

Pt4 - postgraphile & GraphQL & NodeJS & Express.js

//*****//

NodeJs es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación

Express.js es un marco de aplicación web para Node.js. Está diseñado para construir aplicaciones web y APIs.

//*****//

Ya que en postgraphile no se puede realizar operaciones crud(sin definir "mutaciones"), he realizado un simple CRUD con node.js y express.js definiendo las mutaciones para poder realizar operaciones tales como borrar, insertar, mostrar y modificar de un campo de una tabla específica.



```
1
2 mutation {
3   createProducte(
4     input : {producte : { id: "1", part: "Disco duro"}}
5   ){
6     producte {
7       id
8       part
9       createdAt
10    }
11  }
12 }
```



```
{
  "errors": [
    {
      "message": "Schema is not configured for mutations.",
      "locations": [
        {
          "line": 2,
          "column": 1
        }
      ]
    }
  ]
}
```

Conexión a postgresQL

Primero para comenzar a configurar un conexión con el nodeJs y el postGraphile se configura un adaptador el cual hace que se conecte a una base de datos de postgresQL y poder obtener los datos.

```
js pgAdaptor.js x
1  require('dotenv').config()
2  const pgPromise = require('pg-promise');
3
4  const pgp = pgPromise({}); // Empty object means no additional config required
5
6  const config = {
7    host: process.env.POSTGRES_HOST,
8    port: process.env.POSTGRES_PORT,
9    database: process.env.POSTGRES_DB,
10   user: process.env.POSTGRES_USER,
11   password: process.env.POSTGRES_PASSWORD
12 };
13
14 const db = pgp(config);
15
16 exports.db = db;
17
```

Se crea una variable(db) la cual se exporta y contiene todo los datos de la BD.

Las credenciales de conexión las obtiene a través de un archivo .env el cual hace que se haga un enlace seguro.

```
js pgAdaptor.js x .env x
1  POSTGRES_HOST=localhost
2  POSTGRES_PORT=5432
3  POSTGRES_DB=descompon_pcs
4  POSTGRES_USER=postgres
5  POSTGRES_PASSWORD=140299
6
```

Para comprobar que la conexión es correcta se hace una pequeña prueba de listar todos los registros de una tabla por terminal

****Archivo pgAdaptor.js****

```
const db = pgp(config);

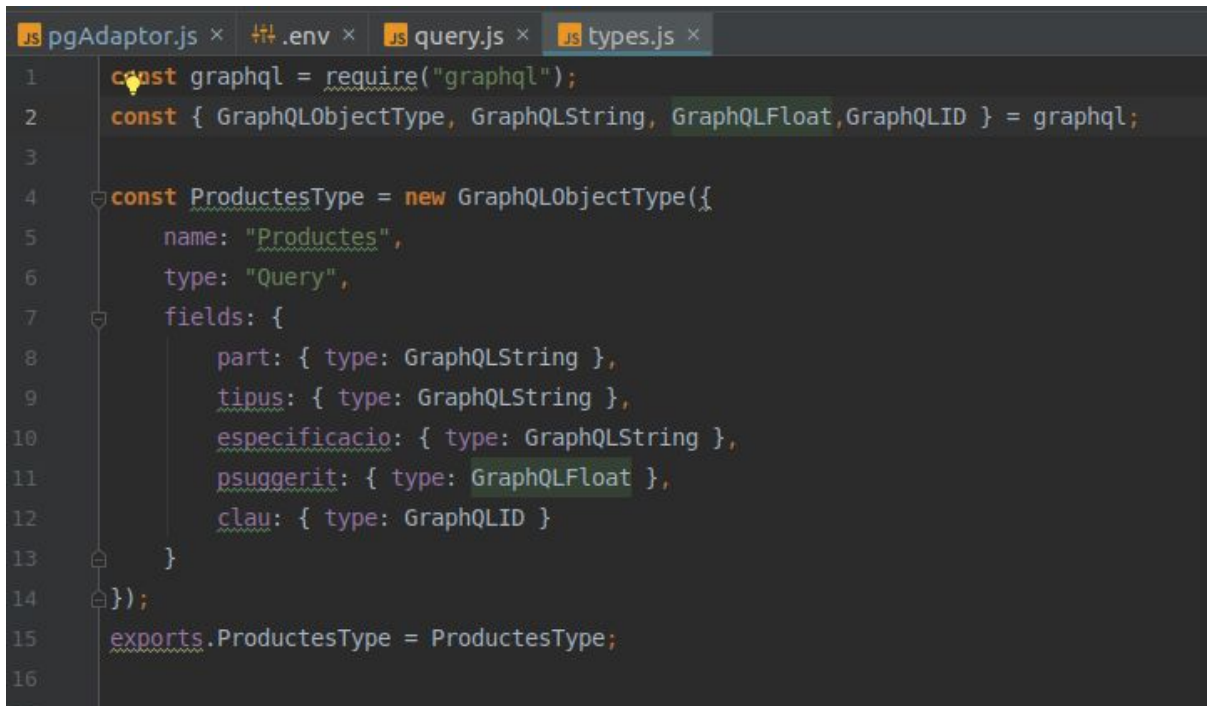
db.query("select * from productes;")
  .then(res => {
    console.log(res);
  }).catch( onrejected: (error) =>{
    console.log(error);
  });

exports.db = db;
```

```
→ crudGraphQLPostgres git:(master) x node pgAdaptor.js
[ { part: 'Processador',
  tipus: '2 GHz',
  especificacio: '32 bits',
  psuggerit: null,
  clau: 1 },
  { part: 'Processador',
  tipus: '2.4 GHz',
  especificacio: '32 bits',
  psuggerit: 35,
  clau: 2 },
  { part: 'Processador',
  tipus: '1.7 GHz',
  especificacio: '64 bits',
  psuggerit: 205,
  clau: 3 },
  { part: 'Processador',
  tipus: '3 GHz',
  especificacio: '64 bits',
  psuggerit: 560,
  clau: 4 },
  { part: 'RAM',
  tipus: '128MB',
  especificacio: '333 MHz',
  psuggerit: 10
```

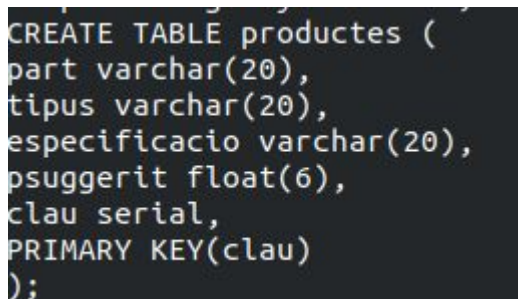
Tipos

Al saber que la conexión se establece correctamente, se procede a definir los tipos los cuales son las tablas



```
1 const graphql = require("graphql");
2 const { GraphQLObjectType, GraphQLString, GraphQLFloat, GraphQLID } = graphql;
3
4 const ProductesType = new GraphQLObjectType({
5   name: "Productes",
6   type: "Query",
7   fields: {
8     part: { type: GraphQLString },
9     tipus: { type: GraphQLString },
10    especificacio: { type: GraphQLString },
11    psuggerit: { type: GraphQLFloat },
12    clau: { type: GraphQLID }
13  }
14 });
15 exports.ProductesType = ProductesType;
```

En la primera línea se define que se requiere graphql el cual se utilizará para indicar los tipos de datos de la tabla el cual se define a la siguiente línea. Los tipos de datos deben ser de acuerdo a los tipos de la tabla real de postgres



```
CREATE TABLE productes (
  part varchar(20),
  tipus varchar(20),
  especificacio varchar(20),
  psuggerit float(6),
  clau serial,
  PRIMARY KEY(clau)
);
```

Query

Se puede crear simple queries para mostrar resultados de una tabla(tipo `|| type`), al cual como argumentos se le debe de pasar primero que todo la conexión a la BD que se encarga de obtener y devolver la sentencia consultada, luego se le indica los tipos de datos y el tipo(tabla) específico. Se crea un nuevo `GraphQLObjectType` al que se le indica que es tipo query y se le pasa como parámetros el tipo de objeto, los argumentos que se le ha de pasar a la sentencia, la sentencia que esté caso es una `SELECT` y por último se le pasa un return de la consulta.

```
const { db } = require("../pgAdaptor");

const { GraphQLObjectType, GraphQLID, GraphQLString, GraphQLFloat } = require("graphql");
const { ProductesType } = require("../types");

const RootQuery = new GraphQLObjectType({
  name: "RootQueryType",
  type: "Query",
  fields: {
    Producte: {
      type: ProductesType,
      args: {
        clau: { type: GraphQLID }
      },
      resolve(parentValue, args) {
        const query = `select * from productes where clau = $1`;
        const values = [
          args.clau
        ];

        return db
          .one(query, values)
          .then( onfulfilled: res => res)
          .catch( onrejected: err => err);
      }
    },
  },
});

exports.query = RootQuery;
```

Para realizar la prueba que funciona correctamente se crea un archivo *app.js*, siendo su función ejecutar las peticiones de las queries que se le pase para eso hago uso de *express.js* el cual apoya con una conexión local usando GraphQL, escuchando el puerto 3000.

```
"use strict";
const graphql = require("graphql");
const express = require("express");
const expressGraphQL = require("express-graphql");
const { GraphQLSchema } = graphql;
const { query } = require("../schemas/query");
const { mutation } = require("../schemas/mutation");

const schema = new GraphQLSchema({
  query,
  mutation,
});

var app = express();
app.use(
  fn: '/',
  expressGraphQL({ options: {
    schema: schema,
    graphql: true
  }})
);

app.listen(3000, () =>
  console.log('GraphQL server running on localhost:3000 ' + ' http://localhost:3000/')
);
```

Se ejecuta el archivo y se abre el enlace

```
→ crudGraphQLPostgres git:(master) x node app.js
GraphQL server running on localhost:3000 http://localhost:3000/
```


Así abriendo la página principal del GraphQL

The screenshot shows the GraphiQL web interface in a browser. The address bar shows 'localhost:3000/?'. The interface has a dark theme and includes a 'Prettify' button and a 'History' button. The main area contains a list of instructions for using GraphiQL, including how to write queries, use keyboard shortcuts, and how the interface handles errors. The instructions are numbered 1 through 30.

```
1 # Welcome to GraphiQL
2 #
3 # GraphiQL is an in-browser tool for writing, validating, and
4 # testing GraphQL queries.
5 #
6 # Type queries into this side of the screen, and you will see intelligent
7 # typeaheads aware of the current GraphQL type schema and live syntax and
8 # validation errors highlighted within the text.
9 #
10 # GraphQL queries typically start with a "{" character. Lines that starts
11 # with a # are ignored.
12 #
13 # An example GraphQL query might look like:
14 #
15 #   {
16 #     field(arg: "value") {
17 #       subField
18 #     }
19 #   }
20 #
21 # Keyboard shortcuts:
22 #
23 #   Prettify Query:  Shift-Ctrl-P (or press the prettify button above)
24 #
25 #   Run Query:      Ctrl-Enter (or press the play button above)
26 #
27 #   Auto Complete:  Ctrl-Space (or just start typing)
28 #
29 #
30 #
```

Ahora si se procede a la prueba

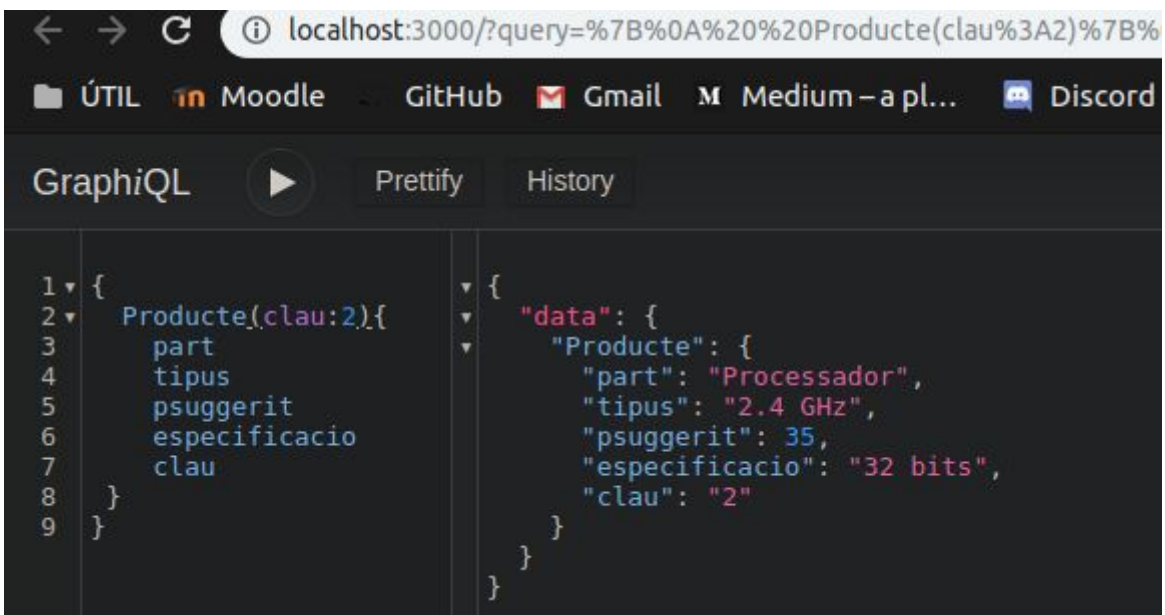
Primero muestro que este tipo de consulta el graphiql por defecto no los tiene

The screenshot shows the PostGraphiQL web interface in a browser. The address bar shows 'localhost:5000/graphiql'. The interface has a dark theme and includes a 'Prettify' button and a 'History' button. The main area contains a GraphQL query for a 'Producte' type. The query is numbered 1 through 9.

```
1 {
2   Producte(clau:1){
3     part
4     tipus
5     psuggerit
6     especificacio
7     clau
8   }
9 }
```

```
{
  "errors": [
    {
      "message": "Cannot query field \"Producte\" on type \"Query\". Did you mean \"producte\", \"allP",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ]
    }
  ]
}
```

Y el hacer la misma consulta creada por mi si que devuelve un resultado correcto



Al postgres se muestra que el resultado es correcto

```
descompon_pcs=# SELECT * FROM productes WHERE clau=2;
   part   | tipus  | especificacio | psuggestit | clau
-----+-----+-----+-----+-----
Processador | 2.4 GHz | 32 bits      |          35 |    2
(1 row)
```


Mutations

Las mutaciones al igual que las queries son consultas creadas a partir de un tipo(tabla), con los argumentos(columnas) correspondientes con la diferencia que se puede realizar sentencias con un poco más de complejidad como `INSERT, DELETE, UPDATE, JOIN, etc`.

INSERT

```
addProducte: {
  type: ProducteType,
  args: {
    part: { type: GraphQLString },
    tipus: { type: GraphQLString },
    especificacio: { type: GraphQLString },
    psuggerit: { type: GraphQLFloat },
    clau: { type: GraphQLID },
  },
  resolve(parentValue, args) {
    const query = `INSERT INTO productes(part, tipus, especificacio, psuggerit, clau) VALUES ($1, $2, $3, $4, $5)`
    const values = [
      args.part,
      args.tipus,
      args.especificacio,
      args.psuggerit,
      args.clau
    ];

    return db
      .one(query, values)
      .then( onfulfilled: res => res)
      .catch( onrejected: err => err);
  }
}
```

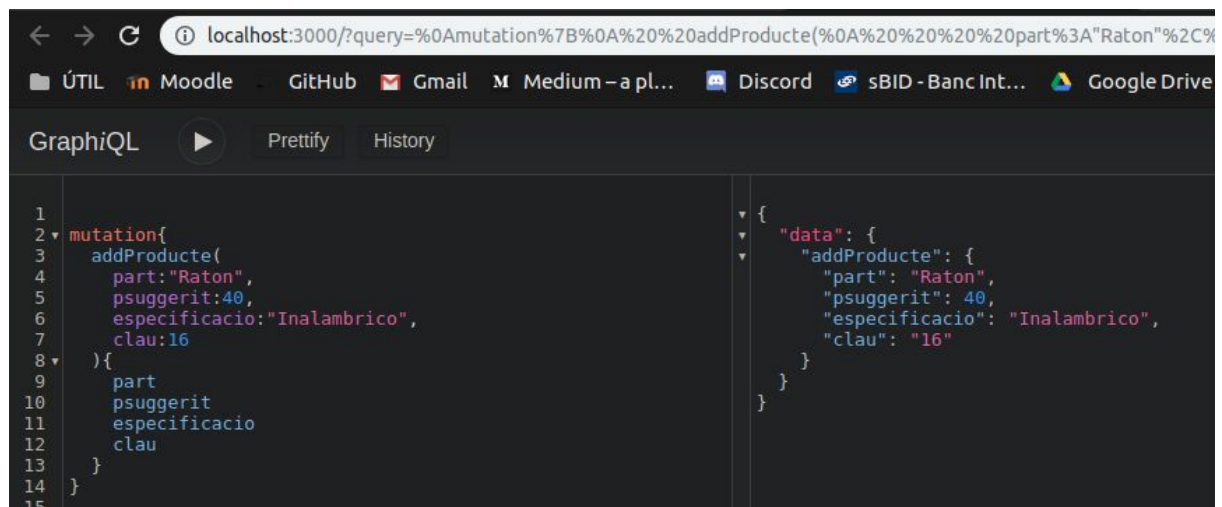
Demostracion

Al postgres no existe ningun registro con la clau = 16

```
descompon_pcs=# SELECT * FROM productes WHERE clau=16;
 part | tipus | especificacio | psuggerit | clau
-----+-----+-----+-----+-----
(0 rows)

descompon_pcs=#
```

Se ejecuta la función

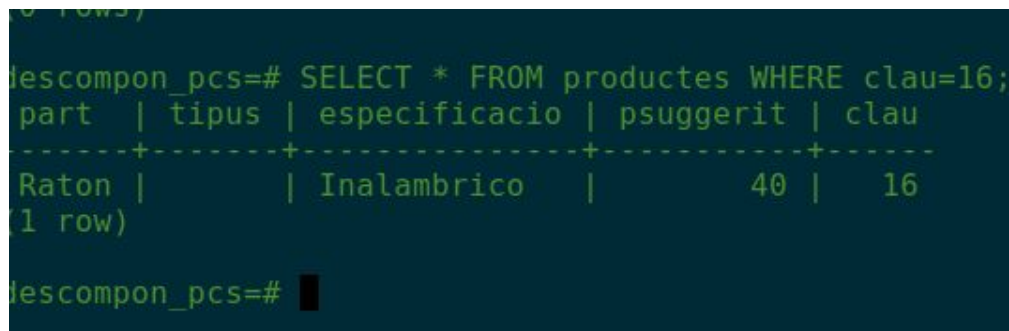


The screenshot shows a web browser at localhost:3000 with a GraphQL query in the GraphiQL interface. The query is a mutation to add a product. The response is a JSON object indicating the product was added successfully.

```
1 mutation{
2   addProducte(
3     part:"Raton",
4     psuggerit:40,
5     especificacio:"Inalambrico",
6     clau:16
7   ){
8     part
9     psuggerit
10    especificacio
11    clau
12  }
13 }
14 }
15 }
```

```
{
  "data": {
    "addProducte": {
      "part": "Raton",
      "psuggerit": 40,
      "especificacio": "Inalambrico",
      "clau": "16"
    }
  }
}
```

Y se comprueba que efectivamente se ha creado



The screenshot shows a SQL query executed in a terminal. The query selects all columns from the 'productes' table where the 'clau' is 16. The result shows one row with the values: Raton, Inalambrico, 40, 16.

```
descompon_pcs=# SELECT * FROM productes WHERE clau=16;
 part | tipus | especificacio | psuggerit | clau
-----+-----+-----+-----+-----
 Raton |      | Inalambrico   |         40 |   16
(1 row)

descompon_pcs=#
```

DELETE



The screenshot shows a GraphQL resolver function for deleting a product. The function takes the parent value and arguments, constructs a SQL DELETE query, and returns the result from the database.

```
deleteProducte:{
  type: ProductesType,
  args:{
    clau: { type: GraphQLID },
  },
  resolve(parentValue, args) {
    const query = `DELETE FROM productes WHERE clau = $1`;
    const values = [
      args.clau
    ];
    return db
      .one(query, values)
      .then( onfulfilled: res => res)
      .catch( onrejected: err => err);
  },
},
```

Se muestra que el producto con la clau=16 existe

```
descompon_pcs=# SELECT * FROM productes WHERE clau=16;
 part | tipus | especificacio | psuggestit | clau
-----+-----+-----+-----+-----
 Raton |      | Inalambrico   |          40 |   16
(1 row)
```

Al ejecutar la función lo primero en aparecer es un error que dice que no hay nada que devolver y en el apartado de data se muestra como null, lo que quiere decir que la función se ha ejecutado bien

```
mutation {
  deleteProducte(clau:16) {
    part
  }
}

{
  "errors": [
    {
      "message": "No se puede borrar un producto que no existe"
    }
  ],
  "data": {
    "deleteProducte": null
  }
}
```

Al buscar el registro, no se encuentra porque se ha borrado

```
descompon_pcs=# SELECT * FROM productes WHERE clau=16;
 part | tipus | especificacio | psuggestit | clau
-----+-----+-----+-----+-----
(0 rows)
```

UPDATE

Función que ejecuta una query de modificación a la columna 'part' de la tabla productos

```
updatePartProducte:{
  type: ProductesType,
  args: {
    part: { type: GraphQLString },
    clau: { type: GraphQLID },
  },
  resolve(parentValue, args) {
    const query = `UPDATE productes SET part=$1 WHERE clau=$2 RETURNING part,clau`;
    const values = [
      args.part,
      args.clau
    ];

    return db
      .one(query, values)
      .then( onfulfilled: res => res)
      .catch( onrejected: err => err);
  }
}
```

Muestro el producto con la 'clau' = 1

```
descompon_pcs=# SELECT * FROM productes WHERE clau=1;
   part   | tipus | especificacio | psuggerit | clau
-----+-----+-----+-----+-----
Processador | 2 GHz | 32 bits      |           |    1
(1 row)

descompon_pcs=#
```

Ejecuto la función y se modifica

```
mutation{
  updatePartProducte(clau:1,part:"Monitor"){
    part
    clau
  }
}
```

```
{
  "data": {
    "updatePartProducte": {
      "part": "Monitor",
      "clau": "1"
    }
  }
}
```

Y se aprecia que ha cambiado

```
descompon_pcs=# SELECT * FROM productes WHERE clau=1;
 part  | tipus | especificacio | psuggestit | clau
-----+-----+-----+-----+-----
 Monitor | 2 GHz | 32 bits      |             | 1
(1 row)

descompon_pcs=#
```

Conclusiones :

1. Postgraphile es una gran forma de conexión rápida a un servidor o una BD
2. Se puede implementar en múltiples lenguajes
3. Para poder trabajar con él y realizar simples consultas es simple
4. Al crear estructuras complejas, se requiere un conocimiento un poco más avanzado para realizar una buena implementación ya que cuenta con una gran cantidad de funcionalidades