

Gql is required to define our types. Here type User is a custom type which we will use to fire a query under type Query.

It will return the list of users where the list and the list of users both can’t be null.

1. Creating only types won’t let us query the schema, we need a resolver function as well to return the data back to our client (typically Apollo-Client). Create ~/schema/resolver.js which will look like:

const { UserList } = require('./FakeData');

const resolvers = {

Query: {

users: () => {

return UserList

}

}

}

module.exports = {

resolvers

}

The Query property means that we are trying to get the data, typically the way we get the data by making some REST API GET call.

Since the type of users is a list of Users with type User (as defined in the type-def.js file) we will return UserList which is a fake data generated online. Instead of this we can make API call as well or query data from DB. For simplification, we have used fake data.

1. The resolver contains following parameters in respective order:

**parent:** It gets the data from its immediate parent. It will return **undefined** if it is not a nested query.

**args:** It is an object which contains all the parameter passed to the resolver function.

**context:** It is one of the important parameters, it could be used for authentication/authorization. It carries some values and could be accessed by all the resolvers. To define these values, we need to add context while creating our ApolloServer.

const server = new ApolloServer({

typeDefs, resolvers, context: ({req}) => {

return { name: 'Roshan Shah' }

}

})

Here, **context** is a function that returns an object which can be accessed by all the resolvers. We also get access to the request that was made as an object parameter to the context function.

**info:** It returns meta data of the graphql query like if the request is a query or a mutation etc.

1. Run **npm run start** to see if everything works fine.

1. The application would be running on <http://localhost:4000>. You can now click on ***Query your server***, it will redirect to <https://studio.apollographql.com/sandbox/explorer>

Here we can make query to test our graphql server.

query GetAllUsers {

users {

id

age

}

}

**query** is a keyword used to query our graphql server, GetAllUsers is just a name given to the query. Users is the resolver function, whereas Id and age is the property which we are querying from the server.

1. We can validate our data by using enum. It will throw an error if the value is not one of the values in enum. We can define enum as:

enum Nationality {

ITALY

CHINA

POLAND

GUAM

ARGENTINA

AUSTRALIA

PORTUGAL

}

1. We can now remove the type String from country property defined in type User.

type User {

id: ID!

name: String!

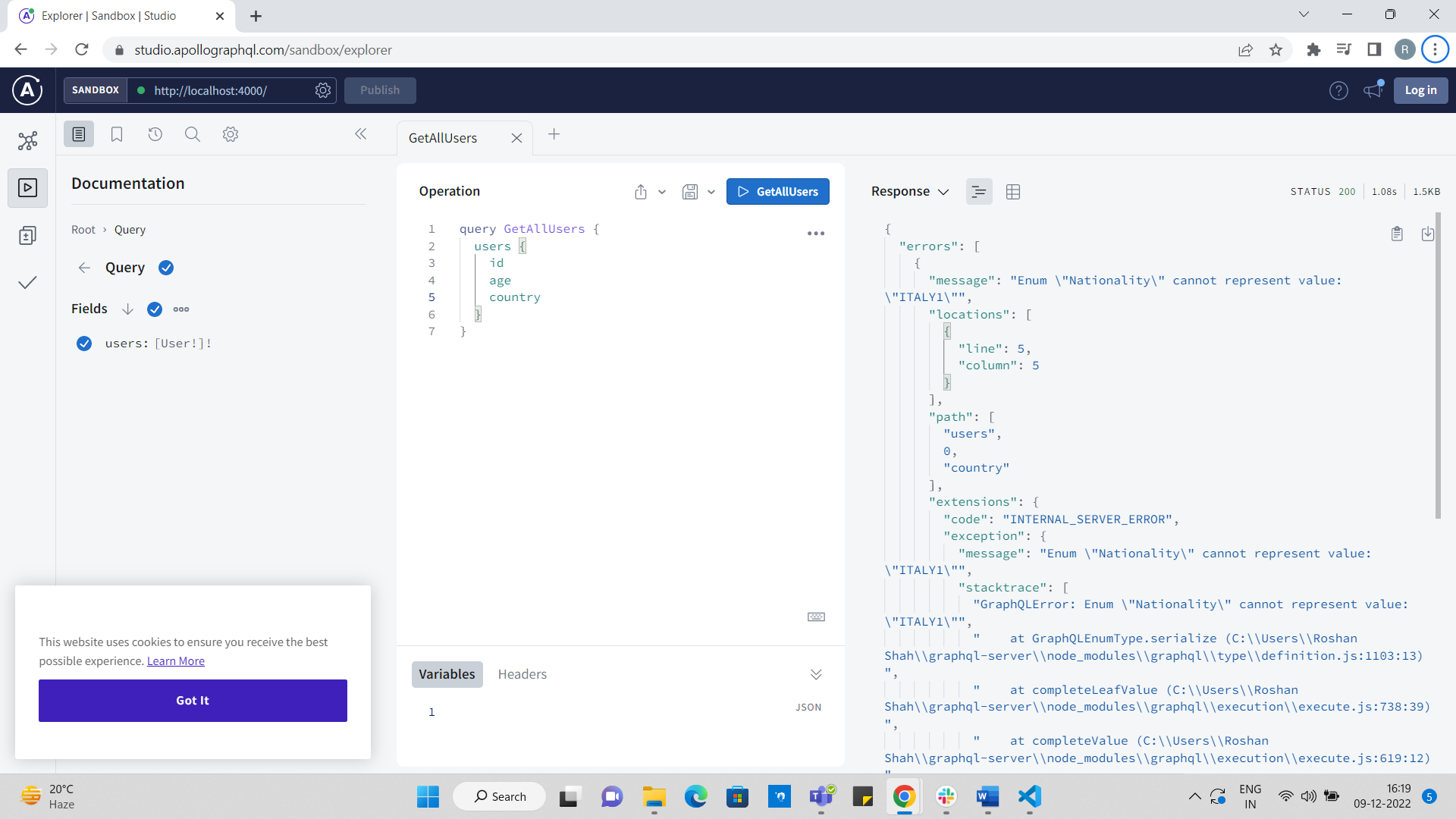
username: String!

age: Int!

country: Nationality!

}

1. Now if we make any change in the country field of fake data, it will throw an error something like this



1. Enums helps to validate the data on the server level itself so that we don’t have to validate the same on the client side.
2. We can use the current defining type as a type to one of our properties.

type User {

id: ID!

name: String!

username: String!

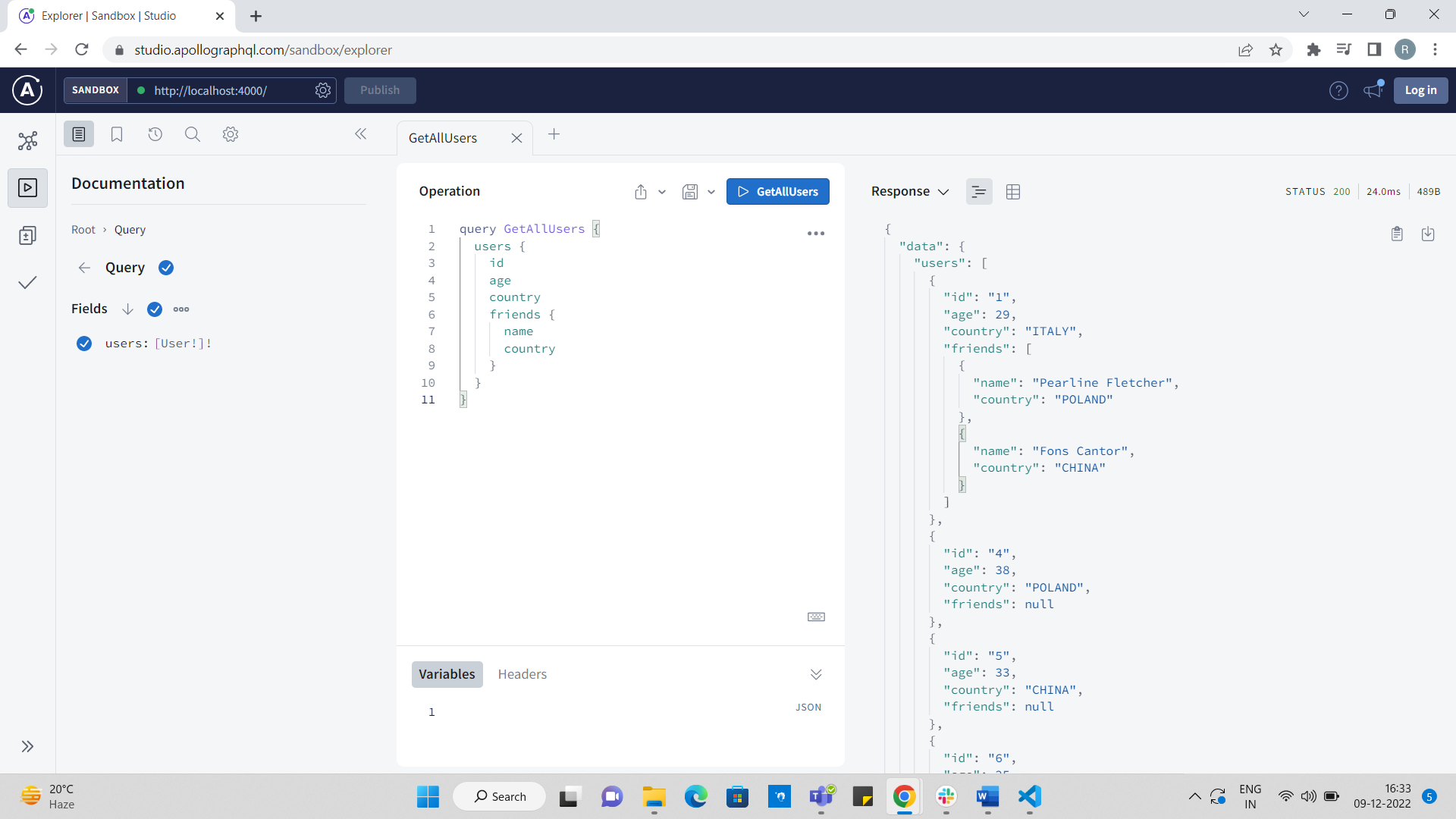
age: Int!

country: Nationality!

friends: [User]

}

Here, friends’ property defines a type of list, which is of type User. Thus, we can assign the type to one of its properties while defining the type itself.



1. If we want to get a result based on a parameter, we can define another query under type Query and accept some parameter which will fetch data of a particular type.

type Query {

users: [User!]!

user(id: ID!): User!

}

Here, we are trying to access a single user based on their Id. Also, we are expecting that the returned value shouldn’t be null.

1. If we have added a query, it means that we need to add a resolver function as well to return the request data.

**Note:** Both the query type name and resolver name should be same.

const resolvers = {

Query: {

users: () => {

return UserList

},

user: (\_, args) => {

const { id } = args;

return UserList.filter(el => el.id == Number(id))[0] // we can make use of find function of loadash

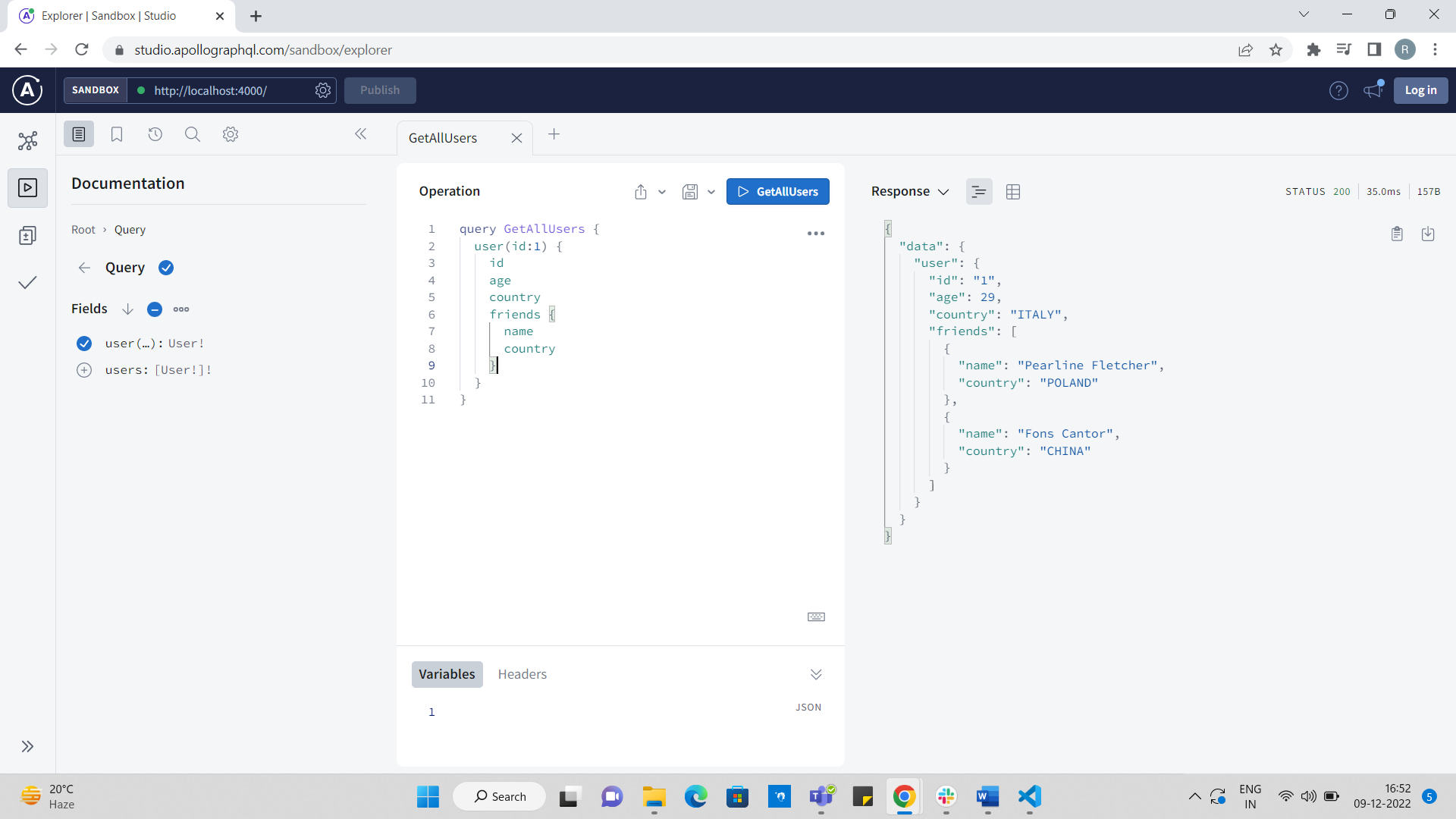
}

}

}

user function takes first parameter as parent and then args as its second argument. Since we were passing Id, we can access it from the args object.

We can query the data in following way:



1. We can nest two different types to get a multiple type data using a single query.

type Movie {

id: ID!

name: String!

yearOfRelease: Int!

isInTheater: Boolean!

}

type Query {

users: [User!]!

user(id: ID!): User!

movies: [Movie!]!

movie(name: String!): Movie!

}

Here, we have created a type Movie and added respective query for it.

1. Below query tries to get different type data from a different location/type and for this kind of scenario we would be using our custom object which has the same name as the custom type

type User {

id: ID!

name: String!

username: String!

age: Int!

country: Nationality!

friends: [User!]

favoriteMovies:[Movie]

}

We are trying to fetch favourite movie data without keeping it in the user fake data. So, you can think of it like a different db/table/api is having some data but since we need to make a single query to fetch both and club it into a single data response, we will create an in our resolver which will be having same name as our custom type defined in type-defs.

In resolver we will add below object:  
 User: {

favouriteMovies: () => {

return MovieList.filter(el => el.yearOfRelease >= 2000 && el.yearOfRelease <= 2010)

}

}

**Note:** The above returned data is just for understanding.

Graphical user interface, application

Description automatically generated

The above screenshot shows the result that we get after executing the above query. We can see that now we are able to get favourite movie as well without adding the data in fake user data. Hence, we can merge data coming from different location by using a single query.

1. Mutation is used to perform add, update or delete operation much like POST, PUT and DELETE operation of REST API.
2. To create a mutation, we need to add mutation object like the query object and place all the functions inside it.

input CreateUserInput {

name: String!

username: String!

age: Int!

country: Nationality = INDIA

}

type Mutation {

createuser(input: CreateUserInput!): User!

}

We are creating an input type so that we don’t have to pass it separately one by one in the mutation function, plus, if we want to make any field mandatory or optional or pass a default value, we can do it which will make it more standard way to pass parameter to a function.

We are returning the User type since it is a standard in graphql mutation to return the same type of data which is being passed in the function as parameters.

We need to add a resolver function which will have the same name as the function name defined in the schema.

Mutation: {

createUser: (parent, args) => {

const { input } = args;

const lastId = UserList[UserList.length - 1].id;

input.id = lastId + 1;

UserList.push(input)

return input;

}

},

createUser is a function which takes same parameter as the query function takes i.e parent and argument.

In the above function we are returning the user data after performing some actions since the output type defined in schema of the function is of type User.