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. 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!

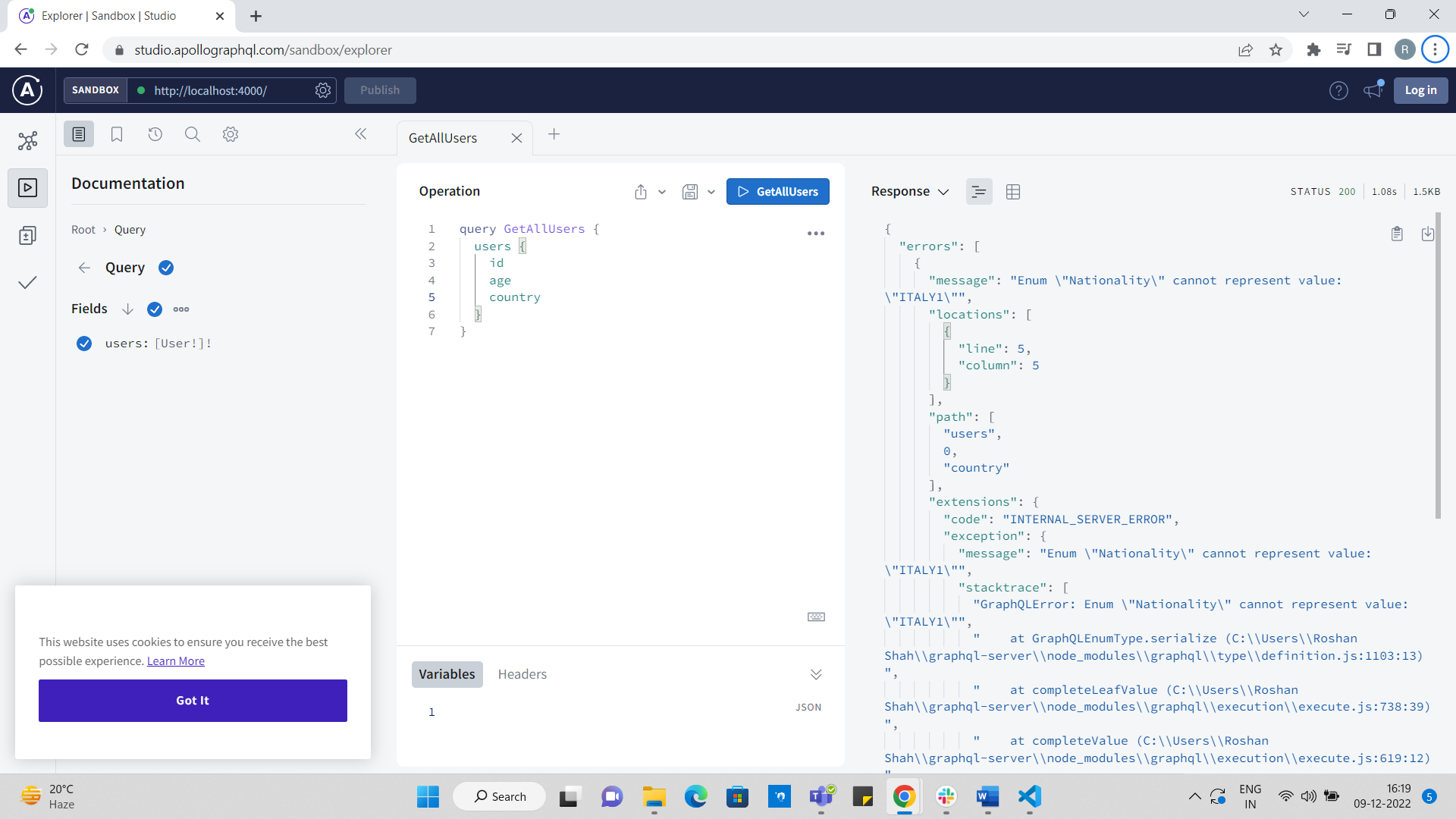
username: String!

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!

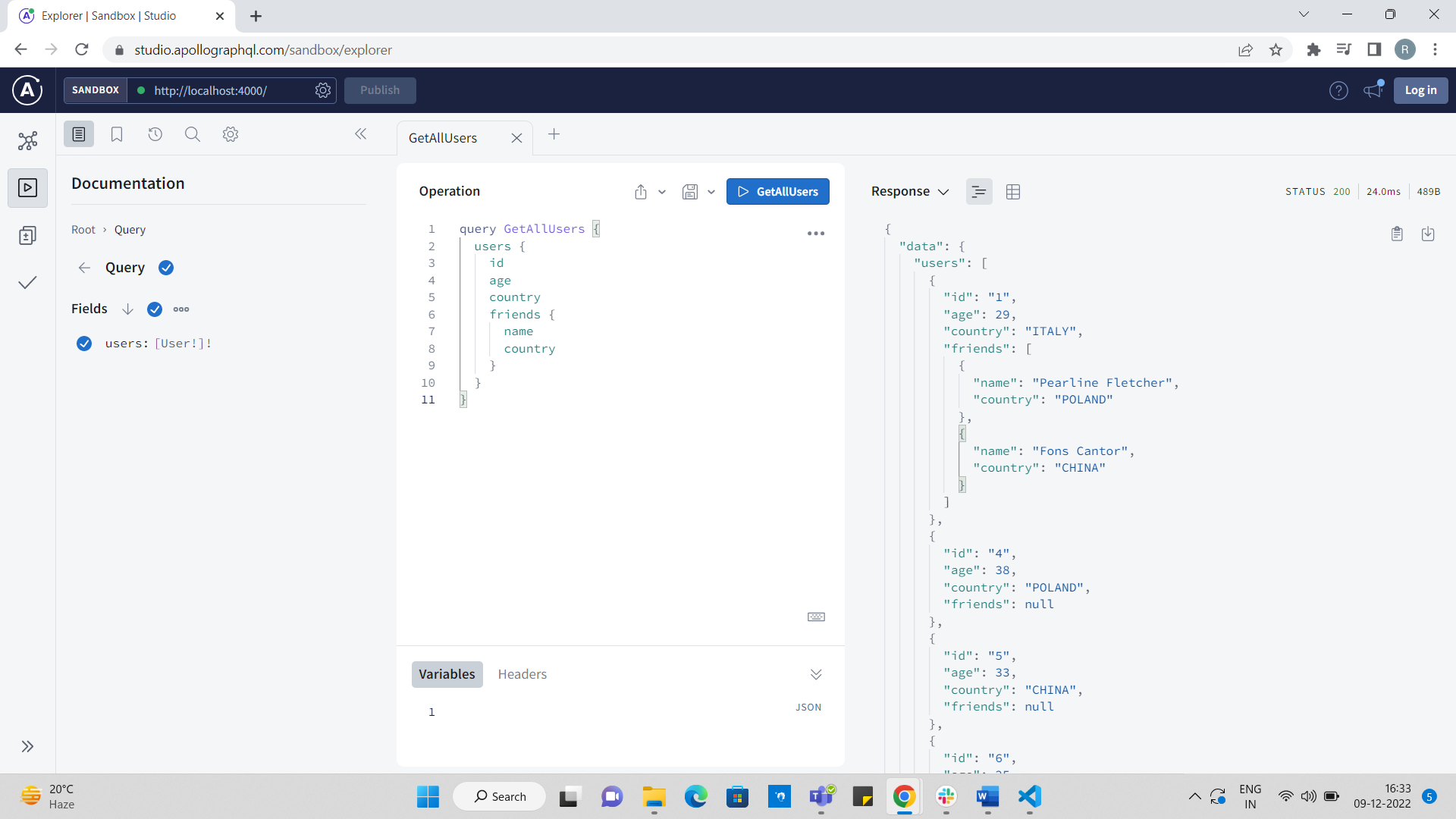
age: Int!

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 == id)[0]

}

}

}

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:

