Domini Code - API GraphQL - Control de inventario 2020-08-08

0.- Presentación

0.1.- Bienvenida

0.2.- Presentación del tema ”API GRAPHQL”

0.3.- Presentación del invitado: El invitado se presenta y describe brevemente a qué se dedica y en donde lo podemos encontrar.

0.4.- Saludos al chat de manera general

LiveCoding

1.- Introducción a GraphQL

1.1.- ¿Qué es?

Una breve explicación para explicar de que trata y en que lenguajes se puede implementar. (documento adjunto del pdf)

1.2.- Comparativa teórica

Explicar el funcionamiento general (con gráfica) y luego hacer la comparativa con las características

1.3.- Ejemplo práctico

Para hacer algo diferente de la API de Star Wars para variar un poco, usaré la API de RestCountries V2 que está tanto para REST como GraphQL.

Objetivo: Obtener los paises que hacen frontera con "España"

REST:
https://restcountries.eu/

Buscar "spain" y vemos que lo tenemos ahí

Ahora vamos a filtrar únicamente España con el filtro que usamos del nombre:

https://restcountries.eu/rest/v2/name/spain

Obtenemos el resultado y en las fronteras, solo obtenemos los AlphaCodes que deberemos de utilizar inidvidualmente para ir haciendo las peticiones:

https://restcountries.eu/rest/v2/alpha/FRA

Conclusión:

- **1 request para obtener los detalles de España**.
- **x request por longitud de paises fronterizos a España. En este caso 5.**
  ==========
  6 request para obtener lo que queremos en este caso

¿Qué pasaría si queremos hacerlo con España, México, y Suiza por poner un ejemplo?

Pues tendríamos **3 request (mínimo)** para detalles de los paises, luego individualmente cogemos los paises fronterizos y hacemos las request de manera individual.

- España 5 paises fronterizos.
- México 3.
- Suiza 5.
- Total 3 + 5 + 3 + 5 = 16 llamadas, esto se soluciona con GraphQL en UNA sola llamada

GraphQL:

https://countries-274616.ew.r.appspot.com/

Mirar la documentación y mirar como podemos obtener la información de los paises fronterizos de España (Spain)

```
query {
  Country (filter: {name: "Spain"}){
      name
      borders {
          name
          nativeName
          population
          populationDensity
      }
  }
}
```

Como se puede apreciar, todo lo que queremos lo obtenemos con una sola llamada y podemos coger solo unas pocas propiedades, haciendo un consumo menor

¿Qué pasaría si queremos hacer con los 2 paises añadidos anteriormente?

Muy fácil, siguiendo la misma estructura y asignando un alias para diferenciar las operaciones con el mismo resolver, hacemos el filtro

2.- Instalaciones, referencias e inicio del proyecto

2.1.- Instalación MongoDB / MongoDB Compass

MongoDB => Docs => Server => Navigation => Installation (MongoDB Community)
Seleccionamos y seguimos los pasos.

MongoDB Compass => Página principal de MongoDB => Software => Try it now.
Seleccionar sistema operativo, versión y descargar. Instalar con asistente.

2.2.- Instalar el generador de proyectos GraphQL

- Acceder a https://github.com/graphql-course/graphql-cli
- Ahi vemos la información del repositorio, lo que podremos encontrar.
- Instalamos el generador de manera global (importante tener Git instalado en nuestra máquina):
- **npm install graphql-course/graphql-cli**
- Para ejecutar **gql**

2.3.- Crear proyecto y primeras configuraciones

Para crear un proyecto y realizar las primeras configuraciones, como la creación del fichero de variables de entorno necesario para especificar la referencia de la Base de Datos. Seguimos los siguientes pasos:

- Ejecutar **gql**
- Seleccionamos la segunda opción: “graphql-hello-world-db”
- Rellenamos todas las preguntas del asistente.
- Crear fichero de variables de entorno .env
- Poner en marcha y hacer un **repaso del playground**, con sus principales opciones para dar una idea de como se trabaja con GraphQL (Docs, Schema,...)

3.- API GraphQL - Control de inventario de almacen

3.1.- Objetivos

3.2.- Funcionalidades que se van a implementar

3.2.1.- Añadir producto

Pasos a seguir:

- Tener a mano la documentación de MongoDB donde vamos a trabajar con CRUD. (Todavía no usaremos)
- Empezar definiendo la operación tipo mutation para añadir productos => **addProduct(product: ProductInput): Boolean**.
  Necesario crear input donde vamos a usar para enviar la información al servidor.
  - **ProductInput (code, name, description, stock, price)**

```
"""
### Para gestionar la información de los productos.
Podremos usarlo para estas **acciones**
* Añadir
* Modificar
"""
input ProductInput {
    "Código identificativo del producto"
    code: ID!
    "Nombre del producto"
    name: String!
    "Descripción general del producto"
    description: String!
    "Precio en euros"
    price: Float!
    "Unidades almacenadas en stock y disponibles para enviar"
    stock: Int!
}
```

- Añadir el nuevo resolver mutation “product.ts”, **añadir la definición para darle solución y unirlo al index.ts.**
- Empezar a dar solución al resolver. Poner mediante un console.log la info del producto, para probar desde el playground.

```
const resolversProductMutation: IResolvers = {
    Mutation: {
        async addProduct(_, { product }) {
            console.log(product);
            // AQUI HACEMOS LA OPERACION DE AÑADIR EL PRODUCTO CON MONGODB
            return true;
        }
    }
};

export default resolversProductMutation;
```

- Resultado esperado (de un ejemplo):

```
{
  code: 'TR-10',
  name: 'Tomb Raider 10',
  description: 'Primer juego en la nueva generación para Playstation 5',
  price: 120,
  stock: 50
}
```

http://localhost:8000/graphql

- Añadir operación de inserción, con MongoDB **(explicar de donde viene el objeto del contexto)**
- Realizar la operación de inserción (añadiendo then y catch)

```
return await db.collection('products')
    .insertOne(product).then(async () => {
        return true;
    }).catch(() => false);
```

- Visualizar resultado en MongoDB Compass e introducir otro ejemplo

3.2.2.- Lista de productos

Pasos a seguir:

- Tener a mano la documentación de MongoDB donde vamos a trabajar con CRUD.
- Empezar definiendo la operación tipo query para ver productos => **products: [Product!]!**.
  Necesario crear type object donde vamos a usar para mostrar la información de los productos (code, name, description, stock, price) (Nos basamos en el Input)
- Añadimos resolver para la lista de productos
- Dar solución con operación en base de datos

```
const queryProductResolvers: IResolvers = {
    Query: {
        async products(_, __, { db }) {
            return await db.collection(COLLECTIONS.PRODUCTS).find().toArray().then(
                (result: Array<object>) => result
            ).catch(() => []);
        }
    }
};
```

3.2.3.- Actualizar producto (por añadir productos o reducir por ventas)

Gestionaremos el incremento (porque añadimos productos al almacen y el decremento, por vender productos. En este caso trabajamos con el tipo de operación de actualizar, donde trabajaremos con “update” y **haremos uso del operator de actualizar $inc** que será el encargado de actualizar con el valor que pasemos (**si + suma, si es - resta**).

https://docs.mongodb.com/manual/tutorial/update-documents/#update-a-single-document

Reference => Operators => Update Operators

- Añadir definición en el mutation (schema) con dos parámetros, stock (entero + / -) y code (string)
- Ir a resolvers y añadir la definición.
- Coger la operación Update y añadirlo.
- **Dentro de esa operación, añadir el operador de incremento $inc (mirar ejemplo haciendo click) y sustituirlo por $set, para gestionar las actualizaciones.**

3.2.4.- Obtener actualizaciones en tiempo real de nuestro inventario completo

3.2.4.1.- Instalación para trabajar con Subscription

Necesitamos instalar para poder configurar la conexión de web socket y enlazarla a nuestra API GraphQL

npm install subscriptions-transport-ws

3.2.4.2.- Configuración del servidor

Referencia que necesitamos: https://github.com/apollographql/graphql-subscriptions

Añadir pubsub y añadirlo en el contexto junto con la instancia de la base de datos
const pubsub = new PubSub(); // Importar dentro de apollo-server-express

Añadirlo en el contexto:

context = { db, pubsub}

Instalar el subscription handle y modificar el mensaje del server:

```
  const httpServer = createServer(app);
  server.installSubscriptionHandlers(httpServer);
  const PORT = process.env.PORT || 2002;
  httpServer.listen(
    {
      port: PORT,
    },
    () => {
      console.log('==================SERVER API GRAPHQL====================');
      console.log(`STATUS: ${chalk.greenBright('ONLINE')}`);
      console.log(`MESSAGE: ${chalk.greenBright('API MEANG - Online Shop CONNECT!!')}`);
      console.log(`GraphQL Server => @: http://localhost:${PORT}/graphql `);
      console.log(`WS Connection => @: ws://localhost:${PORT}/graphql `);
    }
  );
}
```

3.2.4.3.- Escuchar cambios en el stock

Usamos esta referencia para poder trabajar con ello. Definir en el schema el nuevo tipo de raíz (crear fichero subscription.graphql). Esto lo usaremos cuando se actualice el stock y cuando se añada un nuevo producto en el almacen:

type Subscription {
    changeStock: [Product!]!
}

Añadir el resolver correspondiente
(Crear el fichero index.ts para mezclar, el directorio,...), añadir el subscription en el index.ts de los resolvers) Tener en cuenta esta referencia

El resolver

```
import { SUBSCRIPTIONS_EVENT } from '../../config/constants';


const subscriptionStockResolvers: IResolvers = {
    Subscription: {
        changeStock: {
            subscribe: (_, __, { pubsub}) => pubsub.asyncIterator(SUBSCRIPTIONS_EVENT.CHANGE_STOCK),
        },
    }
};

export default subscriptionStockResolvers;
```

Falta publicar el evento (en el mutation):

```
async updateProductStock(_, {code, stock}, {db, pubsub}) {
    return await db.collection(COLLECTIONS.PRODUCTS).updateOne(
        { code },
        {
          $inc: { stock}
        }
    ).then(async () => {
        const list = await db.collection(COLLECTIONS.PRODUCTS).find().toArray();
        pubsub.publish(SUBSCRIPTIONS_EVENT.CHANGE_STOCK, { changeStock: list});
        return true;
    }).catch(() => false);
}
```

DEFINICIÓN DEL SUBSCRIPTIONS Y LA LISTA ACTUALIZADA

EVENTO

Haremos lo mismo con el mutation correspondientea añadir productos

```
async addProduct(_, { product }, { db, pubsub }) {
    console.log(product);
    // AQUI HACEMOS LA OPERACION DE AÑADIR EL PRODUCTO CON MONGODB
    return await db.collection(COLLECTIONS.PRODUCTS)
        .insertOne(product).then(async () => {
            const list = await db.collection(COLLECTIONS.PRODUCTS).find().toArray();
            pubsub.publish(SUBSCRIPTIONS_EVENT.CHANGE_STOCK, { changeStock: list});
            return true;
        }).catch(() => false);
},
```

4.- Preguntas

5.- Final

Repositorio final:
Udemy: https://www.udemy.com/user/anartzmugika/
Cursos PREMIUM:

- GraphQL desde 0 a Experto: https://cursos.anartz-mugika.com/graphql-de-0-a-deploy
- NPM. Crear librerías en JS, TS y Angular: https://cursos.anartz-mugika.com/librerias-js-ts-angular-npm

Próximamente (Curso MEAN+GraphQL => Tienda de videojuegos Online con pagos mediante stripe) (Pequeña demo del apartado de pagos: https://www.youtube.com/watch?v=tV6swbCnLF0)

Comunidad GraphQL en Español (más de 2200 usuarios/as!!): https://www.facebook.com/groups/graphql.es

Puedes encontrarme en las siguientes referencias:

- Github: https://github.com/mugan86
- Twitter (muy activo): https://social.anartz-mugika.com/twitter
- Telegram: https://social.anartz-mugika.com/telegram
- Instagram: https://social.anartz-mugika.com/instagram
- Medium: https://medium.com/@mugan86
  (más de 40 artículos con temáticas de GraphQL, NodeJS, Angular,...)
- Youtube: https://cursos.anartz-mugika.com/yt-anartz