GraphQL: Repaso

GraphQL es excelente para queries. Realizar consultas a la API, especialmente cuando dichas consultas son complejas y tienen múltiples tipos de datos que recibir simultáneamente.

Hasta ahora, vimos que al utilizar APIs tipo REST, para indicar que queremos hacer una operación de escritura o actualización de datos, solamente teníamos que cambiar el tipo de método HTTP. Pero en GraphQL, el tipo de método al igual que la url que consultamos nunca cambian.

Mutations (repaso)

En GraphQL, para impactar cambios en nuestra API, se utiliza otro tipo de request GraphQL que se denomina **mutation**.

Las mutations van en el mismo body de la request de GraphQL, pero se aclara que es una mutación al principio de la misma.

Variables (repaso)

GraphQL como dijimos, es una forma de hacer consultas hacia APIs muy potente. Es tan potente, que las variables son solamente una de las tantas cosas que nos permite utilizar dentro del lenguaje de GraphQL.

Fragmentos (repaso)

Los fragmentos son, como su propio nombre lo describe, trozos de una query.

Supongamos que tenemos múltiples queries hacia una misma entidad y no queremos tener que repetir todo el tiempo las propiedades que estamos necesitando. O bien, tenemos una query que puede devolver dos tipos de datos diferentes y queremos algunas propiedades determinadas según cuál de ellos nos devuelva.

Ya soy pro.

Hasta ahora todo fue muy bonito haciendo APIs de ejemplo, configurando nuestro entorno, utilizando GraphQL de forma pública. Pero que pasa si quiero usar autenticación en mi API?

Eso es muy fácil. Ya que estamos usando Express. Podemos simplemente agregar un middleware que se encargue de autenticar la request. Hay muchas opciones prefabricadas. Pero si optamos por resolverlo nosotros mismos, este sería un buen ejemplo:

```
app.post('/login', (req, res) => {
    if (req.body.password === "pepe") {
      return res.json({ success: true, token: 'token' });
    }

    return res.status(401).json({ success: false, error: 'Invalid Credentials'
});

app.use((req, res, next) => {
    if (req.headers.authorization !== 'token') {
      return res.status(401).json({ error: 'Unauthorized' });
    }

    return next();
});
```

Alternativa a Express GraphQL: Apollo

Apollo es tanto un servidor como un cliente, en otras palabras, es la suite que ofrece GraphQL para trabajar con su tecnología.

En el caso del servidor, nos permite crear una API GraphQL sin necesidad de usar express, aunque también se puede utilizar como middleware si asi se lo prefiera.

```
77  (async () => {
78      const server = new ApolloServer({
79          typeDefs: importSchema('./schema.graphql'),
80          resolvers,
81      });
82
83      const { url } = await startStandaloneServer(server, {
84          listen: { port: 3000 },
85      });
86
87      console.log(`GraphQL API Started: ${url}`);
88  })();
```

GraphQL en el frontend

Para utilizar GraphQL en el frontend, se debe hacer lo mismo que con cualquier API, esto es, crear un cliente o bien utilizar alguno existente.

En el caso de GraphQL, como a diferencia de REST, está estandarizado, existen clientes que ya realizan la mayoría de las cosas de forma automática. El mas conocido es Apollo, y es el cliente oficial de GraphQL.

Los ejemplos que se mostrarán a continuación, utilizan React.

GQL

Es una porción de código GraphQL incrustado en nuestro código el cual contiene la query, variables, y otras cosas necesarias para poder utilizar nuestra API.

```
const LIST_PRODUCTS_QUERY = gql`
  query ListProductQuery {
    ListProducts {
      name
      id
    }
  }
`;

const CREATE_PRODUCT_MUTATION = gql`
  mutation CreateProduct($name: String!) {
    CreateProduct(name: $name) {
      name,
      id
    }
  }
`
```

Hooks

Existen varios React Hooks que simplifican el uso de GraphQL, exportados por Apollo. Por ejemplo:

```
const { loading, error, data } = useQuery(LIST_PRODUCTS_QUERY);
const [ createProduct, createProductStatus ] = useMutation(CREATE_PRODUCT_MUTATION);
```

Buenas Prácticas

GraphQL

- Cada Mutation debería devolver el Objeto producido o actualizado
- Ningún endpoint debería mantener o controlar estados
- Modularizar

NodeJS

- Modularizar
- Manejar errores
- Mantener estructura y esquemas
- Evitar el uso innecesario de librerías
- Es mejor escribir código legible que resumido
- Prevenir código duplicado
- Mantener funciones limpias, que no sean para más de un caso específico