UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Integrando fuentes heterogéneas de datos en Web con GraphQL

Ángel Garrido Román

Julio, 2018

UNIVERSIDAD DE EXTREMADURA
Escuela Politécnica
Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado
Integrando fuentes heterogéneas de datos en Web con GraphQL

Autor/es: Ángel Garrido Román Fdo.:

Director/es: Roberto Rodriguez Echeverria Fdo.:

Tribunal Calificador

Presidente: Álvaro Prieto Ramos
Fdo.:

Secretario: María Encarnación Sosa Sanchez
Fdo.:

Vocal: Alberto Gómez Mancha
Fdo.:

CALIFICACIÓN:
FECHA:

Índice

Tabla de Ilustraciones

Índice de Tablas

Resumen

En este Trabajo Fin de Grado se explora la nueva tecnología de tratamiento de datos GraphQL[i].

Con ella hemos podido conseguir consultar, unificar y estandarizar los atributos en datos en Fuentes de Datos Abiertas totalmente heterogéneas, sin importar como estén nombrados los atributos de estos ni cómo nos devuelvan los datos. De este modo el cliente pueda consultar los datos de las distintas Fuentes de Datos Abiertas desde un único punto de acceso como si estuviese accediendo a una misma Base de Datos.

El Trabajo Fin de Grado consta de dos Proyectos:

Servidor GraphQL: Implementado en JavaScript[ii], es el encargado de unificar las distintas Fuentes de Datos Abiertas y mapear los atributos de manera que para el cliente todos los atributos se llamen de la misma forma sin importar su procedencia; se han tomado como ejemplos los autobuses de Barcelona, Santander, Málaga, Cáceres, y las bicicletas de Barcelona, estas últimas por ilustrar de una manera más clara que pueden unificarse distintos tipos de datos sin problemas en las consultas.

El propio servidor GraphQL dispone de un cliente web, que es el motor gráfico de consultas de llamado GraphiQL[iii], el cual permite acceder al servidor desde la web y realizar consultas mediante una interfaz gráfica.

Cliente Java: Se ha decidido implementarlo en Java con el fin de ilustrar que no es necesario que el cliente y el servidor estén implementados en el mismo lenguaje.

Consta de un Parser para poder crear nuevos modelos para el servidor y unificarlos al esquema del servidor de forma automática mediante plantillas realizadas en formato Json. También consta de varias consultas a modo de ejemplo para poder ver la funcionalidad del servidor.

1. Introducción

1.1 Motivación

Con el avance de la tecnología y la revolución digital el volumen de información a la que tenemos acceso se ha visto incrementado drásticamente, pero la forma de consumirlo no ha variado mucho a lo largo del tiempo.

La forma de consumir datos que hemos estado teniendo hasta ahora no ha tenido en cuenta la posibilidad de conexiones móviles, pues supone que disponemos de conexiones estables y de alta velocidad, las cuales no se ven afectadas por tener que descargar más información extra para poder obtener los mismos resultados.

Este Trabajo Fin de Grado nace como una alternativa a la forma de consumir datos en internet que existe actualmente.

La motivación principal es la de intentar ser lo más eficientes posible a la hora de poder transmitir los datos de los distintos Opendata[iv]. Para esto, se ha buscado una tecnología capaz de poder consumir distintas fuentes de datos abiertos y devolver al usuario única y exclusivamente la información que ha pedido.

Para poder conseguir esto se ha utilizado una nueva tecnología, GraphQL.

1.2 ¿Por qué GraphQL?

Cuando se crearon las primeras APIs allá por el 2000, un simple CRUD era suficiente en muchos casos, pero ya no es así, haciendo de las APIS RESTful algo cada vez más tedioso de mantener. REST ha sido un claro caso de éxito, pero su concepción CRUD basada en recursos, verbos y códigos de respuesta HTTP la hacen tener cada vez más limitaciones debido a su inflexibilidad.

Uno de los principales problemas que nos encontramos con RESTful es que utilizamos una URI para leer o escribir un único recurso, también denominado endpoint o punto de acceso.

Por cada tipo de recurso distinto que queramos manejar necesitaremos un punto de acceso distinto; ahora bien, en la mayoría de los casos reales actuales, cuando se pide información, no solamente se pide un único recurso, lo cual implica varias llamadas a distintos puntos de acceso, las cuales no solamente nos devuelven la información que pedimos, sino que nos devuelven todo el conjunto de datos relativos al recurso que esté alojado en esa URI.

Por ejemplo, si queremos el DNI de una persona, el recurso nos traerá más información extra la cual no nos sirve para nada en nuestra consulta como fecha de nacimiento, nombre, apellidos… Estos datos tienen que ser manejados y filtrados, tarea que habrá que realizar para todos los distintos objetos que tengamos que llamar, algunas opciones para aliviar la carga de estos puntos de acceso es la creación de APIS adhoc en el Backend.

Con la creación de APIS adhoc se crea una pseudo RESTful, saltándose varias reglas principales de RESTful, haciendo que lo que parece una buena idea se convierta en un caos de código duplicado, abstracciones y falta de retro compatibilidad, con un sinfín de pequeños puntos de acceso para cada dato del objeto y luego un punto de acceso extra para el total del objeto.

Pero el mayor problema que tenemos actualmente con la tecnología REST es que no podemos elegir los datos que queremos recibir en nuestro JSON/Payload de respuesta sin el uso de estas APIS adhoc y toda la complejidad que esto entraña.

Este proyecto trata de mostrar una alternativa a las API REST haciendo uso de la tecnología GraphQL, con lo que consumiremos la información proporcionada por los distintos Opendata de ejemplo de manera mucho más eficiente que si de una API REST se tratase, y se mostrará que el

mantenimiento o ampliación del proyecto es trivial comparada con las actuales tecnologías REST.

1.3 Objetivo

El objetivo principal de este Trabajo Fin de Grado es demostrar la ventaja del uso de GraphQL frente a las soluciones API REST actuales en un entorno de consumo de datos de fuentes heterogéneas (Opendata en nuestro caso).

Para la realización de este Trabajo Fin de Grado se han tomado los datos de Autobuses en los Opendata de Barcelona, Cáceres, Santander y Málaga, con distinta forma de entregar los datos unos de otros como las Bicicletas de Barcelona, otro Opendata el cual contiene información totalmente dispar con el resto de los Opendata.

- Barcelona: Mediante el uso de un dataset con su resource id
- Cáceres: Mediante el uso del punto de acceso de SparQL
- Santander: Devuelve un simple Json con toda la información
- Málaga: Mediante el uso de un dataset con su resource id
- Bicicletas Barcelona: Devuelve un simple Json con toda la información.

Ya que GraphQL puede ser implementado en multitud de lenguajes, se ha optado por JavaScript.

Otro objetivo de este Trabajo Fin de Grado es el de unificar todos los nombres de los distintos Opendata para que el cliente no tuviera que preocuparse de cómo se llaman las mismas características de objetos similares en los distintos Opendata, ya que uno de los mayores problemas a la hora de consultar las distintas fuentes de datos abiertos es que no existe homogeneidad ninguna para el tratamiento de datos, es decir, cada Opendata nombra los distintos atributos de los que se componen sus datos de formas totalmente distintas unos de otros.

Con esto conseguimos cumplir los siguientes objetivos:

- ***Un único punto de acceso***: El cliente podrá acceder a todos los servicios desde un único punto de acceso “/GraphQL”

- ***Unificación de nombres:*** El cliente no tiene por qué preocuparse de los distintos nombres que puedan tener los objetos en las distintas fuentes de datos abiertos pues GraphQL se los presentará unificados bajo el mismo nombre.

- ***Filtrado de datos:*** El cliente solamente recibirá los datos filtrados por la consulta realizada en la capa GraphQL de manera que no tendrá que traerse todos los datos.

- ***Múltiples peticiones simultáneas:*** Se podrán realizar desde el lado del cliente una consulta con peticiones a distintas Fuentes de Datos Abiertos y en cada una decidir qué datos quiere traerse, pues la parte de GraphQL del servidor se encargará de devolver un único Json con toda la información requerida.

- ***Unificación de Distintos tipos de fuentes de Datos Abiertos:*** Se podrá acceder con una misma consulta a la información específica correspondiente a Autobuses y Bicicletas, las cuales están en distintos Opendata y son totalmente dispares.

2. Alcance del proyecto

En este Trabajo Fin de Grado se ha desarrollado una API GraphQL que permite acceder a los datos de las distintas Fuentes de datos Abiertos, de manera que, desde el cliente, se pueda acceder a los datos filtrados que nos devuelven las distintas posibles consultas, cumpliendo de esta manera los objetivos propuestos. Para hacer esto diferenciamos tres partes: la parte del Cliente, la parte del Servidor GraphQL y la parte de las distintas bases de datos (Fuentes de datos Abiertos).

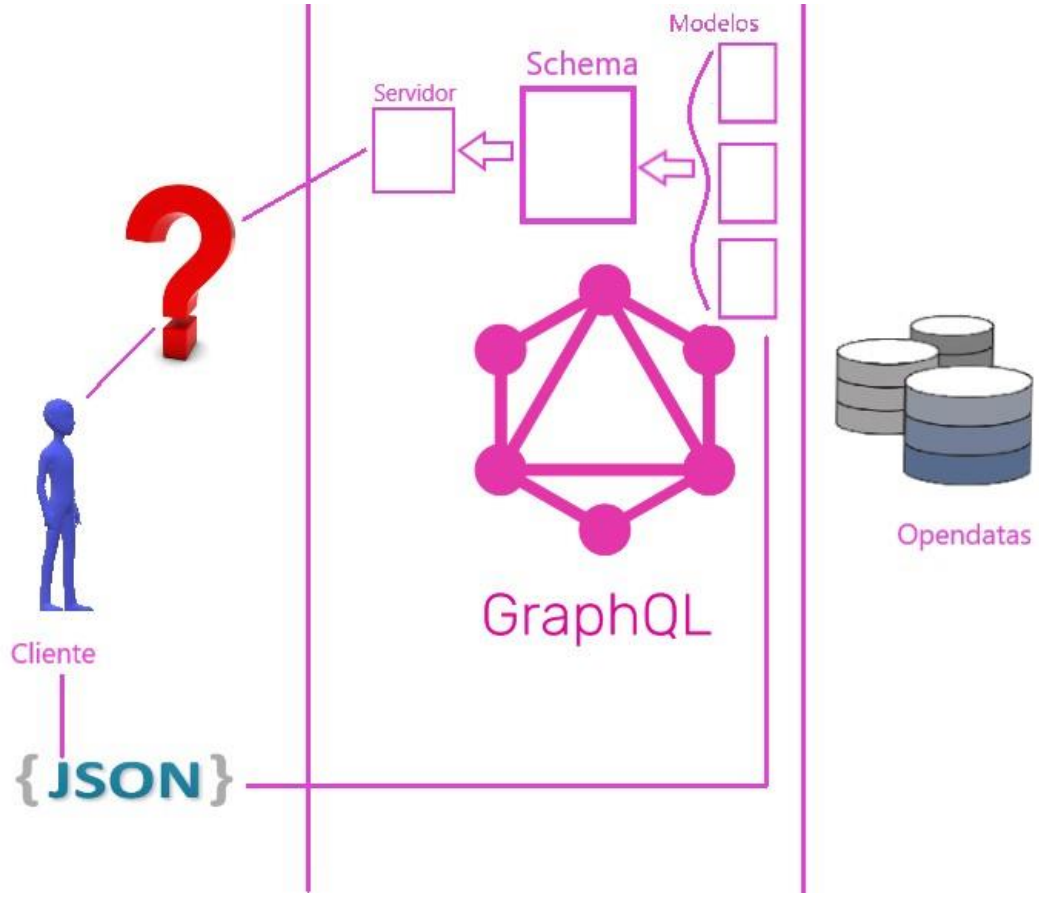


Ilustración 1 Las tres partes del proyecto

En la imagen podemos observar que el proyecto se estructura de la siguiente manera:

Parte del cliente

El cliente es la parte del proyecto con la cual interactuará quien quiera conectarse a la API de GraphQL y está desarrollado en Java.

Puede dividirse a su vez en dos partes muy diferenciables, Parser y Consultas:

Parser: Encargado de transformar un objeto Json siguiendo la estructura dada para poder añadir una nueva Fuente de Datos Abiertos al Esquema del servidor GraphQL en caso de no existir en este, y, gracias a esto añadirlo también al Esquema global para que pueda ser consultado.

Si tenemos algún nuevo objeto Json con una Fuente de Datos Abierta no existente, al ejecutar el cliente el parser puede transformar esos datos en un archivo JavaScript con el modelo de la nueva Fuente de Datos Abiertos e incluirlo en el Esquema. En el caso de nuestro Trabajo Fin de Grado el parser ha sido diseñado para trabajar con autobuses, aunque podrían crearse distintos parser para distintos tipos de datos.

Consultas: Se dispone de un menú principal con varias consultas a modo de ejemplo para nuestro Trabajo Fin de Grado, las consultas disponibles son las siguientes:

Una consulta de introspección, mediante la cual nos devolverá los distintos Opendata disponibles en la API junto con los campos de cada uno de estos. Una consulta con dos fuentes de datos distintas, Cáceres y Barcelona con distintos filtros en la búsqueda.

Una consulta que mezcla cuatro fuentes de datos distintas, Santander, Barcelona, Cáceres y Málaga y en cada una pide distinto número de datos y distintos campos del objeto con distintos filtros en cada uno de los Opendata.

Una consulta que mezcla en una misma petición un número de autobuses de Barcelona y unas bicicletas, llamando de esta forma a dos fuentes de datos distintas y además no relacionadas para nada entre sí.

Una última consulta que se encarga de pedir todos los datos de todas las distintas fuentes de datos de las que disponemos en el ejemplo de nuestro Esquema (Autobuses de Cáceres, Málaga, Barcelona y Santander y Bicicletas de Barcelona).

Capa GraphQL:

Parte del Trabajo Fin de Grado que abarca el Servidor, desarrollado en JavaScript, compuesta por las siguientes partes:

Servidor: Es la parte encargada de exponer el servidor GraphQL al exterior y activar la opción de GraphiQL (la interfaz gráfica del servidor GraphQL para consultas web)

Esquema: El Esquema es el eje central del Servidor GraphQL; es donde tendremos la Consulta principal, desde la cual podremos acceder a todos los modelos de los que disponga el servidor. En esta parte del Servidor se especifican los puntos de acceso a las distintas fuentes de datos y todos los filtros que quieran aplicarse a las futuras consultas a realizar sobre estas.

Modelos: Los modelos son los que especifican los campos de cada uno de los objetos de las Fuentes de Datos Abierto a los que hagan referencia. Pueden ser campos creados a partir de la unión de varios campos del objeto o el resultado de funciones lógicas a partir del campo del objeto. La unión de todos los distintos modelos son la que conforman el Esquema del Servidor, y también permite renombrar los campos con lo cual es el lugar en el cual se realiza el mapeo de todos los campos de los distintos modelos.

Opendata

Parte externa al proyecto que conforma las distintas Fuentes de Datos Abiertos en los que realizaremos las peticiones de datos.

3. Desarrollo del plan del Proyecto

Las bases para la realización del proyecto se establecieron en junio de 2017 y se empezó a trabajar en él en Julio de 2017.

Diferenciaremos este desarrollo en dos partes. En una hablaremos de la cronología del desarrollo y en otra la estimación temporal de las horas dedicadas.

3.1 Fases del Proyecto

3.1.1 Toma de contacto

La primera fase y quizás la más importante de todas fue la toma de contacto. Al tratarse de una tecnología totalmente nueva y de la que desconocía todo, lo primero que se realizó fue una búsqueda exhaustiva de información en la red, toda la información de GitHub [v] fue de gran utilidad para este fin.

Primero aprendiendo qué era GraphQL, y cómo podía mejorar las funcionalidades REST actuales, congresos y charlas especializadas a las cuales se ha podido acceder gracias a estar en YouTube[vi] y una gran cantidad de páginas con información desde lo más básico a operaciones y ejemplos más complejos.

3.1.2 Realización de ejemplos

Una vez entendido que era GraphQL y cómo funcionaba se realizaron varios ejemplos [vii]en bases de datos locales para aprender la estructura de un proyecto de estas características.

Para ello se montó una pequeña Base de Datos con Mongo DB en la cual se crearon distintos tipos de datos y mediante un pequeño Esquema y un par de Modelos en GraphQL se realizaron consultas de prueba.

3.1.3 Primera versión

En una primera versión solamente disponíamos de un Esquema, con su correspondiente consulta principal, sin estructurar los modelos aparte. En poco tiempo se volvería demasiado grande y difícil de manejar, con lo que se tomó la determinación de modularizarlo, haciendo, de esta manera, un proyecto mucho más manejable y fácil de ampliar.

3.1.4 Versión final

Como la utilización de GraphQL necesitaba de la instalación de NodeJS y varios paquetes de dependencias (GraphQL, express, express-GraphQL [viii], request y request-promise), la instalación y la dependencia del cambio de versiones acabó siendo un problema, con lo que se decidió optar por la implementación del servidor en un contenedor Docker[ix].

La versión final del Trabajo Fin de Grado se ha dividido en dos proyectos: la parte de cliente, la cual no existía en una primera versión, y la parte de servidor. La parte del cliente al principio solo iba a servir para demostrar que se podía crear un cliente en otro lenguaje (Java en este caso) para poder ver como con una sola consulta se podían recuperar datos específicos de distintas Fuentes de Datos Abiertos. No obstante, al final se tomó la decisión de ampliarlo con la posibilidad de poder parsear mediante un objeto Json los modelos de distintas Fuentes de Datos Abiertos (para el caso particular de los Autobuses), y crear nuevos modelos en JavaScript para, después, añadirlos de forma automática al Esquema principal del servidor, con lo que también se añadió un menú para poder escoger entre algunas de las consultas.

En la parte del servidor se decidió añadir un modelo que no tuviese nada que ver con todos los demás (Bicicletas de Barcelona) y unirlo en el Esquema principal para, así, demostrar que en un mismo Esquema pueden convivir Fuentes de Datos Abiertos totalmente heterogéneas y que estas pueden ser accesibles con una misma consulta.

Se definieron modelos de cada una de las Fuentes de Datos Abiertos y se aprovechó la elección de JavaScript como lenguaje de la parte del servidor para poder realizar el mapeo de los nombres de los atributos en el propio modelo.

También se descubrió la posibilidad de poder realizar funciones lógicas en la parte del modelo con cada uno de los atributos recuperados de manera que podíamos unir varios atributos, como, por ejemplo, latitud y longitud para crear un nuevo atributo “Coordenadas” o aprovechar algunos atributos que devolvían Verdadero o Falso para indicar la dirección del Autobús.

3.1.5 Estimación Temporal

Las dos primeras fases del proyecto fueron realizadas desde mediados de julio hasta octubre. Desde octubre y hasta diciembre, se realizó una primera versión, la cual tuvo que descartarse por completo porque algunas dependencias sufrieron cambios tan grandes que el proyecto dejó de ser operativo. La versión final comenzó a finales de diciembre y se terminó a principios de marzo; en abril se introdujo el modelo de Bicicletas para ilustrar de una manera más clara la unión de datos heterogéneos en un mismo Esquema, y en junio se empezó a desarrollar la memoria.

Actividad	Inicio	Final	15-jun	01-jul	15-jul	01-ago	15-ago	01-sep	15-sep	01-oct	15-oct	01-nov	15-nov	01-dic	15-dic
Establecer Bases y Propuesta de TFG	15/06/2017	01/07/2017													
Toma de Contacto con GraphQL	15/07/2017	15/10/2017													
Realización de Ejemplos y Casos Base	05/09/2017	15/10/2017													
Primera Versión del Proyecto	15/10/2017	20/12/2017													

Ilustración 2 Diagrama de Gantt 1

Aquí podemos apreciar el diagrama de Gantt de lo que sería la primera parte del Trabajo Fin de Grado, desde el establecimiento de las bases y propuesta por parte del tutor del Trabajo Fin de Grado hasta el final de la primera versión del proyecto.

Actividad	Inicio	Final	01-ene	15-ene	01-feb	15-feb	01-mar	15-mar	01-abr	15-abr	01-may	15-may	01-jun	15-jun	01-jul	15-jul
Segunda Versión del Proyecto	22/12/2017	15/03/2018														
Ampliación Bicicletas	01/04/2018	15/04/2018														
Revisión de Fallos y Correcciones del Proyecto	01/05/2018	01/06/2018														
Redacción de la Memoria Final	01/06/2018	01/07/2018														

Ilustración 3 Diagrama de Gantt 2

A continuación, se muestra el diagrama de Gantt desde que se comienza con la segunda versión del proyecto hasta el fin de la redacción de la memoria final del mismo.

4. Estado del Arte

A continuación, se mostrarán las herramientas y diferentes posibilidades para realizar el proyecto que existen en la actualidad y también especificaremos por qué hemos decidido usar las que usamos en el proyecto para resolver el problema planteado.

4.1 Fuentes de Datos Abiertos (Opendata)

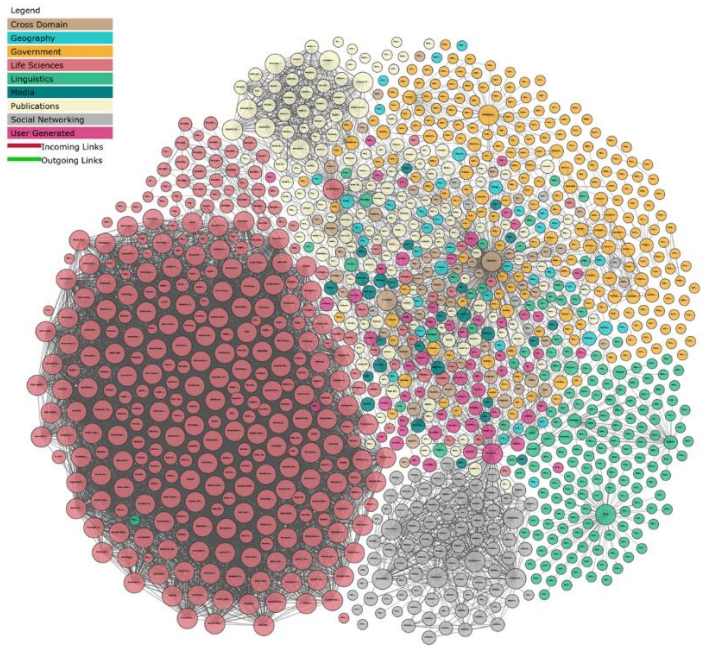


Ilustración 4 Nube de Datos Abiertos [x]

El concepto de Fuente de Datos Abiertos es una filosofía y práctica que persigue que determinado tipo de datos estén disponibles de forma libre para todo el mundo, sin restricciones ni derechos de autor, patentes u otros mecanismos de control, permitiendo su reutilización.

Actualmente en España existe un problema con dichas Fuentes de Datos Abiertos, y este es que no están estandarizadas. En la mayoría de los casos no cumplen un mínimo de accesibilidad, ponen restricciones y mecanismos de control, con estas trabas, se ha tenido que reducir la muestra a los Opendata de Barcelona, Cáceres, Málaga y Santander, algunas Fuentes de Datos Abiertos en España, que, aunque todas guardaban los datos de los Autobuses de las respectivas ciudades, eran totalmente heterogéneas entre ellas.

Los objetivos del movimiento de datos abiertos son los siguientes:

- **Acceso abierto**: Hacer públicas y libres las publicaciones técnicas universitarias en internet.

- **Contenido abierto**: Dirigir los recursos para un público humano (texto, fotos, videos) accesibles de forma libre.

- **Conocimiento abierto**: Apertura en datos abiertos (Científicos, históricos, geográficos, música, películas, libros, administración pública y gobierno)

- **Ciencia de datos abiertos**: Aplicación de los datos abiertos a métodos científicos, incluyendo experimentos fallidos y conjuntos grandes de datos experimentales.

- **Software libre**: Licencias bajo las que se pueden distribuir programas informáticos, no relacionado con los datos en sí normalmente.

- **Ciencia abierta**: Aproximación para los conjuntos de datos científicos interrelacionados, métodos y herramientas para conseguir transparencia, escalabilidad e investigaciones entre disciplinas.

4.1.1 Conjunto de datos (Dataset)

Un dataset (Conjunto de datos) hace referencia a una agrupación de datos sobre una misma categoría.

En muchos Opendata los datos en bruto están organizados en categorías para su mejor explotación; estos Opendata también cuentan con campos dentro de los Dataset para informar de la descripción y frecuencia de actualización, e incluso el formato en el que son presentados los datos.

4.2 Docker

Docker usa el kernel de Linux y funciones de este como Cgroups y namespaces para segregar procesos, y, de esta manera, poder ejecutarlos de manera independiente.

Esta independencia es justamente la que buscamos en el proyecto, ya que con esto conseguimos poder ejecutar una serie de procesos y aplicaciones (con todas sus respectivas dependencias) de manera que no se vean afectados por el resto del entorno, manteniendo su propia infraestructura y la seguridad que tendríamos con sistemas separados.

Ilustración 5 Docker[xi]

Docker ofrece un modelo de implementación basado en imágenes, esto permite que la misma aplicación pueda ser compartida junto con todas sus dependencias en entornos diferentes. También se encarga de automatizar la implementación de la aplicación dentro del entorno del contendor, dotándolo de una capacidad de implementar y controlar las versiones y distribución de las aplicaciones sin precedentes.

Hay que tener en cuenta que la tecnología Docker no es la misma que la de los contenedores tradicionales de Linux

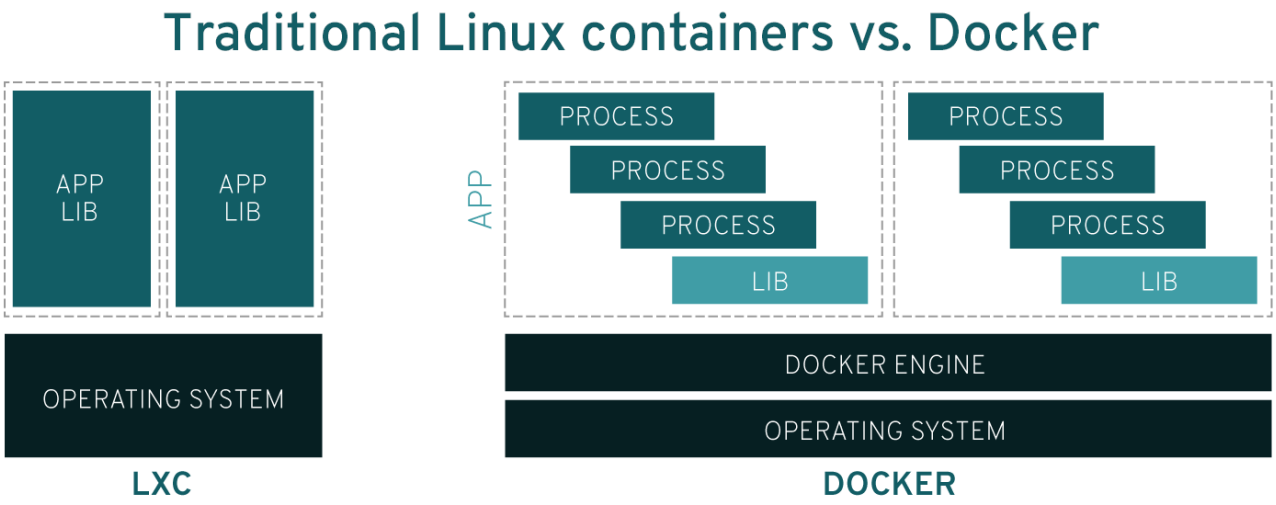


Ilustración 6 Contenedores tradicionales vs Docker[xii]

En un comienzo se desarrolló basándose en la tecnología LXC (tecnología de virtualización para Linux a nivel de sistema operativo la cual permite ejecutar múltiples instancias de sistemas operativos aislándolos en un mismo servidor físico, es un entorno virtual con su propio espacio de procesos y redes, ya que no provee una máquina virtual) pero no tenía buena experiencia de usuario o desarrollador, la tecnología de Docker aparte de aportar la capacidad de ejecutar contenedores también facilita el proceso de crear y desarrollar contenedores, enviando versiones de imágenes de dichos contenedores entre otras cosas, como por ejemplo el dividir los procesos de las aplicaciones que ejecutan dotándolo de un enfoque modularizado con muchas ventajas sobre todo a nivel de micro servicios.

Cada archivo de imagen Docker está formado por varias capas, las cuales se combinan en una única imagen; esta se vuelve a crear una vez la imagen cambia, las capas se crean cuando los usuarios ejecutan un comando (copiar y ejecutar…). Al utilizar estas capas para construir nuevos contenedores, hace que el proceso de construcción sea mucho más rápido, los cambios intermedios se comparten entre imágenes, mejorando la velocidad tamaño y eficiencia. Esto también hace que el control de versiones sea inherente a la creación de capas, lo cual facilita de paso la restauración a imágenes anteriores (versiones anteriores).

La implementación de Docker tarda segundos, ya que, al crear un contendor por cada proceso, puede compartir estos procesos rápidamente, y, debido a que no necesita de ningún sistema operativo para agregar o mover contenedores, el tiempo de implementación es significativamente menores.

4.2.1 ¿Por qué usar Docker en el proyecto?

Por todas las ventajas mencionadas anteriormente se decidió mantener el proyecto en Docker, ya que cuando empezamos a trabajar con GraphQL estábamos trabajando con una tecnología muy cambiante y eso mismo nos hizo perder meses al tener que empezar de nuevo, con lo cual para evitar esos problemas de nuevo se decidió esta solución, aislando nuestra aplicación del resto y manteniendo un control total sobre las versiones y la posibilidad de volver atrás sin riesgos en caso de cualquier problema.

4.3 Node Js

Node.js es un entorno de código abierto, en tiempo de ejecución multiplataforma para la capa del servidor, asíncrono con entrada y salida de datos en una arquitectura orientada a eventos basado en el lenguaje de programación ECMAScript, (basada en el motor V8 de Google), se ha incluido como capa del servidor en el Trabajo Fin de Grado pues fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables lo cual se ajusta a la idea de nuestro proyecto.

Ilustración 7 Logo Node.js[xiii]

Creado en 2009 por Ryan Dahi Node.js es similar en cuanto a propósito a Twisted o Tornado de Python, pero al contrario que la mayoría del código de JavaScript, no se ejecuta en un navegador, sino en la parte del servidor. Node.js implementa algunas especificaciones de CommonJS e incluye un entorno de depuración interactiva (REPL).

Levantar un servidor con Node.js es algo muy sencillo de realizar. En este Trabajo fin de Grado no se usan más de 14 líneas de código, incluyendo imports, para poder levantarlo, más adelante en este documento se mostrarán y explicarán con más detenimiento.

Node.js utiliza un único hilo de ejecución como modelo de evaluación, con entradas y salidas asíncronas las cuales se ejecutan concurrentemente hasta ciento de miles sin que esto incurra en costes asociados al cambio de contexto (ejecución de una rutina perteneciente al núcleo del SO multitarea de un ordenador, destinado a parar la ejecución de un proceso para dar paso a otro distinto).

Un problema del único hilo de ejecución es que Node.js necesita de módulos adicionales para escalar la aplicación con el número de núcleos de procesamiento de la máquina en la que se esté ejecutando como Cluster.

Node.js está formado por módulos: incorpora una serie de módulos "básicos" precompilados en el propio binario, como el de red o asíncrono, path, buffer… y otros de propósito más general como Stream; se pueden utilizar módulos adicionales que aumenten la utilidad de Node.js utilizados por terceros, ya sean precompilados como archivos ".node" o archivos JavaScript planos (estos últimos se implementan siguiendo la especificación CommonJS para módulos).

Esto nos ha servido especialmente de utilidad en el Trabajo fin de Grado pues hemos utilizado Node.js para incluir los módulos de Express, Express-GraphQL, GraphQL, graphql-tools, lodash, request y request-promise.

Para poder incluir módulos en un entorno de Node.js lo único que hay que hacer es abrir una terminal en el proyecto en el cual queramos incluirlos y mediante el comando npm install <nombre_del_móduo> (ese comando lanza el Node Package Manager) nos descarga el paquete y lo instala en nuestro entorno, incluyéndolo en el package.json (Archivo .json en el cual gestiona todas las dependencias y las versiones de estas en el proyecto).

THE BATTERY OPEN-SOURCE SOFTWARE INDEX

Rank	Project	Project Rating	Category	Sample of Related Companies
1	Linux	100.00	IT Operations	Red Hat, Ubuntu
2	Git	31.10	DevOps	GitHub, GitLab
3	MySQL	25.23	Data & Analytics	Oracle
4	Node.js	22.75	DevOps	NodeSource, RisingStack
5	Docker	22.61	DevOps	Docker
6	Hadoop	16.19	Data & Analytics	Cloudera, Hortonworks
7	Elasticsearch	15.72	Data & Analytics	Elastic
8	Spark	14.99	Data & Analytics	Databricks
9	MongoDB	14.68	Data & Analytics	MongoDB

Source: TC

Ilustración 8 Popularidad Node.js[xiv]

Según The Battery Ventues Open Source Software Index, Node.js ha subido hasta el cuarto puesto en cuanto a popularidad con más de 40.000

estrellas en GitHub, 9 millones de instancias, más de 4800 paquetes publicados y, 3 billones de paquetes descargados mediante npm semanalmente y más de 1500 usuarios contribuyendo al proyecto Node.js

4.3.1 ¿Por qué usar Node.js en el proyecto?

Al ser Node.js un entorno multiplataforma el cual puede componerse de infinidad de módulos y actualmente tiene un gran soporte y popularidad se optó por esta opción en el Trabajo fin de Grado para montar la parte del servidor haciendo uso de Express y como GraphQL ha sido implementado en JavaScript, se han añadido módulos mediante npm para poder gestionar la parte de GraphQL de manera que solamente tengamos que preocuparnos de la lógica de esta parte y con un simple export pueda ser expuesto en un servidor, facilitando enormemente el despliegue y comunicación con el otro proyecto o con cualquier otro cliente que se conectase.

4.4 Express

Express.js, también conocido como Express es un framework para aplicaciones web de código abierto bajo una licencia MIT, y está diseñado para facilitar la creación de aplicaciones web y APIs; se ha llegado a convertir en el componente por defecto para crear aplicaciones web o APIs con Node.js es un componente muy minimalista, escrito en JavaScript con la posibilidad de añadir muchas funcionalidades extras mediante plugins, Express está relacionado con el Backend.

Ilustración 9 Express.js[xv]

En este Trabajo Fin de Grado se ha utilizado Express.js junto con Node.js para poder levantar el servidor de una manera rápida y eficiente y de esta manera poder realizar las peticiones a GraphQL desde una interfaz web, haciendo uso de GraphiQL, o bien desde el proyecto java que se ha realizado para poder realizar las consultas.

4.4.1 ¿Por qué usar Express en el proyecto?

Al poderse utilizar como módulo de Node.js para montar un servidor parecía una opción a investigar, después de ver el funcionamiento y la facilidad para poder exponer nuestra aplicación de GraphQL al exterior y que era la opción más usada para estos fines se decidió implementar el servidor utilizando Express, y, con menos de 15 líneas de código teníamos nuestra aplicación expuesta al exterior.

4.5 JavaScript

JavaScript ha sido el lenguaje que hemos decidido utilizar para la implementación de GraphQL. Es un lenguaje de programación interpretado, con un dialecto estándar ECMAScript, definido como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Ilustración 10 JavaScript logo[xvi]

Principalmente es utilizado del lado del cliente como parte de un navegador permitiendo mejoras substanciales tanto en la interfaz de usuario como en páginas web dinámicas.

También se utiliza, aunque en menor medida, en la parte del servidor; en este Trabajo fin de Grado se ha utilizado Express como parte del servidor con lo cual también hemos hecho uso de JavaScript para la parte del servidor.

JavaScript se ha convertido en uno de los lenguajes de programación más populares, aunque al principio muchos desarrolladores renegaban de él, ya que en un principio el público al que iba dirigido este lenguaje era para aficionados y creación de artículos, pero esto cambió con la llegada de Ajax (Asynchronous JavaScript And XML) el cual es una técnica de desarrollo web para crear aplicaciones interactivas RIA (Rich Internet Applications) las cuales se ejecutan en la parte del cliente, mientras mantienen una comunicación asíncrona con el servidor en segundo plano, consiguiendo, de esta forma, realizar cambios en las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y reusabilidad de dichas aplicaciones.

A partir de 2015 con la nueva publicación del estándar ECMAScript se dotó a JavaScript de una serie de características que todavía le faltaban a este lenguaje, como son el uso de clases para permitir la programación orientada a objetos, expresiones de flecha, iteradores, generadores y promesas para programación asíncrona entre otras.

Las principales características de JavaScript serían las siguientes:

1. **Imperativo y Estructurado**

Es compatible con gran parte de la estructura de programación de C, sentencias if, for, bloques Switchs con algunas diferencias, por ejemplo, las variables las cuales en C solo afectan al bloque para el cual fueron definidas mientras que en JavaScript en un principio afectaban a toda la función, más tarde con una nueva versión de se introduce el uso de Let el cual hace que la

variable sea de uso dentro del scope utilizada, asemejándolo de esta manera más a C.

Otra singularidad de JavaScript sería la posibilidad de omitir los dos puntos a la hora de finalizar una sentencia, pues JavaScript te permite omitirlos y los pone a la hora de compilar.

2. Dinámico

Al ser un lenguaje de Scripting, el tipo está asociado al valor no a la variable, es decir una variable puede estar en un momento asociada a un número y actuar como un Integer y más adelante estar asociada a una cadena y actuar como un String, por eso en JavaScript a la hora de comprobar una variable disponemos de varias opciones:

- **==**

Los dos iguales denotan igualdad en el operador, pero no en el tipo, es decir el intérprete intentará convertir las dos variables de la comprobación al mismo tipo si fueran de tipos distintos y entonces comprobar si son iguales, es decir se podría hacer la comprobación 10==’10’ y nos diría que es cierto, ya que intentaría transformarlos al mismo tipo, y, una vez hecho comprobaría si son iguales.

- **===**

La triple igualación denota igualdad tanto en el operador como en el tipo de la variable, el intérprete no intentará convertir ninguna de las variables de la comprobación al mismo tipo, sino que comprobará las dos variables directamente, en este caso si la comprobación fuera la misma que la anterior 10===’10’ nos diría que es falsa pues uno es un entero y el otro es un String.

3. Objetual

Está formado en su mayoría por objetos, los cuales son arrays asociativos mejorados con los prototipos, esto significa que los nombres de las propiedades de los objetos son claves de tipo cadena, es decir: obj.x = 5 y obj[‘x’] = 5 son equivalentes, sus propiedades y valores pueden ser creados,

modificados o eliminados en tiempo de ejecución JavaScript está dotado con un pequeño grupo de objetos predefinidos como pueden ser Function o Date. Una de las funciones que incluye JavaScript es eval, la cual permite evaluar expresiones en forma de cadenas en tiempo de ejecución.

4. **Funcional**

Las funciones poseen propiedades y métodos, como, por ejemplo, .bind(). Las funciones pueden estar anidadas unas dentro de otras, de manera que son creadas cada vez que la función externa (la que contenga a las demás) sea invocada.

5. **Prototipos**

Para el uso de herencia JavaScript utiliza prototipos, llegando a emular muchas de las características tradicionales de las clases en lenguajes orientados a objetos

4.2.1 ¿Por qué usar JavaScript en el proyecto?

Como hemos podido ver anteriormente JavaScript es un lenguaje muy versátil, y solo hemos nombrado algunas de las características más importantes de dicho lenguaje, lo que de verdad nos hizo decantarnos por este lenguaje a la hora de implementar GraphQL fueron varios factores:

Cuando comenzamos con el proyecto la implementación que tenía más desarrolladores y que estaba más revisada y en la cual se estaban centrando más era la de JavaScript, con lo cual teníamos muchos más ejemplos donde fijarnos y aprender, de todas formas, esto no llego a ser un factor decisivo.

El factor decisivo fue cuando nos enfrentamos a la decisión de realizar el mapeo de todos los diferentes nombres de los campos que componían los objetos en las distintas Fuentes de Datos Abiertos, ya que nuestra decisión fue que de cara al cliente todos se llamasen igual, y, en las Fuentes de Datos Abiertos cada campo tenía un nombre muy distinto al de otro sitio, para ello utilizamos los Resolvers de JavaScript, los cuales nos permitían llamar a una

variable como nosotros queríamos, para más tarde resolver ese nombre internamente con el nombre que tuviese el campo del objeto que fuera en ese momento, consiguiendo de esta forma un mapeo interno, rápido, eficiente y sin necesidad de tener que crear un archivo XML excesivamente grande, o varios archivos, uno por cada Opendata utilizado y que tuviese que ir allí a por las referencias necesarias, al hacerse de esta manera todo el proceso ha podido ser automatizado como se podrá ver más adelante cuando hablemos de la implementación.

4.6 GraphQL

GraphQL no es un lenguaje para hacer consultas a una base de datos graph (aunque el nombre parezca indicar lo contrario) es una capa API, al igual que REST para administrar peticiones por parte del cliente hacia distintas bases de datos heterogéneas.

Hasta ahora teníamos ese mismo servicio con las api REST, la gran diferencia es que REST necesita de varios puntos de acceso dependiendo de qué datos quieras, y, en caso de necesitar varios datos dependientes unos de otros tendríamos que sumar su rountrip time, es decir, el coste de la operación es igual a la suma del coste de todas las operaciones necesarias para obtener el resultado, si a eso le ponemos una conexión móvil por ejemplo, el consumo de datos se vuelve mucho más importante, por si esto no fuese poco, cuando se cambia de una versión a otra , ya sea por actualización del servicio o por cualquier otro motivo normalmente la cantidad de puntos de acceso crece, y la complejidad que estos tienen crecen a su vez, hasta llegar un momento en el que se vuelven muy complejos y muchos de ellos no sirven y tienen que ser deprecados, todos estos problemas vienen ya que REST no fue pensado para lo que se está usando actualmente, podemos ver un ejemplo grafico simplificado en esta imagen:

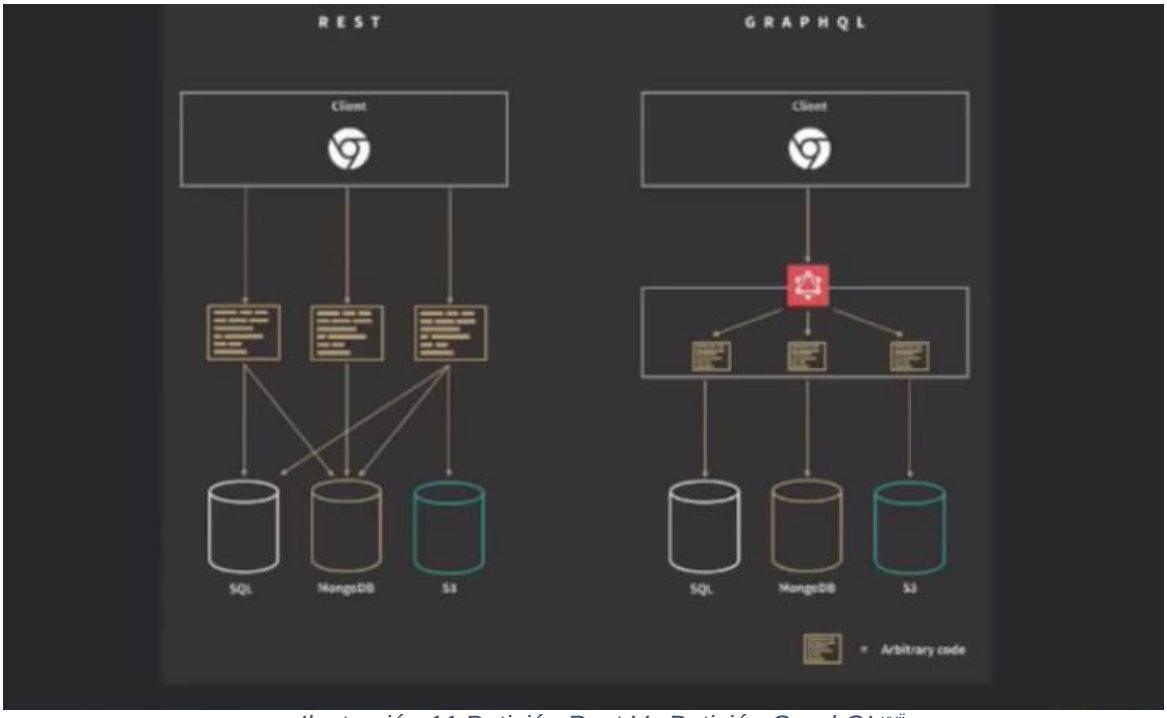


Ilustración 11 Petición Rest Vs Petición GraphQL[xvii]

Debido a todos estos problemas Facebook desarrolló una nueva API llamada GraphQL, esta nueva API revoluciona la forma de realizar peticiones y su coste, empezando por que solo necesita un punto de acceso (normalmente /GraphQL) esta API puede utilizarse tanto como Wrapper de la antigua API REST como sustituyéndola completamente, con el cambio de versiones o con actualizaciones no necesita de nuevos puntos de acceso, simplemente se aumenta su esquema en el servidor con lo cual el cliente puede realizar nuevas consultas, se pueden pedir varios datos del mismo tipo a la vez (por ejemplo podemos pedir datos de dos usuarios a la vez sin necesidad de realizar dos peticiones desde el cliente para ello) eliminando el problema de roundtrip time en el cual necesitábamos el primer dato antes de pedir el segundo, aparte de estas mejoras también resuelve el problema de n+1.

Otra gran ventaja de GraphQL es la auto documentación, al poder rellenar el campo de descripción del atributo, GraphQL nos puede generar una documentación obteniendo el tipo de dato de cada uno de los atributos y las descripciones.

4.6.1 Principios fundamentales de GraphQL

A continuación, pasamos a explicar los principios fundamentales en los que se basa GraphQL así como los elementos necesarios para crear una estructura de cliente y servidor y los tipos de peticiones que son posibles realizar con este lenguaje, para ello utilizaremos a modo de ejemplo SWAPI[xviii], uno de los ejemplos más famosos de implementaciones de GraphQL basado en el mundo de Star Wars.

Consiste en Tipos y Esquemas:

- Objetos
- Campos
- Tipos Enum:
- Escalares
- Listas
- No Nulos

Lenguaje fuertemente tipado, funciona con cualquier tipo de base de datos que se tenga, a la hora de recoger datos funciona como si de un Árbol se tratase:

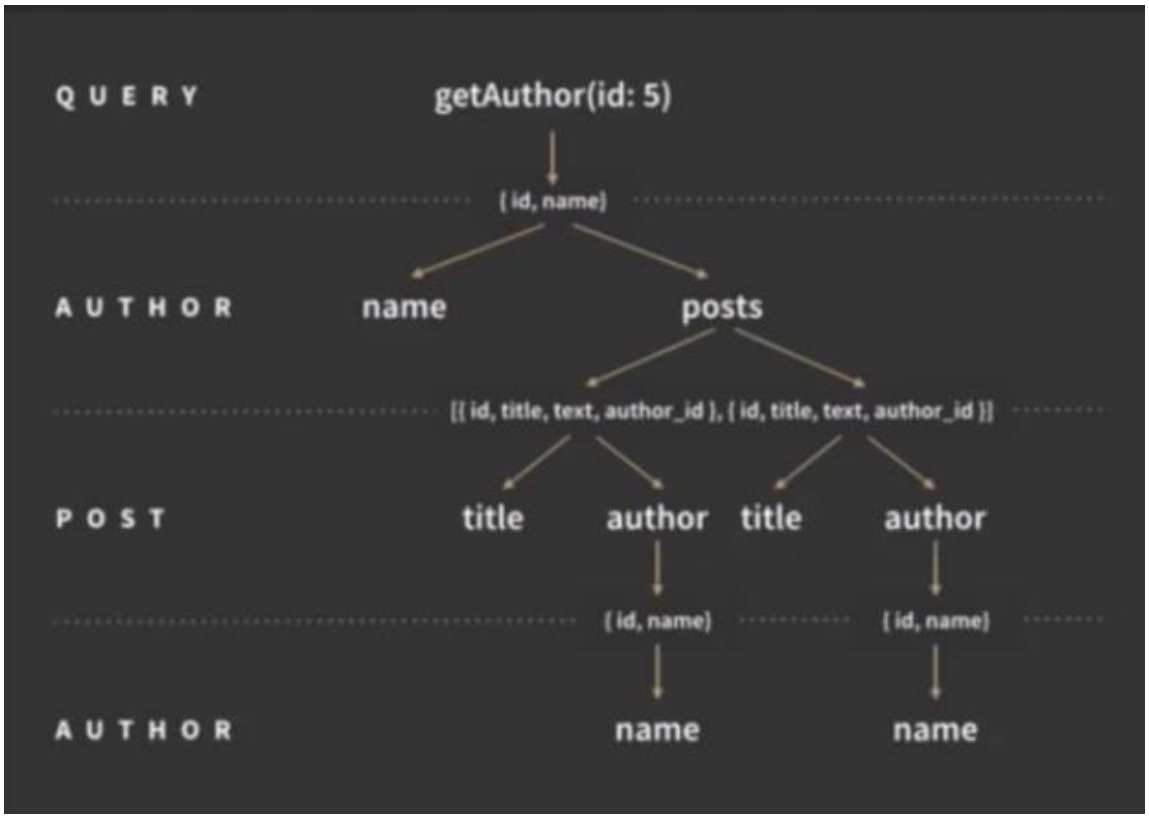


Ilustración 12 Estructura de Datos en GraphQL

Aunque es un lenguaje de “consultas” no expone la estructura de la base de datos tal y como es en el cliente, si no que funciona como un DSL, el esquema que expone no tiene por qué ser exactamente como el de la base de datos, es más bien una forma de describir la petición del cliente de forma estructurada, es en el Backend donde se realiza la petición una vez recibida la consulta del cliente GraphQL.

Si se necesitan añadir nuevas especificaciones, ya sea poder consultar información adicional o distintos tipos de datos, lo único que necesitaríamos sería modificar la consulta en el cliente sin necesidad de tocar el Backend.
GraphQL no se conecta directamente a la base de datos, como hemos visto antes es una capa API (tal como puede ser REST), con la diferencia de que solo necesita un punto de acceso y no varios como REST.

A la hora de realizar una consulta tenemos normalmente el punto de acceso /GraphQL e interactuamos mediante el método POST para decirle que datos queremos obtener, modificar o crear, para esto nos valdremos de Consultas, Mutaciones y Subscripciones.

La capa de GraphQL estaría situada después de que el usuario se autentique (en caso de que fuera necesaria autenticación para el uso del servicio)

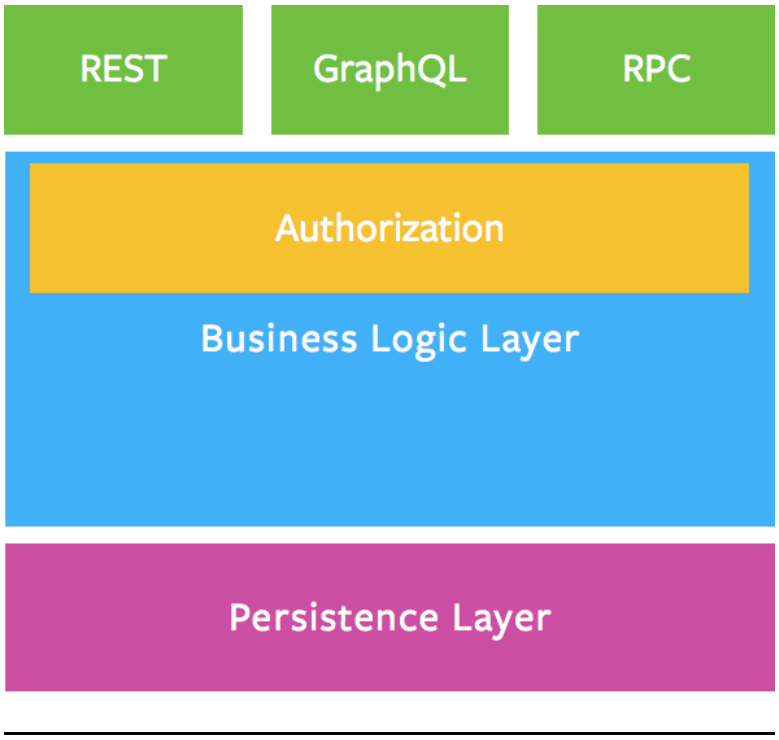


Ilustración 13 Capas de una Aplicación

4.6.2 Tipos y Esquemas

No siempre queremos devolver un valor cuando se realiza una petición, (por ejemplo, si un usuario A quiere ver el mail del usuario B en vez del suyo), para estos casos tenemos dos opciones:

1. Devolver nulo en el campo requerido (funciona en la mayoría de los casos)

2. Devolver un error, es una opción más correcta si el cliente necesita saber el porqué de que no tenga esos datos.

Los componentes más básicos en GraphQL son los Objetos, los cuales toman la siguiente forma:

```
type Character {
        name: String!
        appearsIn: [Episode]!
}
```

Este ejemplo es el del Objeto Character compuesto por dos campos, name y appearsIn, los campos que llevan una exclamación al final indican que no pueden ser nulos, en el caso de episodio (el cual representa un array de objetos) indica que este array tiene que tener cero o más objetos, pero nunca nulo.

4.6.3 Escalares

Representarían las hojas en el árbol de GraphQL, los conjuntos de escalares por defecto son:

- Int
- Float
- String
- Boolean
- ID

El escalar ID representa un identificador único, serializado de la misma forma que un String, pero que al estar definido como ID significa que no es un valor que tenga que ser legible para los humanos.

Existen formas de crear escalares propios, como por ejemplo fecha, el cual depende de la implementación para definir como tiene que ser serializado, deserializado y validado el nuevo tipo.

4.6.4 Enum

En este ejemplo podemos ver cómo funcionaría un enum, el tipo Episode podría tener alguno de estos tres valores, dependiendo del lenguaje escogido para implementar GraphQL la forma de actuar con los enums puede variar (por ejemplo, en JavaScript tendrían que ser mapeados a un conjunto de enteros)

```
enum Episode {
        NEWHOPE
        EMPIRE
        JEDI
}
```

4.6.5 Directivas

La especificación de GraphQL incluye dos directivas, las cuales deben estar siempre disponibles en cualquier tipo de especificación de servidor que se realice:

- @include (if:Boolean) Solo incluye este campo si el resultado del argumento es TRUE
- @skip(if:Boolean) Omite este campo si el resultado del argumento es TRUE

  Las directivas pueden ser útiles para resolver situaciones, que, de otra forma, necesitarían de manipulación de cadenas de texto para añadir o quitar campos en la consulta.

4.6.6 Meta Campos

GraphQL dispone de un servicio para determinar cómo manejar los datos desde el cliente, utilizando meta campos disponibles podemos obtener información de cómo están estructurados los datos, qué tipo de datos o consultas disponemos en el servidor, esto forma parte del sistema de introspección de GraphQL, los Meta Campos comienzan con __ por ejemplo: __schema → nos devolvería el esquema del servidor.

Ilustración 14 Query de Introspección

Tabla 1 Ejemplo Tipo de datos GraphQL

{
__type (name: "Droid") {
name
kind
}
}
	{
	"data": {
	"__type": {
	"name": "Droid",
	"kind": "OBJECT"
	}
	}
	}

{
__type (name: "Character") {
name
kind
}
}
	{
	"data": {
	"__type": {
	"name": "Character",
	"kind": "INTERFACE"
	}
	}
	}

En este caso el primero nos dice que Droid es un Objeto, mientras que la segunda consulta nos indica que Character es de tipo Interfaz. De la misma manera podríamos ampliar el ejemplo para obtener toda la información de un objeto de la siguiente manera:

Tabla 2 Obtención de información de un objeto en GraphQL

```
{
  __type (name: "Droid") {
            name
            fields {
                  name
                  type {
                        name
                        kind
                  }
             }
  }
]
```

De esta manera obtendríamos el nombre del objeto, y sus campos con el nombre de estos y de qué tipo son (también el nombre de dichos campos y el tipo).

4.6.7 Interfaz

En este ejemplo devolveremos ID y name (no pueden ser nulos), en el caso de que tuviese amigos (este campo sí puede ser nulo) se devolvería el array de amigos y también los posibles episodios en los que apareciese (si no aparece en ninguno, al no poder ser nulo devolvería cero). Una interfaz es útil cuando queremos devolver un objeto o una colección de objetos, estos pueden ser de distintos tipos.

```
interface Character {
        id: ID!
        name: String!
        friends: [Character]
        appearsIn: [Episode]!
}
```

4.6.8 Union

Similar a las interfaces, pero no especifican ningún campo en común entre los tipos:

```
union SearchResult = Human | Droid | Starship
```

Con esta petición podemos obtener un Human, Droid o Starship, las Union tienen que realizarse entre objetos no pueden ser realizadas entre interfaces.

4.6.9 Input

Se pueden pasar objetos complejos, una consulta puede obtener varios objetos con sus respectivos campos con una sola petición, en vez de realizar varios roundtrips como sería el caso de REST:

```
input ReviewInput {
        stars: Int!
        commentary: String
}
```

Tabla 3 Ejemplo de Objeto Complejo GraphQL

Example:	Returns

{
hero {
name
appearsIn
}
}
	{
	“data”: {
	“hero”: {
	“name”: “R2-D2”,
	“appearsIn”: [
	“NEWHOPE”,
	“EMPIRE”,
	“JEDI”
	]
	}
	}
	}

Como podemos observar nos devuelve la información en forma de árbol y de la manera exacta que hemos pedido.

4.6.10 Argumentos

Podemos añadir argumentos a los campos, en un sistema REST solo puedes pasar un conjunto de argumentos, mientras que en GraphQL cada campo u objeto anidado puede tener su propio conjunto de argumentos, se pueden pasar argumentos incluso en los tipos escalares para implementar transformaciones en los datos una vez en el servidor en vez de hacerlo en cada cliente por separado. Se pueden realizar peticiones al mismo campo con distintos argumentos (para esto se utilizarían Alias, con el fin de renombrar el objeto en la petición y evitar así conflictos).

Cada campo de GraphQL puede tener cero o más argumentos, estos tienen que tener nombre y pueden ser opcionales u obligatorios, por ejemplo:

```
type Starship {
  id: ID!
  name: String!
  length (unit: LengthUnit = METER): Float
}
```

Los objetos de tipo Input no pueden tener argumentos en sus campos.

4.6.11 Variables

En la mayoría de las aplicaciones los argumentos no son fijos, sino que son dinámicos, no es una buena idea pasar los argumentos directamente en las consultas, con ese fin disponemos de las Variables, las cuales se pasan de forma separada como un diccionario de variables. Se necesitan realizar tres pasos para poder utilizar variables:

1 Reemplazar el valor estático en la consulta con la siguiente forma $variable
2 Declarar $variable como una de las variables aceptadas por la consulta
3 Pasar en el diccionario de variables, (usualmente un JSON) variable: valor.

Se pueden asignar valores por defecto a las variables en las consultas añadiéndolo como valor por defecto al final de la declaración:

```
query HeroNameAndFriends ($episode: Episode = "JEDI") {
  hero (episode: $episode) {
    name
    friends {
                name
        }
    }
}
```

En este caso el valor por defecto de la variable $episode sería “JEDI”

4.6.12 Fragmentos

Los fragmentos permiten construir conjuntos de campo, para luego incluirlos en las peticiones que sean necesarios:

Tabla 4 Fragmentos GraphQL

```
{
        Leftcomparison: hero (episode: EMPIRE) {
                …comparisonFields
}
Rightcomparison:hero(episode: JEDI){
        …comparisonFields
}
}
```

```
Fragment comparisonFields on Character {
        Name
        appearsIn
        friends {
                name
        }
}
```

Como podemos ver en este ejemplo, describimos un Fragment (comparisonFields) en el entorno de Character al cual añadimos los campos Name, appearsIn y friends (este a su vez con el campo name), para, luego, utilizarlo en una misma consulta en la cual se realizan dos comparaciones con una sola petición.

4.6.12.1 Inline Fragments

Si vamos a realizar una consulta que nos puede devolver un conjunto de datos distintos dependiendo de la unión, tenemos que utilizar Inline Fragments para especificar los valores:

Tabla 5 Inline Fragments para consultas complejas

query HeroForEpisode($ep: Episode!) {
  hero (episode: $ep) {
  name
  ... on Droid {
  primaryFunction
  }
  ... on Human {
  height
  }
  }
}
	{
	  "data": {
	  "hero": {
	  "name": "R2-D2",
	  "primaryFunction": "Astromech"
	  }
	  }
	}

Variables
{
  "ep": "JEDI"
}	

En este caso, dependiendo del episodio, el héroe puede ser un Human o un Droid, la parte …on Droid se ejecutará si el campo Hero es de ese tipo de esa manera en la misma consulta podemos ver si es de un tipo u otro el campo y dependiendo de qué tipo sea retornar o bien su primaryFunction (caso …on Droid) o su height (caso …on Human).

4.6.13 Validación

Utilizando el sistema de tipos podemos predeterminar si una consulta GraphQL es válida o no, para ser válida tiene que seguir las siguientes reglas: Un fragmento no puede hacer referencia a sí mismo o crear un ciclo de referencias:

```
fragment NameAndAppearancesAndFriends on Character {
        name
         appearsIn
        friends {
                ...NameAndAppearancesAndFriends
        }
}
```

Ese ejemplo estaría prohibido, ya que se crearía un bucle infinito.

Cuando se realiza una consulta tiene que ser sobre campos existentes.

Cuando al realizar una consulta el campo retorne algo diferente a un escalar o enum, tenemos que especificar qué tipo de dato estamos esperando de ese campo.

Si el campo es un escalar, no tiene sentido realizar consultas de campos adicionales sobre ese campo, ya que los escalares son la mínima expresión de las consultas en GraphQL.

4.6.14 Ejecución

Podemos pensar que cada campo en una consulta es una función o método del tipo previo el cual nos devuelve el siguiente tipo, así, hasta llegar a un valor escalar (como un String o Int) en ese punto la ejecución termina. Las consultas siempre terminan en un valor escalar.

4.6.14.1 Root Fields & Resolvers

En el nivel más alto de cada servidor GraphQL existe un tipo que representa todos los posibles puntos de entrada a la API, se llama Root Type o Query Type:

```
Query: {
  human(obj, args, context) {
    return context.db.loadHumanByID(args.id).then(
            userData => new Human(userData)
    )
  }
}
```

En este ejemplo (escrito en JavaScript) el elemento Root sería Human.

4.6.15 Mutaciones

Es la forma que tiene GraphQL de permitir a los clientes modificar la base de datos (es decir realizar las operaciones de escritura y borrado del CRUD) una vez realizada la mutation query, esta nos devolverá los valores introducidos, de manera que no tenemos que realizar una nueva petición para ver los valores nuevos/actualizados.

Las mutaciones pueden tener múltiples campos (como cualquier otra consulta), con una importante diferencia, mientras que en una consulta normal los campos se ejecutan en paralelo, los campos de las mutaciones se ejecutan en serie, para así evitar el problema de “race condition”, un ejemplo de mutación sería:

Tabla 6 Mutación GraphQL

mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
createReview(episode: $ep, review: $review) {
stars
commentary
}
}
	{
	"data": {
	"createReview": {
	"stars": 5,
	"commentary": "This is a great movie!"
	}
	}
	}

{
"ep": "JEDI",
"review": {
"stars": 5,
"commentary": "This is a great movie!"
}
}	

4.6.16 Problemas con GraphQL

No todo son ventajas con GraphQL, también tenemos posibles problemas, ya que GraphQL soporta anidamiento de peticiones, potencialmente estamos arriesgándonos a tener el problema de “turtles all the way down” (problema de regresión infinita), lo cual puede suponer un duro golpe para el backend, disponemos de tres formas de mitigar este problema:

1 Realizar una inspección AST en el esquema para validar que la consulta no sea muy “compleja” rechazando así este tipo de peticiones “maliciosas”.
2 Poner un umbral de “timeout”, si no se ha resuelto la petición en un tiempo dado se deshecha.
3 La opción que ha tomado Facebook: implementar una “cache de consultas” mediante el cual las consultas que pueden realizarse están cacheadas y los clientes acceden a ellas mediante un ID en vez de realizar una consulta, básicamente es hacer una whitelist de consultas.

5 Metodología para el tratamiento de datos

En la capa del servidor se tratan los datos para, de esta manera, poder enviar única y exclusivamente la información que pide el usuario.

En GraphQL podemos diferenciar dos partes en el tratamiento de datos, el Esquema y el Modelo.

Modelo: El modelo se basa en crear una lista de objetos encargados de recoger la información de los objetos de la Base de Datos que se consulte.

Dentro de cada atributo o campo disponemos de varias funcionalidades:

Nombre del Modelo: Nombre que recibirá el modelo.
Description: Descripción del Modelo.
Fields: Los distintos campos del Modelo:
Name: Nombre que recibe el campo en la Base de Datos a la que hacemos las consultas.
Type: Tipo de dato a devolver (Entero, Cadena, Boolean..)
Description: Breve descripción del campo, si rellenamos este campo dentro del atributo el propio motor de GraphQL en su interfaz gráfica de GraphiQL nos creará una documentación utilizando dichas descripciones.

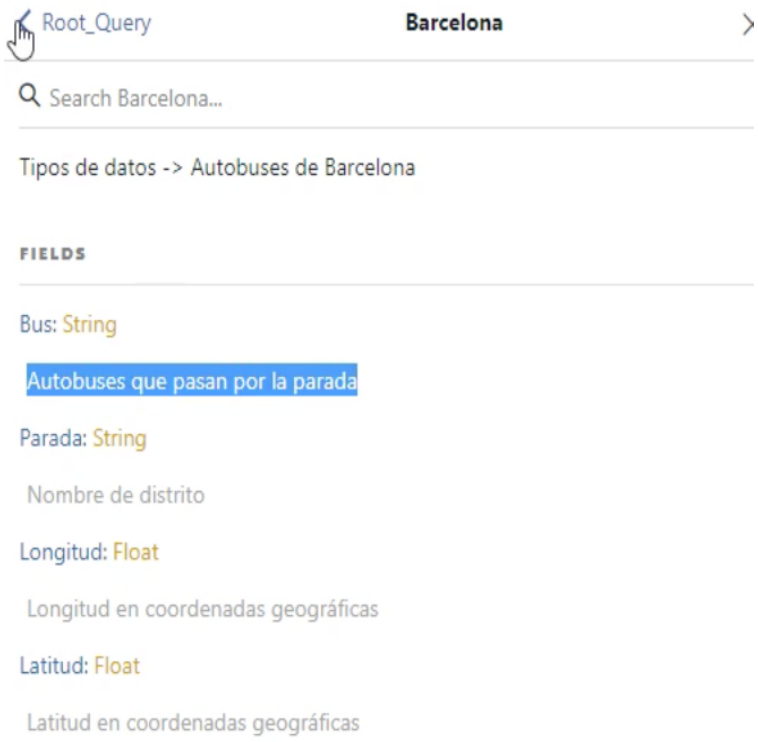


Ilustración 15 Ejemplo de Auto Documentación interfaz GraphiQL

Resolve: Tratamiento del campo del objeto consultado, aquí podemos devolver el campo directamente si nos vale la información tal y como la contiene o podemos realizar operaciones lógicas (ifs) sobre él, incluso podemos mezclar varios campos del objeto consultado y devolverlo en un nuevo campo, como, por ejemplo, utilizar Longitud y Latitud para devolver un nuevo campo llamado “Coordenadas GPS”.

En el caso de este Trabajo Fin de Grado hemos tenido que tener en cuenta que cada Fuente de Datos Abiertos nombra los distintos campos del objeto de una forma distinta, sin tener un estándar entre ellos, para poder estandarizarlo nosotros hemos nombrado los campos de los distintos atributos de igual manera, consiguiendo, en el modelo, el mapeado de todos los atributos del objeto sin el coste adicional de crear un mapeo XML de los distintos atributos.

Esquema: Es el resultado de la unión de todos los distintos Modelos que pueden llegar a ser consultados. Importa los módulos de “request-promise” y “lodash” aparte de todos los Modelos que vayan a ser consultados.

Formado por una Consulta Principal (Root Query) se asemeja mucho a como se tratan los objetos en los distintos Modelos, en este caso los distintos campos de la consulta serían los modelos y estarían formados por:
Nombre del Campo: Nombre con el cual invocaremos al Modelo para realizar las consultas.

Type: Tipo del campo, normalmente serán listas de objetos ya que facilita el devolver más de un resultado y también puede devolver un solo resultado siendo una lista.

Args: Argumentos que pueden ser utilizados para filtrar los resultados devueltos al cliente en la consulta.

En este campo hemos introducido la posibilidad de realizar consultas filtrando por todos los campos del objeto para poder recuperar, desde el objeto completo, hasta un solo campo de dicho objeto, también se incluye en un argumento extra llamado “limit” mediante el cual podemos limitar el número de coincidencias a devolver.

Resolve: Aquí utilizamos Lodash para incluir la instancia de la lista de objetos a devolver, también se le pasan los argumentos utilizados para filtrar la consulta.

Mediante el uso de Request-Promise hacemos la función Asíncrona:

Request: Dependiendo de la Base de Datos y cómo tengamos que acceder a los datos tendremos que introducir un tipo de información u otra, para todos los casos es necesario siempre introducir la URI: Dirección web del recurso.

Dependiendo del tipo de Base de dato en el campo adicional Query Selector (QS) tendríamos que rellenar lo siguiente:

API Rest→ QS: necesitaríamos introducir el id del recurso para acceder a la API.

Json → No sería necesario el campo QS

SparQL → Se necesitarían dos campos:

Query: Query que seleccione todos los campos necesarios del objeto a consultar.

Format: Formato en el que queremos que nos devuelva la consulta, como GraphQL trabaja con formato Json, podremos especificarlo a SparQL que el resultado sea en formato Json.

Promise: En la promesa es donde realizaremos toda la filtración de los argumentos utilizados para la consulta.

Primero, comprobaremos que existen para evitar errores, ya que, si intentamos consultar un campo, que por cualquier circunstancia no

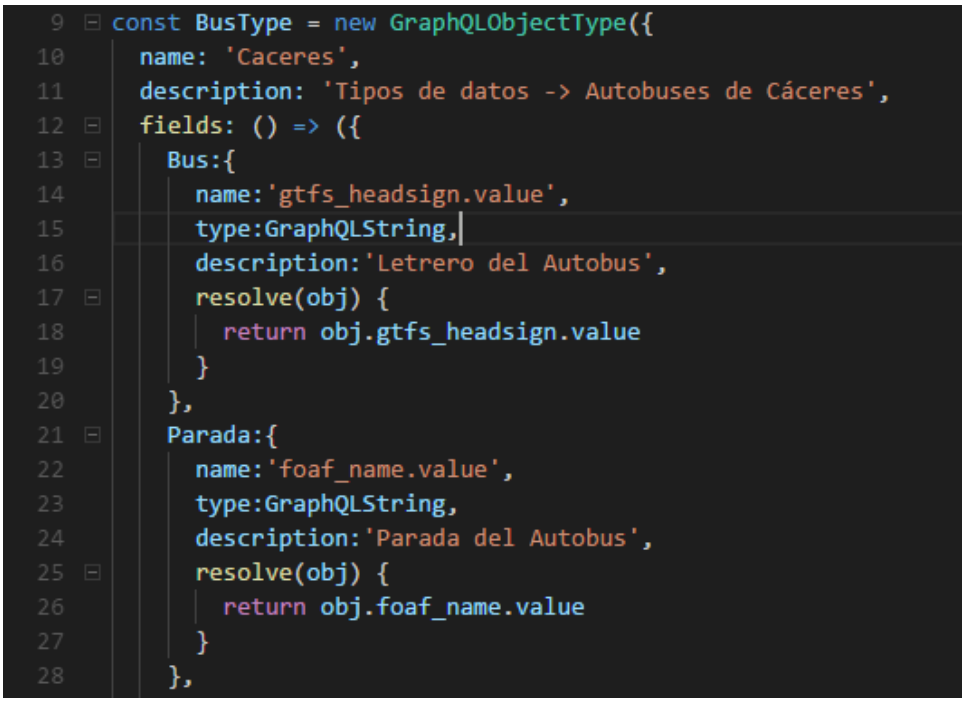
exista en la Base de Datos, nos devolvería un error e impediría que el resto de las consultas o campos pudieran ser devueltos.
Una vez comprobada la existencia del campo se recogerán las coincidencias y se concatenaran a una nueva lista en formato Json que es la que se le devolverá al cliente.

6 Diseño e Implementación

En este apartado se hablará de la solución adoptada para conseguir la integración de múltiples Fuentes de Datos Abiertos y los problemas que se han sufrido a lo largo del desarrollo del Trabajo Fin de Grado.

La idea principal era conseguir que el servidor de GraphQL pudiese manejar desde un único punto de acceso las consultas a distintas Fuentes de Datos Abiertos unificando a su vez los nombres de los atributos y la forma en la que los datos son devueltos, para, de esta manera, poder trabajar de una forma uniforme con todas las Fuentes de Datos Abiertos sin importar como estas trabajasen internamente.

Para conseguir esto se siguió la filosofía de Facebook (creadores de GraphQL) de mantener un único esquema con los distintos modelos, cada modelo haría referencia a una Fuente de Datos Abiertos, y no sería otra cosa que la reconstrucción del objeto a recuperar al hacer la consulta, aprovechándonos de poder implementar GraphQL en una gran variedad de lenguajes, nos decantamos por JavaScript, pues al principio la idea era crear un mapeo para la unificación de los nombres de los atributos de todas las Fuentes de Datos Abiertos, pero gracias a JavaScript y su manera de trabajar con las variables, este paso se pudo hacer de una manera mucho más sencilla como puede verse en la imagen:

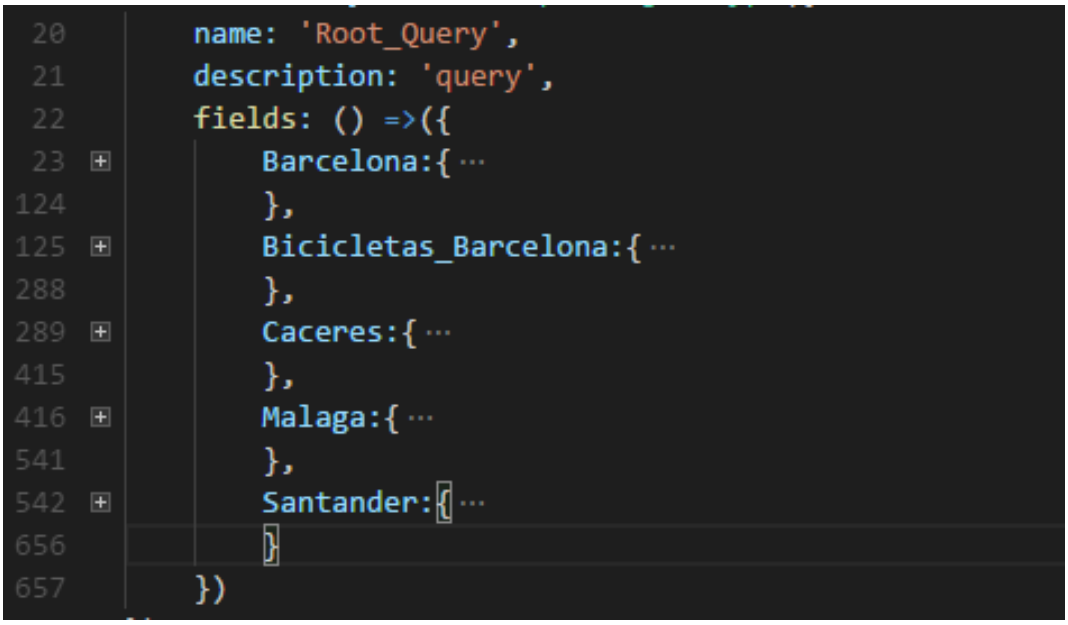


```
const BusType = new GraphQLObjectType({
  name: 'Caceres',
  description: 'Tipos de datos -> Autobuses de Cáceres',
  fields: () => ({
    Bus:{
      name:'gtfs_headsign.value',
      type:GraphQLString,
      description:'Letrero del Autobus',
      resolve(obj) {
        return obj.gtfs_headsign.value
      }
    },
    Parada:{
      name:'foaf_name.value',
      type:GraphQLString,
      description:'Parada del Autobus',
      resolve(obj) {
        return obj.foaf_name.value
      }
    },
```

Ilustración 16 Ejemplo Modelo Servidor GraphQL

Como podemos ver en esta imagen en los distintos campos del objeto ya se está haciendo el mapeo de los nombres de manera que nos ahorramos mucho trabajo.

Cada modelo contendrá los campos necesarios para poder realizar las consultas y luego se exportarán al esquema único, en el cual, serán expuestos al cliente.



```
  name: 'Root_Query',
  description: 'query',
  fields: () =>({
      Barcelona:{ ···
      },
      Bicicletas_Barcelona:{ ···
      },
      Caceres:{ ···
      },
      Malaga:{ ···
      },
      Santander:{ ···
      }
  })
```

Ilustración 17 Ejemplo de Esquema GraphQL

En este esquema podemos ver como los distintos modelos han sido añadidos para poder ser consultados.

Para simplificar el despliegue hicimos uso de la tecnología de Docker, mediante la cual, y configurando una única vez el archivo Dockerfile podíamos construir una imagen del Servidor GraphQL para ser desplegada en cualquier entorno con Docker instalado sin depender de absolutamente nada más:

```
1   FROM node:carbon
2
3   WORKDIR /usr/app
4
5   COPY package*.json ./
6   RUN npm install
7   RUN npm install graphql --save
8   RUN npm install express express-graphql graphql --save
9   RUN npm install --save request
10  RUN npm install --save request-promise
11  COPY . .
12
13  EXPOSE 49160
14
15  CMD [ "npm", "start" ]
```

Ilustración 18 Configuración de Dockerfile

Con este archivo configurado lo único que necesitaremos para crear la imagen de Docker sería escribir el siguiente comando en la consola:

docker build -t <Nombre_de_la_imagen> .

Con esto indicamos que queremos construir una imagen de nuestro actual proyecto y con –t indicamos el nombre que queremos ponerle a la imagen.

6.1 Dataset Utilizados

En este apartado mostraremos los dataset utilizados en el proyecto indicando tanto su tipo como atributos y como estos han sido mapeados en nuestro esquema:

Barcelona [xix]

Tanto los autobuses como las bicicletas son devueltos en formato Json y se accede a estos datos mediante una API con la clave proporcionada en la misma página de la Fuente de Datos Abiertos, en el caso de los Autobuses estos son los campos, tipos de datos y como han sido mapeados a nuestro servidor GraphQL:

En este caso hemos utilizado dos campos del Opendata de Autobuses de Barcelona para crear un único campo mapeado como Parada, el resto de los campos sigue el mapeado indicado en la tabla y son devueltos sin ningún tratamiento especial de los mismos.

Tabla 7 Fuente de Datos Abiertos de Autobuses Barcelona

Campo	**Tipo**	**Mapeo**
EQUIPAMENT	String	Bus
LONGITUD	Float	Longitud
LATITUD	Float	Latitud
NOM_DISTRICTE	String	Parada
NOM_BARRI	String	

En el caso de las Bicicletas de Barcelona contamos con más campos, los cuales han sido mapeados y devueltos sin ningún tratamiento especial con excepción de los siguientes, streetName y streetNumber han sido unificados para crear el campo Calle, y, tanto el campo Estado como el campo Bici son tratados en el modelo con operaciones lógicas para devolver un dato u otro, el campo Bici para saber si son eléctricas o normales y Estado para saber si están disponibles o fuera de Servicio.

Tabla 8 Fuente de Datos Abiertos de Bicicletas Barcelona

Campo	Tipo	Mapeo
BIKE	String	Bici
streetName	String	Calle
streetNumber	String	
Longitude	Float	Longitud
Latitude	Float	Latitud
Bikes	String	NumeroBicis
Slots	String	Espacio
nearbyStations	String	Cercanas
Status	String	Estado

Cáceres [xx]

Accedemos a la Fuente de Datos Abiertos de Cáceres mediante la API de Sparql en la cual tenemos que indicar que los datos nos sean devueltos en formato JSON, retornando en cada campo el valor, menos con Dirección en el cual lo necesitamos tratar en el propio modelo para, dependiendo del valor, retornar si dichos Autobuses son de Ida o Vuelta.

Tabla 9 Fuente de Datos Abiertos de Autobuses Cáceres

Campo	Tipo	Mapeo
Gtfs_headsign.value	String	Bus
Foaf_name.value	String	Parada
Geo_long.value	Float	Longitud
Geo_lat.value	Float	Latitud
Gtfs_direction.value	String	Direccion

Málaga [xxi]

Los datos son devueltos en formato Json y se accede a estos mediante una API con la clave proporcionada en la misma página de la Fuente de Datos Abiertos, estos datos no necesitan ningún tratamiento especial aparte del mapeado.

Tabla 10 Fuente de Datos Abiertos de Autobuses Málaga

Campo	Tipo	Mapeo
nombreLinea	String	Bus
nombreParada	String	Parada
Lon	Float	Longitud
Lat	Float	Latitud
Sentido	String	Direccion

Santander [xxii]

Los datos de esta Fuente de Datos abiertos son devueltos directamente en formato JSON, el único problema consistió en poder mapearlos ya que son devueltos con un nombre que dificulta mucho su lectura en un JSON como es el uso de las comillas simples o de los dos puntos como parte del nombre.

Tabla 11 Fuente de Datos Abiertos de Autobuses Santander

Campo	Tipo	Mapeo
Ayto:numero	String	Bus
Ayto:parada	String	Parada
Wgs84_pos:long	Float	Longitud
Wgs84_pos:lat	Float	Latitud
Ayto:sentido	String	Direccion

La forma de unificar y homogeneizar los campos de los distintos Opendata es la misma para todos los casos, aprovechando los resolve en los modelos de la siguiente manera:

```
Parada:{
    name:'foaf_name.value',
    type:GraphQLString,
    description:'Parada del Autobus',
    resolve(obj) {
      return obj.foaf_name.value
    }
  }
```

Como podemos apreciar en este ejemplo en el campo name se tendría que poner el nombre del atributo al que hacemos referencia en la Fuente de

Datos Abiertos de origen (en este caso la Parada de Autobús del Opendata de Cáceres) como el tipo del atributo a devolver es un String, tenemos que indicar a GraphQL que sea de tipo GraphQLString. En descripción ponemos una breve descripción del atributo para la autogeneración de la descripción en el entorno gráfico de GraphQL.

La parte del resolve de este ejemplo simplemente nos devuelve del objeto recogido el valor del atributo al que estamos haciendo referencia bajo el nombre indicado al principio de Parada; con esto conseguimos homogeneizar todos los campos de las distintas Fuentes de Datos Abiertos a la vez que pedimos los atributos.

En este otro ejemplo se puede observar como en el resolve se aplica una pequeña lógica mediante el uso de condicionales ifs:

```
Direccion:{
    name:'gtfs_direction.value',
    type:GraphQLString,
    description:'Dirección que lleva el Autobus',
    resolve(obj) {
    if (obj.gtfs_direction.value==true)
      return 'Ida'
    else
      return 'Vuelta'
    }
}
```

En este caso lo que conseguimos es recoger un valor que en la Fuente de Datos Abiertos de origen es Booleano, y, mediante un if en el resolve dependiendo del valor obtenido retornaremos ‘Ida’ o ‘Vuelta’. Este ejemplo sirve para ver que podemos renombrar el atributo, realizar operaciones con él y devolverlo como otro tipo de dato.

En este último ejemplo de atributo en el modelo podemos ver cómo utilizamos el resolve para combinar dos atributos de la Fuente de Datos Abiertos a la que hacemos referencia y lo devolvemos como un único atributo:

```
Parada:{
        name:'NOM_DISTRICTE + NOM_BARRI',
        type:GraphQLString,
        description:'Nombre de distrito',
        resolve(obj) {
         return obj.NOM_DISTRICTE+"  "+obj.NOM_BARRI
        }
}
```

También, como en todos los otros casos, el nombre devuelto queda homogenizado para que todas las Fuentes de Datos Abiertos tengan el mismo nombre para los mismos atributos.

6.2 Problemas durante el desarrollo

El mayor de los problemas que hemos tenido durante el desarrollo del Trabajo de Fin de Grado ha sido el depender de Babel para realizar las importaciones de las clases y varias partes del léxico dentro de la API. Babel dejó de dar soporte de “traductor”, que eran las funciones que se utilizaban, y, básicamente se tuvo que rehacer el proyecto de nuevo, esta vez, sin depender de traductores, este fue el momento en el cual se decidió hacer uso de Docker para poder crear un contenedor con el proyecto, y, que no volviese a ocurrir ningún problema de compatibilidad como este, consiguiendo como beneficio extra el que la instalación del Trabajo Fin de Grado pasase por descargarse la imagen de Docker y ejecutarla, aunque esto supuso una gran ventaja para el proyecto también hizo que toda la planificación temporal hasta el momento dejase de ser válida.

Conseguir la forma más eficaz de utilizar un solo punto de acceso mediante la unión de todos los modelos en un mismo Esquema común, después de mucho investigar distintas páginas se llegó a la conclusión de que la mejor manera de realizar dicha tarea era seguir las directrices de Facebook (Creadores de GraphQL) y estas son crear distintos modelos, uno por cada Fuente de Datos Abiertos en nuestro caso, y unir los modelos en un mismo Esquema, el cual se encarga de manejar los filtros de las consultas y sus puntos de acceso.

Poder crear las llamadas con el mismo nombre de atributos, hay que tener en cuenta que cada Fuente de Datos Abiertos tiene su propio nombre para los distintos atributos, en nuestro Trabajo Fin de Grado queríamos conseguir que esta parte fuese totalmente transparente para el cliente, de manera que todos los atributos tenían que acabar teniendo el mismo nombre, en un principio se pensó en crear un mapeo en XML de todos los atributos de las distintas Fuentes de Datos Abiertos, pero eso sería muy complejo, por suerte, haciendo uso de los resolvers de JavaScript en la parte de los modelos se ha podido arreglar este problema, haciendo que cada modelo se encargue de renombrar cada uno de sus atributos con un mismo nombre, para, de esta manera, a la hora de realizar la llamada no importe como se llame internamente en el Opendata original el atributo, sino que tendremos el mismo nombre para atributos similares en las distintos Fuentes de Datos Abiertos.

Las distintas formas de nombrar los atributos en las distintas Fuentes de Datos Abiertos también resultó ser un gran problema, no solo porque cada uno lo llamase de forma distinta y se necesitase crear un mapeo, sino, porque algunas Fuentes de Datos Abiertos como el caso de Santander devolvía un Json formado de una manera muy compleja, uno de los campos era de la siguiente forma [‘ayto:parada‘], esto hacía mucho más complicado poderlo manejar en un Json, y, aún más complejo el poder crear un parser capaz de traducir ese tipo de campos, aunque se consiguió hacer, consiguiendo un parser mucho más completo.

También tuvimos un problema a la hora de conseguir los datos de las distintas Fuentes de Datos Abiertos, ya que cada una los servía de una manera distinta (Api Rest con un id para acceder a los datos, Api SparQL, o en formato Json) esto no solamente hizo más complejo el Esquema, también hizo más complejo el parser para que pudiera entender los distintos tipos de Fuentes de Datos Abiertos.

Tuvimos un problema con los contenedores Docker al principio, y, es que, al ser Docker un contenedor que está principalmente pensado para entornos Linux, cuando se utiliza bajo Windows existe la posibilidad de que la imagen

que está siendo lanzada no se borre de forma correcta al pararla, de manera que no te dejará volver a iniciarla posteriormente, a no ser, que ejecutes una serie de comandos específicos para obligar a todas las imágenes (incluso las no listadas como activas) a parar, y, después, reinicies el servicio de Docker.

7 Manual de usuario

En este manual de usuario vamos a diferenciar la parte del servidor de la parte del cliente:

Servidor: En esta parte para poder ejecutarlo lo único que tenemos que hacer es utilizar la imagen de Docker, para ello necesitaremos instalar Docker en nuestro ordenador y desde una terminal lanzar la imagen como se indica en la siguiente ilustración:

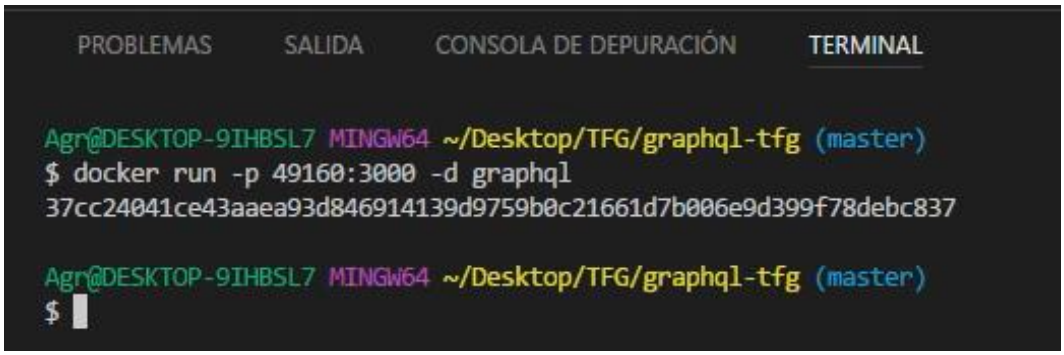


Ilustración 19 Lanzar Docker desde Terminal

El primer puerto será el puerto mediante el cual accederemos al contenedor Docker y el segundo puerto el del servidor de GraphQL que se ha levantado dentro del contendor.

No necesitaremos instalar nada ni configurar nada aparte ya que la imagen de Docker viene totalmente preparada para su utilización.

Si deseamos ampliar con una nueva Fuente de Datos Abiertos solamente tendremos que crear el modelo y ampliar el esquema principal con dicho modelo y con el comando Docker build tendremos la imagen actualizada y lista para volver a utilizar sin más configuración.

Si queremos añadir un nuevo modelo, estos pueden añadirse de dos formas distintas, o bien, para nuestro ejemplo de autobuses añadiendo el archivo Json en la carpeta Parser_JSON del cliente:

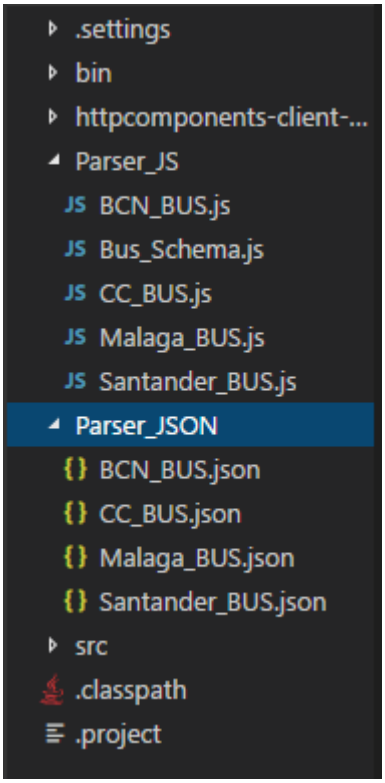


Ilustración 20 Parser_JSON

Con esto, al iniciar el cliente lo primero que hará será buscar si no existe el mismo archivo en Parser_JS y, de ser así, generará el archivo en JavaScript del modelo automáticamente y regenerará el archivo de Bus_Schema para añadir el modelo y todas sus características al esquema.

Para poder generar automáticamente el archivo JavaScript y el resto del proceso sería necesario rellenar un Json como el siguiente, que, como podemos ver, no tiene más de 40 líneas en el cual se pide el nombre de la Fuente de Datos Abiertos, una breve descripción y los campos de los que está compuesta con su nombre, tipo y descripción:

```
{
   "Name":"Málaga",
   "Description":"Tipos de datos -> Autobuses de Malaga",
   "fields":{
      "Parada":{
```

```
        "name":"nombreParada",
        "type":"GraphQLString",
        "desc":"Parada del Autobus"
    },
    "Bus":{
        "name":"nombreLinea",
        "type":"GraphQLString",
        "desc":"Letrero del Autobus"
    },
    "Direccion":{
        "name":"sentido",
        "type":"GraphQLString",
        "desc":"Dirección que lleva el Autobus",
        "resolve":
            {
                "true":"cabeceraIda",
                "false":"cabeceraVuelta"
            }
    },
    "Longitud":{
        "name":"lon",
        "type":"GraphQLFloat",
        "desc":"Longitud en coordenadas geográficas"
    },
    "Latitud":{
        "name":"lat",
        "type":"GraphQLFloat",
        "desc":"Latitud en coordenadas geográficas"
    }
},
"Query":{
    "endpoint":"datastore",
    "uri":"http://datosabiertos.malaga.eu/api/3/action/datastore_search",
    "resource_id":"d7eb3174-dcfb-4917-9876-c0e21dd810e3",
```

```
        "results":"result.records"
    }
}
```

Si por el contrario queremos añadir modelos de forma manual, necesitaríamos incluir todos los tipos propios de GraphQL, crearnos el objeto de tipo BusType con su nombre, descripción y campos, en los campos se realiza el mapeo como puede verse, ya que se están renombrando a la vez que en el resolve se devuelve el campo con el nombre correspondiente en la Fuente de Datos Abiertos a la que el modelo hará referencia, y finalmente, exportar el objeto:

```
let {
 GraphQLString,
 GraphQLInt, //distances
 GraphQLObjectType,
 GraphQLBoolean,
 GraphQLFloat
} = require('graphql');

const BusType = new GraphQLObjectType({
  name: 'Malaga',
  description: 'Tipos de datos -> Autobuses de Malaga',
  fields: () => ({
    Bus:{
          name:'nombreLinea',
          type:GraphQLString,
          description:'Letrero del Autobus',
          resolve(obj) {
          return obj.nombreLinea
          }
     },
    Parada:{
          name:'nombreParada',
          type:GraphQLString,
```

```
            description:'Parada del Autobus',
            resolve(obj) {
            return obj.nombreParada
            }
      },
    Longitud:{
            name:'lon',
            type:GraphQLFloat,
            description:'Longitud en coordenadas geográficas',
            resolve(obj) {
            return obj.lon
            }
      },
    Direccion:{
            name:'sentido',
            type:GraphQLString,
            description:'Dirección que lleva el Autobus',
            resolve(obj) {
      if (obj.sentido==true)
          return obj.cabeceraIda
          else
            return obj.cabeceraVuelta
            }
      },
    Latitud:{
            name:'lat',
            type:GraphQLFloat,
            description:'Latitud en coordenadas geográficas',
            resolve(obj) {
            return obj.lat
            }
      }
   })
})
```

```
module.exports= BusType;
```

Hasta ahora el proceso es bastante parecido y corto, pero con esto solo tendríamos un nuevo modelo creado y ahora necesitaríamos incorporarlo al esquema general, para ello deberíamos incluirlo en la Root Query, en el ejemplo de Barcelona para Autobuses el código sería el siguiente, una lista con el nombre de Barcelona la cual está formada por objetos BCN_BUS que hacen referencia al modelo de Autobuses de Barcelona, los argumentos por los cuales se podrán filtrar las consultas, en el resolve se hace la petición a la Fuente de Datos Abiertos y en la promesa es donde se implementa la lógica necesaria para realizar los filtros en la parte del servidor y mandar al cliente única y exclusivamente los datos que ha pedido:

```
    Barcelona: {
      type: new GraphQLList(BCN_BUS),
      args: {
        limit: {type:GraphQLInt},
        Bus:{type:GraphQLString},
        Parada:{type:GraphQLString},
        Longitud:{type:GraphQLFloat},
        Latitud:{type:GraphQLFloat}
      },
      resolve: (_,args) => rp({
        uri: `http://opendata-
ajuntament.barcelona.cat/data/api/action/datastore_search`,
        qs:{
          resource_id: '2d190658-93ac-4c43-a23f-c5d313b1ae9c'
        }
      }).then(function(responseJson) {
        var parsed = JSON.parse(responseJson);
        var results = parsed.result.records;
        var response=[];
        var bool=false;
        if (typeof args.Bus !== "undefined"){
          bool=true;
```

```
    if (response.length>0){
        results=[];
        for (i = 0; i < response.length; i++)
            if (response[i].EQUIPAMENT == args.Bus)
                results=results.concat(response[i]);
        response=results;
    }else{
        for (i = 0; i < results.length; i++)
            if (results[i].EQUIPAMENT == args.Bus)
                response=response.concat(results[i]);
        results=response;
    }
}
if (typeof args.Parada !== "undefined"){
    bool=true;
    if (response.length>0){
        results=[];
        for (i = 0; i < response.length; i++)
            if (response[i].NOM_DISTRICTE == args.Parada)
                results=results.concat(response[i]);
        response=results;
    }else{
        for (i = 0; i < results.length; i++)
            if (results[i].NOM_DISTRICTE == args.Parada)
                response=response.concat(results[i]);
        results=response;
    }
}
if (typeof args.Longitud !== "undefined"){
    bool=true;
    if (response.length>0){
        results=[];
        for (i = 0; i < response.length; i++)
            if (response[i].LONGITUD == args.Longitud)
```

```
                results=results.concat(response[i]);
            response=results;
        }else{
            for (i = 0; i < results.length; i++)
                if (results[i].LONGITUD == args.Longitud)
                    response=response.concat(results[i]);
            results=response;
        }
    }
    if (typeof args.Latitud !== "undefined"){
        bool=true;
        if (response.length>0){
            results=[];
            for (i = 0; i < response.length; i++)
                if (response[i].LATITUD == args.Latitud)
                    results=results.concat(response[i]);
            response=results;
        }else{
            for (i = 0; i < results.length; i++)
                if (results[i].LATITUD == args.Latitud)
                    response=response.concat(results[i]);
            results=response;
        }
    }
    if(typeof args.limit !== "undefined"){
        bool=true;
        if(response.length>0){
            results=[];
            if (response.length>args.limit){
                for (i = 0; i < args.limit; i++)
                    results=results.concat(response[i]);
                response=results;
            }
        }else{
```

```
                if (results.length>args.limit)
                    for (i = 0; i < args.limit; i++)
                        response=response.concat(results[i]);
                else
                    response=results;
            }
        }
        if (bool)
            return response;
        else
            return results;
    })
},
```

Todo este código es el que nos ahorraría el parser en la parte del cliente, y no sólo eso, sino que como ya hemos visto, dependiendo de la Fuente de Datos Abiertos que consultemos esta tendrá una forma de entregar los datos u otra, esas variantes se tienen en cuenta en el parser, de hacerlo de forma manual el usuario necesitaría conocer toda esa información también.

Una vez añadidos los modelos y estos enlazados en el esquema con el servidor levantado podemos acceder a la herramienta web que levanta el motor de GraphQL llamada GraphiQL y realizar consultas en esta:

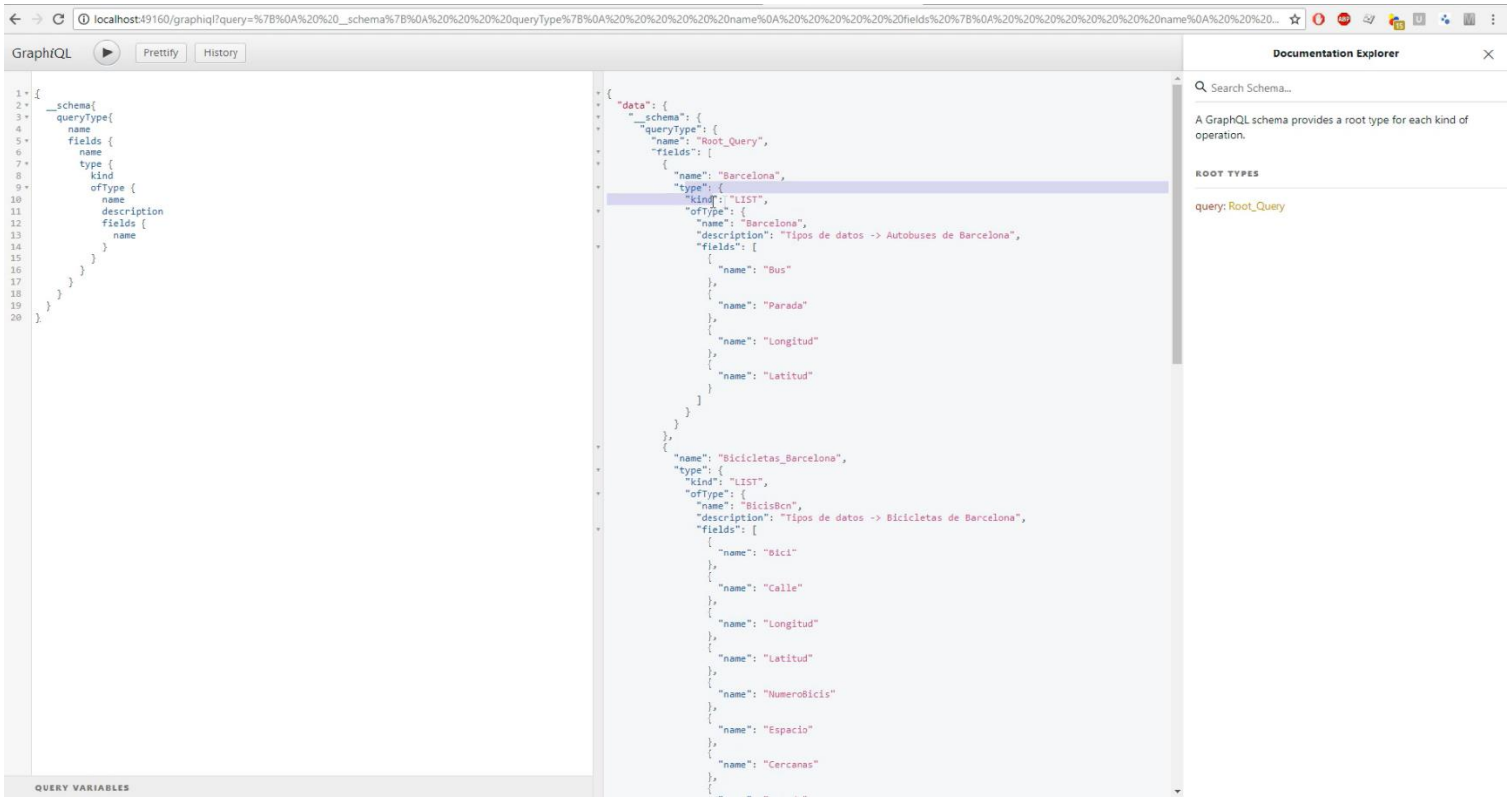


Ilustración 21 GraphiQL Consulta de Introspección

Como tenemos el servidor levantado en local solo tendríamos que acceder a: localhost:49160/GraphiQL y tendríamos todo listo para empezar a realizar consultas.

Cliente: Para poder ejecutar el cliente necesitaremos levantar primero el servidor, una vez levantado podremos ejecutar el cliente (aplicación java) lo primero que hará será buscar en la carpeta Parser_JSON y, si tenemos algún modelo en formato Json nuevo lo parseará y creará el modelo en JavaScript y lo unirá al esquema general para poder ser utilizado en el servidor GraphQL como hemos explicado anteriormente, una vez terminado ese proceso nos aparecerá un menú para escoger que hacer:

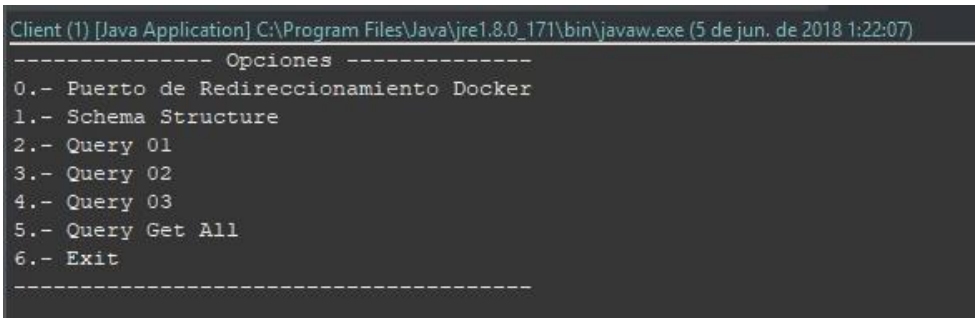


Ilustración 22 Menú Cliente

Lo primero que haremos será introducir el puerto de redireccionamiento Docker mediante el cual accederemos al servidor alojado dentro del contenedor Docker.

Una vez realizado ese paso podremos consultar una de las 4 consultas de ejemplo que tenemos y las cuales arrojaran los valores en formato Json.

Ambos proyectos pueden ser descargados de los repositorios de Bitbucket:

El proyecto que compone la parte del servidor:
Graphql-tfg: https://bitbucket.org/Heavyr/graphql-tfg/src/master/

El proyecto que compone la parte del cliente:
Graphql-tfg-client:https://bitbucket.org/Heavyr/graphql-tfg-client/src/master/

8 Pruebas de concepto

En este apartado mostremos algunas de las consultas que hemos realizado y como pueden ser utilizados en el mundo real.

8.1 Primera Consulta

En este ejemplo podemos ver como con una sola consulta estamos pidiendo datos de dos Fuentes de Datos Abiertos totalmente distintas y heterogéneas, y, como resultado obtendremos el nombre y calle de las tres

primeras bicicletas eléctricas de Barcelona y los cuatro primeros autobuses de Barcelona con su nombre, parada, longitud y latitud, el resultado en la interfaz gráfica sería el siguiente:

```
{
Bicicletas_Barcelona(limit:3, Bici:"BIKE-ELECTRIC"){
  Bici
  Calle
}
  Barcelona(limit:4){
    Bus,
    Parada,
    Longitud,
    Latitud
  }
}
```

Ilustración 23 Consulta 1


```
{
  "data": {
    "Bicicletas_Barcelona": [
      {
        "Bici": "Bicicleta Electrica",
        "Calle": "(PK) C/ JAUME FABRA 12"
      },
      {
        "Bici": "Bicicleta Electrica",
        "Calle": "(PK) C/ PADILLA 159"
      },
      {
        "Bici": "Bicicleta Electrica",
        "Calle": "(PK) C/ URGELL 14"
      }
    ],
    "Barcelona": [
      {
        "Bus": "BUS -59-V27-136--",
        "Parada": "Ciutat Vella  la Barceloneta",
        "Longitud": 2.194744,
        "Latitud": 41.386198
      },
      {
        "Bus": "BUS -46-65-79-109-165-L70-L72-L80-L81-L86-L87-L94-L95-H12-H16-E95--",
        "Parada": "Sants-Montjuïc  Hostafrancs",
        "Longitud": 2.144913,
        "Latitud": 41.371972
      },
      {
        "Bus": "BUS -39-55-D40--",
        "Parada": "Horta-Guinardó  el Guinardó",
        "Longitud": 2.171964,
        "Latitud": 41.418606
      },
      {
        "Bus": "NITBUS -N1--",
        "Parada": "Nou Barris  la Prosperitat",
        "Longitud": 2.177548,
        "Latitud": 41.44159
      }
    ]
  }
}
```


Ilustración 24 Resultado Consulta 1

Como podemos apreciar estos datos vienen devueltos en una estructura de formato Json la cual consumiría el cliente, para, de esta forma poder convertir los resultados en objetos y poder operar con ellos.

En este primer ejemplo hemos visto como se han recuperado datos específicos de dos Fuentes de datos Abiertos totalmente distintos en una sola consulta, mediante el uso de un único punto de acceso y sin tener que consumir datos adicionales que no queríamos.

8.2 Segunda Consulta

En esta segunda consulta se piden datos a dos Fuentes de Datos Abiertos que, aunque ambas son de autobuses, tienen nombres totalmente distintos y formas de entregar la información totalmente distinta, mientras que la de Barcelona es una API Rest la de Cáceres está montada sobre un punto de acceso de SparQL.

Al cliente de GraphQL no le afecta ya que el servidor se encarga de pedir los datos y transformarlos igualmente en un objeto Json de texto plano con únicamente los datos que hemos consultado, en este caso, Parada y Bus para Barcelona y Parada y Dirección en Cáceres, pero en el caso de Barcelona solamente los datos de los autobuses que paren en Sant Andreu y de Cáceres solamente la información de los autobuses RC – AV. I. De Moctezuma-Campus Univ.

```
{
  Barcelona(Parada: "Sant Andreu"){
    Parada
    Bus
  }
  Caceres(Bus:"RC - AV. I. DE MOCTEZUMA-CAMPUS UNIV."){
    Parada
    Direccion
  }
}
```

Ilustración 25 Consulta 2

```
{
  "data": {
    "Barcelona": [
      {
        "Parada": "Sant Andreu  el Bon Pastor",
        "Bus": "BUS -11-B23--"
      },
      {
        "Parada": "Sant Andreu  la Sagrera",
        "Bus": "NITBUS -N3--"
      },
      {
        "Parada": "Sant Andreu  Navas",
        "Bus": "BUS -192--"
      },
      {
        "Parada": "Sant Andreu  Navas",
        "Bus": "BUS -62-H8--"
      },
      {
        "Parada": "Sant Andreu  la Sagrera",
        "Bus": "BUS -126-V27--"
      },
      {
        "Parada": "Sant Andreu  la Sagrera",
        "Bus": "NITBUS -N9--"
      },
      {
        "Parada": "Sant Andreu  Sant Andreu",
        "Bus": "BUS -60-B19--"
      },
      {
        "Parada": "Sant Andreu  el Congrés i els Indians",
        "Bus": "NITBUS -N1-N4--"
      },
      {
        "Parada": "Sant Andreu  Sant Andreu",
        "Bus": "BUS -H4-V31--"
      },
      {
        "Parada": "Sant Andreu  Sant Andreu",
        "Bus": "BUS -V31--"
      },
      {
        "Parada": "Sant Andreu  Baró de Viver",
        "Bus": "BUS -11-60-B19-M28--"
      }
    ],
    "Caceres": [
      {
        "Parada": "Espacio deportivo instalación: Ciudad Deportiva De La Junta De
Extremadura",
        "Direccion": "Ida"
      },
      {
        "Parada": "Complejo Deportivo Valdesalor",
        "Direccion": "Ida"
      },
      {
        "Parada": "Campo De Fútbol Sergio Trejo Vaca",
```


Ilustración 26 Resultado Consulta 2

Los resultados, una vez más, podemos comprobar como son devueltos en formato Json con única y exclusivamente la información que hemos

decidido consultar, para acceder a esta información hemos utilizado el mismo punto de acceso que para la consulta anterior y para todas las consultas que realizaremos, lo único que cambia en GraphQL es la consulta, el punto de acceso siempre es el mismo.

8.3 Tercera Consulta

En esta tercera consulta hemos pedido información a las cuatro Fuentes de Datos Abierto de las que disponemos en nuestro Trabajo Fin de Grado sobre autobuses, todas con distintos parámetros de consulta y filtros.

Como podemos observar para el cliente todos los campos de las distintas Fuentes de Datos Abiertos se llaman igual, aunque en realidad todas sean distintas y las consulta de igual manera, pero, cada Fuente de Datos Abiertos devuelve los resultados de formas distintas, para el cliente el resultado a obtener es un simple Json:

```
{
  Santander(Direccion: "Lluja" Parada: "Ines Diego Del Noval 25"){
    Bus
    Parada
    Longitud
    Direccion
    Latitud
  }
  Barcelona(limit:1){
    Bus
    Parada
    Longitud
    Latitud
  }
  Caceres(limit:3 Direccion: "Vuelta" Parada: "Ciudad Deportiva De La Junta De Extremadura"){
    Parada
    Bus
    Direccion
  }
  Malaga(Direccion:"Churriana"){
    Bus
    Parada
    Longitud
    Direccion
    Latitud
    }
}
```

Ilustración 27 Tercera Consulta

```
{
  "data": {
    "Santander": [
      {
        "Bus": "256",
        "Parada": "Ines Diego Del Noval 25",
        "Longitud": -3.7987308620172127,
        "Direccion": "Lluja",
        "Latitud": 43.48410808706857
      }
    ],
    "Barcelona": [
      {
        "Bus": "BUS -59-V27-136--",
        "Parada": "Ciutat Vella  la Barceloneta",
        "Longitud": 2.194744,
        "Latitud": 41.386198
      }
    ],
    "Caceres": [
      {
        "Parada": "Ciudad Deportiva De La Junta De Extremadura",
        "Bus": "LC - CAMPUS UNIV.-PLAZA DE AMERICA",
        "Direccion": "Vuelta"
      },
      {
        "Parada": "Ciudad Deportiva De La Junta De Extremadura",
        "Bus": "L1 - ALDEA MORET-PLAZA O. GALARZA",
        "Direccion": "Vuelta"
      },
      {
        "Parada": "Ciudad Deportiva De La Junta De Extremadura",
        "Bus": "L1 - ALDEA MORET-PLAZA O. GALARZA",
        "Direccion": "Vuelta"
      }
    ],
    "Malaga": [
      {
        "Bus": "Alameda Principal - Churriana por C/ Torremolinos",
        "Parada": "Avda. de Velázquez - Avda. Moliére",
        "Longitud": -4.456351,
        "Direccion": "Churriana",
        "Latitud": 36.691254
      }
    ]
  }
}
```


Ilustración 28 Resultado Consulta 3

Vemos que todos los campos tienen el mismo formato y solamente recibimos los datos consultados sin necesidad de acceder a varios puntos de acceso como cliente o tener que tratar ningún dato.

Con estos ejemplos podemos ver como GraphQL es una herramienta muy potente, capaz de unificar Fuentes de Datos totalmente heterogéneas de una forma rápida y eficaz sin coste para el cliente, haciendo posibles consultas muy complejas de forma muy sencilla y un tratamiento de datos mínimo para realizar cualquier operación.

Un ejemplo posible sería ver las estaciones de autobuses que pasan cerca de una estación de bicicletas dada, de manera que en una sola consulta tendríamos los datos necesarios, la parada de bicicleta y las distintas paradas de autobuses y el cliente solamente tendría que compararlas para obtener un resultado.

9 Conclusión

9.1 Cumplimiento de los objetivos y limitaciones

Con este Trabajo Fin de Grado hemos conseguido cumplir con todos los objetivos propuestos, ya que conseguimos:

- Recuperar desde la parte cliente única y exclusivamente los datos que queremos despreciando los demás en el lado del servidor y ahorrando el envío de mucha información innecesaria.
- Unificar y estandarizar los nombres de los atributos de las distintas Fuentes de Datos Abiertos (para el caso de ejemplo de los autobuses).
- Poder consultar Fuentes de Datos Abiertos sin relación alguna, como son los autobuses y las bicicletas mediante el uso del mismo punto de acceso y una única consulta.
- Creación de un parser en el lado del cliente para la creación de modelos para el servidor de forma automática y de manera que estos

se enlacen al esquema general del servidor GraphQL de forma automatizada.

- Simplificar toda la instalación del proyecto mediante el uso de una tecnología como Docker para no tener que depender de ningún otro tipo de instalación o configuración.

Como limitación cabría destacar:

GraphQL necesita trabajar con los datos en formato Json con lo cual si la Base de Datos no nos los ofrece en dicho formato necesitaríamos pasarlos en el propio esquema.

Para la unión de todos los modelos es necesario que estén enlazados en el mismo esquema haciendo que este pueda acabar siendo demasiado grande, aunque este podría fragmentarse en varios archivos con el fin de hacerlo más manejable al ser humano.

La necesidad de controlar un gran problema para el Backend como puede ser el problema de regresión infinita, mediante filtrado de consultas que pueden realizarse, consultas que no podrían realizarse o como en el ejemplo del cliente simplemente ofrecer un número determinado de consultas que sean las más usadas.

9.2 Reflexión Personal

El desarrollo de un Trabajo Fin de Grado sobre una tecnología que desconocía por completo y en un lenguaje (JavaScript) que tampoco entendía muy bien ha supuesto un reto y a la vez me ha motivado a aprender muchas cosas, al principio era todo demasiado caótico pues no sabía muy bien como plasmar en algo tangible lo que teníamos en mente, aunque el proyecto siempre estuvo definido, el problema en este caso fue plasmar esa definición en código.

El hecho de tener que trabajar con Fuentes de Datos Abiertos ha supuesto otro reto, tiene muchas posibilidades pero existe un descontrol total en cuanto a homogenización de datos, no hay ningún estándar aplicado a la hora de nombrar los datos, cada grupo encargado de gestionar las distintas Fuentes de Datos Abiertos los trata, nombre y hace disponibles de una manera distinta, pero al final hemos conseguido una forma de unificar y homogenizar todos estos datos heterogéneos de una manera muy eficaz.

En definitiva, ha sido un proyecto que me ha enseñado nuevas tecnologías y formas de tratar los datos y hemos aprendido una forma muy viable para realizar consultas de datos mucho más eficaces.

9.3 Trabajos Futuros

Podría ampliarse con más Fuentes de Datos Abiertos como parques, farmacias, museos u otros puntos de interés, y, aprovechando que los datos son devueltos en formato Json, no sería complicado crear objetos a partir de estos en el cliente, de manera que por ejemplo se pudieran realizar comprobaciones para saber si un autobús pasa cerca de algún museo, farmacia o parque, o donde dejar la bicicleta para poder coger un autobús que te lleve a la parada deseada.

También podría utilizarse como servidor intermedio entre las distintas Fuentes de Datos Abiertos y los clientes de manera que el cliente puede acceder a los datos de, por ejemplo, autobuses, de toda España desde un mismo punto de acceso, con una unificación de nombres en los atributos, recuperando única y exclusivamente la información que desea y obtener toda esa información en formato Json, facilitando su tratamiento en la parte cliente.

Las posibilidades de trabajos futuros y de explotar esta tecnología de transmisión de datos como puede verse es muy amplia.

10 Enlaces de referencia:

[i] https://github.com/facebook/graphql

[ii] https://github.com/graphql/graphql-js

[iii] https://github.com/graphql/graphiql

[iv] https://es.wikipedia.org/wiki/Datos_abiertos

[v] https://github.com/chentsulin/awesome-graphql

https://github.com/graphql/graphql-relay-js

https://github.com/facebook/dataloader

https://www.howtographql.com/basics/0-introduction/

https://www.compose.com/articles/use-all-the-databases-part-1/

https://dev-blog.apollodata.com/graphql-explained-5844742f195e

https://github.com/graphql/graphql-js/blob/master/src/type/introspection.js

https://github.com/graphql/graphql-js/tree/master/src/validation

http://facebook .github.io/relay/docs/thinking-in-graphql.html

https://facebook.github.io/relay/docs/graphql-relay-specification.html#content

[vi] YouTube:

1. Por qué API REST está muerto y debemos usar APIs GraphQL - José María Rodríguez Hurtado
2. Probablemente necesites GraphQL - Belen Curcio
3. Zero to GraphQL in 30 Minutes – Steven Luscher
4. Exploring GraphQL
5. Introducción a GraphQL - Pablo Chiappetti
6. Lessons from 4 Years of GraphQL
7. React London Meetup August 2015, Viktor Charypar, GraphQL at The Financial Times
8. Joseph Savona - Relay- An Application Framework For React at react-europe 2015
9. Lee Byron - Exploring GraphQL at react-europe 2015

10. Nick Schrock & Dan Schafer - Creating a GraphQL Server at react-europe 2015
11. ElixirConf 2015 - What's Next for Phoenix by Chris McCord
12. F8 2015 - React Native & Relay- Bringing Modern Web Techniques to Mobile
13. React.js Conf 2015 - Data fetching for React applications at Facebook
14. Øredev 2015 - Lee Byron - INTRODUCTION TO GRAPHQL
15. React London Meetup October 2015, Nick Schrock, GraphQL
16. Dan Schafer - GraphQL at Facebook at react-europe 2016
17. Laney Kuenzel & Lee Byron - GraphQL Future at react-europe 2016
18. Martijn Walraven - Building native mobile apps with GraphQL at react-europe 2016
19. Build a GraphQL server for Node.js, using PostgreSQL-MySQL
20. Building and Deploying Relay with Facebook
21. GraphQL and Relay
22. GraphQL server tutorial for Node.js with SQL, MongoDB and REST

[vii] https://api.github.com/graphql

[viii] https://github.com/graphql/express-graphql

[ix] https://www.redhat.com/es/topics/containers/what-is-docker

[x] https://es.wikipedia.org/wiki/Datos_abiertos#/media/File:Lod_2017-08-22.png

[xi] https://www.docker.com/

[xii] https://www.redhat.com/en/topics/containers/what-is-docker

[xiii] https://nodejs.org/es/

[xiv] https://techcrunch.com/2017/04/07/tracking-the-explosive-growth-of-open-source-software/?guccounter=1

[xv] http://expressjs.com/es/

[xvi] https://www.javascript.com/

[xvii] http://bearcatjs.org/graphql-versus-rest-api/

[xviii] https://github.com/graphql/swapi-graphql

[xix] http://opendata-ajuntament.barcelona.cat/

[xx] http://opendata.caceres.es/sparql/

[xxi] http://datosabiertos.malaga.eu/

[xxii] http://datos.santander.es/api/rest/datasets/paradas_bus.json