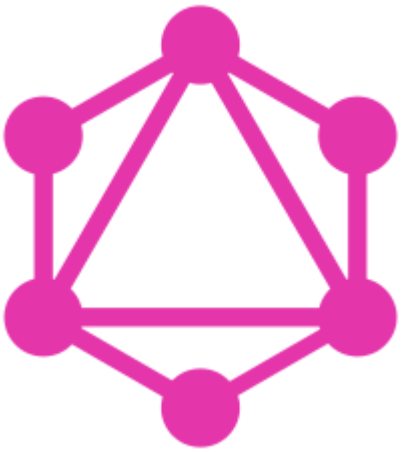
GraphQL en profundidad

Aplicaciones Distribuidas en Internet

Luis Jiménez Tortosa



Índice

Introducción

El tema elegido para el trabajo son las APIs **GraphQL en profundidad**. El objetivo de este trabajo es principalmente desarrollar un tema que en clase solo hemos visto brevemente de la forma más ilustrativa posible. En este caso se trata de una de las alternativas actuales al paradigma REST para la construcción de APIs. GraphQL se centra en la flexibilidad a la hora de recuperar los datos. Primero vamos a ver en qué consiste, por qué surgió y qué pretende aportar frente al paradigma dominante de las APIs REST y a continuación explicaremos la especificación de GraphQL y veremos un ejemplo de implementación de un API para la consulta y modificación de recursos.

Motivación

Aunque REST es el patrón más extendido para la transferencia de datos en la web y el principal que hemos visto en clase, puede dar ciertos problemas conforme las aplicaciones se vuelven más complejas. También hay otras situaciones como las APIs orientadas a operaciones o el streaming de datos en tiempo real en los que no es el más apropiado. GraphQL fue diseñado por Facebook para resolver los problemas que surgían en aplicaciones complejas por cierta falta de flexibilidad de REST. A continuación vamos a plantear un ejemplo de situación en la que REST puede ser un poco incómodo de usar y la solución que plantea GraphQL.

En REST el concepto principal son los recursos, cada recurso tiene asociado su endpoint que representa los recursos de un tipo, así podemos acceder a todos los recursos de un tipo, o especificando su identificador, a un recurso en concreto:

/coches	⟶	`{`
		`  "coches": [`
		`  {`
		`  id: 1`
		`  nombre: "Kia Ceed",`
		`  },`
		`  {`
		`  id: 2,`
		`  nombre: "Seat Ibiza",`
		`  }`
		`  ]`
		`}`

/coches/2	⟶	`{`
		`  "coche": {`
		`  id: 2,`
		`  nombre: “Seat Ibiza”,`
		`  }`
		`}`

Pero normalmente no tenemos solo recursos, también tenemos relaciones entre ellos, los recursos pueden tener subrecursos o estar relacionados con otros. Por ejemplo vamos a

suponer un portal web de películas donde los usuarios pueden ver películas, su puntuación y hacer reseñas. La película y la reseña podrían ser algo así:

/movies/{id}	⟶	`{ "movie": { id: 432, title: “Titanic”, year: 1997, cast: { ... }, ... } }`
/reviews/{id}	⟶	`{ "review": { id: 5678, movieId: 432, userId: 896, review: “...”, rating: 7.5, date: “15/06/2014” } }`

Sin embargo es evidente que con esto no bastaría, el recurso películas tiene una gran cantidad de datos sobre la película pero además está conectado a otros recursos, por ejemplo, una película tendrá reseñas que hayan hecho los usuarios o los actores que han participado. En aplicaciones de mayor complejidad un recurso estará relacionado con muchos otros recursos de distintas formas. Conectarlos y navegar entre ellos usando los ID no es viable haciendo peticiones a través de una red, podríamos acabar necesitando muchas peticiones. Por eso necesitaríamos otro endpoint para obtener las reviews de una película:

/movies/{id}/reviews	⟶	`{ "reviews": [ { id: 5678, movieId: 432, ... }, { id: 8312, movieId: 432, ... } ] }`

Ahora desde nuestro frontend si queremos ver los detalles de la película y sus reseñas necesitaremos 2 peticiones, algo que es bastante asequible. Sin embargo la película también tiene actores, un director, un reparto… Los actores podrían ser otro recurso ya que participarán en otras películas y podría interesarnos a partir de un actor obtener otras películas que haya hecho. Si lo diseñamos de la misma forma para cargar los datos de la película ya necesitaremos 3 peticiones. En este momento podemos pensar en el uso que se le va a dar al API, probablemente cuando queramos ver los detalles de una película nos interese ver también los actores que han participado en ella, así que para evitar otra petición añadiremos la información de los actores al endpoint **/movies/{id}**. Si recuperamos toda la información de cada actor tendremos un montón de información en un endpoint de películas que no nos sirve, así que únicamente recuperaremos sus datos básicos como su nombre y su ID. De esta forma hemos evitado añadir otra petición, hasta aquí nuestra API REST funciona bien.

Pero desde el frontend necesitamos usar la información desde varios sitios, queremos la información de las películas desde un listado de películas pero también al ver el detalle de la película con sus reviews. No usamos la misma información en el listado de películas donde nos basta con los datos básicos que en el detalle desde donde probablemente necesitaremos no sólo la información de la película sino también la de recursos relacionados (reviews y actores). Aquí empezamos a intuir el problema, cualquier información de otros recursos relacionados que necesitemos y no incluyamos en el endpoint va a necesitar (al menos) una petición más, esto se llama **under-fetching**. Cualquier información relacionada de otros recursos que incluyamos para evitar el peticiones extras en la vista detallada va a ser inútil desde otros lugares en los que necesitemos menos información, esto se llama **over-fetching**. Se trata del problema de la granularidad de los recursos visto en clase, la granularidad de los recursos REST es fija, no podemos especificar que datos de un subrecurso queremos y cuáles no.

Nuestro ejemplo es muy sencillo y las APIs REST funcionarán perfectamente, hay buenas prácticas de diseño para endpoints RESTful que nos ayudarán a minimizar el problema. Pero conforme los requerimientos crecen habrá más recursos con más relaciones más complejas. Si no añadimos información de las relaciones del recurso necesitaremos no una ni dos sino varias peticiones y evitar estas peticiones extra supondrá añadir más información que hará en otros casos que el problema de over-fetching sea mayor.

GraphQL pretende resolver este problema permitiendo al cliente especificar claramente los datos que necesita al servidor, que le responderá con los datos exactos que ha pedido. Además permite al cliente especificar la forma en la que se devuelven los datos. En GraphQL hay un único endpoint para el API al que se envía una query especificando los datos que se quieren recibir. De esta forma si quiero la lista de películas, pero los únicos datos que me interesan de cada película son el identificador y el nombre lo indico en la petición:

```
{
    movies:
        id,
        title
}
```

→

```
{
    "movies": [
        {
            id: 1
            title: "Titanic",
        },
        {
            id: 2,
            title: "Forest Gump",
        }
    ]
}
```

Si quiero todos los detalles de una película también puedo indicarlo. GraphQL me permite especificar los subrecursos a los que quiero acceder y qué datos del subrecurso quiero en la misma petición, ya no hay distintos endpoints para distintos recursos. Si quiero todos los datos de la película lo indico en la query. Si además quiero las reviews y los actores también puedo obtenerlos. Además puedo especificar que quiero todos los datos de las reviews incluido el nombre del usuario que las ha publicado, pero en cambio de los actores sólo me interesa su nombre y su identificador:

```
{
    movie(432):
        title,
        year,
        cast: {
            name,
            id
        },
        ...
        reviews: {
            id,
            user: {
                id,
                name
            },
            review,
            rating,
            date
        }
}
```

→

```
{
    "movie": {
        title: “Titanic”,
        year: 1997,
        cast: [
            {
                name: “Kate Winslet”,
                id: 126
            },
            ...
        ],
        ...
        reviews: [
            {
                id: 5678,
                user: {
                    id: 8666,
                    name: “Johnny77”
                },
                review: “...”,
                rating: 7.5,
                date: “15/06/2014”
            }
        ]
    }
}
```

Esto soluciona el problema de over-fetching y el de under-fetching, liberando al cliente de la tarea de ir haciendo peticiones a distintos endpoints. Además el cliente sabe exactamente qué datos va a recibir.

¿Qué es GraphQL?

GraphQL fue desarrollado internamente por un equipo de ingenieros de Facebook en 2012 cuando preparaban su nueva aplicación para iOS. La API del feed de noticias para la web no les servía y tenían que diseñar una nueva, su decisión inicial fue hacer una API RESTful pero pronto encontraron varios problemas, principalmente la necesidad de múltiples peticiones y al alto acoplamiento entre el cliente y la API del servidor. Su solución fue adoptar un enfoque distinto que fue el origen de GraphQL. Desde entonces Facebook siguió utilizándolo internamente y a partir de 2015 pasó a ser open-source. A día de hoy no depende de Facebook y es mantenido por la GraphQL foundation.

La definición de GraphQL de la página de la fundación que mantiene su especificación dice que es “un lenguaje de consulta para APIs y un entorno de ejecución para llevar a cabo esas consultas con tus datos” que “proporciona una descripción completa y comprensible de los datos en tu API, permitiendo al cliente pedir exactamente lo que necesita y nada más, haciendo más fácil evolucionar las APIs a lo largo del tiempo”.

Es decir, GraphQL es simplemente una especificación de una forma de recibir recursos de un servidor como alternativa a REST y el uso de distintos endpoints para solicitar la información. No es un lenguaje de programación ni una nueva tecnología, sino una forma distinta de usar las herramientas que ya tenemos para solicitar información a través de HTTP. No necesitamos nada nuevo para empezar a usar GraphQL y hay multitud de librerías que nos permiten empezar a usarlo fácilmente.

Especificación

En este apartado vamos a aprender cómo funciona GraphQL y cómo usarlo. Veremos en qué se basa, cómo hacer consultas y cómo modificar datos del servidor. Sin embargo aún no vamos a implementar nada, ya que GraphQL es simplemente una forma de comunicación entre el cliente y el servidor. Será en el próximo apartado en el que pongamos en práctica todo lo que hemos visto aquí y utilicemos una librería para implementar nuestra API.

Introducción

GraphQL es un lenguaje de consulta que permite al cliente especificar exactamente los datos que quiere recibir. No está vinculado a ninguna base de datos ni ninguna otra tecnología. Este es un ejemplo básico en el que un servidor expone los datos de un usuario y el nombre de sus amigos:

<table>
<tr>
<td>

```
{
    user {
        id,
        name,
        friends {
            name
        }
    }
```

</td>
<td>→</td>
<td>

```
{
    “user”: {
        id: “R2-D2”,
        name,
        friends [
            {
                name: “Luke Skywalker”
            },
```

</td>
</tr>
</table>

}
 	
		{
		name: “Han Solo”
		}
		]
		}
		}

Un servicio GraphQL se crea definiendo tipos de datos y sus atributos y proporcionando funciones que devuelvan los datos para los atributos de cada tipo. Por ejemplo para el servicio que hemos definido necesitamos un tipo Query, ya que se trata de una consulta, y dentro de ese tipo Query hay un tipo User del que podemos solicitar información con sus atributos. Podríamos verlos así:

```
type Query {
    user: User
}

type User {
    id: ID
    name: String!
    friends: [User]
}
```

Estos tipos no están definidos en ningún lenguaje en concreto, ya que según el lenguaje de programación lo haremos de una forma u otra, es solo una forma explicativa de entender que GraphQL es un lenguaje tipado, los atributos tienen tipos y en nuestro servicio GraphQL estos tipos habrán sido definidos de alguna forma. A continuación vamos a explicar los tipos en GraphQL.

Tipos

Los tipos en GraphQL son:

- **Escalares**: son los tipos básicos, aquellos que no están compuestos, pueden ser `Int`, `Float`, `String`, `Boolean` o `ID`.
- **Enumeraciones**: datos de tipo escalar con la particularidad de que están restringidos a un set de valores.
- **Objetos**: los componentes básicos de GraphQL que indican que objetos puedes recuperar del servidor y qué atributos tiene.
- **Listas** y **no-nulos**: podemos tener lista de tipos escalares y no escalares (objetos) y también podemos indicar que algún tipo de dato es no-nulo, para indicar que siempre tiene que devolver un valor (o recibirlo como argumento).
- Objetos **Query** y **Mutation**: son dos objetos especiales del *schema*. Todos los esquemas GraphQL tienen un tipo Query y algunos un tipo Mutation. El objeto Query contiene los tipos que se pueden consultar como hemos visto en el ejemplo anterior, el objeto Mutation contiene las consultas que realizan modificaciones en los recursos.

Hay algunos tipos más como las interfaces, los tipos unión y los tipos input pero ya hemos visto lo básico. Los tipos input los veremos más en detalle cuando hablemos de las mutaciones.

Queries

Con los ejemplos sencillos de queries que hemos visto hasta ahora ya tenemos lo básico. Para crear queries basta con indicar los tipos y atributos de esos tipos que queremos recuperar. Los tipos de datos estarán definidos previamente y podremos navegar entre ellos para recuperar datos de sus subrecursos y recursos relacionados. Pero al hacer consultar normalmente queremos enviar parámetros para poder ordenar o filtrar los resultados o especificar cualquier otro tipo de información. Por eso GraphQL permite el uso de parámetros:

```
{
    user(id: 23) {
        name,
        height
    }
}
```

→

```
{
    “data”: {
        “user”: {
            name: “Luke Skywalker”,
            height: 1.72
        }
    }
}
```

Además GraphQL permite que cada campo y cada objeto anidado reciba su propio set de parámetros. Podríamos utilizarlos para filtrar los subrecursos, por ejemplo obtener solo los amigos del usuario que cumplen cierto requisito o darle otros usos como especificar el formato en el que quieres recibir cierto dato:

```
{
    user(id: 23) {
        name,
        height(unit: FOOT)
    }
}
```

→

```
{
    “data”: {
        “user”: {
            name: “Luke Skywalker”,
            height: 5.6430448
        }
    }
}
```

Esto permite peticiones mucho más específicas que REST pero a la vez mucho más claras y legibles. Podemos pasar como argumentos los tipos de GraphQL o tipos propios que hemos definido siempre que sean serializables.

Un pequeño problema a la hora de hacer queries como las hemos hecho hasta ahora es que si pedimos datos de dos objetos distintos del mismo tipo en la respuesta ambos objetos se llamarán igual. Cuando hacemos esto podemos definir **alias** de forma que la respuesta los utilice y sea más fácil recuperar los datos de la respuesta:

```
{
    me: user(id: 23) {
        name
    },
    you: user(id: 25) {
        name
    },
}
```

→

```
{
    “data”: {
        “me”: {
            name: “Luis”,
        },
        “you”: {
            name: “Pepe”,
        }
    }
}
```

GraphQL tiene más funcionalidades interesantes en para las consultas como los fragmentos para reutilizar queries, las variables o las directivas que permiten modificar la querie en función del valor de parámetros. Pero con lo que hemos visto ya podemos construir todas nuestras queries.

Mutations

Aunque la mayoria de las peticiones a las son para pedir datos, necesitamos también una forma de modificar los datos del servidor. En GraphQL la forma de indicar que una petición puede modificar los datos es con una mutación. Hasta ahora no hemos indicado de que tipo eran nuestras peticiones, si no se indica se asume que es de tipo Query, para que sea de tipo Mutation hay que indicarlo:

```
mutation {
    addBook(title: “...”) {
        id,
        title
    }
}
```

→

```
{
    “data”: {
        “addBook”: {
            id: 15,
            title: “...”
        }
    }
}
```

En este ejemplo vemos cómo añadir un nuevo libro a nuestra colección. Indicamos que es una mutación, después el nombre de la mutación definida en el *schema* que vamos a usar con los parámetros que recibe. Si la mutación devuelve un objeto podemos indicar los atributos que queremos recibir como hacíamos en las consultas.

Para hacer más fáciles las mutaciones podemos usar variables y objetos *input*. Las variables ayudan al evitar tener que introducir directamente el valor del parámetro en cada sitio en el que se use mientras que el objeto input permite pasar un objeto entero como parámetro en lugar de los valores individuales de los atributos. Vamos a ver un ejemplo en el que añadimos una review de una película:

```
mutation newReview($id: ID, $review:
ReviewInput!){
    addReview (movieId: $id, review:
$review) {
```

→

```
{
    “data”: {
        “addReview”: {
            id: 687
```

```
        id
    }
}
```

Variables:

```
{
    “id”: 432,
    “review”: {
        review: “...”,
        rating: 7.5

    }
}
```

```
        }
    }
}
```

Con esto hemos repasado los aspectos más relevantes de GraphQL y estamos listos para ponerlo en práctica implementando nuestra propia API. Hay algunos pequeños detalles que he omitido y están enfocados principalmente en mayor comodidad y facilidad a la hora de crear nuestras peticiones, pero tienen que ver con un uso más avanzado y profesional y buenas prácticas de uso. En realidad con lo que hemos visto ya podemos hacer un uso completo de la especificación de GraphQL.

Ejemplo de implementación

Ahora que conocemos bien cómo usar GraphQL vamos a realizar un ejemplo de implementación de una API GraphQL. Hay multitud de librerías que implementan GraphQL para distintos lenguajes de programación, tanto para comunicar el frontend con el backend como para comunicar dos backends distintos. Nosotros vamos a realizar un servidor con Node.js y express. Empezamos iniciando nuestro proyecto de node y después instalamos las dependencias que vamos a utilizar:

```
> npm init
> npm i express express-graphql graphql
```

Y creamos nuestro servidor básico usando express que utilice GraphQL. Al poner el parámetro graphiql a true le estamos diciendo que queremos utilizar GraphiQL, una interfaz que nos proporciona la librería para hacer nuestras consultas, muy útil para el desarrollo.

```
const express = require('express')
const { graphqlHTTP } = require("express-graphql")
const app = express()

app.use('/graphql', graphqlHTTP({
    graphiql: true
}))
app.listen(5000., () => console.log('Server is running'))
```

Ya tenemos nuestro servidor funcionando usando GraphQL en el puerto 5000. Nuestro único endpoint es `/graphql` al que enviaremos todas nuestras peticiones. Si probamos el endpoint obtenemos un error que dice no se ha encontrado un *schema*. Recuerda que

dijimos que GraphQL es un lenguaje tipado y que para crear nuestro servicio definimos distintos tipos de datos e indicar la forma de obtener los datos de cada tipo, a esto se le denomina *schema*. Vamos a definir el nuestro, primero importamos los tipos para GraphQL que nos proporciona la librería:

```
const {
    GraphQLSchema,
    GraphQLObjectType,
    GraphQLString
} = require('graphql')
```

Estos son los tipos que vamos a usar de momento, el *schema*, el tipo objeto que es el que usan todos los objetos y el tipo *string*. Si usamos cualquier otro tipo de los que vimos en el apartado anterior lo añadimos aquí. Y definimos el *schema*:

```
const schema = new GraphQLSchema({
    query: new GraphQLObjectType({
        name: 'HelloWorld',
        fields: () => ({
            message: {
                type: GraphQLString,
                resolve: () => 'Hello World'
            }
        })
    })
})

app.use('/graphql', graphqlHTTP({
    schema: schema,
    graphiql: true
}))
```

En el esquema definimos la query, en este caso es un ejemplo y hemos definido un único objeto llamado `HelloWorld`. Indicamos sus atributos, este solo tiene un campo `message` de tipo `GraphQLString`, en `resolve` indicamos cómo obtener los datos. Por último indicamos el *schema* que estamos usando. Ahora si accedemos al endpoint desde el navegador vemos la interfaz gráfica de GraphiQL y podemos hacer nuestra query:

`query {`
`  message`
`}`
	→	`{`
		`  “data”: {`
		`  “message”: “Hello World”`
		`  }`
		`}`

No hay que indicar el objeto `HelloWorld` ya que en nuestro caso hemos dicho que `query` es igual a `HelloWorld`. Al hacer una query ya estamos dentro del objeto HelloWorld y podemos acceder a sus atributos. A continuación vamos a hacer un ejemplo más completo en el que tendremos distintos objetos y usaremos un objeto raíz para agruparlos en la query y poder acceder a todos.
En el ejemplo práctico que vamos a construir tendremos distintos objetos relacionados entre sí, vamos a hacer una API en la que podremos consultar las autonomías, las provincias y los municipios de España. Queremos poder recuperar el listado con todos los recursos pero

también queremos poder recuperar una única provincia o un único municipio y navegar entre sus atributos: obtener los municipios de una provincia, las provincias de una autonomía etc. Por lo tanto nuestros objetos principales de tipo `ObjectType` son `Autonomia`, `Provincia` y `Municipio`. Vamos a agrupar los tres dentro de un objeto raíz que nos permite poder consultar cualquiera de ellos al hacer una query.

Empezaremos por lo más sencillo que es recuperar una lista con las autonomías. En este ejemplo no vamos a usar una base de datos ya que no es nuestro objetivo, la información la guardamos en arrays de javascript en memoria. Nuestras autonomías en formato JSON van a ser así:

```
{
    "autonomia_id": "01",
    "nombre": "Andalucía"
}
```

Las vamos a agrupar en un array y a continuación vamos a definir el tipo `RootQuery` en el que indicamos todos los campos que podemos consultar. Cómo queremos obtener una lista de autonomías definimos el campo `autonomias` de tipo `List`. Tendremos que definir a parte un `ObjectType` que represente una autonomía indicando sus campos y sus tipos:

```
const autonomias = require('./data/autonomias.json')

const AutonomiaType = new GraphQLObjectType({
    name: 'Autonomia',
    description: 'Representa una autonomia',
    fields: () => ({
        autonomia_id: { type: GraphQLNonNull(GraphQLString) },
        nombre: { type: GraphQLNonNull(GraphQLString)}
    })
})

const RootQueryType = new GraphQLObjectType({
    name: 'Query',
    description: 'Root Query',
    fields: () => ({
        autonomias: {
            type: new GraphQLList(AutonomiaType),
            description: 'Lista de autonomías',
            resolve: () => autonomias
        }
    })
})

const schema = new GraphQLSchema({
    query: RootQueryType
})
```

Indicamos cómo obtener la lista de autonomías que simplemente es el array que hemos definido. Los campos de `AutonomiaType` no hace falta resolverlos porque se corresponden con el nombre de los atributos de cada elemento de nuestro array.
Podemos comprobar la consulta en GraphiQL:

<table>
<tr>
<td>

```
query {
  autonomias {
    autonomia_id,
    nombre
  }
}
```

</td>
<td>→</td>
<td>

```
{
  "data": {
    "autonomias": [
      {
        "autonomia_id": 1,
        "nombre": "Andalucía"
      },
      {
        "autonomia_id": 2,
        "nombre": "Aragón"
      },
      ...
    ]
  }
}
```

</td>
</tr>
</table>

De la misma forma vamos a añadir la lista de provincias que va a ser así:

```
{
    "autonomia_id": "08",
    "provincia_id": "02",
    "nombre": "Albacete"
}
```

Añadimos un nuevo campo provincias que es una lista a `RootQuery` y definimos el tipo `ProvinciaType`. Y ahora también podemos consultar la lista de provincias.

```
fields: () => ({
    ...
    provincias: {
        type: new GraphQLList(ProvinciaType),
        description: 'Lista de provincias',
        resolve: () => provincias
    }
})
```

```
const ProvinciaType = new GraphQLObjectType({
    name: 'Provincia',
    description: 'Representa una provincia',
    fields: () => ({
        autonomia_id: { type: GraphQLNonNull(GraphQLString) },
        provincia_id: { type: GraphQLNonNull(GraphQLString) },
        nombre: { type: GraphQLNonNull(GraphQLString)}
    })
})
```

Sin embargo esto sólo nos permite obtener los listados, aún no podemos navegar entre sus campos para obtener todas las provincias de una autonomía o saber el nombre de la autonomía en la que se encuentra una provincia. Para poder navegar desde una provincia y obtener los datos de su autonomía tenemos que añadirle un nuevo campo `autonomia` de tipo `AutonomiaType` a `ProvinciaType` e indicarle cómo resolverlo:

```
fields: () => ({
```

```
    autonomia_id: { type: GraphQLNonNull(GraphQLString) },
    autonomia: {
        type: AutonomiaType,
        resolve: (provincia) => {
                return autonomias.find( autonomia => autonomia.autonomia_id ===
provincia.autonomia_id )
        }
    },
    provincia_id: { type: GraphQLNonNull(GraphQLString) },
    nombre: { type: GraphQLNonNull(GraphQLString)}
})
```

Una vez le añadimos el nuevo atributo y le indicamos cómo obtener el valor ya podemos navegar desde la provincia hacia su autonomía en nuestras queries, de forma que podemos hacer esta consulta:

```
query {
  provincias {
    nombre,
    autonomia{
      nombre
    }
  }
}
```

→

```
{
  "data": {
    "provincias": [
      {
        "nombre": "Albacete",
        "autonomia": {
          "nombre": "Castilla - La Mancha"
        }
      },
      {
        "nombre": "Alicante/Alacant",
        "autonomia": {
          "nombre": "Comunitat Valenciana"
        }
      },
      ...
    ]
  }
}
```

Nos falta el otro lado de la relación, poder obtener a partir de una comunidad autónoma su listado de provincias. Ya sabemos como hacerlo, añadimos a `AutonomiaType` su atributo provincias que será una lista de `ProvinciaType` e indicamos cómo resolverlo. En nuestro caso filtramos el array de javascript para obtener los elementos que nos interesan, en otras aplicaciones llamaríamos a nuestros métodos de servicio que a su vez llamarían a la base de datos, pero esos detalles están fuera de GraphQL y dependen de nuestro proyecto. Vamos a ver como añadir el listado de provincias:

```
fields: () => ({
    autonomia_id: { type: GraphQLNonNull(GraphQLString) },
    nombre: { type: GraphQLNonNull(GraphQLString)},
    provincias: {
        type: GraphQLList(ProvinciaType),
        resolve: (autonomia) => {
```

```
                return provincias.filter( provincia => provincia.autonomia_id ===
autonomia.autonomia_id)
        }
    }
})
```

Este es el resultado de las consultas en las que pedimos las provincias de las autonomías:

Los municipios se añadirían igual que hemos hecho hasta ahora, definimos un `MunicipioType`, añadimos a `RootQuery` un nuevo campo `municipios` que es una lista para poder consultarlo y lo relacionamos con su provincia añadiendo el campo provincia e indicando cómo resolverlo. Si queremos poder ver los municipios de una provincia también podemos añadirle a la provincia el campo municipios. Una vez relacionado el tipo municipio con el tipo provincia, también podremos relacionar autonomías y municipios gracias a las relaciones ya existentes sin añadir nada más de código. Aquí empezamos a ver las ventajas de GraphQL y podemos empezar a pensar en grafos, una vez conectado el nuevo nodo (municipios) con uno ya existente (provincias) no solo podemos navegar entre ellos sino también hacia el resto de nodos con los que las provincias estuvieran relacionadas sin añadir nada de código.

Hasta ahora hemos estado devolviendo la lista entera de elementos, pero esto es muy limitado. Uno de los casos más comunes en las consultas es querer recuperar los datos de un elemento concreto definido por un identificador. Para hacer esto usamos los parámetros. Vamos a ver el ejemplo para devolver una sola provincia según su identificador. En el objeto `RootQuery` añadimos un nuevo campo provincia del tipo `ProvinciaType` que ya definimos, pero indicamos que puede recibir un argumento y en el resolve le decimos cómo usar el argumento para recuperar la información:

```
fields: () => ({
        ...
        provincia: {
            type: ProvinciaType,
            description: 'Una provincia',
            args: {
                id: { type: GraphQLString }
            },
                    resolve: (parent, args) => provincias.find( provincia =>
provincia.provincia_id === args.id)
        },
        ...
    })
```

Simplemente añadiendo esto ya podemos recuperar una provincia dado su identificador. Y al ser de un tipo ya definido seguimos pudiendo acceder a sus atributos y obtener también sus municipios y la comunidad autónoma en la que se encuentra:

```
{
  provincia(id: "03"){
    municipios{
      nombre
    }
  }
}
```

→

```
{
  "data": {
    "provincia": {
      "municipios": [
        {
          "nombre": "Agost"
        },
        {
          "nombre": "Agres"
        },
        ...
      ]
    }
  }
}
```

Los parámetros no sólo sirven para recuperar objetos por su identificador. Como ya vimos anteriormente se pueden utilizar para hacer consultas más específicas de muchas formas. Para filtrar los datos, para ordenarlos, para paginarlos, para especificar el formato en el que se quieren recibir… Yo he añadido otro parámetro a los tipos lista para poder filtrar por un término de búsqueda y que sólo se devuelvan los resultados cuyo nombre contiene el término de búsqueda.

Además como podemos especificar argumentos para cada campo que solicitamos podemos hacer consultas como esta en la que queremos los municipios de alicante cuyo nombre contiene “beni”:

```
{
  provincia(id: "03"){
    nombre
    municipios(nombreContains:
"beni"){
      nombre
    }
```

→

```
{
  "data": {
    "provincia": {
      "nombre": "Alicante/Alacant",
      "municipios": [
        {
          "nombre": "Beniarbeig"
```

```
  }
}
```

```
        },
        {
          "nombre": "Beniardá"
        },
        ...
      ]
    }
  }
}
```

Conclusión

GraphQL nos ofrece una alternativa interesante a REST para el diseño de APIs. El concepto de GraphQL para hacer queries es muy sencillo e intuitivo pero a la vez muy flexible y es especialmente útil desde el frontend al permitirnos especificar exactamente los datos que queremos recuperar. Tiene otras ventajas como el endpoint único que nos evita tener que definir nuevos endpoints para nuevas peticiones, con GraphQL en su lugar definimos nuevos tipos que además son reusables. Esto junto con las numerosas librerías que nos ayudan a implementarlo lo hacen muy interesante. Sin embargo esto no quiere decir que sean mejores que las APIs REST o que vayan a sustituirlas, el paradigma REST sigue teniendo un gran peso e igualmente nos permite hacer la mayoría de cosas que permite GraphQL. Es nuestra decisión personal en base a nuestras preferencias o nuestra experiencia decidir cómo vamos a diseñar nuestras APIs.

Referencias

Página oficial de GraphQL, la mayoría de información la saqué del apartado learn.
https://graphql.org

Videotutorial de GraphQL con NodeJS y express
https://www.youtube.com/watch?v=ZQL7tL2S0oQ

Una serie de artículos sobre GraphQL muy interesantes. Explica muy bien la paginación.
https://buddy.works/tutorials/what-is-graphql-and-why-facebook-felt-the-need-to-build-it