# Instalar

* yarn add graphql

# Apollo

<https://www.apollographql.com/docs/apollo-server/getting-started>

* yarn add @apollo/server
* yarn add --save-dev typescript @types/node

crear el tsconfig.json para las configuraciones de TS

{

    "compilerOptions": {

      "rootDirs": ["src"],

      "outDir": "dist",

      "lib": ["es2020"],

      "target": "es2020",

      "module": "esnext",

      "moduleResolution": "node",

      "esModuleInterop": true,

      "types": ["node"]

    }

  }

En package.json hay que agregar

{

  …

  "type":"module",

"scripts": {

   ,"compile":"tsc"

    ,"start":"npm run compile && node ./dist/index.js"

  },

  …

}

## Errores

<https://www.apollographql.com/docs/apollo-server/data/errors>

import { ApolloServerErrorCode } from '@apollo/server/errors';

import { GraphQLError } from 'graphql';

//dentro de algún resolver

throw new GraphQLError(“messageError”, {

                    extensions: {

                        code: 'YOUR\_ERROR\_CODE'

                        , myCustomExtensions // opcional

                    },

                  });

### Códigos de error

|  |  |
| --- | --- |
| **CODE** | **DESCRIPTION** |
| [GRAPHQL\_PARSE\_FAILED](https://www.apollographql.com/docs/apollo-server/data/errors/#graphql_parse_failed) | The GraphQL operation string contains a syntax error. |
| [GRAPHQL\_VALIDATION\_FAILED](https://www.apollographql.com/docs/apollo-server/data/errors/#graphql_validation_failed) | The GraphQL operation is not valid against the server's schema. |
| [BAD\_USER\_INPUT](https://www.apollographql.com/docs/apollo-server/data/errors/#bad_user_input) | The GraphQL operation includes an invalid value for a field argument. |
| [PERSISTED\_QUERY\_NOT\_FOUND](https://www.apollographql.com/docs/apollo-server/data/errors/#persisted_query_not_found) | A client sent the hash of a query string to execute via [automatic persisted queries](https://www.apollographql.com/docs/apollo-server/performance/apq), but the query was not in the APQ cache. |
| [PERSISTED\_QUERY\_NOT\_SUPPORTED](https://www.apollographql.com/docs/apollo-server/data/errors/#persisted_query_not_supported) | A client sent the hash of a query string to execute via [automatic persisted queries](https://www.apollographql.com/docs/apollo-server/performance/apq), but the server has disabled APQ. |
| [OPERATION\_RESOLUTION\_FAILURE](https://www.apollographql.com/docs/apollo-server/data/errors/#operation_resolution_failure) | The request was parsed successfully and is valid against the server's schema, but the server couldn't resolve which operation to run.  This occurs when a request containing multiple named operations doesn't specify which operation to run (i.e.,operationName), or if the named operation isn't included in the request. |
| [BAD\_REQUEST](https://www.apollographql.com/docs/apollo-server/data/errors/#bad_request) | An error occurred before your server could attempt to parse the given GraphQL operation. |
| [INTERNAL\_SERVER\_ERROR](https://www.apollographql.com/docs/apollo-server/data/errors/#internal_server_error) | An unspecified error occurred.  When Apollo Server formats an error in a response, it sets the code extension to this value if no other code is set. |

### Respuestas a un error

{

"data": {

"addTipo1": null

},

"errors": [

{

"message": "el mensaje",

"locations": [

{

"line": 2,

"column": 3

}

],

"path": [

"addTipo1"

],

"extensions": {

"code": "GRAPHQL\_PARSE\_FAILED",

"stacktrace": [

"GraphQLError: el mensaje",

" at Object.addTipo1 (file:///D:/Programacion/Proyectos/GraphQL/Proyecto1/dist/index.js:52:23)",

….

]

}

}

]

}

# Componentes principales de un proyecto

* origen de datos
* definición de tipos
* Solucionadores
* Mutaciones
* Suscripciones

# Definir tipos de datos

const typeDefs=`

    type TipoDeDato {

atributoId: ID!

        atributo1: Tipo @deprecated(reason: "mensaje del porque")

        atributo2: [Tipo]

    }

type Query {

        nombreQueryLista:[TipoDeDato1]

nombreQueryGet(atributoId: ID!):TipoDeDato1

nombreQueryGet2(

atributoId: ID!

atributo2: String

):TipoDeDato1

    }

type Mutation {

        addTipo1(

            atributo1:String!

            atributo2:String!

        ):TipoDeDato1

    }

type Subcription {

        onAddTipo1: TipoDeDato1

    }

`

El tipo puede ser String, un arreglo seria [Tipo], un id seria ID,

Un tipo boligatorio seria Tipo!

## Definir querys querys

Hay que agregarle siempre al final el tipo Query que define como van a ser nuestras consultas, estas consultas son de tipo de obtención, osea no realizan cambios en los datos

Para habilitar una lista de un tipo

nombreQueryLista:[TipoDeDato1]

Para una consulta que solo retorne un solo elemento y pida algún argumento (tipo un búsqueda de un elemento por su id)

nombreQueryGet(atributoId: ID!):TipoDeDato1

## definir mutaciones querys

Las mutaciones son querys que de alguna formar van a modificar los datos

type Mutation {

        addTipo1(

            atributo1:String!

            atributo2:String!

        ):TipoDeDato1

    }

## Definir Suscripciones

Son eventos que se llama cuando cambia algo

type Subcription {

        onAddTipo1: TipoDeDato1

    }

## Decoradores

### Definir depreciaciones (deprecated)

Luego de la declaración de un atributo se le agrega

 @deprecated(reason: "mensaje del porque")

Para indicar que este atributo, aunque esta disponible ya esta obsoleto

### Excluir (skip)

Si se desea no incluir un atributo de la respuesta si se cumple alguna condición, se le pone al final de este

@skip(if: true)

### Incluir (include)

Si lo que se desea es incluir un atributo solo si se cumple una condición

@include(if: true)

# Definir Resolvers

import { PubSub } from "graphql-subscriptions";

const pubsub=new PubSub();

const resolvers ={

TipoDeDato1:{

        atributoCreado:(parent,args,context,info)=>{

            return parent.atributo1+" - "+parent.atributo2;

        }

    }

    ,Query:{

// consultar un bd y retornar esto

        tipos1:()=>datosDeTipos1,

tipo:(parent,args,context,info)=>{

            const id=parseInt(args.id);

            return datosDeTipos1.find((dato)=>dato.id===id);

        }

    }

,Subscription:{

        onAddTipo1: {

            subscribe: () => pubsub.asyncIterator(['NOMBRE\_EVENTO'])

        }//(parent, args, context, info) => pubsub.asyncIterator('onAddTipo1')

    }

,Mutation:{

        addTipo1:(parent,args,context,info)=>{

            const tipo1={...args,id:1};

            datosDeTipos1.push(tipo1);

            return tipo1;

        }

,editTipo1:(parent,args,context,info)=>{

            //editar el tipo 1

            pubsub.publish('NOMBRE\_EVENTO',{

                "atributoEvento1":"valor1"

                ,"atributoEvento2":"valor2"

            });

            return args;

        }

    }

}

De la manera anterior es si la forma en que están definidos los datos en los typeDefs es igual a como están en los origines de los datos (datosDeTipos1)

En estos es donde se realizan las búsquedas por querys, básicamente hay que implementar cada query

## mutaciones

Al igual que las querys aquí también se declaran como se definen las mutaciones

## Definir atributos tipos de datos

Se pueden también definir como se crean los atributos de un tipo de dato

De un TipoDeDato1 aquí se van declarando como resolver sus atributos personalizados (estos atributos personalizados deben estar declarados en el typeDefs)

TipoDeDato1:{

        atributoCreado:(parent,args,context,info)=>{

            return parent.atributo1+" - "+parent.atributo2;

        }

    }

## Subscriptions

<https://www.apollographql.com/docs/apollo-server/data/subscriptions/>

import { PubSub } from "graphql-subscriptions";

const pubsub=new PubSub();

,Subscription:{

        onAddTipo1: {

            subscribe: () => pubsub.asyncIterator(['NOMBRE\_EVENTO'])

        }

    }

Y dentro de alguna mutación llamar a este evento pasándole algunos argumentos para transmitir información

,Mutation:{

,editTipo1:(parent,args,context,info)=>{

            //editar el tipo 1

            pubsub.publish('NOMBRE\_EVENTO',{

                "atributoEvento1":"valor1"

                ,"atributoEvento2":"valor2"

            });

            return args;

        }

    }

# Declarar el servidor

import { ApolloServer } from "@apollo/server"

import { startStandaloneServer} from '@apollo/server/standalone'

const server =new ApolloServer({

    typeDefs,resolvers

});

const { url } =await startStandaloneServer(server,{

    listen:{port:4000}

})

Va ha estar en http://localhost:4000/

# Lenguaje Petición

Se pueden pedir tipos de datos(en la forma de atributos definidos dentro de resolvers en Query ) ya definidos y de estos atributos ya definidos

## Para pedir una lista

query{

tipos1 {

atributo1

atributo2

}

}

## Para pedir un solo elemento por su id

query  nombreQurey {

  tipo(atributoId:1) {

    atributo1

    atributo2

  }

}

## Variables en querys

Para ponerle variables a las querys para poder usarlas después es declarándolas antes del cuerpo {} de la query y después del nombre si tiene $nombreVariable: Tipo

query qurey($variable:ID!) {

  tipo(atributoId:$variable) {

    atributo1

    atributo2

  }

}

Esto se ejecuta unido a un json que tiene las variables con sus valores en esta consulta

{

$variable: valor

}

## Múltiples consultas

Por su naturaleza dentro de una Query se pueden llamar a varias peticiones (querys) dentro del { } de la query

query  nombreQurey {

  tipo(atributoId:1) {

    atributo1

    atributo2

  }

tipos1 {

atributo1

atributo2

}

}

## Mutaciones

mutation($atributo1: String!, $atributo2: String!){

  addTipo1(atributo1: $atributo1, atributo2: $atributo2) {

    atributo1

    atributo2

  }

}

## Suscripciones

subscription PostFeed {

postCreated {

author

comment

}

}

Una ves se lanzan, esta se llama automáticamente cuando ocurra un evento de su tipo

# Conectarse a Express

* yarn add @graphql-tools/schema
* yarn add body-parser
* yarn add express
* yarn add graphql-subscriptions
* yarn add graphql-ws
* yarn add ws

import { makeExecutableSchema } from "@graphql-tools/schema";

import  e  from "express";

import { createServer } from "http";

import  WebSocket from  "ws";

import {useServer} from "graphql-ws/lib/use/ws"

import { ApolloServerPluginDrainHttpServer } from "@apollo/server/plugin/drainHttpServer";

import bodyParser from "body-parser";

import { expressMiddleware} from '@apollo/server/express4'

const schema=makeExecutableSchema({typeDefs,resolvers})

const app=e()

const httpServer=createServer(app)

const wsServe=new WebSocket.WebSocketServer({

    server:httpServer,

    path:'/graphql'

})

const wsServerCleanup=useServer({schema},wsServe)

const server =new ApolloServer({

    schema,

    plugins:[

        ApolloServerPluginDrainHttpServer({

            httpServer

        }),

        {

            async serverWillStart()  {

                return {

                    async drainServer(){

                        await wsServerCleanup.dispose()

                    }

                }

            }

        }

    ]

});

await server.start()

app.use('/graphql',bodyParser.json,expressMiddleware(server))

httpServer.listen(4000,()=>{

    `server readi at: http://localhost:4000/graphql`

})

yarn add mongoose

yarn add @types/mongoose