Universidad Zaragoza

Trabajo Fin de Grado

GraphQL aplicado a catálogos de datos abiertos

GraphQL applied to open data catalogs

Autor

Diego García Muro

Director

Francisco Javier López Pellicer

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2023

GraphQL aplicado a catálogos de datos abiertos

RESUMEN

Hoy en día grandes plataformas de catálogos de datos como pueden ser el portal oficial de datos de la Unión Europea[1] o el de España[2] ofrecen información muchas veces imprecisa, ambigua o desactualizada. Gran parte de culpa la tiene RDF (Resource Description Framework), un estándar de representación de información en la web legible por las máquinas, que al adquirir un gran nivel de complejidad las consultas se vuelven ineficientes y las relaciones entre los recursos se entrelazan y solapan.

Con el fin de analizar esto, se ha estudiado el modelo DCAT (Data Catalog Vocabulary), una especificación de RDF, que en relación con los catálogos de datos como los mencionados o como el del Gobierno de Aragón[3] proporciona un marco común para describir y organizar la información sobre sus conjuntos de datos a través de una serie de metadatos y relaciones. De esta forma facilita el descubrimiento y mantenimiento de los datos, pero, como se verá tiene sus limitaciones, entre ellas las consultas limitadas o la escalabilidad.

El proyecto demuestra como GraphQL, quien que se ha convertido actualmente en un estándar industrial de facto para acceder a modelos complejos de datos, es la solución a muchos de estos problemas, mejorando la estructura de los datos y el acceso a los mismos. El objetivo es crear un envoltorio o capa sobre el portal de datos del Gobierno de España, con el fin de estructurar los más de 66.000 conjuntos de datos que contiene este portal en un nuevo modelo GraphQL basado en los estándares empleados en dichos portales (DCAT 2 y 3) y persistirlos en una base de datos. Para verificar el resultado y las consultas a los datos la aplicación cuenta con una interfaz de usuario donde el usuario final es capaz de realizar consultas, visualizar conjuntos de datos concretos, navegar entre los recursos y aplicar una gran variedad de filtros para obtener una información organizada y accesible. Todo ello contando con las ventajas que ofrece GraphQL: posibilidad de realizar complejas sin tener que conocer los API específicos de los portales, de solicitar únicamente los campos requeridos evitando así la sobrecarga de datos y, además, flexibilidad para modificar y escalar el esquema o modelo de datos.

[1] https://data.europa.eu/es/

[2] https://datos.gob.es/es/

[3] https://opendata.aragon.es/

Índice

Acrónimos

RDF Resource Description Framework

DCAT Data Catalog Vocabulary

SPA Single-Page Application

API Application Programming Interface

REST Representational State Transfer

CSW Catalogue Service for the Web

UI User Interface

ECAS European Commission Authentication Service

CMS Content Management System

FME Feature Manipulation Engine

XML Extensible Markup Language

JSON JavaScript Object Notation

CSV Comma-separated Values

TURTLE Terse RDF Triple Language

W3C World Wide Web Consortium

DCAT-AP DCAT Catalogue Application Profile

CKAN Comprehensive Knowledge Archive Network

MQA Metadata Quality Assurance

SDL Schema Definition Language

CRUD Create, Read, Update and Delete

ORM Object-Relational Mapping

Glosario

Open Data Se refiere a la práctica que persigue proporcionar información y ciertos datos de forma libre y accesibles para que cualquier persona pueda acceder a ellos, los pueda reutilizar o redistribuir sin restricciones, promoviendo la transparencia y la colaboración.

Metadatos Consiste en información descriptive que proporciona detalles sobre otros datos, como la fecha de creación, el formato, el autor... Ayudan a comprender y organizar los conjuntos de datos facilitando su búsqueda, interpretación y uso.

Catálogo de datos Se trata de una plataforma o recurso en línea que organiza y enumera conjuntos de datos disponibles para el público, proporcionando metadatos sobre cada conjunto, facilitando así la búsqueda y acceso a los datos abiertos.

Varnish Es un servidor proxy y acelerador de contenido de código abierto. Actúa como una capa intermedia entre el servidor web y los usuarios, almacenando temporalmente copias de las páginas web y recursos, lo que reduce la carga en el servidor.

Proxy Inverso Es un servidor que actúa como intermediario entre clientes y uno o más servidores. Cuando un cliente solicita un recurso, como una página web, el proxy recibe la solicitud y la reenvía al servidor correspondiente. A continuación, recibe la respuesta y la envía al cliente. Se diferencia de un proxy convencional en la dirección del flujo de datos, pues en este caso trabaja en nombre de los servidores para gestionar las solicitudes de los clientes.

CKAN Plataforma de código abierto diseñada para gestionar y publicar conjuntos de datos.

DRUPAL Sistema de gestión de contenidos de código abierto que permite crear y administrar distintos tipos de sitios web y aplicaciones en línea.

Apache SOLR Plataforma de búsqueda y análisis de código abierto basada en el motor de búsqueda de Apache Lucene. Está diseñado para realizar búsquedas y recuperar información de forma rápida y eficiente en grandes volúmenes de datos.

Gazetteer Se trata de un nomenclátor geográfico.

Harvester Componente software que recupera información de diferentes fuentes y la consolidan para su posterior procesamiento o presentación.

Virtuoso Quad Store Virtuoso es un sistema de base de datos multimodelo que combina la gestión de datos relacionales, grafos y RDF, entre otros. El término Quad Store hace referencia a la gestión de los datos RDF, para los que usa cuádruplas: sujeto-predicado-objeto-contexto.

SPARQL manager Herramienta diseñada para administrar consultas y operaciones SPARQL, lenguaje usado para consultar y manipular datos en RDF.

PIWIK Ahora denominado Matomo, es una plataforma de análisis web de código abierto que analiza el tráfico en los sitios web.

MQA Herramienta desarrollada por el consorcio de data.europa.eu para estudiar la calidad de los metadatos desarrollados.

Linked Data API Conjunto de convenciones para construir API REST que gestionan y exponen conjuntos de datos enlazados.

PostgreSQL Sistema de gestión de bases de datos relacionales de código abierto.

MYSQL Sistema de gestión de bases de datos relacionales de código abierto. Se enfoca más en la velocidad de las consultas que en la integridad de los datos, la flexibilidad y la capacidad de extensión.

API Son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos.

JUnit Es un marco de pruebas unitarias para el lenguaje de programación Java que puede también utilizarse con Kotlin. Proporciona un entorno y herramientas para escribir y ejecutar pruebas automatizadas de manera eficiente.

Plugin Es un módulo que se integra en herramientas y entornos de desarrollo para ampliar la funcionalidad base de una aplicación sin necesidad de modificar su código fuente principal.

Gradle Es una herramienta de construcción y automatización de tareas en el desarrollo de software. Se utiliza para compilar, probar, empaquetar y distribuir aplicaciones.

Dependencia Es un componente externo, como una biblioteca o un módulo, que se utiliza en un proyecto para agregar funcionalidad o recursos específicos. Pueden ser bibliotecas de código, frameworks, plugin u otros recursos.

Caché Es un componente que almacena temporalmente datos con el fin de acelerar el acceso futuro. La idea es que, en lugar de tener que cargar los datos de la fuente original (por ejemplo, una base de datos) cada vez que se solicitan, los datos son cargados desde un almacenamiento más rápido, como puede ser una RAM.

ORM Es un modelo de programación que permite mapear las estructuras de una base de datos relacional sobre una estructura lógica de entidades con el fin de simplificar y acelerar el desarrollo de las aplicaciones. Mapea las tablas y relaciones de la base de datos a objetos en código, actuando como una abstracción entre la base de datos y la aplicación.

Corrutinas (coroutines) Es mecanismo de programación asíncrona que permite escribir código concurrente y asíncrono de manera más estructurada y legible, evitando bloqueos y permitiendo la ejecución en suspensión para realizar operaciones concurrentes de manera más eficiente.

Listado de figuras

Listado de tablas

Listado de códigos

1 Introducción

Este Trabajo de Fin de Grado analiza la aplicación de la tecnología **GraphQL** [14] aplicada al descubrimiento de conjuntos de datos abiertos ofrecidos por los portales de datos europeos. Actualmente, la descripción de estos datos para su búsqueda se hace a través de un modelo común denominado **DCAT** (Data Catalog Vocabulary) [8]. En este proyecto se va a realizar un análisis de las ventajas y desventajas que ofrece GraphQL, como se pueden exprimir al manipular catálogos de datos abiertos y como se resuelven los problemas que presentan tecnologías como **RDF** (Resource Description Framework) [59] y **SPARQL** [60], utilizadas en dichos portales.

A lo largo de esta sección se ofrece una visión general del proyecto, aclarando cuales son los objetivos, el contexto y el alcance. Por otro lado, se presenta la estructura que sigue el documento, de forma que el lector pueda navegar fácilmente a través de su contenido.

1.1 Contexto

Para entender el contexto es necesario comprender que sucede hoy en día con los datos y es que, en la sociedad actual, inmersa en datos de todo tipo, el manejo efectivo de estos se ha vuelto crucial para las empresas, que utilizan información para tomar decisiones, desarrollar estrategias y anteponerse a la competencia [1]. Con el fin de organizar y dotar de una estructura a los datos de forma que la información sea útil, administraciones públicas y empresas privadas empezaron a adoptar las denominadas estrategias de **Open Data** [2]. Estos datos se sustentan en principios como la disponibilidad pública y el uso de formatos abiertos y se encuentran respaldados por licencias que permiten su libre reutilización. De esta forma, se consigue fomentar la transparencia y el desarrollo de aplicaciones y servicios basados en datos [5], como, por ejemplo, una aplicación que informe en tiempo real sobre horarios, rutas o retrasos en el transporte público.

Los datos abiertos se agrupan y organizan en **catálogos de datos** [73], que son plataformas encargadas de describir y brindar acceso a dichos conjuntos de datos, con el fin de facilitar, en última instancia, el descubrimiento y acceso a los mismos (figura 1). Estos catálogos utilizan un estándar basado en **RDF** llamado **DCAT** que garantiza la interoperabilidad y la consistencia de los conjuntos. DCAT se puede entender como un lenguaje común en la web semántica [71], en el que una serie de etiquetas se asignan a los conjuntos de datos con el fin de organizarlos y describirlos. De esta forma ayuda a alcanzar el objetivo de la web semántica, que es ir más allá de enlazar páginas, sino lograr que las máquinas sean capaces de entender la información, optimizando así el intercambio y descubrimiento de datos. Estas etiquetas son los denominados **metadatos** [72], que, simplificando, consisten en información adicional agregada a un conjunto de datos con el fin de facilitar su descubrimiento y proporcionar contexto sobre su contenido, origen o uso.

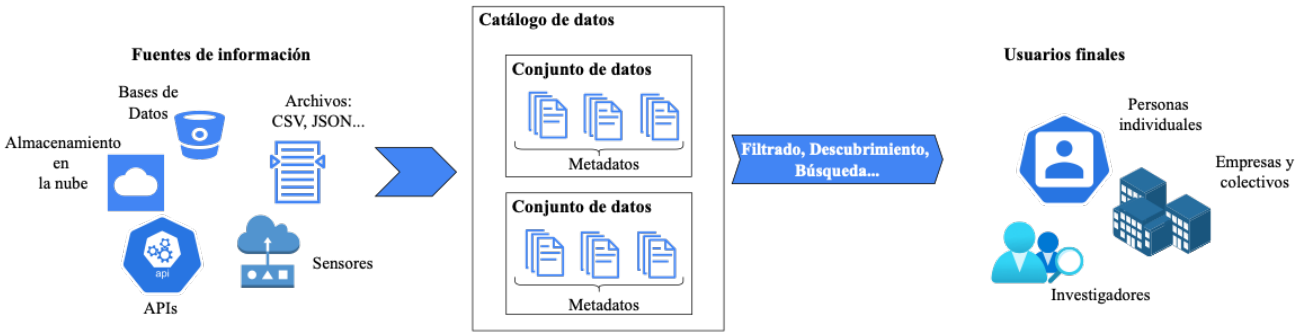


Figura 1 Representación visual sin detalle para entender que un catálogo de datos abiertos

Sin embargo, los catálogos de datos abiertos presentan desafíos [4], relacionados principalmente con la estructuración de la información y la forma en la que las API (Application Programming Interface) acceden a esta. Por un lado, la estructura existente debe redefinirse

siempre que se desee extender el modelo DCAT, ya que sus clases y propiedades se encuentran predefinidas, lo que puede convertirse en un auténtico dolor de cabeza. Por otro lado, las consultas implican sobrecargas de datos innecesarios (no es posible solicitar solo aquellos que interesan) y el lenguaje utilizado, como es el caso de **SPARQL**, no resulta sencillo, lo que hace que no cuente con una gran audiencia.

Según el Estudio del impacto de los datos abiertos en España [81], las principales barreras en cuanto al acceso y puesta a disposición de los datos abiertos es la información homogénea, que cuenta con datos de poca calidad y poco actualizados. No hay que ahondar mucho para encontrarse con alguno de los problemas derivados, con una simple búsqueda en portales de datos abiertos como el de la Unión Europea [74] o el de España [6] se encuentran enlaces desfasados, sin contenido que mostrar o poco precisos, como se muestra en la sección de análisis. Además, en cuanto a las consultas que se pueden realizar a través la API que ofrecen, estos portales, existe una limitación. Y es que los endpoints definidos no permiten búsquedas relativamente complejas y, en ocasiones, es necesario llamar a varios de ellos en lugar de obtener el resultado con una sola invocación. Por ejemplo, en el portal español, para filtrar conjuntos de datos por palabras clave o categoría es necesario invocar dos endpoints diferentes (figura 2). Los **endpoints** se pueden entender como una puerta de entrada, o salida, que permite obtener información remota o realizar acciones.

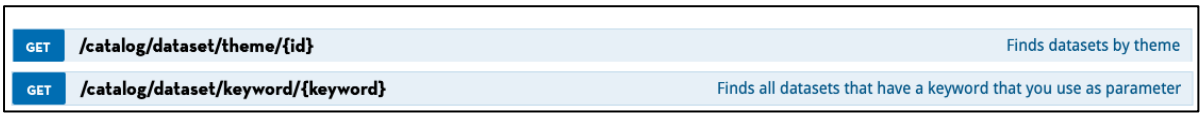


Figura 2 Endpoints API del portal de datos del Gobierno de España

Aquí es donde entra en juego **GraphQL**, un lenguaje de consulta y una especificación para la obtención y manipulación eficiente de datos en API. Esta tecnología se ha convertido en un estándar industrial de facto para acceder a modelos complejos de datos y es que una de sus grandes ventajas, es la capacidad que ofrece al cliente de solicitar únicamente aquellos campos que necesita [7], haciendo más eficientes las consultas (ya no hay que consultar varios endpoints en muchas ocasiones) y evitando la sobrecarga. Por otro lado, no hay complicaciones a la hora de actualizar el modelo y es que los cambios se pueden realizar de manera incremental sin afectar la compatibilidad hacia atrás, lo que facilita la gestión y extensión de este. Con ello se puede lograr que los datos sean más accesibles, fáciles de buscar y se encuentren mejor estructurados, es decir, se ofrece una visión más unificada.

Actualmente hay una línea de investigación en el grupo IAAA del I3A que está considerando el uso de GraphQL en catálogos de datos. Esta línea ha dado como resultado anteriormente el Trabajo Fin de Grado realizado por Alejandro Magallón[4] que se centró en los catálogos de datos espaciales, atacando los denominados servicios CSW (Catalog Service for the Web). El proyecto presentado en esta memoria no comparte ningún detalle de código o soluciones implementadas con Trabajos Fin de Grado anteriores al centrarse en un aspecto diferente como es el modelo DCAT y los catálogos de datos abiertos.

1.2 Objetivo

Este Trabajo Fin de Grado tiene como principal objetivo analizar las ventajas y desventajas que tiene **GraphQL** y como se pueden aprovechar para manipular catálogos de datos abiertos con el fin de resolver las debilidades que tecnologías como **RDF** y **SPARQL** presentan,

[4] https://github.com/IAAA-Lab/space-ql

ofreciendo al usuario una forma más flexible y amplia de acceder a la información. Con el fin de materializar lo señalado, este Trabajo Fin de Grado ha desarrollado una aplicación web completa, con una interfaz que permite al usuario interactuar con los diferentes datos, pudiendo realizar búsquedas filtradas y visualizarlos de forma clara y sencilla.

La web desarrollada es de tipo **SPA** (Single-Page Application) [75], lo que permite que las diferentes rutas actualicen el contenido en tiempo real, pues no es necesario cargar toda la página, solo se actualiza la parte necesaria. Además, los usuarios deberán poder realizar tanto consultas concretas a través de una plataforma que sirve de API y que es ofrecida por GraphQL, como otras consultas predefinidas a través de la web. El sistema debe ser escalable y flexible, facilitando en un futuro modificar el modelo de datos, por ejemplo, agregando metadatos, sin necesidad de rehacer la estructura.

Para validar la solución desarrollada, se van a cargar los más de 66.000 conjuntos de datos existentes en la web del Gobierno de España con el fin de poder ejecutar múltiples consultas de distinta complejidad. De esta manera, partiendo de los endpoints que ofrece el portal, se desarrollan consultas más flexibles, eficientes y simples, junto con otras que el portal, y la API, no son capaces de realizar o resultan muy enrevesadas. Además de esto, se proporciona al usuario de una navegabilidad más rica entre los recursos a través del interfaz web. Es importante destacar este concepto de navegabilidad. En GraphQL, los diferentes recursos y relaciones en el sistema se representan como un grafo, donde los nodos son tipos y los bordes son relaciones. Para aprovechar este aspecto, el usuario final tiene la capacidad de navegar por el interfaz entre los distintos recursos existentes sin complicación. Por ejemplo, se puede acceder al catálogo que contiene un conjunto de datos concreto y a partir de él, acceder al total de conjuntos de datos que contiene y repetir el proceso para otro conjunto.

1.3 Alcance del proyecto

Hay que tener en cuenta que es un Trabajo de Fin de Grado, por lo que la aplicación final no es un sistema con salida inmediata a producción, sin embargo, sí debe mostrar soluciones a los problemas planteados. El resultado debe permitir al usuario final del sistema poder realizar un mayor abanico de consultas, potentes, de manera mucho más simple que con las tecnologías actuales basadas en RDF y **REST** [76] (Representational State Transfer), y, además, ser capaz de navegar por los diferentes recursos, modelados a partir de DCAT, a través de una interfaz interactiva y fácil de usar. Y es que los portales actuales ponen su foco en los conjuntos de datos, pero no tanto en el resto de los recursos, como pueden ser los catálogos que los contienen. Con ello, se exprime la tecnología GraphQL, aprovechando la comodidad que ofrece a la hora de integrarse con otras, la simplicidad a la hora de implementar y gestionar el código gracias a su motor y las posibilidades y eficiencia al realizar las consultas.

El sistema desarrollado se puede entender como un wrapper, un envoltorio sobre el portal de datos del Gobierno de España. Los distintos conjuntos de datos se cargan junto con sus propiedades y relaciones en un nuevo modelo inspirado en DCAT mucho más flexible, sobre el que se llevarán a cabo las consultas. De esta forma la aplicación cuenta con un cliente y un servidor. El primero es quien ofrece la interfaz y gestiona la interacción con el usuario, permitiéndole filtrar información y navegar por ella. Además, el cliente es quién envía las diferentes consultas GraphQL al servidor. Este se encarga de procesar dichas solicitudes, gestionar la base de datos y lo que se puede considerar el centro de la aplicación, el modelo de datos, en función del cual se definirán las diferentes consultas para el acceso y modificación de los datos. El esquema GraphQL expuesto por el sistema desarrollado se basa en los modelos **DCAT 2** [8] y **DCAT 3** [9] y es posible agregarle progresivamente diferentes metadatos, recursos y relaciones entre ellos. Los datos cargados del portal se adaptan a este y, pese a poderse realizar la transformación al vuelo, se ha decidido persistirlos en una base de datos fiel al nuevo modelo, de esta forma se evitan pérdidas de datos y siempre será posible

extenderlo agregando nuevos recursos, metadatos o enlaces a preguntas SPARQL remotas. Para procesar estos datos el sistema debe ser capaz de soportar diferentes formatos. Se han contemplado dos: JSON (JavaScript Object Notation) y CSV. Todos estos requisitos se incluyen, junto con las tecnologías pertinentes, en el anexo A.4.

1.4 Estructura de la memoria

La memoria se estructura de la siguiente forma. A continuación, una sección de análisis presenta cual es la base del problema, hablando de la web semántica y tecnologías como RDF y SPARQL. Tras explicar que soluciones ofrece GraphQL, se hace hincapié en esta tecnología, detallando sus conceptos clave. El proceso de diseño se detalla en una nueva sección, donde se muestra una evaluación general del sistema, sus requisitos, modelos de datos y el aspecto inicial de la interfaz. Tras esto se incluye la sección de desarrollo. Aquí se detallan los aspectos relacionados con la implementación de los diferentes módulos que conforman el proyecto. Se plasman, también, los diferentes problemas encontrados y las decisiones que se han tomado. Por último, se añade una sección donde se explican las diferentes pruebas realizadas para garantizar el correcto funcionamiento del sistema y una sección con las conclusiones extraídas. Al final del documento se encuentra el anexo, con un mayor detalle sobre los temas tratados en las secciones anteriores. Todo se complementa con una bibliografía.

2 Análisis

Antes de comenzar a diseñar el sistema (arquitectura, modelos de datos...) conviene comprender cual es la base del problema existente y como se va a solucionar. Para ello hay que estudiar la arquitectura de los portales de datos europeos y las tecnologías que utilizan, así como la tecnología GraphQL, empleada para resolver dichos problemas.

2.1 Base del problema

En esta subsección se trata la arquitectura del portal de datos abiertos del Gobierno de España, haciendo referencia a otros como el de la Unión Europea (UE), para visualizar que el problema no solo se da en un ámbito local. Se analiza también el modelo de datos que sirve de base para dichos portales y para este proyecto, DCAT, los formatos en los que se puede exportar la información y la problemática de utilizar RDF y SPARQL.

Arquitectura de los portales de datos europeos

Analizando los portales de datos europeos como son el portal de la UE [51,52] y el de España [53] se ha comprobado que utilizan componentes idénticos o muy similares, con lo que se demuestra que la problemática tratada en este Trabajo de Fin de Grado ocurre a nivel internacional. En la figura 3 se muestra un esquema en alto nivel donde se han combinado, a partir de ambos portales, los componentes existentes. Para entender el funcionamiento, los elementos clave se explican en el anexo A.1.

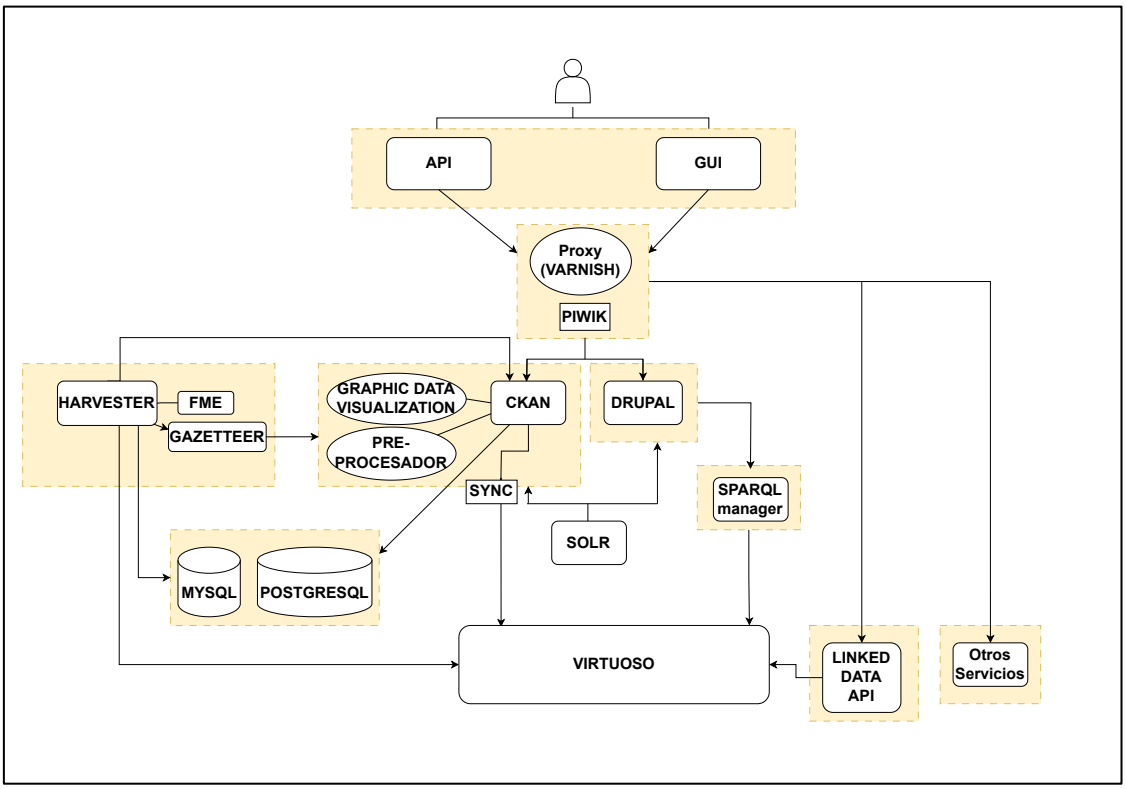


Figura 3 Diseño arquitectural del portal de datos de la UE y del Gobierno de España

Esta arquitectura a primera vista presenta el inconveniente de la complejidad. Conviven múltiples tecnologías y componentes, que, en las webs oficiales, se encuentran relacionados de forma muy poco clara. Además, el uso de SPARQL es una forma compleja de realizar consultas y se haya alejada de los datos. En este Trabajo Fin de Grado se va a aprovechar, en el sentido de validar las consultas que ofrece el portal y las que se pueden efectuar con GraphQL, pero en una versión futura podría emplearse para actualizar los conjuntos de datos

almacenados en este proyecto, cargando los últimos publicados. El sistema desarrollado se podría decir actúa como una capa que simplifica las operaciones de búsqueda y descubrimiento de datos a partir de los contenidos en Virtuoso (un sistema de bases de datos). Con este fin se evita tener un elevado número de endpoints a los que enviar diferentes consultas, pues el motor de GraphQL se encarga de gestionar los diferentes campos requeridos en una consulta por separado. Además, comparte los dos tipos de acceso presentados, por un lado, un usuario es capaz de interactuar a través de una UI, y por otro también es posible utilizar la API de GraphQL y ejecutar cualquier tipo de consultas sobre una interfaz que ofrece el propio motor de GraphQL, denominado GraphiQL o sobre una serie de endpoints predefinidos que se presentan en la nueva interfaz.

Portal de datos del Gobierno de España

El portal de datos abiertos del Gobierno de España se ha utilizado tanto como inspiración para llevar a cabo el diseño de la interfaz y de la arquitectura como para extraer conjuntos de datos y adecuarlos al modelo GraphQL. Ofrece una presentación y un rango de opciones similares a otros como el de la UE y el aragonés [79]. En primer lugar, al acceder al portal se muestra una página inicial con cierta información sobre instrucciones, datos estadísticos, actualidad, aplicaciones o empresas relacionadas. Desde aquí es posible acceder a otras secciones, entre ellas el área **Catálogo de Datos**, la que más interesa para este proyecto, desglosada a su vez en otras tres subsecciones: **Conjunto de Datos**, **API** y **Punto SPARQL**. En la sección API se encuentran todos los endpoints e instrucciones para realizar consultas sobre los conjuntos de datos. Dicha acción puede también realizarse a través del punto SPARQL, no obstante, se necesitan mayores conocimientos técnicos. Estas dos últimas secciones resultan útiles pues permiten probar con GraphQL las diferentes consultas existentes. Por otro lado, la sección Conjuntos de Datos sirve de base para la nueva interfaz del usuario. Se pueden diferenciar dos tipos de páginas: la primera (figura 4) contiene un listado con todos los conjuntos de datos y cierta información para cada uno, como el publicador o los formatos en los que está disponible, junto con una sección de filtros donde se pueden filtrar los conjuntos, por ejemplo, por categorías. Cuenta también con una barra de búsqueda, el número de recursos encontrados y los enlaces para descargarlos en diferentes formatos: RDF, CSV y ATOM. La otra página (figura 5) hace referencia a la información propia de cada conjunto de datos seleccionado en la página que se acaba de describir. En ella se incluyen datos como el publicador, el título, o enlaces para descargar los metadatos que describen el conjunto de datos en un formato concreto: XML (Extensible Markup Language), JSON, CSV, TURTLE (Terse RDF Triple Language) y RDF. Estos serán útiles en el proceso ETL para el mapeo y carga de datos en el esquema GraphQL de la aplicación.

Como se puede ver, la información que se muestra es relativamente simple, echándose de menos una mayor navegabilidad entre los recursos que aproveche mejor el modelo DCAT, basado en RDF que sigue una estructura en forma de grafo. Por ejemplo, a partir de un publicador poder acceder al resto de conjuntos de datos que ofrece y a los catálogos contenedores, o acceder a información de todos los catálogos existentes.

Figura 4 Página correspondiente a la sección Conjunto de Datos del portal de datos del Gobierno de España

Figura 5 Página correspondiente a la información sobre un conjunto de datos concreto en el portal de datos del Gobierno de España

Problemas de la web semántica: las desventajas de utilizar RDF y SPARQL

Al hablar de catálogos de datos abiertos conviene hablar de la Web Semántica [10], encargada de describir y etiquetar los conjuntos de datos, sus propiedades y sus relaciones, de forma que sean comprensibles tanto por las personas como por las máquinas. Analizándola se van a comprender cuales son las debilidades de RDF y SPARQL, permitiendo identificar aquellos aspectos en los que GraphQL ofrece una mejor solución.

Actualmente la Web Semántica se enfrenta a diferentes dificultades que han afectado su popularidad y utilidad a lo largo del tiempo, como se puede observar en la figura 7, que

representa su índice de popularidad desde 2004 hasta 2022. La principal causa de ello es que resulta demasiado compleja [67]. Su sintaxis, basada en tripletas (sujeto-predicado-objeto) [66] **RDF**, se ha demostrado difícil de manipular [68] y de leer, incluso con formatos como TURTLE o XML. RDF destaca por su inferencia lógica [4], que requiere una estructura de modelo clara y coherente para descubrir nueva información de manera efectiva, sin embargo, en la práctica, la falta de esta estructura coherente suele ser una barrera. Los datos en RDF siguen un modelo normalizado, donde la información se descompone en nodos y relaciones, lo que hace que en los grafos dirigidos complejos los límites entre los objetos no son claros, dificultando las consultas. Por ejemplo, si se define un grafo para representar datos de la ciudad de Zaragoza como monumentos o transporte público, surgen entrelazamientos y superposiciones que dificultan establecer unos límites claros en el sentido de donde termina una categoría de información y donde comienza la siguiente. Puede darse el caso de que ciertos monumentos estén ubicados cerca de puntos de transporte público, lo que hace que la información sobre los mismos esté vinculada tanto a la categoría de monumentos como a la de transporte. Algo parecido ocurre en la figura 6, donde los resultados de búsqueda a la consulta "fútbol en zaragoza" incluyen información sobre los votos a los candidatos al Consejo de Circoscrizionale para el distrito Oporto – Zaragoza. Además, muchos de los conjuntos de datos se encuentran desfasados. Es fácil localizar enlaces que no llevan a ningún sitio o que carecen de contenido, tal y como se muestra en la figura 8.

A esto se suma **SPARQL** [11], el lenguaje utilizado para realizar consultas en el grafo RDF. De nuevo la sintaxis es compleja y al trabajar con un esquema fijo, definido por las ontologías, las consultas deben ajustarse a esa estructura predefinida. Además, como RDF se base en tripletas, las consultas pueden llegar a ser ineficientes cuando los datos crecen notablemente, lo que implica un escaneo completo de estos y la resolución de múltiples combinaciones de tripletas. Los resultados son devueltos en formatos normalizados, como el ya mencionado TURTLE, menos intuitivos que, por ejemplo, JSON. Todo esto puede desencadenar en resultados ambiguos y poco precisos. Un ejemplo de consulta SPARQL, suponiendo que se está trabajando con un sistema que almacena información sobre monumentos y artistas, se muestra en el Código 1. En ella se obtiene la fecha de nacimiento del artista que creó un monumento específico, así como una lista de todas sus obras. Como se puede ver implica varias tripletas y restricciones que pueden llegar a complicarse mucho más si se desea obtener información más concreta o extensa.

```
PREFIX : <http://example.org/monumentos#>
SELECT ?fechaNacimiento ?nombreObra
WHERE{
  ?monumento :id "Puerta del Carmen"
  ?monumento :tieneArtista ?artista .
  ?artista :nacimiento ?fechaNacimiento .
  ?artista :tieneObra ?obra .
  ?obra :nombre ?nombreObra .
}
```

Código 1 Una query de ejemplo en SPARQL

Es aquí donde entra en juego GraphQL, que logra parte de lo que SPARQL y RDF prometieron, pero no llegaron a cumplir completamente [4]. GraphQL evita lidiar con la estructura completa de un documento cuando se quiere acceder a información específica del mismo, pues permite realizar consultas precisas para obtener solo la información o conjunto

de datos deseados. Además, GraphQL está centrado en JSON, lo que permite realizar consultas complejas y obtener los resultados en un formato estructurado y fácil de entender y permite actualizaciones del contenido de la base de datos desde el servidor con un tipo de consultas independientes al sistema denominadas mutaciones. Otras ventajas que presenta esta tecnología son la facilidad de uso, la independencia del tipo de base de datos y la capacidad de definir la estructura de los datos en un esquema muy flexible que describe los tipos, sus campos y la relaciones entre ellos.

De este modo, GraphQL evita problemas derivados de consultas o modelos poco precisos y complejos. La correspondiente consulta GraphQL a la de SPARQL anterior se presenta en el Código 2. Ya a simple vista la sintaxis es mucho más intuitiva y si se quisieran obtener otros metadatos, simplemente se agregan a la consulta como lo está `nacimiento`.

```
query{
    monumento(id: "Puerta del Carmen"){
        artista{
        nacimiento
        obras{
            nombre
        }
    }
  }
}
```

Código 2 Ejemplo query GraphQL

Figura 6 Muestra de búsqueda con resultados poco precisos. Fuente: https://data.europa.eu/data/

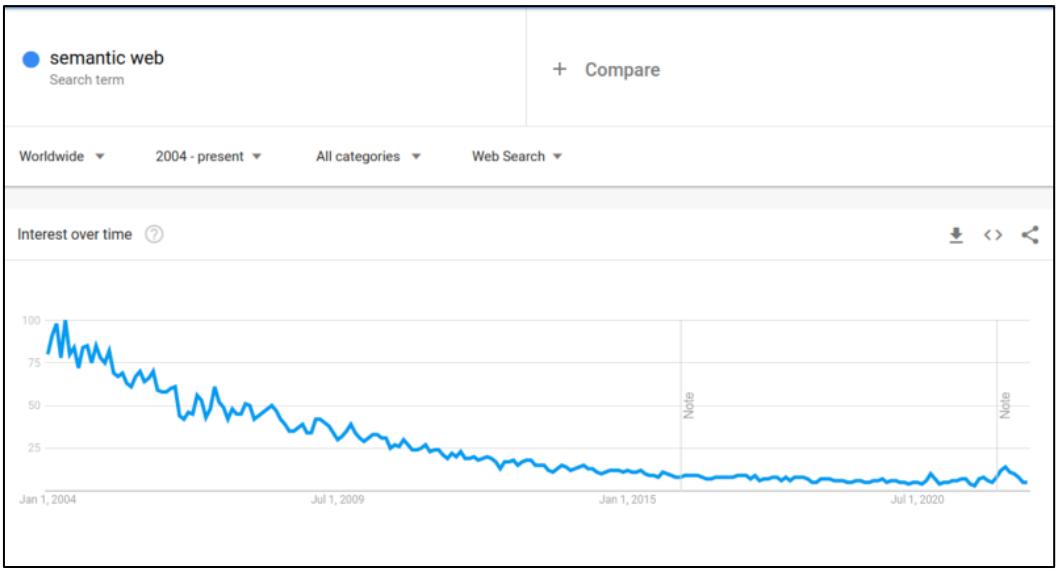


Figura 7 Evolución del interés por la web semántica en el periodo 2004-2022, extraído de: https://terminusdb.com/blog/the-semantic-web-is-dead/

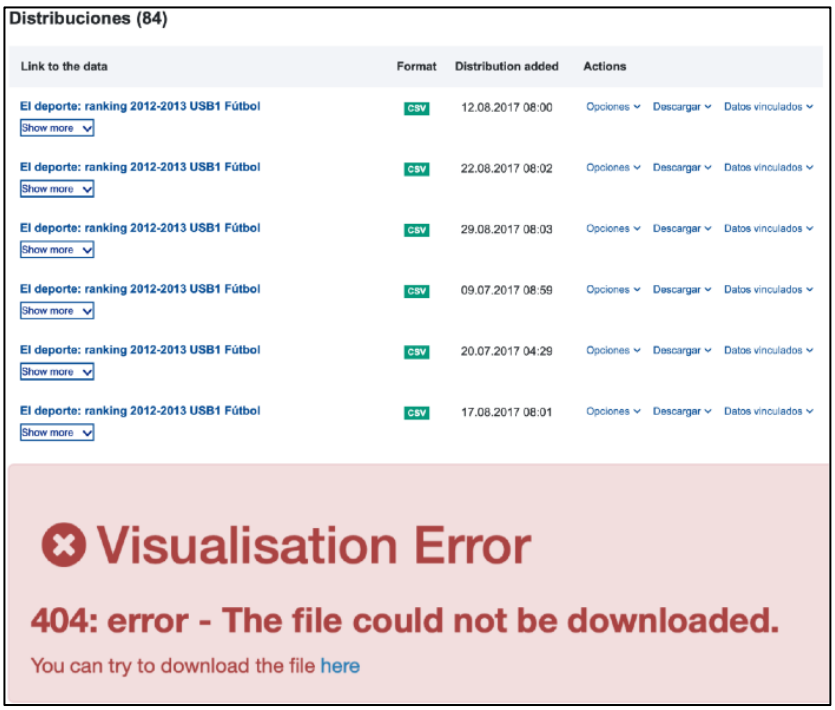


Figura 8 Ejemplo de información desactualizada. Al tratar de acceder al contenido no solo se muestra ese error si no que el navegador no encuentra el servidor. Fuente: https://data.europa.eu/data

Modelo de información en el que se basa la arquitectura: DCAT

Con el fin de desarrollar la iniciativa Open Data y asegurar la interoperabilidad entre los catálogos de datos publicados por diferentes portales de la web, el W3C (World Wide Web Consortium) publicó **DCAT**. Se trata de un vocabulario RDF [56] que describe los catálogos de datos a través de tres conceptos clave: catálogo (catalog), conjunto de datos (dataset) y distribución (distribution). A partir de este vocabulario la Comisión Europea impulsó una plataforma colaborativa (JoinUp) en la que se desarrolló el perfil de **Aplicación de DCAT para portales de datos europeos** (DCAT-AP) [57]https://joinup.ec.europa.eu/solution/dcat-application-profile-data-portals-europe, una especificación en la que se identificaron los elementos y atributos necesarios de este [55] y donde se describen diversas restricciones sobre

el modelo, con el objetivo de facilitar la homogeneización y la búsqueda cruzada mediante el uso de metadatos entre distintos portales europeos. Esta especificación, cuenta con una extensión para describir colecciones de conjuntos de datos (dataset series) y servicios de datos (data services) y otras extensiones para describir conjuntos de datos geoespaciales y estadísticos (GeoDCAT-AP y StatDCAT-AP). De esta forma se identifican los siguientes elementos fundamentales en la tabla 1. Para comprender mejor como se relacionan se muestra un grafo en el anexo A.2.2.

Tabla 1 Entidades u objetos del modelo DCAT extraídas al esquema GraphQL

Objeto	Descripción
Catalog	Colección de metadatos que describe uno o más conjuntos de datos.
Data Service	Entidad que proporciona acceso a conjuntos de datos o recursos de información mediante servicios en línea como API o puntos SPARQL.
Dataset	Colección de datos que comparten un cierto tema o propósito.
Dataset Series	Colección de conjuntos de datos que comparten un tema o propósito común.
Catalog Record	Describe mediante metadatos un recurso incluido en un Catalog.
Distribution	Describe como se ponen a disposición del usuario los datos de un dataset, indicando, por ejemplo, su formato.

En este proyecto se va a analizar el modelo DCAT (ver anexo A.2.1), llevando a cabo un proceso con unos puntos similares a los de la Comisión Europea, mencionado en esta sección. Se analizan los objetos, atributos y relaciones del modelo y se comienzan a extraer aquellos imprescindibles para el proyecto, modelando nuevas relaciones y otras ya existentes y agregando nuevos metadatos y propiedades a medida que el desarrollo del proyecto avanza. De esta forma se consigue un modelo compatible con el portal de datos del Gobierno de España, simplificado y preciso.

Formatos de exportación disponibles en el portal

El portal ofrece tres escenarios posibles a la hora de visualizar los conjuntos de datos: pueden verse todos los conjuntos disponibles, un subconjunto de estos de acuerdo con algún filtro o un conjunto de datos concreto. Para estos escenarios, el portal ofrece diferentes formatos de exportación. Para los dos primeros escenarios los formatos disponibles son: **RDF**, **CSV** y **ATOM**. El primero describe los recursos mediante tripletas sujeto-predicado-objeto, CSV almacena datos tabulares en forma de texto plano, donde cada línea representa una fila de datos y cada columna los valores separados por coma, y ATOM utiliza etiquetas XML para estructurar la información (por ejemplo, `<dcat:Dataset>`). De estos se va a utilizar CSV para la carga de datos, pues resulta sencillo leer un fichero CSV y estructurar su contenido, ya que cada línea representa los metadatos y relaciones, separados por columnas, de un conjunto de datos. Por ejemplo, una columna puede indicar el metadato correspondiente a la fecha de creación o los títulos. En el caso de un conjunto de datos concreto existen también varios formatos de descarga: **XML**, **JSON**, **CSV**, **TURTLE** y **RDF**. EL formato XML estructura los datos mediante etiquetas y atributos de forma legible tanto para humanos como para máquinas. En JSON los datos se representan en pares clave-valor y se basa en objetos, arrays o matrices. Por su parte, TURTLE representa datos semánticos en forma de tripletas de una manera más compacta y legible para los humanos. De estos se ha decidido centrarse en JSON por su amplia compatibilidad, legibilidad y fácil manipulación y extracción de los datos deseados.

Así, es posible realizar una primera aproximación sobre un conjunto de datos concreto a partir de su descripción en un fichero JSON y cuando la aplicación esté lista para cargar un mayor número de datos realizar una segunda prueba de concepto cargando múltiples conjuntos de datos a partir de un fichero CSV. Ambos ficheros se describen en el anexo A.5.

2.2 GraphQL: Un lenguaje de consultas moderno

GraphQL es, desde el punto de vista del **frontend**, un lenguaje de consulta y manipulación para API (permite hacer preguntas a un servidor especificando que campos se requieren) y desde el punto de vista del **backend** una capa en tiempo de ejecución, responsable de interpretar las consultas y responder con los datos adecuados. Esto se entiende como una arquitectura Cliente-Servidor: un cliente, con el que el usuario final interactúa, envía consultas a un servidor GraphQL, que las resuelve de acuerdo con el modelo y devuelve una respuesta, tal y como demuestra el diagrama de secuencia presentado en el anexo A.3.

Ofrece múltiples ventajas como la capacidad del cliente de solicitar únicamente aquello que necesita [13], evitando sobrecargas y facilitando la evolución de las API a lo largo del tiempo. Se resuelven así algunas limitaciones de la arquitectura **API REST** [12], lo que se conoce como **overfetching** (se evita extraer todos los datos relativos a un endpoint) y **underfetching** (se evitan múltiples llamadas a diferentes endpoints). A esto hay que añadir que el resultado se devuelve en formato JSON con una estructura muy similar a la consulta, lo cual es una gran ventaja a la hora de manipular los valores obtenidos. Además, el modelo es relativamente fácil de extender, pues lo único que se debe hacer es modificar el esquema agregando nuevos elementos, ya sean tipos, operaciones u otras propiedades [14]. En el Código 3 se muestra un ejemplo de una consulta en la que se recupera información relativa a un dataset. Cabe destacar que, entre los metadatos, se recupera el formato (`format`) de la distribución correspondiente.

```
query Datasets($filter:[MapInput!], $type:String!,$page: Int!){
    resourcesByFilter(filters:$filter, type:$type, page: $page){
        ... on Dataset{
            id
            distributions{
                format
            }
        }
    }
}
```

Código 3 Ejemplo de consulta GraphQL para obtención de información sobre datasets

Conceptos clave de GraphQL: Esquema

Uno de los principales elementos de GraphQL es el esquema o **schema**. En él se define la estructura de los datos que pueden manipularse (mutaciones) y consultarse (consultas), declarando los tipos de objetos, sus campos y las relaciones entre ellos, así como las operaciones de consulta y mutación. Puede interpretarse como un contrato entre el cliente y el servidor en el que se especifican que operaciones están permitidas y cómo deben estructurarse. El hecho de poder representar las relaciones entre los diferentes objetos permite disponer de una especie de grafo de todo el modelo, lo que hace posible que con consultas relativamente simples es posible recorrerlo al completo desde cualquier punto. Los tipos (**types**) son las estructuras de datos fundamentales en un esquema GraphQL y pueden representar objetos (las entidades), escalares (tipos de datos primitivos como String o personalizados), enumeraciones, uniones o interfaces. Por último, existen dos operaciones fundamentales en GraphQL: consultas (**queries**) y mutaciones (**mutations**). Las consultas se emplean para recuperar una serie de datos solicitados al servidor y las mutaciones sirven para modificar los datos. Se les pueden pasar parámetros para, por ejemplo, filtrar, ordenar o paginar. Un ejemplo de esquema GraphQL puede verse en el Código 4:

```
schema{
  query: Query
  mutation: Mutation
}

type Query{
  catalog(id: ID): Catalog
}

type Mutation{
  createCatalog(input: CatalogInput):Catalog
}

type Catalog{
  id: ID!
  title: [LangString!]
  description: String!
  catalogs: [Catalog!]
}
```

Código 4 Ejemplo de schema GraphQL

2.3 Tecnologías en las que se apoya el desarrollo con GraphQL

Dado que el proyecto tiene como base GraphQL, es importante contar con tecnologías que se integren fácilmente con esta y ofrezcan herramientas que simplifiquen la implementación del código y que se adapten a la estructura del proyecto. Por ello, se ha decidido utilizar como lenguaje común a toda la aplicación, **Kotlin**, aprovechando **Kotlin Multiplatform** [18] para implementar la parte del cliente y del servidor, pues esta tecnología se ha diseñado para simplificar el desarrollo de proyectos multiplataforma, evitando que se restrinja únicamente a una tecnología nativa o web [19]. Además, se integra muy bien con GraphQL. Entre las ventajas de Kotlin está el hecho de que permite manejar los tipos nulos de forma explícita, lo que supone una ventaja al trabajar con GraphQL, quien tiene una manera de tratarlos denominada **nullable types** [16] y es un detalle que se da bastante en el código. Otra ventaja de utilizar este lenguaje es su compatibilidad con Java, lo que resulta útil pues muchas tecnologías utilizadas tienen este lenguaje como base.

Otras tecnologías que merece la pena mencionar son Apollo Kotlin y Netflix DGS. En el proyecto, **Apollo Kotlin** [17] se emplea para enviar solicitudes GraphQL al servidor, simplificando notablemente la comunicación. Todo está automatizado, pues adapta las consultas GraphQL a Kotlin generando el correspondiente código. Además, evita lidiar con el análisis de un tipo JSON para obtener los campos de la respuesta a la consulta, que se estructura en base a los tipos de GraphQL generados en Kotlin. Por último, el framework **Netflix DGS** (Netflix Domain Graph Service) [15] consiste en una biblioteca de código abierto que combina los principios de GraphQL con **Spring Boot** (utilizado en el servidor) para facilitar la creación de servicios GraphQL en entornos Java. En cuanto a sus características ofrece un framework de pruebas para test unitarios que permite escribir consultas y un plugin que automatiza la generación de código (Java o Kotlin) a partir del esquema [20], entre otros aspectos. Un examen más profundo sobre la relación que existe entre las distintas tecnologías y los requisitos no funcionales que debe cumplir este proyecto se explica en el anexo A.4.2.

3 Diseño

Esta sección ofrece una visión global de la aplicación en alto nivel y explica el modelo de datos GraphQL. Finalmente se muestra un mockup de la interfaz, siguiendo los requisitos funcionales (anexo A.4.1).

3.1 Diseño del esquema GraphQL partiendo del modelo DCAT

El diseño del esquema se basa, en las versiones 2 y 3 del modelo **DCAT**. De ellas se han extraído los objetos y relaciones más importantes y se han agregado las propiedades oportunas, evitando aquellos aspectos ambiguos o redundantes. Esto se puede comprender mejor a través del diagrama de clases de la figura 9. En él se representan las principales entidades, metadatos y relaciones. Como se puede comprobar se han definido una serie de clases que permiten distinguir entre recursos (**ResourcesInCatalog**) y datasets (**DatasetsInCatalog**). Esto es debido a que las entidades base (ver tabla 1) comparten la mayoría de los atributos y, como se muestra en los modelos DCAT (ver anexo B.2), existe una relación jerárquica que se resume en una entidad padre denominada **Resource** de la que heredan el resto. Aquí se ha ido un paso más y se ha dividido dicha clase en dos, pues **DataService**, al contrario que el resto, se identifica como Resource, pero no se incluye en **DatasetsInCatalog**, debido a que sirve o da acceso a los recursos, pero no a sí mismo, con lo que surgía un conflicto si todos se trataban como recursos. De dichas clases se han evaluado sus propiedades extrayendo aquellas que resulten útiles para el esquema. Estas propiedades, recogidas en la tabla 2, son metadatos, que se organizan a través de un espacio de nombres denominado **`DCT`** (Dublin Core Terms), en la versión 2, y que posteriormente ha evolucionado a **`DCTERMS`** (Dublin Core Metadata Initiative Terms), empleado en la versión 3, que proporciona una variedad más amplia y rica de términos para describir recursos en la Web Semántica. Para un mayor detalle sobre los rangos y las transformaciones correspondientes, así como aspectos de implementación, se puede consultar el anexo A.2.3.

Tabla 2 Mapeo metadatos DCAT 2 y 3

Elemento	Rango	Notas
<dc:title>	rdfs:Literal	Es un nombre dado a distribuciones, recursos o records. Se almacena en un objeto junto con su idioma para el caso de los 2 primeros (cadenas de texto) y una sola cadena de texto para el tercero.
<dct:accessUrl>	rdfs:Resource	Es una URL del recurso que brinda acceso a una distribución del conjunto de datos, por ejemplo, un punto de consulta SPARQL.
< dcat:byteSize >	xsd:decimal (DCAT 2) xsd:nonNegativeInteger (DCAT 3)	Tamaño de la distribución en bytes. Se utiliza como un entero no negativo.
< dct:format >	dct:MediaTypeOrExtent	El formato de archivo de la distribución. Se utilizan tipos MIME.
< dct: identifier >	rdfs:Literal	Es un identificador único del recurso que se está describiendo o catalogando.
< dct:description >	rdfs:Literal	Una descripción textual libre del recurso o la distribución.
< dct:language >	dct:LinguisticSystem	Se refiere al idioma natural utilizado para los metadatos textuales (títulos, descripciones, etc.) de un recurso o distribución. Se asigna al recurso.
< dct:issued >	rdfs:Literal	Fecha de emisión del recurso.
< dct:modified >	rdfs:Literal	Fecha de la última modificación del recurso.
< dcat:theme >	skos:Concept	Representa una categoría de un recurso, puede haber varias. Como cadena de texto.

< dct:publisher >	foaf:Agent	Entidad que pone el recurso a disposición. Con un objeto se almacena su identificador y nombre. Por ejemplo: “A0DAT0002” – “ANIMSA”. A partir de la primera letra del código (“A0...”) se puede obtener el tipo de organismo. La ‘A’ corresponde con Administración Autonómica.
< dct:keyword >	rdfs:Literal	Etiqueta que describe el recurso. Lleva asociado también el idioma de esta.
< dct:spatial >	dct:Location	El área geográfica cubierta por el conjunto de datos. Se representa como una cadena de texto. Es propia del dataset.
< dct:AccrualPeriodicity >	dct:Frequency	Frecuencia en la que se publica el dataset. Se representa mediante un objeto que indica la frecuencia (p.e: en días, horas...) y el periodo (número decimal). Por ejemplo: time:years 1.0. Esto se pude también mapear y establecer, para el ejemplo, que es una frecuencia anual.
< dct:temporal >	dct:periodOfTime	Periodo temporal que abarca el dataset. Viene dado por una fecha de inicio y otra de finalización.
< dct:rights >	dct:RightsStatement	Derechos no abordados por dcterms:license o dcterms:accessRights, se ha decidido utilizar para registrar otras regulaciones asociadas al dataset.
< dct:relation >	•	Representa un recurso con una relación no especificada con el recurso catalogado. Se utiliza para contener las URL de los recursos relacionados.
< dct:license >	dct:LicenseDocument	Un documento legal bajo el cual se pone a disposición el recurso. Se almacena la URL correspondiente. Se relaciona con el Lincensing Assistant de la arquitectura de este tipo de portales.

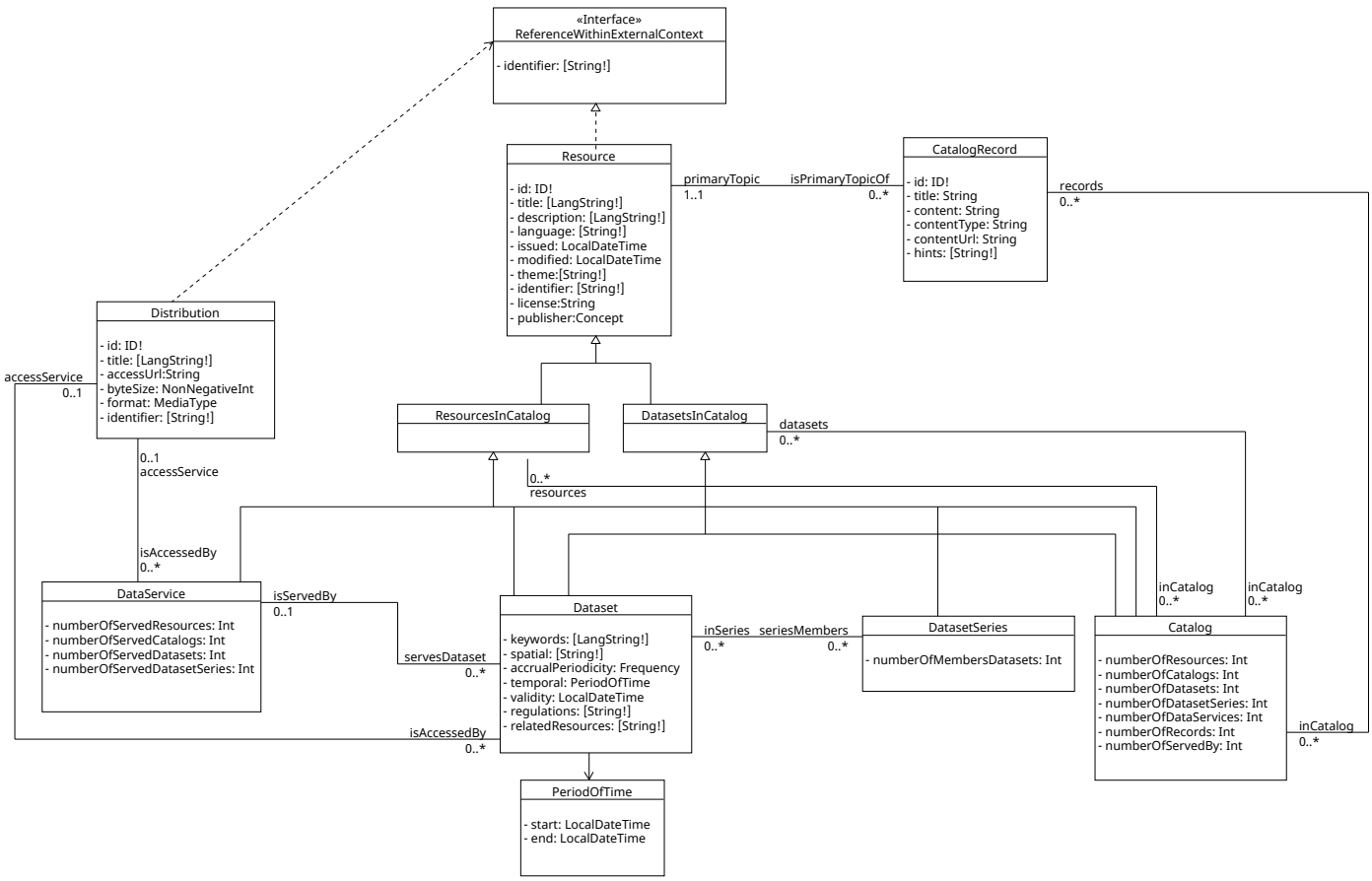


Figura 9 Diagrama de clases UML del esquema GraphQL inferido a partir de DCAT

3.2 Visión general del sistema: arquitectura y patrones de diseño utilizados

La aplicación consta de tres módulos bien diferenciados (figura 10): cliente, servidor GraphQL y base de datos, y se diferencian dos maneras en las que el usuario puede hacer uso de ella. Por un lado, puede atacar directamente la API de GraphQL, a través de un IDE denominado GraphiQL [83], o tiene la opción de interactuar con ella a través de una UI en una página web del tipo SPA, donde se simula parte de la API del portal español para ejecutar algunas consultas predefinidas . Esta interfaz forma parte del **Cliente**, encargado no solo de mostrar información al usuario sino también de enviar solicitudes (consultas o mutaciones) al **Servidor GraphQL**. El servidor se puede dividir en dos submódulos, por un lado, se encuentra la lógica **GraphQL**, encargada de gestionar el esquema, resolver consultas y responder al cliente y, por otro lado, se encuentra todo lo relacionado con el proceso ETL, el **Harvester**, encargado de acceder a las fuentes de datos externas, estructurar los datos correspondientes y cargarlos en la **base de datos**. Esta conforma el tercer módulo, consistente en aquellas bases de datos utilizadas para persistir todos los valores de los metadatos que componen el esquema. Su diseño se explica en el anexo B.3. Como fuente de datos se toman los conjuntos de datos ofrecidos por datos.gob.es.

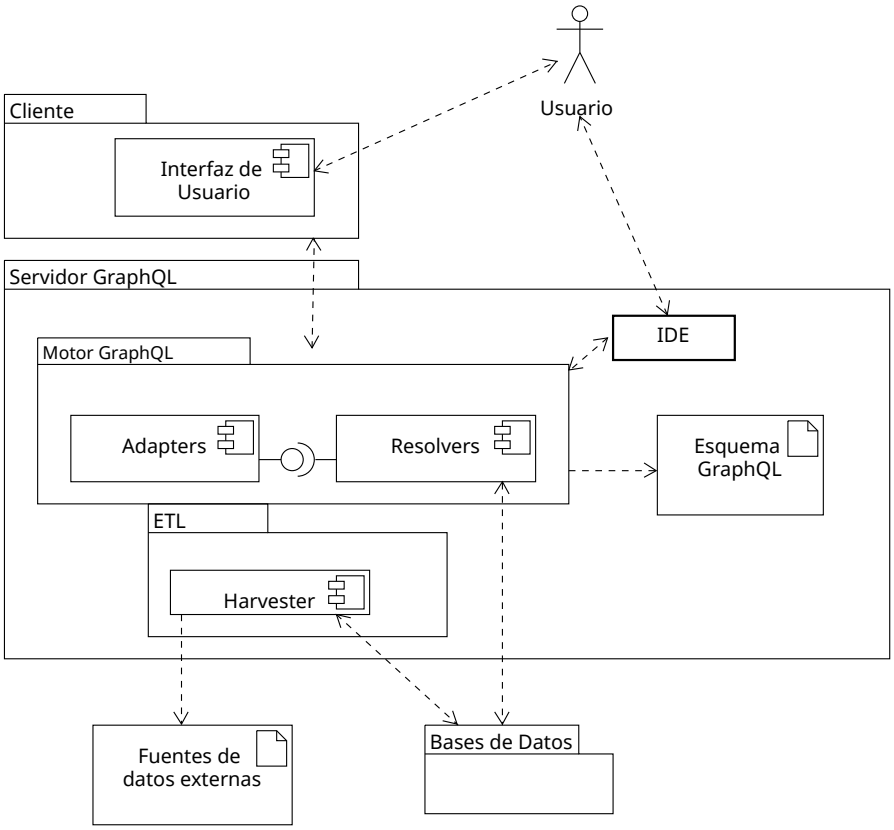


Figura 10 Esquema que muestra cómo se organiza el sistema a desarrollar

Si se entra más en detalle la disposición de estos módulos sigue un modelo de arquitectura hexagonal [58] parcial. El objetivo es reducir el acoplamiento de manera que sea inexistente o muy limitado, con el fin de aislar la lógica de negocio y poder modificar, por ejemplo, la capa de acceso a datos, sin afectar al resto. Se dice que es parcial pues no aísla al completo el núcleo de cualquier detalle técnico, en especial, de lógica relacionada con las entidades de la base de datos, donde se utilizan anotaciones de un **ORM** (Object-Relational Mapping) [70] para determinar la clave primaria, columnas o tabla a la que se asignan. Esta arquitectura se

puede ver como un hexágono en cuyo centro se encapsulan las funcionalidades y las reglas de negocio y se rodea por capas de puertos y adaptadores que se comunican con el exterior. Los adaptadores se encargan de ajustar la información y las solicitudes entre el mundo exterior y el núcleo de la aplicación y los puertos son interfaces que definen cómo el núcleo de la aplicación interactúa con los adaptadores, definiendo la estructura de los datos y las operaciones que deben implementarse. De esta forma en el núcleo se encuentra la implementación de los servicios relacionados con el proceso ETL, los resolvers o con la comunicación cliente-servidor. Además, también se encuentra la lógica GraphQL, tanto el esquema como ciertas consultas o mutaciones implementadas y las entidades de la base de datos. Por otro lado, la capa de puertos la componen las interfaces que dan acceso a estos servicios, y son utilizados o extendidos por la capa de adaptadores (en función de la comunicación, de dentro hacia fuera o al revés). Aquí se encuentran los relacionados con la base de datos (denominado **adaptador de persistencia**), el harvester con acceso a fuentes externas y los adaptadores de GraphQL para realizar y procesar las consultas. La explicación detallada se encuentra en el anexo B.1.

3.3 Diseño del proceso ETL

Una etapa crucial en el desarrollo de soluciones de gestión de datos es el proceso ETL. Durante esta fase se traza el camino que van a seguir los datos desde el origen hasta su destino final, atravesando una serie de puntos intermedios con transformaciones y mejoras para finalmente ser cargados en el sistema de almacenamiento correspondiente. De este procedimiento se encarga el módulo **Harvester**. El origen de los datos es el portal de datos del Gobierno de España, de donde se descargan en el formato correspondiente para que, posteriormente, un servicio de este módulo acceda localmente a ellos, con el fin de aplicar, junto a otros servicios, las correspondientes transformaciones y por último ser cargados en la base de datos.

Tal y como se ha mencionado en la sección de análisis, los formatos planteados a partir de los que extraer los metadatos referentes a los conjuntos de datos son dos: **JSON** y **CSV**. La diferencia principal entre ambos es que el primero solo mantiene la información relativa a un conjunto de datos, mientras que el segundo contiene todos. Esto hace que el proceso ETL en el caso del JSON resulte ser muy lento si se va a tratar una enorme cantidad de conjuntos de datos. Y hay que tener en cuenta que se van a cargar más de los 60.000 datasets que contiene portal. Por ello, se ha acabado utilizando CSV, tal y como se detalla en la sección 4.1. No obstante, la aplicación se ha diseñado con la capacidad de agregar nuevos modos de procesamiento para futuros formatos sin afectar al resto. Todo este proceso puede lanzarse a través de una mutación de GraphQL, indicando o bien la ruta a la fuente de información o bien pasando el contenido directamente como una cadena de texto.

3.4 Diseño de la interfaz de usuario

A la hora de diseñar la interfaz se busca un formato con el que el usuario está familiarizado por lo que el diseño se basa en los portales de datos abiertos europeos mencionados. De estos se ha aprovechado todo aquello necesario, como por ejemplo el apartado de filtros, y se le ha dado una vuelta, consiguiendo un resultado intuitivo y fácil de utilizar con mayor navegabilidad entre los recursos. Se han evitado enlaces a fuentes externas o información que no guarda relación con los datos que se muestran.

De esta forma se distinguen tres páginas principales: la primera (figura 11) muestra un listado de los recursos existentes, por ejemplo, los datasets existentes, junto con el apartado de filtros. Los filtros seleccionados se pueden visualizar en la parte superior, debajo del valor que indica el número de elementos encontrados, lo que facilita al usuario conocer exactamente por qué valores está buscando. Complementario a esto es posible la ordenación por diferentes temas y paginar. La segunda (figura 12) contiene información en detalle del elemento correspondiente

al recurso seleccionado en el listado de la primera página. En esta página es posible navegar a través de enlaces por los diferentes elementos del grafo que intervienen en dicho recurso, y seleccionar o editar los filtros para una nueva búsqueda. Por último, la tercera página (figura 13) hace referencia a la API del portal. En ella se presentan una serie de consultas predefinidas que resultan interesantes y demuestran la capacidad de búsqueda que ofrece GraphQL. En las tres páginas existe una cabecera con un menú que permite seleccionar entre los diferentes recursos tratados (Catalog, Catalog Record, Dataset Series, Data Service, Dataset y Distribution), mostrando la primera página para cada uno, y la API. El diseño se ha realizado a través de Figma [78].

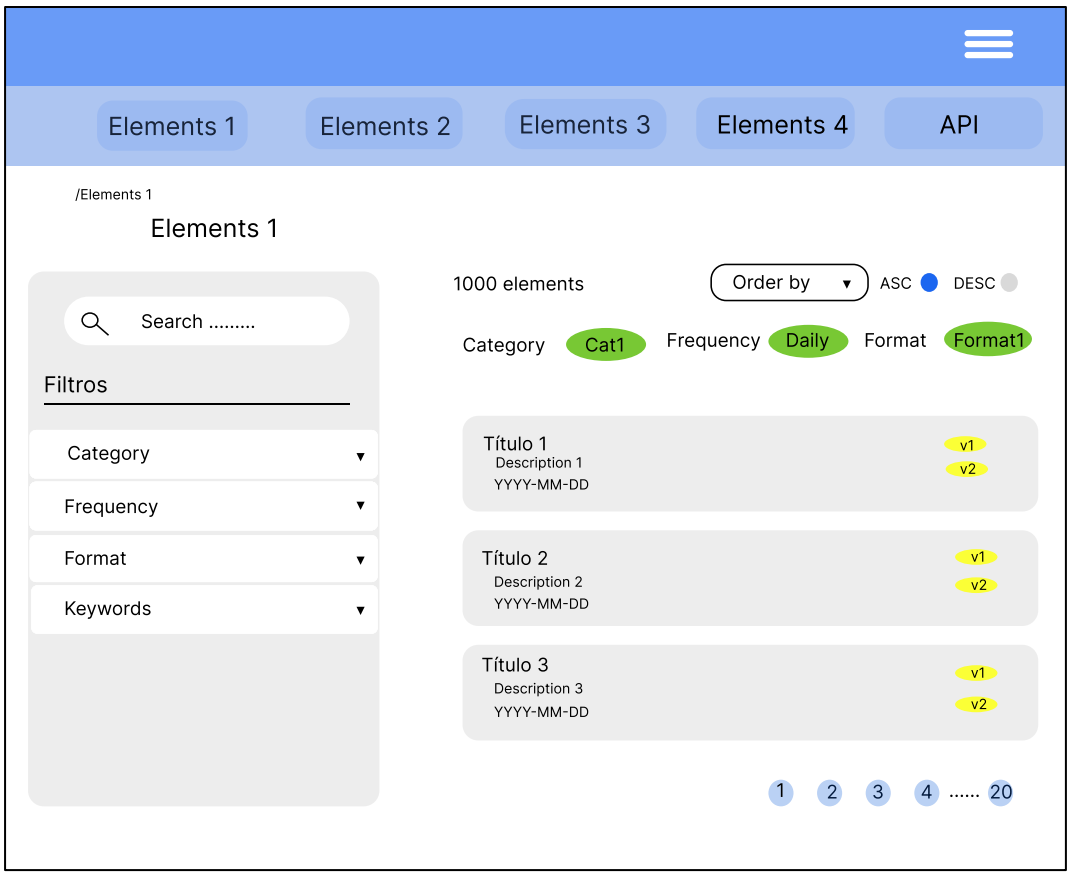


Figura 11 Diseño inicial de la primera página, con el listado, el área de filtros y el menú desplegado

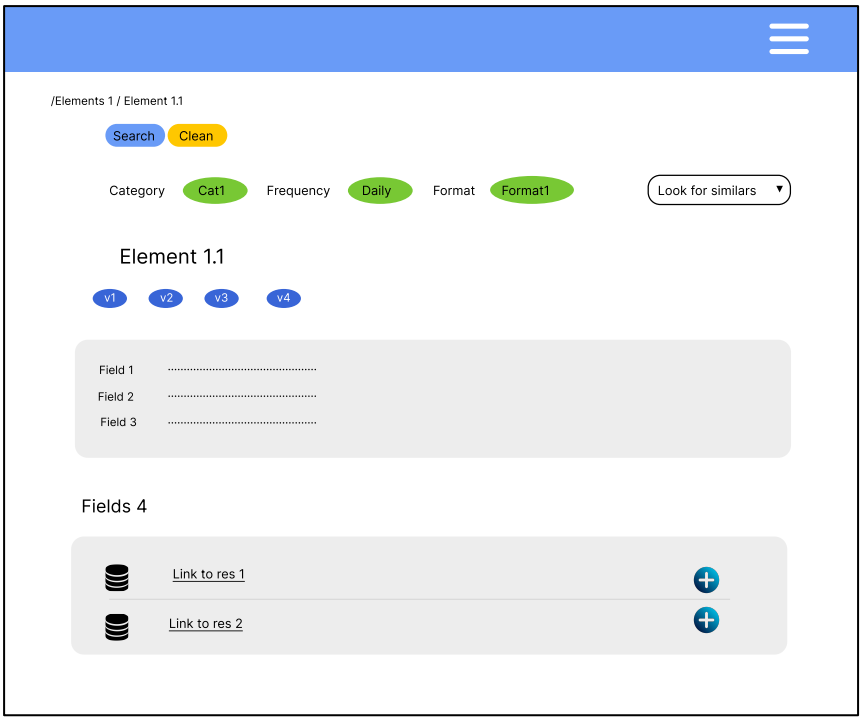


Figura 12 Diseño inicial de la segunda página con listado de filtros seleccionados, compartidos con la primera

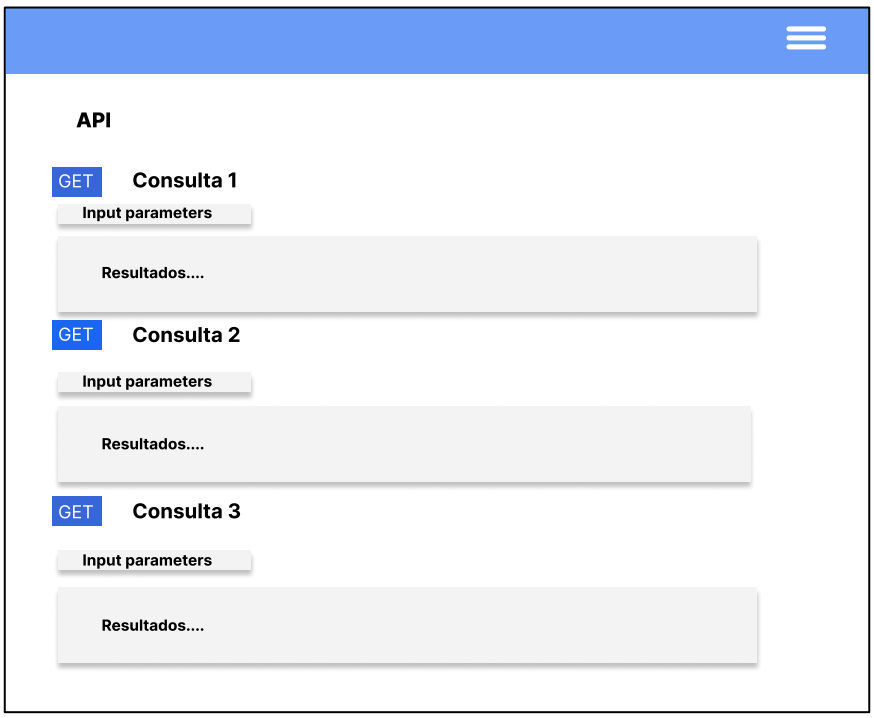


Figura 13 Diseño inicial de la tercera página con una serie de consultas que es posible realizar

4 Desarrollo

En esta sección se incluyen los detalles más reseñables de la implementación final de los módulos que conforman el sistema principalmente (figura 10): cliente, servidor GraphQL y bases de datos, configurados en la aplicación a través de la organización que ofrece Kotlin Multiplatform (anexo C.1). Además, se indican los problemas encontrados durante el proceso. Todo ello en base a los aspectos tratados en el apartado de diseño.

4.1 Implementación del proceso ETL

Como se ha visto en la sección de análisis, el fichero CSV almacena el conjunto de metadatos que describen los datasets existentes en el portal. Para poder adaptar dichos datasets al modelo GraphQL del proyecto hay que aplicar un proceso de extracción y posteriormente uno de transformación sobre los datos correspondientes. Con Kotlin esto es muy simple, pues existen librerías de código abierto (como **`csvReader`**) que permiten cargar el contenido del fichero y procesarlo por lotes, pudiendo extraer y modelar los metadatos de cada línea en un mismo proceso. La implementación correspondiente se muestra en el Código 5, donde por cada línea del fichero CSV se crea, a partir del contenido de sus columnas, una estructura de datos denominada `DatasetCSVModel` que va a contener los metadatos correspondientes (anexo C.3.3). La carga de estos datos se realiza en una base de datos **PostgreSQL**, que permite manejar y persistir grandes volúmenes de datos. Además, se despliega con Docker, lo que aumenta la disponibilidad y portabilidad y su configuración e integración con la aplicación es muy sencilla, tal y como se muestra en el anexo C.2.1. El proceso puede iniciarse mediante una mutación GraphQL (anexo C.3.4). También cabe mencionar que previamente se barajaron otras opciones, tal y como se indica en el anexo C.3.1, en lo referido a la base de datos utilizada y al formato de la fuente de información, donde se planteó el uso de JSON, cuyo proceso ETL se explica en el anexo C.3.2).

```
fun processCsv(inputStream: InputStream): List<DatasetCSVModel> = csvReader().open
(inputStream) {
        readAllWithHeaderAsSequence().map {
            DatasetCSVModel(
                it["URL"],
                fieldOrNull(it["IDENTIFICADOR"]),
                titleDescription(it["TÍTULO"]),
                titleDescription(it["DESCRIPCIÓN"]),
                fieldSplit(it["TEMÁTICAS"]),
                keywords(it["ETIQUETAS"]),
                localDateTimes(it["FECHA DE CREACIÓN"]),
                localDateTimes(it["FECHA DE ÚLTIMA MODIFICACIÓN"]),
                frequency(it["FRECUENCIA DE ACTUALIZACIÓN"]),
                fieldSplit(it["IDIOMAS"]),
                fieldOrNull(it["ÓRGANO PUBLICADOR"]),
                fieldOrNull(it["CONDICIONES DE USO"]),
                fieldOrNull(it["COBERTURA GEOGRÁFICA"]),
                periocity(it["COBERTURA TEMPORAL"]),
                localDateTimes(it["VIGENCIA DEL RECURSO"]),
                relatedResourcesAndRegulations(it["RECURSOS RELACIONADOS"]),
                relatedResourcesAndRegulations(it["NORMATIVA"]),
                distributions(it["DISTRIBUCIONES"]),
            )
        }.toList()
    }
```

Código 5 Función de extracción y transformación de recursos en un fichero CSV

4.2 Servidor GraphQL

El servidor contiene la implementación en Kotlin de los distintos servicios, como pueden ser los resolvers, el código correspondiente a la base de datos (entidades y repositorios) y los adapters de Netflix DGS. Además, gestiona el esquema GraphQL.

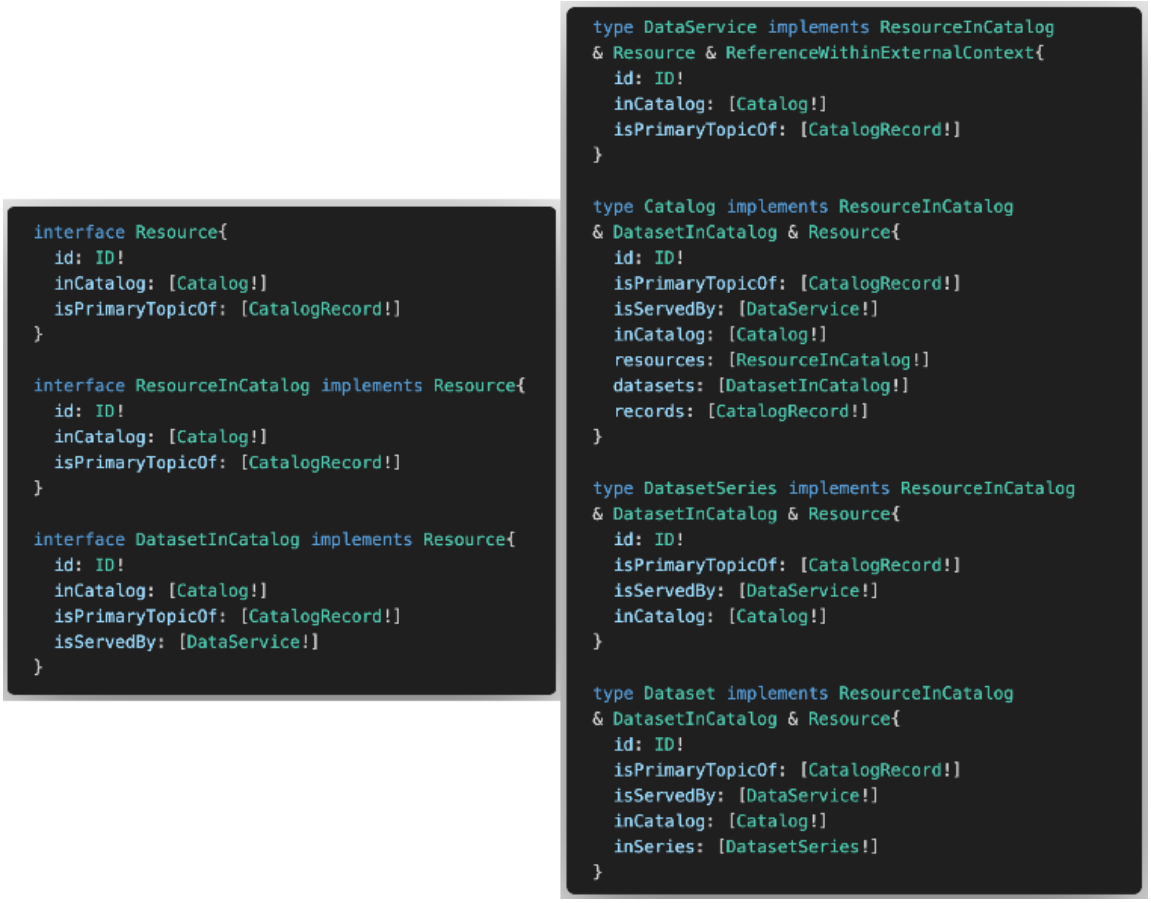
Esquema GraphQL

La definición del esquema se realiza en un fichero con formato **`graphqls`**. En los Código 6 y Código 7 se muestran los tipos de objetos principales que lo componen. Como ya se ha explicado en la sección de diseño, existe una especialización de la clase **`Resource`** en dos subclases, ya que de los cuatro tipos de objetos principales: data service, catalog, dataset series y datasets, los tres últimos pueden ser servidos por el primero, así se pueden clasificar como **`DatasetsInCatalog`**. Esto es una forma de organizar la información, pues comparten la mayoría de los atributos. Para implementarlo se utilizan las interfaces de GraphQL. La ventaja es que una interfaz puede extender otras y a su vez ser implementadas por un tipo de objeto, heredando todos sus atributos, que también deben incluirse en el objeto o la interfaz que implementa. Por otro lado, las clases correspondientes a catalog record y distribution se definen como tipos de objetos y ya en sus propiedades se establecen las relaciones que formarán parte del grafo del modelo. Al igual que el resto de las propiedades estas se declaran con el siguiente formato: <nombre>:<tipo>, donde <tipo> puede ser un escalar, un tipo primitivo (por ejemplo, Boolean) u otro tipo de objeto. Para verlo mejor, en el Código 6, el tipo de objeto **`Distribution`** contiene la propiedad `accessUrl` cuyo tipo es una cadena de texto (`String`), contiene también la propiedad `title`, cuyo tipo es una colección de `LangString`, que es un objeto compuesto por dos cadenas de texto y dispone a su vez de la propiedad `isDistributionOf`, cuyo tipo es una colección de objetos del tipo `Dataset`.

```
type CatalogRecord{
  id: ID!
  title: String
  contentType: String
  content: String
  contentURL: String
  hints: [String!]
  primaryTopic: ResourceInCatalog!
  inCatalog: [Catalog!]
}
```

```
type Distribution{
  id: ID!
  title: [LangString!]
  accessUrl:String
  isDistributionOf: [Dataset!]
  accessService: [DataService!]
  byteSize: NonNegativeInt
  format: MediaType
}
```

Código 6 Tipos de objetos que representan las clases catalog record y distribution respectivamente



```graphql
interface Resource{
  id: ID!
  inCatalog: [Catalog!]
  isPrimaryTopicOf: [CatalogRecord!]
}

interface ResourceInCatalog implements Resource{
  id: ID!
  inCatalog: [Catalog!]
  isPrimaryTopicOf: [CatalogRecord!]
}

interface DatasetInCatalog implements Resource{
  id: ID!
  inCatalog: [Catalog!]
  isPrimaryTopicOf: [CatalogRecord!]
  isServedBy: [DataService!]
}

type DataService implements ResourceInCatalog
& Resource & ReferenceWithinExternalContext{
  id: ID!
  inCatalog: [Catalog!]
  isPrimaryTopicOf: [CatalogRecord!]
}

type Catalog implements ResourceInCatalog
& DatasetInCatalog & Resource{
  id: ID!
  isPrimaryTopicOf: [CatalogRecord!]
  isServedBy: [DataService!]
  inCatalog: [Catalog!]
  resources: [ResourceInCatalog!]
  datasets: [DatasetInCatalog!]
  records: [CatalogRecord!]
}

type DatasetSeries implements ResourceInCatalog
& DatasetInCatalog & Resource{
  id: ID!
  isPrimaryTopicOf: [CatalogRecord!]
  isServedBy: [DataService!]
  inCatalog: [Catalog!]
}

type Dataset implements ResourceInCatalog
& DatasetInCatalog & Resource{
  id: ID!
  isPrimaryTopicOf: [CatalogRecord!]
  isServedBy: [DataService!]
  inCatalog: [Catalog!]
  inSeries: [DatasetSeries!]
}
```

Código 7 Esquema GraphQL mostrando la jerarquía de clases

Adaptadores GraphQL-Kotlin utilizando Netflix DGS y resolución de solicitudes GraphQL.

Hasta aquí se ha visto como se estructura el modelo con la declaración del esquema, pero ¿cómo interactúa el servidor con él? Para ello están los adaptadores. Estos permiten conectar el código Kotlin con el esquema GraphQL, de forma que su propio motor identifica que operación se ejecuta y que campos se solicitan. Dichos campos se resuelven ejecutando los servicios Kotlin correspondientes. Además, es posible implementar escalares personalizados y que sean reconocidos por el esquema.

La tecnología utilizada para su implementación es Netflix DGS, que ofrece una serie de anotaciones entendibles por su motor de resolución DGS (por ejemplo, `@DgsQuery`), de forma que es capaz de determinar automáticamente que método Kotlin debe ejecutarse para resolver cada campo. Estos métodos se denominan **resolvers** y consisten en interfaces que actúan como servicios. En su implementación se accede a la base de datos y se aplican las transformaciones necesarias a los datos obtenidos con el fin de devolver el campo solicitado en la consulta correctamente. Además, es capaz de generar en tiempo de compilación el esquema GraphQL en código Kotlin, de forma que es posible acceder a los diferentes elementos en los distintos servicios. Todo esto de forma automática y con una configuración muy simple. Todo lo tratado en este punto se explica en el anexo C.5.

4.3 Cliente

En el cliente se lleva a cabo la implementación de la interfaz del usuario mediante Kotlin JS, la opción que ofrece Kotlin para trabajar con la librería de JavaScript React. Además, se implementan las consultas con el servidor GraphQL, ejecutadas a través de Apollo Kotlin.

Interfaz de usuario mediante React y Kotlin

React es una librería de JavaScript enfocada en crear interfaces de usuario. Su característica distintiva es que se basa en componentes, y cada uno puede controlar su propio estado. Esto permite construir interfaces más complejas y resolver funcionalidades de manera sencilla. Además, permite combinar código HTML con JavaScript gracias a la sintaxis JSX de sus componentes. Dado que React está escrito en JavaScript, se requiere de una forma de comunicar Kotlin con este lenguaje y para ello existe **Kotlin JS** y los denominados **Kotlin Wrappers** [69]. El primero permite compilar el lenguaje de programación Kotlin en código JavaScript y los segundos permiten interactuar desde Kotlin con bibliotecas y frameworks escritos en JavaScript, como es React (anexo C.4.1). Por ejemplo, el wrapper **kotlin-react**, permite crear y manipular componentes de React mediante código Kotlin en lugar de JavaScript (ver el Código 8).

```
val RadioGroup = FC<Props>{
    FormControl{
        sx{ marginLeft = 60.pct}
        RadioGroup{
            row = true
            FormControlLabel{
                value = "o"
                control = Radio.create(){size = Size.small}
                label = ReactNode("O")
            }
        }
    }
}
```

Código 8 Ejemplo de componente React en Kotlin JS

Elaboración de consultas sobre el servidor GraphQL mediante Apollo Kotlin

Apollo Kotlin es un cliente de GraphQL con el que se generan modelos Kotlin y Java a partir de operaciones de consulta o mutación. Gracias a él, es posible interactuar con el servidor GraphQL. Por ejemplo, se pueden solicitar mediante una consulta ciertos campos, como el título, de un conjunto de datasets y mostrarlos al usuario final en una lista. Además, ofrece tareas que automatizan la generación de código Kotlin en tiempo de compilación a partir de GraphQL facilitando su invocación en el proyecto. En el Código 9 se muestra la ejecución mediante Apollo de la consulta mostrada en el Código 3. El parámetro `DatasetQuery` es la consulta GraphQL en código Kotlin generada por Apollo y el resto son parámetros propios de ella. Mediante `.data` se obtiene el resultado a la consulta, que se puede ver como una jerarquía similar a la consulta y cuya raíz es este valor. Por ejemplo, para acceder al siguiente nivel habría que utilizar `.data.resourcesByFilter`. Se pueden ver más detalles en el anexo C.4.2.

```
apolloClient.query(
    DatasetsQuery(filter=Optional.present(filter),type=values, page = page)
).execute().data
```

Código 9 Ejemplo de ejecución de la consulta mostrada en Código 3 mediante Apollo Kotlin

4.4 Adaptador de persistencia

El adaptador de persistencia permite interactuar entre el sistema de almacenamiento de datos, en este caso PostgreSQL, y la aplicación. Actúa como un intermediario que maneja la lógica de como los datos se almacenan y recuperan desde la base de datos, abstrayendo las operaciones de acceso a esta. Puede verse como una parte clave de la denominada capa de repositorios, quién proporciona métodos para realizar operaciones **CRUD** (Create, Read, Update and Delete) en la base de datos, como se ve a continuación.

Los repositorios consisten en una abstracción utilizada con el fin de acceder y manipular los datos de una base de datos y se implementan mediante interfaces. Se suele tener un repositorio por cada tabla o entidad de la base de datos. Las tecnologías utilizadas ofrecen anotaciones para identificar clases Kotlin como repositorios, además, al ser interfaces son capaces de extender otras predefinidas que ofrecen diferentes métodos para facilitar las consultas a la base de datos. Un ejemplo son los que ofrece **JPA** (`JpaRepository`), como es el caso de `findAll()`, que permite recuperar todos los registros de una entidad de la base de datos. Además, se pueden declarar métodos personalizados sin necesidad de implementación, ya sea mediante una sintaxis basada en los atributos de la clase entidad o usando SQL, tal y como se muestra en el anexo C.2.2.

4.5 Pruebas unitarias y de integración

La fase de pruebas tiene un peso muy grande en el proyecto. Cada funcionalidad y cada método se ha sometido a una serie de pruebas antes de incorporarse a este, comprobando su correcto funcionamiento y verificando que no se introducen errores que afecten al resto de componentes. Las pruebas se dividen en dos tipos: unitarias y de integración. Los primeros, a diferencia de los segundos, validan una funcionalidad aislada sin interacción con otros módulos [46]. Un ejemplo de test de integración es que al validar una consulta GraphQL, se verifica el proceso ETL, los adaptadores y resolvers y los resultados de la consulta. Cada prueba se implementa en un método anotado con la anotación del marco de pruebas unitarias JUnit, **`@Test`**. Dentro de estas pruebas se encuentran aquellas que validan los **repositorios de datos**, donde se verifican los métodos del ORM JPA, otras que validan **las consultas y mutaciones GraphQL**, donde se verifican que los resultados son coherentes con el contenido de la base de datos y otras **pruebas secundarias** relacionadas con expresiones regulares, el formateo de fechas y transformaciones del proceso ETL. El entorno de pruebas es relativamente fácil de configurar tal y como se muestra en el anexo C.7.

4.6 Problemas encontrados y decisiones tomadas

En esta sección se explican las distintas decisiones y problemas encontrados a lo largo del proceso de implementación de los diferentes módulos.

Decisiones y problemas que han afectado a todos los módulos de la aplicación

Durante la implementación de la aplicación algunas de las tecnologías utilizadas han sufrido actualizaciones, como es el caso de Netflix DGS [50]. Dado que en la dependencia se había indicado que utilizase la última versión disponible, surgieron incompatibilidades con el resto de las dependencias que no se actualizaban automáticamente. La solución fue actualizar los plugin y dependencias afectadas en el Gradle de acuerdo con el manual de la nueva versión y evitar actualizar automáticamente la versión de dichas dependencias.

Otro de los contratiempos encontrados tiene que ver con el tamaño máximo de memoria asignado al entorno de ejecución y es que a la hora de cargar los datasets del fichero CSV en

la base de datos la aplicación emitió el error “Java out of memory”. La solución ha sido configurar el fichero gradle.properties indicando el tamaño máximo de memoria con la siguiente instrucción **`kotlin.daemon.jvmargs=-Xmx6g`**. Donde “6” es la cantidad máxima de memoria que el programa puede utilizar para funcionar sin problemas.

Por último, hubo problemas en cuanto a la comunicación entre el cliente y el servidor debido a que CORS bloqueaba el tráfico. Con el fin de permitir las solicitudes de origen cruzado se implementó una clase de configuración tal y como se muestra en el anexo C.6.

Problemas y decisiones que han afectado al módulo cliente

En lo referido al cliente surgieron dos dificultades principalmente: por un lado, adaptar la sintaxis TypeScript utilizada, por ejemplo, en los componentes de MUI Material, a Kotlin JS y, por otro lado, mantener un estado compartido entre las dos páginas de la aplicación con el fin de manejar filtros seleccionados por el usuario u otros detalles. Esto se solucionó creando un contexto común (anexo C.4.3).

Problemas y decisiones que han afectado al módulo servidor GraphQL

En un primer enfoque del diseño de la base de datos se contempló utilizar una tabla para la clase padre (ResourceEntity) y otra por cada uno de los hijos. No obstante, debido a que la mayor parte de los atributos de las hijas son compartidos se decidió tener una única tabla tanto para el padre como para las hijas (single-table). Con ello se reduce la complejidad del esquema y el número de joins (uniones) para recuperar información, Además, al no haber duplicación de columnas, el almacenamiento es más eficiente. Por otro lado, a la hora de crear las entidades y definir sus relaciones se han fijado una serie de directrices:

- El sistema al arrancar por primera vez contiene una instancia de la entidad de catalog con identificador "root". Si se piensa en un sistema de ficheros, hace referencia a "/".
- Un dataset puede ser servido por varios data services. Se da en el mundo real, por ejemplo, muchos de los conjuntos de datos que publica el Centro Nacional de Información Geográfica en el Centro de Descargas [80] son utilizados por muchas autonomías para sus servicios de mapas.
- Sólo se puede deducir que un recurso es miembro de un catalog si es tópico primario de un catalog record que se encuentra registrado en dicho catalog.
- Existe una operación de mutación que permite crear un catalog record y otros recursos relacionados a partir de una URL a un metadato externo o un fichero de metadatos.

En cuanto a las fuentes de datos utilizadas se detectaron inconsistencias como la mostrada en el Código 10, donde un mismo título aparece registrado con dos idiomas distintos. Por esta razón, la entidad que representa los títulos mantiene una relación muchos-a-muchos con la entidad que contiene los lenguajes. Además, a la hora de procesar los ficheros JSON se da el caso de que un mismo metadato puede aparecer como **`JSONArray`**, **`JSONObject`** o **`String`**, por lo que es necesario contemplar estos casos (Proceso de extracción).

```
{"dct:title":[
    {"language": "es",
      "value": "Abastecimiento. Contadores (municipal)"
    },
    {"language": "eu",
      "value": "Abastecimiento. Contadores (municipal)"
    }
  ]
}
```

Código 10 Inconsistencia en la propiedad `dct:title` de un fichero JSON

5 Validación

Una vez se ha implementado el modelo GraphQL y el resto de la aplicación es importante contrastar el resultado obtenido con el portal original. Para ello se deben ejecutar diferentes consultas existentes en este y comprobar si con la aplicación GraphQL es posible su ejecución. Además, se plantean otra serie de consultas que no son ofrecidas por el portal o son demasiado complejas para el mismo, pero que resultan relevantes.

5.1 Consultas que ofrece el portal a través de endpoints

El portal de datos ofrece una API con una serie de enlaces mediante los que recuperar distintos conjuntos de datos, pudiéndose aplicar filtros como puede ser recuperar los dataset emitidos por un publicador y ordenar los resultados en función del título. La principal desventaja de estos endpoints es que no ofrecen mucha flexibilidad en cuanto a los filtros que se pueden aplicar, pues, por ejemplo, al filtrar no es posible filtrar publicador y categoría a la vez o utilizando varios publicadores. A continuación, se presentan varias consultas GraphQL que es posible hacer a través de los endpoints expuestos por la API.

Obtener todos los datasets

El portal ofrece el método GET a la dirección **`catalog/dataset`** para obtener el conjunto de todos los dataset existentes, ordenados por alguno de sus campos y con paginación. Sin embargo, devuelve todos los campos disponibles, por lo que no es posible seleccionar solo aquellos deseados. Esto puede causar sobrecarga y hace necesario extraer los campos correspondientes del conjunto manualmente. Con GraphQL es, además, posible seleccionar solo los campos requeridos tal y como se muestra en el Código 11 y el resultado queda estructurado en un JSON mucho más simple (ver el anexo C.7). Además, a través de esta consulta es posible filtrar los datasets de acuerdo con diferentes campos, como puede ser título, publicador o fecha de creación entre otros y ordenarlos. Si se desean obtener otras propiedades que describen al dataset, se modifican los campos requeridos en la consulta.

```
query Datasets($filter:[MapInput!], $page: Int!){
    resourcesByFilter(filters:$filter, page: $page){
        ... on Dataset{
            id
            title
            publisher
            description
            distributions{
                format
            }
        }
    }
}
```

Código 11 Consulta GraphQL para obtener todos los datasets

Obtener datasets en función de la categoría y el publicador

Para llevar a cabo esta consulta el portal ofrece dos endpoints diferentes que deben lanzarse por separado:

`/catalog/dataset/publisher/{id}` y **`/catalog/dataset/theme/{id}`**.

El primero permite obtener los datasets en función del publicador con identificador dado en `{id}` y el segundo hace lo mismo, pero para la categoría dada. Aquí se puede ver como inconveniente que solo es posible filtrar por un publicador o categoría. Con GraphQL esto se consigue también a través de la consulta anterior (ver el Código 11), y solo es necesario invocarla una vez. En el parámetro de filtros se incluyen el publicador y la categoría, pudiéndose indicar varios de cada uno y obtiene aquellos datasets que o bien contienen las categorías indicadas o bien han sido emitidos por los publicadores dados (ver el Código 12).

```
{"filter":[
  {"key": "Categoría", "values": ["Hacienda","Deporte"]},
  {"key": "Publicador", "values": ["Ayuntamiento de Alcobendas"]},
  {"key": "Formato", "values": [".rss"]}
],
  "type": "dataset",
  "page":0
}
```

Código 12 Parámetros utilizados en la consulta para paginar y filtrar por categoría, formato y publicador

5.2 Otras consultas más complejas que no ofrece el portal

Obtener datasets en función del publicador y los servicios de datos que contienen los catálogos a los que pertenecen dichos datasets

La consulta planteada en el Código 13 filtra todos los recursos de forma que se obtengan solo los datasets con un publicador concreto y de ellos obtiene el id, los títulos y catálogos que los contiene junto con todos los recursos que contienen dichos catálogos del tipo `data_service`. De dichos servicios obtiene las distribuciones por las que son accedidos, en particular el id y el título. Además, obtiene también los formatos en los que se puede distribuir a partir de estas.

```
query resourcesByFilter($filters:[MapInput!], $type: String!, $page: Int!){
    resourcesByFilter(filters:$filters, type: $type, page: $page){
        ... on Dataset{
            id
            title
            distributions{
                format
            }
            inCatalog{
                id
                resources{
                    ... on DataService{
                        id
                        isAccessedBy{
                            id
                            title
                        }
                    }
                }
            }
        }
    }
}
```

Código 13 Consulta GraphQL a través de la que se accede a datasets, catalog y data services

Obtener todas las palabras claves de los datasets que sirve un publicador y localizar otros publicadores que sirvan datasets con alguna de esas palabras clave

En este caso habrá que efectuar dos consultas relativamente simples (ver el Código 14), pues GraphQL no permite anidamiento, es decir, se necesita una consulta que obtenga las palabras clave deseadas y otra que filtre los recursos por dichas palabras clave. Para probarlo se utiliza

el publicador con notación “EA0040819” correspondiente a “Agencia Estatal Boletín Oficial del Estado”. Las palabras clave de la primera consulta son fáciles de obtener, simplemente se extraen del JSON usando la siguiente ruta:

```
data.publisher.resources[*].keywords[*].literal
```

```
query Publishers{
  publisher(notation:"EA0040819", label:null){
    notation
    label
    resources(page:0){
      id
      ... on Dataset{
        keywords
      }
    }
  }
}
```

```
query Datasets($filter:[MapInput!], $type:String!,$page: Int!){
    resourcesByFilter  filters:$filter, type:$type,page: $page){
        ... on Dataset{
          id
          keywords
          publisher
    }
  }
}
```

Código 14 Consultas GraphQL que involucran recursos, publicadores y palabras clave

6 Conclusiones y resultados obtenidos

En este apartado aporta una serie de conclusiones extraídas del proyecto. Tanto análisis del resultado, lecciones aprendidas y valoración personal como posibles mejoras futuras.

6.1 Resultado del proyecto

Se ha conseguido mapear los modelos DCAT a un esquema GraphQL y en una base de datos. También se ha logrado crear una serie de consultas GraphQL para manipular y acceder a los datos volcados en la base de datos aprovechando el potencial y simplicidad de GraphQL. Además, los datos se logran cargar de una fuente de datos externa real, siendo posible estructurarlos para adaptarlos al modelo. Por último, se ha desarrollado un cliente capaz de enviar solicitudes GraphQL al servidor y presentarlos en una interfaz amigable al usuario, quien también es capaz de navegar por los datos y operar con ellos (filtros, ordenación...). En conclusión, se han utilizado una serie de tecnologías a priori desconocidas por mí, se han integrado bien con GraphQL y se han demostrado sus ventajas de las que se habla en este proyecto. En el anexo e se muestra el aspecto final de la interfaz desarrollada y en el anexo f la gestión del tiempo de acuerdo con las diferentes partes implicadas en este Trabajo de Fin de Grado.

6.2 Valoración personal

El proyecto es un gran punto de inflexión en cuanto a aprendizaje. Tecnologías desconocidas y contar únicamente con el apoyo de un director hace que sea algo diferente al resto de proyectos llevados a cabo durante la carrera, pese a que he cursado la rama de Sistemas de Información, donde se han aprendido muchas tecnologías y se han realizado varios proyectos FullStack . No obstante, gracias al apoyo del profesor acabas comprendiendo el proyecto como una serie de objetivos que con una buena organización y planificación es posible cumplir.

6.3 Lecciones aprendidas

La principal lección aprendida ha sido el seguir una buena estrategia de planificación y desarrollo y una buena organización del código. Al principio no se llevó a cabo la mejor estrategia desarrollo generando algún que otro cuello de botella, además, la existencia de ciertas herramientas como Detekt han permitido generar un código mucho más legible, que también ha ayudado a agilizar este proceso. Por último, el hecho de seguir una metodología basada en tests (se prueba cada funcionalidad antes de agregarla al proyecto) ha logrado simplificar y hacer más eficiente el desarrollo de la aplicación.

Desde mi punto de vista la mayor parte de las tecnologías al estar bien documentadas han sido fáciles de integrar y desarrollar, sin embargo, algunas como Apollo o Kotlin JS de las que no hay mucha documentación han llevado un poco más de tiempo de entender. Quizá el punto de mayor estancamiento se produjo a la hora de definir el esquema GraphQL, pues los modelos DCAT eran algo completamente nuevos para mí y su interpretación y modelado no fue lo suficientemente fluida como me habría gustado. Pese a ello, considero que, aunque la aplicación pueda tener ciertas limitaciones, cumple con los objetivos planteados y se ha obtenido un producto que satisface las expectativas.

6.4 Mejoras futuras

El proyecto es completo en cuanto al objetivo planteado, no obstante, tiene una serie de limitaciones que si se resolviesen darían lugar a un sistema más ambicioso. La primera de las limitaciones, y la principal, se relaciona con la forma en que se accede a los datos del portal y

se cargan en el sistema. Es cierto que el proceso actual es simple, pues ejecutando la mutación GraphQL e indicando la ruta del fichero local o remoto los datos se extraen y cargan en la base de datos, sin embargo una mejora sería tener un script que accediese a la fuente externa automáticamente e invocase este proceso. Con ello se lograría también tener los conjuntos de datos actualizados en la base de datos.

Por otro lado, el sistema se centra en el modelo DCAT y en particular, en los datos del portal del Gobierno de España. Pese a que el sistema se ha implementado de forma que se puedan agregar nuevos formatos y procesamientos de diferentes fuentes de datos, siempre existe la posibilidad de tener que revisar el sistema.

Por último, las consultas ofrecidas y ejecutadas en la aplicación son aquellas consideradas relevantes para entender las ventajas que tiene GraphQL, pero siempre podrían agregarse más facetas con el fin de enriquecer las opciones que se ofrecen al usuario.

Bibliografía

[1]: Martín, A. M. (24 de 01 de 2023). ¿Qué beneficios aporta el Open Data a las empresas? Obtenido de Caja Siete: https://www.cajasietecontunegocio.com/temas/legislacion/item/que-beneficios-aporta-el-open-data-a-las-empresas

[2]: La importancia de los datos. (s.f.). Obtenido de Ayuware: https://www.ayuware.es/blog/importancia-de-los-datos/

[3]: Aragón, G. d. (s.f.). Curso Open Data (DCAT). Obtenido de Aragón Open Data: https://opendata.aragon.es/static/public/campus/curso/html/3_dcat.html

[4]: Cagle, K. (03 de 09 de 2021). Why GraphQL Will Rewrite the Semantic Web. Obtenido de Tech Target: https://www.datasciencecentral.com/why-graphql-will-rewrite-the-semantic-web/

[5]: Castellón, D. d. (s.f.). ¿Qué son los datos abiertos? Obtenido de Datos Abiertos Diputación de Castellón: https://datosabiertos.dipcas.es/pages/opendata/?flg=es

[6]: Gobierno de España. (s.f.). Portal de datos abiertos del Gobierno de España. Obtenido de Iniciativa de datos abiertos del Gobierno de España: https://datos.gob.es/es/

[7]: ¿Qué es GraphQL? (08 de 01 de 2019). Obtenido de Red Hat: https://www.redhat.com/es/topics/api/what-is-GraphQL

[8]: Data Catalog Vocabulary (DCAT) - Version 2. (04 de 02 de 2020). Obtenido de W3C: https://www.w3.org/TR/vocab-dcat-2/

[9]: Data Catalog Vocabulary (DCAT) - Version 3. (07 de 03 de 2023). Obtenido de W3C: https://www.w3.org/TR/vocab-dcat-3/

[10]: Riart, I. (12 de 01 de 2023). ¿Qué es la web semántica y en qué consiste? Descubre hacia dónde va el SEO. Obtenido de Cyberclick: https://www.cyberclick.es/numerical-blog/que-es-la-web-semantica-y-en-que-consiste

[11]: Dresslar, P. (10 de 10 de 2019). Ready for GraphQL Sidebar: GraphQL vs. SPARQL? Obtenido de Medium: https://medium.com/@peterdresslar/ready-for-graphql-sidebar-graphql-vs-sparql-4f2eb5246c12

[12]: Farías, I. H. (25 de 10 de 2022). *Intro a Patrones de arquitectura de APIs: GraphQL y API REST.* Obtenido de Thoughtworks: https://thoughtworks-es.medium.com/intro-a-patrones-de-arquitectura-de-apis-GraphQL-y-api-rest-2d3f5350e37

[13]: Acosta, J. (29 de 06 de 2021). *GraphQL: Qué es y qué ventajas ofrece.* Obtenido de OpenWebinars: https://openwebinars.net/blog/GraphQL-que-es-y-que-ventajas-ofrece/

[14]: *What is GraphQL?* (s.f.). Obtenido de hygraph: https://hygraph.com/academy/what-is-graphql

[15]: Strawderman, P. (2023, 06 12). *Netflix/dgs-framework Public.* Retrieved from GitHub: https://github.com/Netflix/dgs-framework

[16]: Stubailo, S. (2023, 05 10). *Using nullability in GraphQL.* Retrieved from Apollo Blog: https://www.apolloGraphQL.com/blog/GraphQL/basics/using-nullability-in-GraphQL/

[17]: *Introduction to Apollo Kotlin*. (n.d.). Retrieved from Apollo DOCS: https://www.apolloGraphQL.com/docs/kotlin/

[18]: *Kotlin Multiplatform.* (2023, 06 09). Retrieved from Kotlin: https://kotlinlang.org/docs/multiplatform.html#full-stack-web-applications

[19]: *Kotlin Multiplatform: La mejor tecnología para desarrollar Apps*. (2021, 05 31). Retrieved from Zimaltec Soluciones: https://www.zimaltec.es/blog/desarrollo-app-a-medida-kotlinmultiplatform#:~:text=Kotlin%20Multiplatform%20brinda%20la%20oportunidad,coste%20de%20desarrollo%20mucho%20menor.

[20]: *DGS Framework*. (2023, 06 08). Retrieved from DGS: https://netflix.github.io/dgs/generating-code-from-schema/

[21]: Caules, C. Á. (2019, 12 03). *Java Adapter Pattern y su utilidad*. Retrieved from Arquitectura Java: https://www.arquitecturajava.com/java-adapter-pattern-y-su-utilidad/

[22]: Michaels, P. (2023, 01 21). *The Service / Repository Pattern*. Retrieved from The Long Walk: https://pmichaels.net/service-repository-pattern/#:~:text=What%20is%20the%20Service%20%2F%20Repository,var%20salesOrder%20%3D%20salesOrderService

[23]: *Extracción, transformación y carga de datos (ETL)*. (s.f.). Obtenido de Microsoft: https://learn.microsoft.com/es-es/azure/architecture/data-guide/relational-data/etl#:~:text=Extracción%2C%20transformación%20y%20carga%20(ETL)%20es%20una%20canalización%20de,almacén%20de%20datos%20de%20destino

[24]: *CÓDIGO DE IDIOMAS SEGÚN ISO 639-1 (2 letras)*. (s.f.). Obtenido de Mucattu: http://utils.mucattu.com/iso_639-1.html

[25]: *Identificación de los organismos públicos*. (s.f.). Obtenido de Gobierno de España: https://datos.gob.es/es/recurso/sector-publico/org/Organismo

[26]: *Lista completa de tipos MIME*. (s.f.). Obtenido de MDN: https://developer.mozilla.org/es/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types

[27]: KeepCoding, R. d. (24 de 08 de 2022). *Componentes en React*. Obtenido de keep Coding: https://keepcoding.io/blog/componentes-en-react/

[28]: *Qué es el DOM*. (18 de 12 de 2021). Obtenido de Desarrollo web: https://desarrolloweb.com/articulos/que-es-el-dom.html

[29]: *Comaparing the HashRouter and the BrowserRouter in React applications*. (19 de 07 de 2021). Obtenido de Wanago: https://wanago.io/2021/04/19/hashrouter-browserrouter-react/

[30]: *Getting Started with ReduxKotlin*. (s.f.). Obtenido de Redux Kotlin: https://reduxkotlin.org/introduction/getting-started

[31]: Alvarez, M. A. (02 de 03 de 2018). *Qué es Redux*. Obtenido de Desarrollo web: https://desarrolloweb.com/articulos/que-es-redux.html

[32]: *kotlin-mui* . (s.f.). Obtenido de Sonatype: https://central.sonatype.com/artifact/org.jetbrains.kotlin-wrappers/kotlin-mui/5.11.10-pre.557

[33]: *Kotlin MUI Showcase*. (s.f.). Obtenido de https://karakum-team.github.io/kotlin-mui-showcase/

[34]: Popoff, A. (14 de 06 de 2023). *kotlin-mui-showcase* . Obtenido de Github: https://github.com/karakum-team/kotlin-mui-showcase

[35]: *Move faster with intuitive React UI tools*. (s.f.). Obtenido de MUI: https://mui.com

[36]: *Gradle plugin configuration.* (s.f.). Obtenido de Apollo Docs: https://www.apollographql.com/docs/kotlin/advanced/plugin-configuration/

[37]: *Adding Custom Scalars*. (08 de 06 de 2023). Obtenido de DGS: https://netflix.github.io/dgs/scalars/

[38]: todys, D. p. (11 de 06 de 2022). *GraphQL en Springboot con Kotlin*. Obtenido de YouTube: https://www.youtube.com/watch?v=HcnFi2wYjnQ&t=1695s

[39]: Dodino, F. (15 de 06 de 2022). *eg-profesores-graphql-kotlin* . Obtenido de Github: https://github.com/uqbar-project/eg-profesores-graphql-kotlin/blob/master/src/main/kotlin/com/uqbar/profesores/graphql/URLScalar.kt

[40]: *Reglas de transformación del modelo E/R al modelo relacional*. (s.f.). Obtenido de Daumier Web and Programing: https://daumindex.wordpress.com/2013/03/18/reglas-de-transformacion-del-modelo-er-al-modelo-relacional-2/

[41]: Krajcik, A. (s.f.). *Get Started with Kotlin Multiplatform and Spring Boot Kotlin*. Obtenido de https://morioh.com/p/f18659194586

[42]: *Build a web application with React and Kotlin/JS — tutorial*. (16 de 06 de 2023). Obtenido de Kotlin: https://kotlinlang.org/docs/js-react.html#7f191a6b

[43]: Mathur, G. (15 de 06 de 2023). *How to Setup Webpack with React.js?* Obtenido de Knowledgehut: https://www.knowledgehut.com/blog/web-development/create-react-app-webpack

[44]: SebastianAigner. (10 de 08 de 2022). *kotlin-hands-on* . Obtenido de Github: https://github.com/kotlin-hands-on/jvm-js-fullstack/blob/final/build.gradle.kts

[45]: *Build a full-stack web app with Kotlin Multiplatform*. (16 de 06 de 2023). Obtenido de Kotlin: https://kotlinlang.org/docs/multiplatform-full-stack-app.html#edit-configuration https://kotlinlang.org/docs/js-ir-migration.html#make-boolean-properties-nullable-in-external-interfaces

[46]: Calle, N. R. (23 de 11 de 2021). *Ejemplos de Testing en Spring Boot*. Obtenido de Refactorizando: https://refactorizando.com/ejemplos-testing-spring-boot/

[47]: Webb, P., & Wilkinson, A. (s.f.). *Annotation Interface SpringBootTest*. Obtenido de Spring Docs: https://docs.spring.io/springboot/docs/current/api/org/springframework/boot/test/context/SpringBootTest.html

[48]: *Testing*. (08 de 06 de 2023). Obtenido de DGS: https://netflix.github.io/dgs/query-execution-testing/

[49]: *TrimIndent*. (s.f.). Obtenido de Kotlin: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/trim-indent.html

[50]: *Netflix/dgs-framework*. (s.f.). Obtenido de Github: https://github.com/Netflix/dgs-framework/releases

[51]: *Check out the source code of the Portal*. (03 de 01 de 2023).Obtenido de data.europa.eu - The official portal for European data: https://data.europa.eu/en/publications/datastories/check-out-source-code-portal

[52]: *Understanding the European Data Portal High level presentation of the architecture.* (s.f.). Obtenido de European Data Portal: https://data.europa.eu/sites/default/files/edp_factsheet_portal_architecture_online.pdf

[53]: *Tecnología.* (s.f.). Obtenido de Gobierno de España: https://datos.gob.es/es/tecnologia

[54]: Calderón, N. (07 de 03 de 2021). *Entendiendo a la arquitectura limpia.* Obtenido de Medium: https://nescalro.medium.com/entendiendo-a-la-arquitectura-limpia-7877ad3a0a47

[55]: *DCAT-AP, perfil de aplicación de DCAT para portales Open Data Europeos.* (03 de 12 de 2019). Obtenido de Datos.gob.es: https://datos.gob.es/es/documentacion/dcat-ap-perfil-de-aplicacion-de-dcat-para-portales-open-data-europeos

[56]: *DCAT-AP y sus extensiones: Contexto y evolución.* (23 de 02 de 2018). Obtenido de Datos.gob.es: https://datos.gob.es/es/documentacion/dcat-ap-y-sus-extensiones-contexto-y-evolucion

[57]: *Perfil de aplicación de DCAT para portales de datos europeos.* (s.f.). Obtenido de PAe: https://administracionelectronica.gob.es/ctt/verPestanaGeneral.htm?idIniciativa=dcatap

[58]: Woltmann, S. (18 de 01 de 2023). *HEXAGONAL ARCHITECTURE - WHAT IS IT? WHY SHOULD YOU USE IT?* Obtenido de Happy Coders: https://www.happycoders.eu/software-craftsmanship/hexagonal-architecture/

[59]: Group, R. W. (25 de 02 de 2014). *RDF.* Obtenido de W3C Semantic Web: https://www.w3.org/RDF/

[60]: *Punto SPARQL.* (s.f.). Obtenido de Gobierno de España: https://datos.gob.es/es/sparql

[61]: *The world's leading open source data management system* . (s.f.). Obtenido de CKAN: https://ckan.org

[62]: *Drupal 10.1.* (s.f.). Obtenido de Drupal: https://www.drupal.org

[63]: *What is FME?* (s.f.). Obtenido de FME: https://docs.safe.com/fme/html/FME-Form-Documentation/FME-Form/Workbench/What_is_FME.htm

[64]: *Qué es Piwik.* (s.f.). Obtenido de Arimetrics: https://www.arimetrics.com/glosario-digital/piwik

[65]: *Metadata quality* (s.f.). Obtenido de data.europa.eu: https://data.europa.eu/mqa/methodology?locale=es

[66]: *What is a triple?* (s.f.). Obtenido de RDFOX: https://www.oxfordsemantic.tech/faqs/what-is-a-triple

[67]: Galvin. (12 de 08 de 2022). *The Semantic Web is Dead – Long Live the Semantic Web!* Obtenido de TerminusDB: https://terminusdb.com/blog/the-semantic-web-is-dead/

[68]: Cagle, K. (03 de 07 de 2016). *Why the Semantic Web Has Failed.* Obtenido de Linkedin: https://www.linkedin.com/pulse/why-semantic-web-has-failed-kurt-cagle/

[69]: Popoff, A. (08 de 08 de 2023). *Kotlin Wrappers.* Obtenido de Github: https://github.com/JetBrains/kotlin-wrappers

[70]: Muro, J. A. (s.f.). *¿Qué es un ORM?* Obtenido de Deloitte.: https://www2.deloitte.com/es/es/pages/technology/articles/que-es-orm.html

[71]: *¿En qué consiste la web semántica?* (22 de 12 de 2021). Obtenido de Digital Guide IONOS: https://www.ionos.es/digitalguide/online-marketing/marketing-para-motores-de-busqueda/web-semantica/#:~:text=La%20web%20semántica%20añade%20información,pueden%20tener%20el%20mismo%20significado

[72]: Lema, J. C. (30 de 05 de 2017). *Metadatos en Datos Abiertos.* Obtenido de Medium: https://medium.com/@jclema/metadatos-en-datos-abiertos-520a4685ab63#:~:text=DCAT%20es%20el%20estándar%20de,y%20serializar%20en%20diferentes%20formatos

[73]: *Catálogo de datos.* (s.f.). Obtenido de IBM: https://www.ibm.com/es-es/topics/data-catalog

[74]: *El portal oficial de datos europeos*. (s.f.). Obtenido de Data Europa: https://data.europa.eu/es/

[75]: Alvarez, M. A. (26 de 11 de 2016). *Qué es una SPA*. Obtenido de DesarrolloWeb: https://desarrolloweb.com/articulos/que-es-una-spa.html

[76]: *API REST: qué es y cuáles son sus ventajas en el desarrollo de proyectos*. (23 de 03 de 2016). Obtenido de BBVA API Market: https://www.bbvaapimarket.com/es/mundo-api/api-rest-que-es-y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos/#:~:text=Buscando%20una%20definición%20sencilla%2C%20REST,posibles%2C%20como%20XML%20y%20JSON

[77]: *RDF Graph Model Engine*. (s.f.). Obtenido de http://virtuoso.openlinksw.com/DAV/virtuoso2.openlinksw.com/content/rdf-quad-store.html

[78]: *How you design , align , and build matters. Do it together with Figma.* (s.f.). Obtenido de Figma: https://www.figma.com/

[79]: *Aragón Open Data*. (s.f.). Obtenido de Gobierno de Aragón: https://opendata.aragon.es/

[80]: Nacional, I. G. (s.f.). *Centro de Descargas.* Obtenido de Gobierno de España: https://centrodedescargas.cnig.es/CentroDescargas/catalogo.do?Serie=MAUT

[81]: *Disponible la 11ª edición del Informe del Sector Infomediario de ASEDIE.* (13 de 04 de 2023). Obtenido de Datos.gob.es Reutiliza la información pública: https://datos.gob.es/es/noticia/disponible-la-11a-edicion-del-informe-del-sector-infomediario-de-asedie

[82]: *EU Login - European Commission Authentication Service.* (s.f) Obtenido de European Commission: https://wikis.ec.europa.eu/display/NAITDOC/EU+Login+-+European+Commission+Authentication+Service

[83]: Rikki Schulte (08 de 08 de 2023). *GraphiQL*. Obtenido de *Github:* https://github.com/graphql/graphiql/tree/main/packages/graphiql

Anexo A Análisis detallado

En este anexo se explica en mayor detalle aspectos tratados en la sección de análisis que pueden resultar complejos con el fin de facilitar al lector su comprensión

A.1 Arquitectura de los portales de datos abiertos

En la figura 14 se muestra un esquema en alto nivel donde se han combinado, a partir de ambos portales, los componentes existentes. En primer lugar, hay dos formas por las que un usuario es capaz de acceder al portal: a través de una **API** o mediante una interfaz de usuario (**UI)**. Las distintas acciones y solicitudes, así como el acceso se gestionan a través de un **proxy**, que se encarga del enrutamiento a los diferentes servicios. En el caso del Gobierno de España se utiliza, además de un proxy inverso (encargado de administrar y optimizar el tráfico entrante hacia los servidores), un **varnish**, que no es más que un servidor proxy cuya función es almacenar en caché el contenido del sitio web y servirlo de forma eficiente, acelerando el rendimiento del sitio. La UI se basa en dos componentes: **CKAN** (Comprehensive Knowledge Archive Network) [61] y **DRUPAL** [62]. El primero hace referencia al catálogo de datos, el portal donde las organizaciones o gobiernos correspondientes publican los conjuntos de datos y se apoya en diferentes componentes que facilitan la visualización de estos. El segundo es el sistema de gestión de contenidos (CMS) utilizado para desarrollar y administrar el sitio web del portal. Proporciona la página de inicio con contenido editorial (por ejemplo, tweets o artículos) y otros enlaces. Las búsquedas de contenido se realizan mediante el motor de búsqueda **SOLR**, permitiendo indexaciones por separado tanto en DRUPAL como en CKAN. Los datos que conforman dicho contenido son recopilados de diferentes fuentes y API en distintos formatos. De ello se encarga el módulo **Harvester**, quién actúa como punto de entrada único para todos los metadatos, los transforma en el esquema JSON de CKAN y los carga en su repositorio. Este módulo se apoya en la plataforma para datos espaciales **FME** [63] y el componente **Gazetteer**. El primero ayuda en el proceso **ETL** (Extract, Transform, Load) [23], y el segundo en mejorar la calidad de los metadatos con información geoespacial mejorando la funcionalidad de búsqueda en el portal. Dichos metadatos se almacenan en un repositorio central, por ejemplo, PostgreSQL (usado en el portal de la UE y España) o MYSQL (usado en el portal de España), que, junto con CKAN, construyen el portal. Además, los metadatos se replican en otro repositorio de **Virtuoso Quad Store** [77], una plataforma de base de datos. Virtuoso ofrece soporte para datos enlazados (**Linked Data API**) y consultas SPARQL (**SPARQL manager**) no presentes en el primer repositorio. La arquitectura se completa con otros servicios que mejoran la calidad de los datos y la experiencia del usuario, como componentes que monitorizan el tráfico y las acciones de los usuarios (PIWIK [64] o MQA [65]), servicios de traducción u otros encargados de gestionar el registro o inicio de sesión de usuarios para acceder a funcionalidades extras, como es el caso del servicio de autenticación ECAS [82] (European Commission Authentication Service).

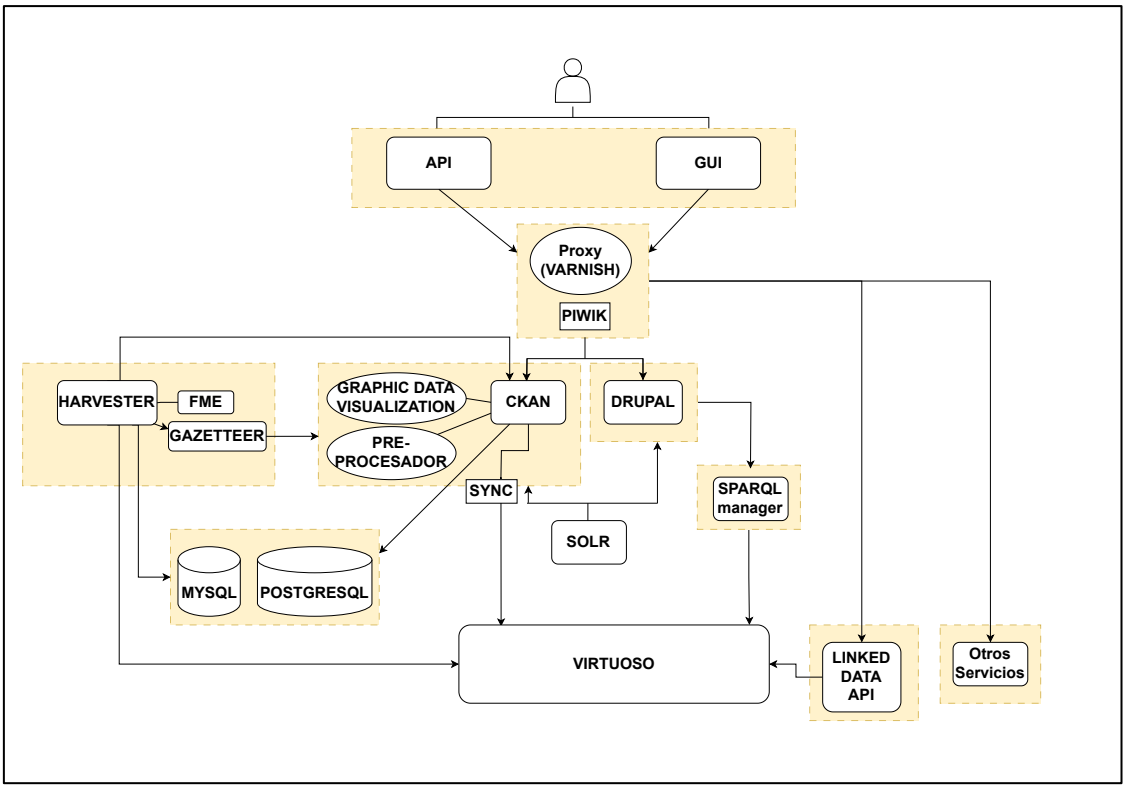


Figura 14 Diseño arquitectural del portal de datos de la UE y del Gobierno de España

A.2 DCAT en detalle

A continuación se muestra cómo se relacionan los principales recursos de DCAT y cuáles son los metadatos que interesan para este proyecto y que decisiones se han tomado para, posteriormente, definirlos en el esquema GraphQL y la base de datos.

A.2.1 El modelo DCAT al detalle

En las figuras 15 y 16 se muestran los modelos UML correspondientes a DCAT 2 y 3 respectivamente. Ambos tienen una estructura similar, no obstante en la versión 3 la estructura se ha simplificado, eliminando del diagrama las entidades foaf:Agent, skos:Concept y skos:ConceptSchema. Los agentes son entidades involucradas en la creación, mantenimiento o publicación de los conjuntos de datos, por otro lado, **SKOS** (Simple Knowledge Organization System) es un estándar para la representación de vocabularios controlados y conceptos y se utiliza para describir conceptos o términos utilizados en los metadatos. El esquema de conceptos SKOS proporciona un marco para organizar y estructurar los conceptos en un vocabulario. Otra diferencia es que la versión 2 describe los metadatos con el acrónimo DCT, mientras que en la 3 se lleva a cabo con DCTERMS, una extensión del anterior que incluye un conjunto más amplio de términos de metadatos que permiten describir recursos con más detalle. No todo son diferencias y es que ambas versiones comparten la mayoría de las entidades base, que son Resource, Catalog, DataService, Dataset, Distribution y CatalogRecord, sin embargo en la última versión se ha añadido la entidad DatasetSeries, que permite organizar conjuntos de datos relacionados, por ejemplo, bajo un tema o categoría común. Por último, existe la entidad Relationship, que permite describir conexiones específicas entre diferentes recursos DCAT, ya sean conjuntos de datos o catálogos, y adjuntar información adicional a esas relaciones.

En cuanto a la forma de relacionarse de estas entidades o recursos, se identifica una relación de generalización entre el recurso Resource y los recursos DataServices y Dataset, quien también tiene una relación de herencia con Catalog y con DatasetSeries (en la versión 3). En

cuando al Catalog, tiene relaciones con Resource, Catalog, DataService y Dataset, que indican los distintos tipos de recursos que contiene. Los catálogos también almacenan los CatalogRecord correspondientes, y tal y como se ha mencionado en la sección 4.6, sólo se puede deducir que un recurso es miembro de un catálogo si es tópico primario de un registro de catálogo que se encuentra registrado en dicho catálogo. El tópico primario se identifica como la relación primaryTopic que se da entre CatalogRecord y Resource y se entiende como el recurso DCAT descrito en el registro de catálogo. Para finalizar, queda explicar las relaciones que asocian Dataset, DataService y Distribution, más completas en la versión 3. Por un lado se tiene que un dataset puede tener varias distribuciones, lo que significa que puede haber múltiples versiones, formatos o medios a través de los cuales los usuarios pueden acceder a los datos (dcat:distribution). Hay que tener en cuenta que cada distribución está asociada con un conjunto de metadatos que describe cómo se puede acceder a los datos en esa distribución específica. Por otro lado, las distribuciones se relacionan con los servicios de datos (DataService) mediante dcat:accessService. Esta propiedad se utiliza para describir servicios que proporcionan acceso a los datos en una distribución particular. En lugar de simplemente proporcionar un enlace para descargar un archivo, un accessService podría ser un enlace a una API o una interfaz web. Por ejemplo, si un conjunto de datos contiene información geoespacial y una de las distribuciones está en formato GeoJSON, un accessService podría ser una API en línea que permite hacer consultas específicas en esa información geoespacial y obtener resultados personalizados según los criterios de búsqueda. Para finalizar, hay que mencionar que los recursos que sirve un servicio de datos se indican mediante la relación servesDataset.

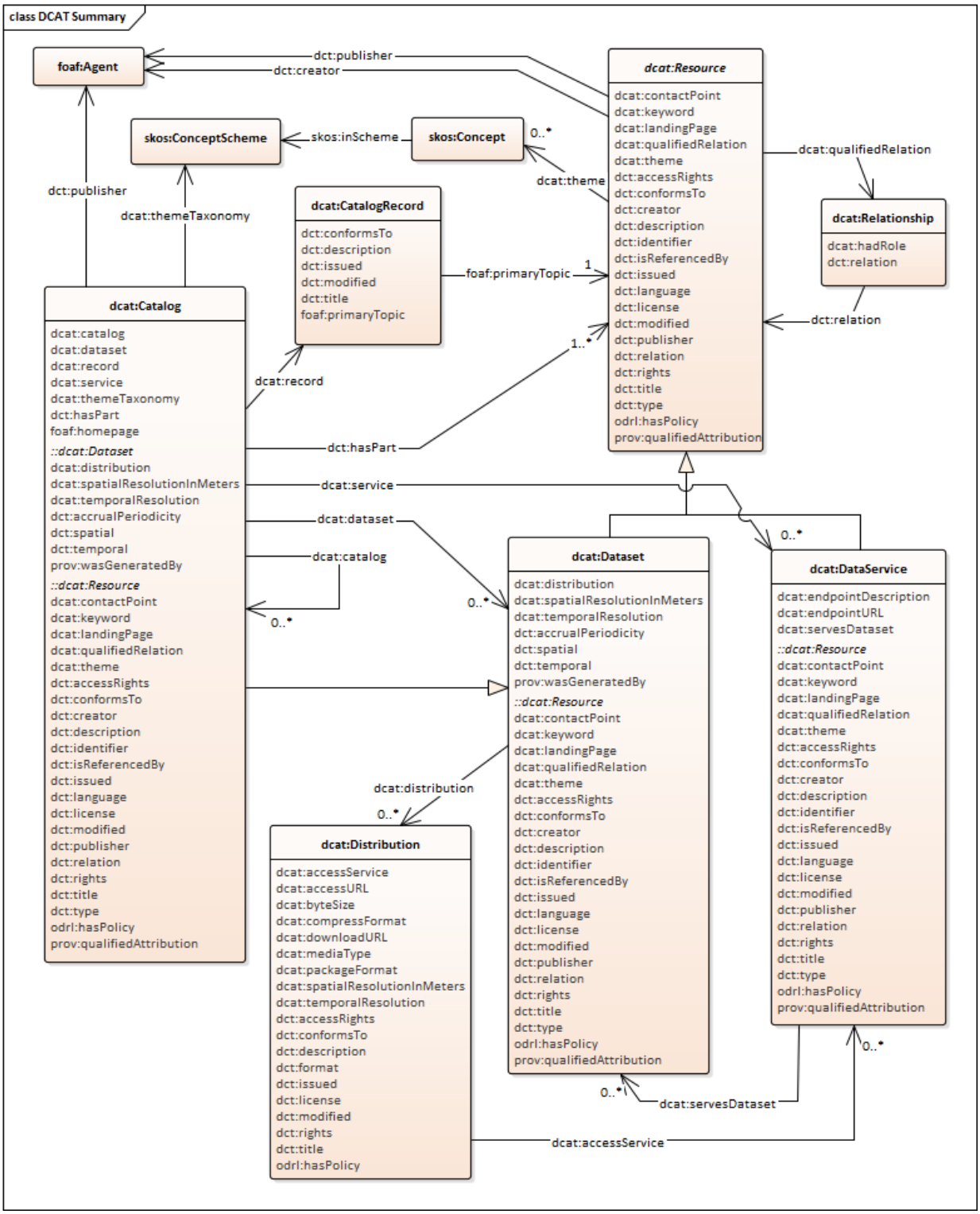


Figura 15 Modelo DCAT 2. Fuente: https://www.w3.org/TR/vocab-dcat-2/

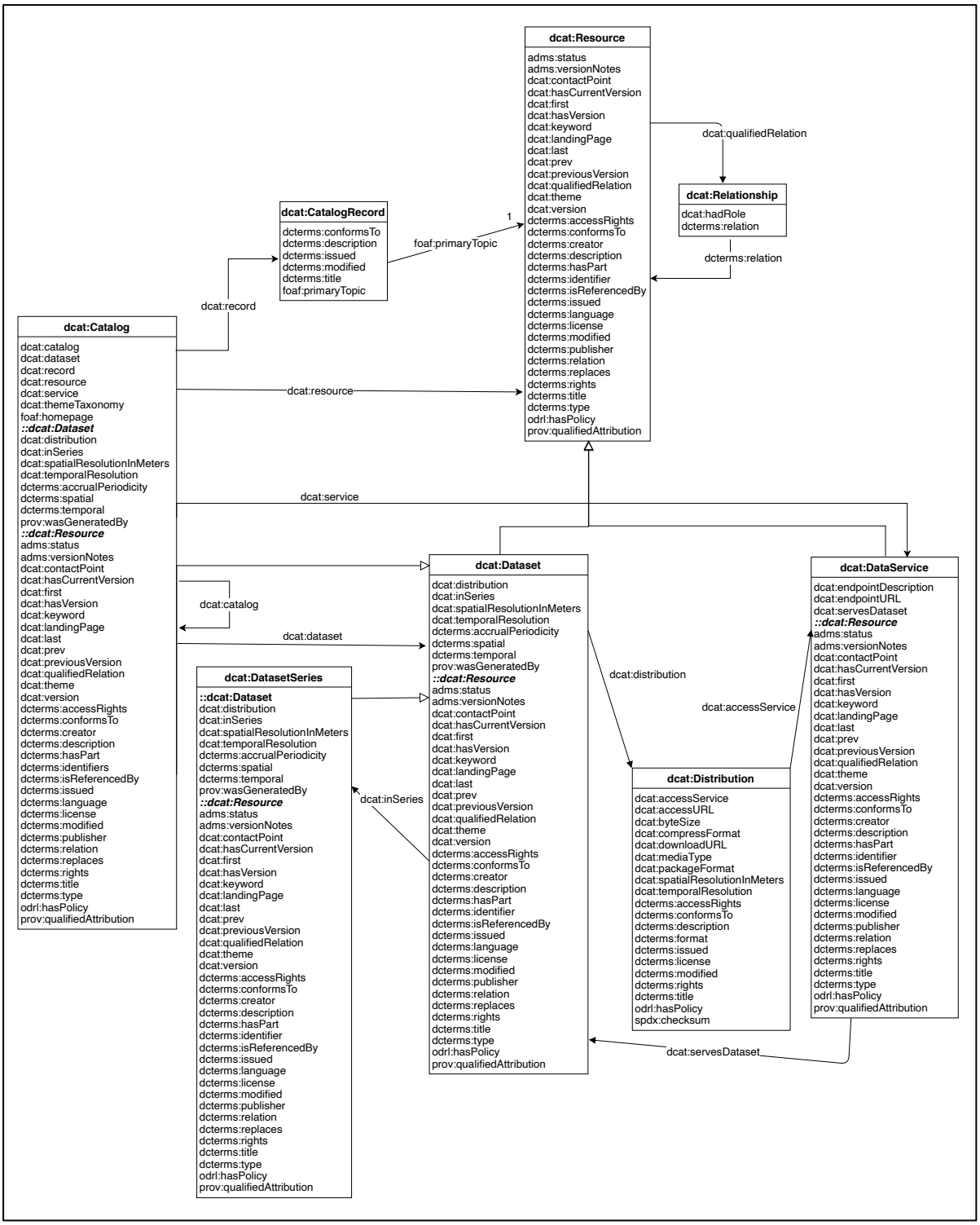


Figura 16 Modelo DCAT 3. Fuente: https://www.w3.org/TR/vocab-dcat-3/

A.2.2 Principales relaciones entre los recursos

Para comprender mejor como se relacionan los componentes presentados en la tabla 1 se ha diseñado el grafo de la figura 17. En él se representan cuatro **catalog record** (“CR1”,”CR2”,”CR3” y “CR4”), que describen respectivamente cuatro recursos: un **catalog** (“ROOT”), un **dataset series** (“DS1”), un **dataset** (“D1”) y un **data service** (“DSER1”). Junto a estos se incluye una **distribution** (“DIST1”).

Para entender las relaciones entre estos recursos se han utilizado enlaces (flechas) etiquetados con el nombre de la relación. Un catálogo almacena un conjunto de recursos (catalogs, datasets, dataset series o data services) y registros (catalog records). En este grafo, los registros son los cuatro catalog records mencionados y el nombre utilizado para la relación es **records**. Por otro lado, los recursos contenidos se dividen en dos grupos, uno que incluye los data services y otro que no. La razón se explica en la 3.1, pero es por una cuestión de organizar la información siguiendo el modelo DCAT, ya que un servicio de datos proporciona acceso a los recursos de tipo catalog, dataset y dataset series pero no data services. Por ello se identifican dos tipos de relaciones a las que se ha nombrado **resources** y **datasets**. Hablando de los servicios de datos (data services) en el grafo se identifica un tipo de relación a la que se ha nombrado **servesDataset**. Con ella se indican los recursos a los que proporciona acceso el servicio. Por último, queda de hablar de la distribución y el dataset. La distribución se relaciona con el servicio de datos mediante la relación etiquetada como **accessService**. Esta relación indica el servicio que da acceso a la distribución del dataset. Además, también existe la relación llamada **distributions**, para indicar el dataset que pone a disposición. Por último, los dataset se pueden agrupar en conjuntos denominados dataset series. Para representar el conjunto al que pertenece se ha creado la relación llamada **inSeries**.

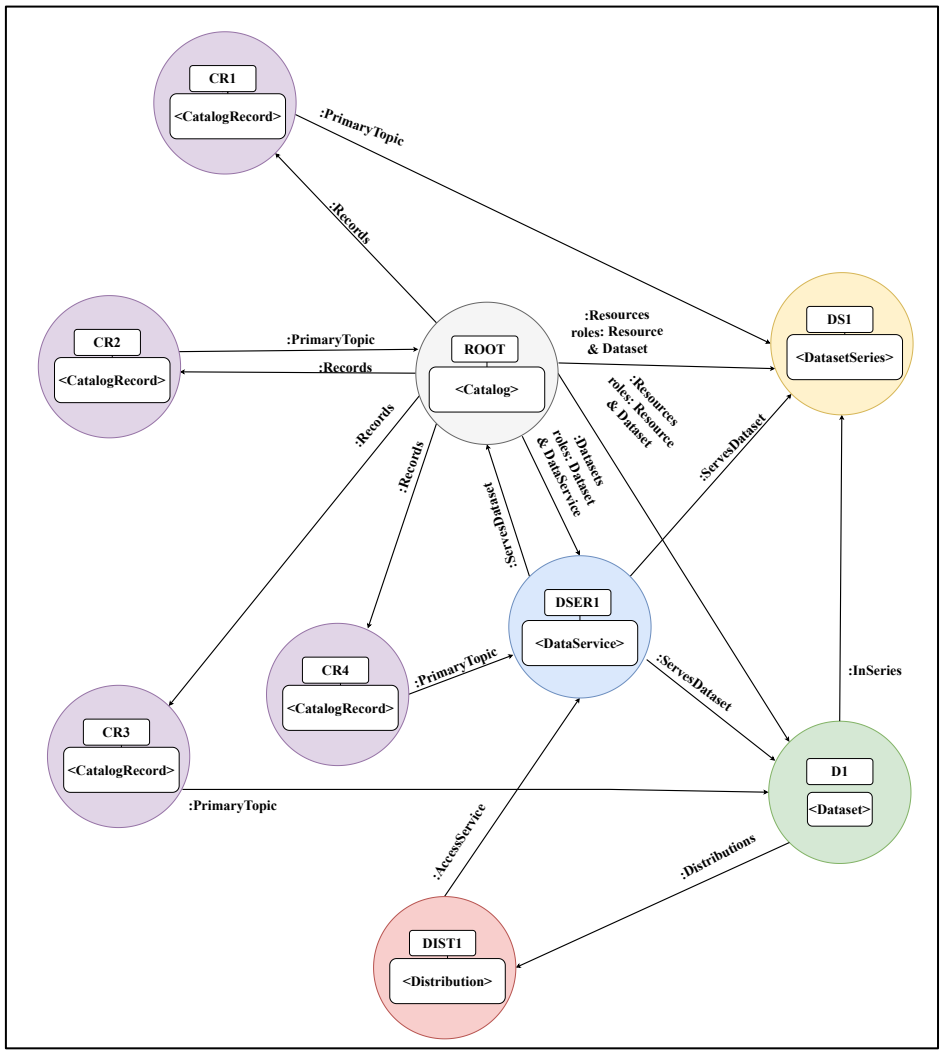


Figura 17 Grafo con relaciones y relaciones inversas entre los tipos GraphQL

A.2.3 Análisis de los metadatos

En este apartado se detallan las decisiones tomadas sobre los metadatos de los diferentes objetos de modelos DCAT 2 y 3 para adaptarlos al esquema GraphQL, explicando dichos metadatos.

Entidad Resource

Tabla 3 Mapeo de metadatos entre DCAT y el modelo GraphQL para la entidad "Resource"

DCAT2	DCAT3	GraphQL
dct:description (rdfs:Literal)	dcterms:description (rdfs:Literal)	description ([LangSpring!])
dct:title (rdfs:Literal)	dcterms:title (rdfs:Literal)	title ([LangSpring])
dct:language (dct:LinguisticSystem)	dcterms:language (dcterms:LinguisticSystem)	language ([String!])
dct:issued (rdfs:Literal)	dcterms:issued (rdfs:Literal)	issued (LocalDateTime!)
dct:modified (rdfs:Literal)	dcterms:modified (rdfs:Literal)	modified (LocalDateTime!)
dcat:theme (skos:Concept)	dcat:theme (skos:Concept)	theme ([String!])
dct:identifier (rdfs:Literal)	dcterms:identifier (rdfs:Literal)	identifier ([String!])
dct:publisher (foaf:Agent)	dcterms:publisher (foaf:Agent)	publisher (Concept)

1. **Description**. Proporciona una descripción textual de un recurso o catálogo, incluyendo, por ejemplo, detalles como el contenido o el contexto. En DCAT su rango es `rdfs:literal`, que representa un valor RDF dentro del rango de cadenas de texto, números, booleanos... En el esquema GraphQL se ha decidido declararlo como `LangString`, un escalar diseñado para albergar 2 cadenas de texto. La primera indica el contenido de la descripción y la segunda su idioma, siguiendo el código de idioma ISO 639-1 [24].

2. **Title**. Describe de forma clara y precisa en qué consiste el recurso. El mapeo, al igual que en el punto anterior, se ha realizado de `rdfs:literal` a `LangString`.

3. **Language**. Especifica el lenguaje en el que se presenta un recurso dentro de un catálogo de datos. En DCAT se le asigna el rango `dct:LinguisticSystem` (`dcterms` para DCAT 3) [9], que representa un sistema lingüístico empleado en un recurso o en un catálogo de datos. Por ejemplo, una URL que identifica el sistema lingüístico. En GraphQL se ha decidido emplear una colección de valores `String`, que siguen el código de idioma ISO 639-1 [24].

4. **Issued**. Indica la fecha en la que un dataset o recurso concreto se expidió o se hizo disponible. El rango empleado para este metadato es `rdfs:literal`, que como se ha mencionado representa también fechas y valores temporales, como horas o segundos. Para representar estos valores el rango sigue una codificación que utiliza la norma ISO 8601 y los tipos de datos XML Schema. Esto especifica que los valores fecha y hora deben codificarse como cadenas de texto de la siguiente forma: “YYYY” representa el año, “MM” el mes y “DD” el día. Para horas se emplea “HH” y así sucesivamente. Además, los valores se encuentran tipados en XML Schema, de forma que propiedades como `xsd:gYear`, `xsd:gYearMonth`, `xsd:date` y `xsd:dateTime` se utilizan para representarlos. En GraphQL se representa como un escalar del tipo `LocalDateTime`.

5. **Modified**. Indica la fecha en la que un dataset o recurso concreto se modificó por última vez. Los rangos y transformaciones son las mismas que las llevadas a cabo para la propiedad `issued`.

6. **Theme**. Especifica la categoría principal de un conjunto de datos. Su rango es `skos:Concept`, un estándar que representa y relaciona conceptos en un vocabulario controlado como una taxonomía. En este caso son conceptos que describen las diferentes categorías. Esta propiedad permite clasificar, de forma coherente, los dataset. Ejemplos de valores que pueden tomar son "Educación" o "Transporte". En el modelo se ha decidido representarlos como una colección de cadenas de texto.

7. **Identifier**. Asigna un identificador único a un recurso o dataset dentro de un catálogo de datos. El rango de dicha propiedad es `rdfs:literal`. Esta propiedad facilita la recuperación de recursos dentro de un catálogo de datos. Se ha decidido modelar como una colección de cadenas de texto.

8. **Publisher**. Indica la entidad u organización encargada de publicar un dataset. Se recomienda que tenga valores de tipo `foaf:Agent`, un vocabulario RDF que hace referencia a entidades que actúan como agentes, personas o instituciones. Se ha modelado como un escalar `Concept`, una estructura que almacena 2 cadenas de texto, de forma que se representa el agente publicador indicando su identificador (`skos:notation`) que identifica organismos públicos y su nombre (`skos:prefLabel`). Los valores de notation son predefinidos, por ejemplo "E00003801" hace referencia a "Ministerio del Interior" [25].

9. **License**. Especifica la licencia asociada a un recurso, que define los términos legales y de uso relacionados con el mismo. Su rango es `dcterms:LicenseDocument` (`dct` en DCAT 2). Puede tomar una URL que referencia un documento con la licencia o un contexto descriptivo. En GraphQL se ha modelado como una cadena de texto.

Entidad Dataset

Tabla 4 Mapeo de metadatos DCAT-GraphQL en entidad Dataset

DCAT2	DCAT3	GraphQL
dct:keyword (rdfs:Literal)	dcterms:keyword (rdfs:Literal)	keywords (LangString)
dct:spatial (dct:Location)	dcterms:spatial (dcterms:Location)	spatial ([String!])
dct:AccrualPeriodicity (dct:Frequency)	dcterms:AccrualPeriodicity (dct:Frequency)	accrualPeriodicity (Frequency)
dct:temporal (dct:periodOfTime)	dcterms:temporal (dcterms:periodOfTime)	temporal (PeriodOfTime)
-	-	validity (LocalDateTime)
dct:rights (dct:RightsStatement)	dcterms:rights (dcterms:RightsStatement)	regulations ([String!])
dct:relation	dcterms:relation	relatedResources ([String!])

1. **Keyword**. Representa una serie de términos que ayudan a clasificar y describir el contenido del recurso. El rango es de tipo `rdfs:literal` y se ha mapeado como `LangString`.

2. **Spatial**. Describe el ámbito espacial en el que se aplica el dataset. El rango asociado a esta propiedad es `dcterms:Location` (`dct` en DCAT 2). Este rango representa una ubicación geográfica y toma valores como latitud y longitud o nombre de la ubicación. En GraphQL se ha mapeado como una colección de cadenas de texto.

3. **AccrualPeriodicity**. Especifica la periodicidad con la que se actualiza un dataset, indicando la frecuencia con la que se realiza. El rango es `dcterms:Frequency` (`dct` en

DCAT 2) y representa la frecuencia de ocurrencia con valores como “Daily” o “Weekly”. En GraphQL se ha modelado mediante el escalar `Frequency`, una estructura de datos compuesta por dos campos: `range` como una cadena de texto y `period` como un número real. El primero representa el periodo, por ejemplo, “years” y el segundo la frecuencia dentro de este periodo. El valor “years 2.0” indica que se actualiza el dataset 2 veces al año.

4. **Temporal**. Describe el periodo de tiempo durante el cual un dataset es relevante o válido. El rango asociado es `dcterms:PeriodOfTime` (`dct` en DCAT 2) y representa intervalos de tiempo mediante propiedades como `dcterms:startDate` y `dcterms:endDate`. En GraphQL se ha modelado mediante el tipo `PeriodOfTime`, que contiene los campos `start` y `end` del tipo escalar `LocalDateTime`. En el fichero JSON se define así esta propiedad.

Tabla 5 Ejemplo propiedad temporal JSON. Fuente: https://datos.gob.es/es

dct:**temporal** <https://datos.gob.es/catalogo/a16003011-tablas-estadisticas-inventario-de-emisiones-de-contaminantes-a-la-atmosfera-de-la-c-a-del-pais-vasco-1990-20181/**PeriodOfTime**-1> ;
<https://datos.gob.es/catalogo/a16003011-tablas-estadisticas-inventario-de-emisiones-de-contaminantes-a-la-atmosfera-de-la-c-a-del-pais-vasco-1990-20181/PeriodOfTime-1> a dct:**PeriodOfTime**; schema:**endDate** "2020-12-31T00:00:00+01:00"^^xsd:dateTime; schema:**startDate** "2020-01-01T00:00:00+01:00"^^xsd:dateTime .

5. **Validity**. Representa la fecha de inicio de vigencia de un recurso. DCAT proporciona propiedades para describir aspectos temporales y ciclos de vida de los recursos, no obstante, no establece una lista específica a utilizarse, si no que recomienda adoptar estándares y prácticas comunitarias existentes que se adecúen al contexto de uso, de ahí que se adopte `dcterms:valid` para representar este metadato. En el modelo se ha utilizado como un tipo `LocalDateTime`.

6. **Regulations**. Describe la normativa asociada a un dataset. En DCAT no existe una propiedad específica para representar este campo, no obstante, tras estudiar las propiedades `odrl:hasPolicy`, `dcterms:accessRights` y `dcterms:rights` la que mejor se adecúa es esta última. Esta propiedad proporciona información acerca de los derechos legales y de propiedad intelectual ligados a un recurso. El rango es `dcterms:RightsStatement` y toma como valores una cadena de texto o algún tipo de URL que referencia a un recurso externo. Se ha modelado mediante el metadato `regulations` de forma que pueda aceptar una colección de cadenas de texto. Estos valores suelen ser URL o texto. El campo se identificó en la cabecera del fichero CSV descargado del portal del Gobierno de España [6].

7. **RelatedResources**. Representa todos los recursos con los que se relaciona el recurso correspondiente. Se identifica como una relación general entre recursos de ahí que se ha decidido emplear la propiedad `dcterms:relation` de DCAT (`dct` en DCAT2). Esta propiedad se utiliza para establecer relaciones generales entre un recurso catalogado y otros recursos relacionados. En DCAT no se especifica un rango para esta propiedad, pues depende del contexto en el que se utilice. En GraphQL se ha modelado como una colección de cadenas de texto. El campo se identificó en la cabecera del fichero CSV [6].

Entidad CatalogRecord

Tabla 6 Mapeo de metadatos DCAT-GraphQL en entidad CatalogRecord

DCAT2	DCAT3	GraphQL+DGS
dct:title (rdfs:Literal)	dcterms:title (rdfs:Literal)	title (String)
-	-	content (String)
-	-	contentType(String)
-	-	contentURL(String)
-	-	hints ([String!])

1. **Title**. Nombre dado al registro de catálogo. Su rango es `rdfs:literal`. En este caso solo se modela en el esquema GraphQL como una cadena de texto.

2. **Content**. En DCAT no existe ninguna propiedad relativa a este campo. Se ha decidido incluir en Catalog Record el metadato `content`, una cadena de texto que contiene, por ejemplo, el fichero JSON de un dataset. Esto resulta útil a la hora de implementar mutaciones para crear recursos.

3. **ContentType**. En DCAT no se define ninguna propiedad en relación con este campo. Se ha decidido que puede resultar importante almacenar como texto el formato del contenido almacenado en el campo `content`. Por ejemplo, si describe un recurso procesado en formato JSON, este valor será "application/json". Esto es útil para las mutaciones, en específico, para determinar cómo procesar la fuente de datos. Se ha modelado como una cadena de texto.

4. **ContentUrl**. En DCAT no se establece ninguna propiedad relativa a este campo para Catalog Record. Es una alternativa a `content`. En lugar de introducir el recurso como una cadena de texto, se introduce la URL que lo referencia. Se ha modelado como una cadena de texto.

5. **Hints**. En DCAT no se establece ninguna propiedad relativa a este campo. Representa una serie de sugerencias adicionales al `contentType` útiles para el proceso ETL. Se han modelado como una colección de cadenas de texto. Un ejemplo del valor que puede contener es el siguiente: ["datos.gob.es"].

Entidad Distribution

Tabla 7 Mapeo de metadatos DCAT-GraphQL en entidad Distribution

DCAT2	DCAT3	GraphQL+DGS
dct:title (rdfs:Literal)	dcterms:title (rdfs:Literal)	title ([LangSpring])
dct:accessUrl (rdfs:Resource)	dcterms:accessUrl (rdfs:Resource)	accessUrl (String)
dcat:byteSize(xsd:decimal)	dcat:byteSize (xsd:nonNegativeInteger)	byteSize (NonNegativeInt)
dct:format (dct:MediaTypeOrExtent)	dcterms:format (dcterms:MediaTypeOrExtent)	format (MediaType)
dct: identifier (rdfs:Literal)	dcterms: identifier (rdfs:Literal)	identifier (String)

1. **Title**. Proporciona un título o nombre a una distribución. Es igual al correspondiente para la entidad Resource.

2. **AccessUrl**. Se utiliza para proporcionar la URL de acceso a un recurso en un catálogo de datos. Su rango, `rdfs:Resource`, representa cualquier recurso identificable en la web, mediante una URL, por ejemplo. Se ha modelado como una cadena de texto.

3. **ByteSize**. Indica el tamaño en bytes de un recurso en su formato original. Su rango es `xsd:nonNegativeInteger` en DCAT 3 y `xsd:decimal` en DCAT 2, lo que indica que el valor asignado a esta propiedad debe ser un número entero no negativo y un decimal respectivamente. El modelo se ha diseñado mediante el escalar `NonNegativeInt` que contiene un valor del tipo número entero no negativo.

4. **Format**. Especifica el formato en el que una Distribution representa un recurso. El rango es `dcterms:MediaTypeOrExtent` (`dct` en DCAT 2). En GraphQL se ha considerado oportuno modelarlo como un tipo MediaType o MIME, cuyo formato es una cadena de texto compuesta por 2 partes separadas con '/'. La primera indica la categoría principal del medio y la segunda la variante específica dentro de dicha categoría. Por ejemplo: "text/plain" [26]. Se ha terminado utilizando un escalar que representa un tipo MIME mediante dos cadenas de texto: `type` y `subtype`.

5. **Identifier**. Consiste en una cadena de texto que se asigna al recurso para proporcionar una referencia inequívoca dentro de un contexto particular. Es igual al correspondiente para la entidad Resource.

A.3 Comunicación entre el cliente y el servidor GraphQL

A continuación, se explica que papel tiene GraphQL en el proyecto, a través de un esquema de secuencia (figura 18), en el que se simula una comunicación entre el cliente, servidor y la base de datos a la hora de ejecutar una consulta. Todo ello en muy alto nivel, para comprender los conceptos clave. En primer lugar, el cliente envía una solicitud al servidor en la que indica el tipo de operación que se desea ejecutar, pudiendo ser una consulta (query) o una mutación (mutation), los campos que se solicitan junto con los argumentos y/o variables opcionales si fuesen necesarias. A continuación, el servidor procesa la solicitud. Lleva a cabo un análisis y valida la consulta, verificando su semántica y estructura. Esto lo hace el motor de GraphQL automáticamente, y a través de los adapters localiza el campo solicitado. Estos se encargan de invocar a los resolvers, unas interfaces cuya implementación accede a la base de datos, si es necesario y devuelve el resultado sobre el campo requerido. Este debe estructurarse para poder devolverse de acuerdo con el esquema planteado. Finalmente se envía la respuesta al cliente.

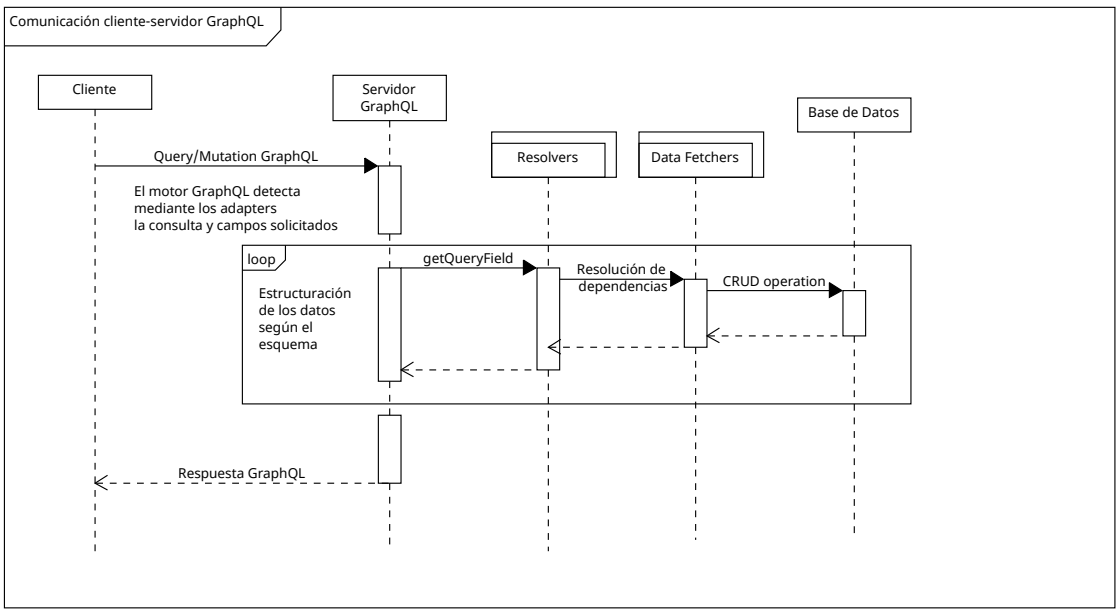


Figura 18 Diagrama de secuencia entre Cliente y Servidor GraphQL

A.4 Requisitos del sistema

En esta sección se muestran los requisitos funcionales y no funcionales que debe cumplir el sistema.

A.4.1 Requisitos funcionales

Los requisitos funcionales proporcionan una descripción clara de cómo debe comportarse el sistema ante la interacción con un usuario o a una situación en particular, describiendo las funciones del sistema. Para la definición de estos se ha estudiado principalmente la web de datos abiertos del Gobierno de España [6]. Se recogen en la tabla 8.

Tabla 8 Requisitos funcionales

Requisito	Nombre	Descripción
RF1	Página de listado	La aplicación debe ser capaz de mostrar la información resumida en forma de lista para cada recurso.
RF2	Página de información concreta	La aplicación debe ser capaz de mostrar para cada elemento de la lista información más exhaustiva del mismo en una página diferente.
RF3	Número de elementos	El usuario debe ser capaz de visualizar el número de elementos encontrados para cada recurso.
RF4	Menú	La aplicación permite visualizar un menú con los diferentes tipos de recursos, para poder navegar entre ellos y visualizar en listas sus elementos.
RF5	Navegabilidad	La aplicación debe contar con enlaces entre recursos y diferentes páginas relativas a los diferentes recursos tratados en el proyecto. De forma que se pueda navegar siguiendo las relaciones definidas en el modelo GraphQL.
RF6	Filtros	La aplicación debe contar con filtros de forma que el usuario pueda seleccionar el contenido de acuerdo con ciertos parámetros.
RF7	Filtros compartidos	La aplicación debe mantener los filtros de manera que desde la página en la que se muestra información concreta de un elemento pueda filtrarse según ciertos campos en la que se muestra el listado de elementos
RF8	Barra de búsqueda	La aplicación contará con una barra de búsqueda que facilitará la selección de los elementos mostrados en un listado.

RF9	Filtración con barra de búsqueda	Cuando el usuario introduzca un término en la barra de búsqueda se le mostrarán al instante elementos que cumplan dicho filtro.
RF10	Ordenación	El usuario debe ser capaz también de ordenar los resultados de búsqueda de acuerdo con ciertos parámetros.
RF11	Interfaz intuitiva	La interfaz de usuario debe ser intuitiva y fácil de utilizar.

A.4.2 Requisitos no funcionales

Los requisitos no funcionales describen las limitaciones del sistema en relación con el rendimiento, seguridad o usabilidad entre otros y no tienen ningún impacto en la funcionalidad de este. Se recogen en tabla 9 relacionándolos con las distintas tecnologías.

Tabla 9 Requisitos no funcionales

Requisito	Tecnología	Descripción
RNF1	GraphQL	El usuario debe ser capaz de seleccionar únicamente los campos requeridos a la hora de realizar consultas, evitando así sobrecargar el sistema.
RNF2	GraphQL	Debe limitarse el número de solicitudes a la API
RNF3	Kotlin Multiplatform	La aplicación no debe restringirse únicamente a tecnología nativa o web y debe poder ampliarse a otras plataformas de manera sencilla.
RNF4	Kotlin, Spring Boot, NetflixDGS, Apollo Kotlin.	Las tecnologías empleadas deben integrarse de forma sencilla y eficaz con GraphQL, facilitando la implementación de los aspectos de este.
RNF5	GraphQL y Apollo Kotlin	El cliente debe poder realizar solicitudes de forma independiente a cualquier cambio que se realice en el servidor.
RNF6	Netflix DGS	Generación automática del schema para simplificar la creación de los servicios y resolvers GraphQL.
RNF7	Netflix DGS	Implementación de consultas flexibles.
RNF8	Todas	El código debe estar bien estructurado, seguir buenas prácticas de programación y ser fácil de mantener y modificar en el futuro.

A.5 Fuentes de información

Los datos cargados en la base de datos se obtienen del portal de datos del Gobierno de España, a partir de una serie de ficheros descargables que ofrece el portal. Estos ficheros pueden ser de diversos formatos, pero, como se ha explicado, los formatos elegidos han sido JSON y CSV. A continuación se realiza un análisis de su contenido.

A.5.1 Fichero JSON

En esta subsección se analizan las distintas entidades o recursos que se pueden encontrar descritos mediante metadatos en un fichero JSON obtenido del portal de datos del Gobierno de España.

Entidad Distribution

En el Código 16 se muestra un objeto JSON que representa una distribución según `@type: dcat:Distribution`. Los metadatos que se identifican son `accessUrl`, `format`, `identifier` y `title`. Este último contiene el valor correspondiente al título y el lenguaje en el que se encuentra. Para el caso del formato, solo se especifica un id. Esto es debido al rango especificado para esta propiedad en DCAT (tabla 7). Para ver el valor correspondiente al formato hay que revisar las propiedades mostradas en el Código 15. El tipo del recurso es

`dct:IMT` lo que hace alusión a **Internet Media Type**. En este caso, el recurso representa el tipo de medio o formato empleado en la distribución con id e identificador, indicados en Código 16. Su valor es "application/json". La propiedad `rdfs:label` define una etiqueta descriptiva del recurso.

```
{
  "@id":
"https://datos.gob.es/catalogo/l01502973-campos-de-futbol/resource/e2dbfba7-2a21-4079-a07d-aaea45b84c26/format",
  "@type": "dct:IMT",
  "rdf:value": "text/csv",
  "rdfs:label": "CSV"
}
```

Código 15 Ejemplo declaración del recurso format en Json

```
{
  "@id": "https://datos.gob.es/catalogo/l01502973-campos-de-futbol/resource/07895a12-b2ab-4a94-b03a-7a7b00f74d5b",
  "@type": "dcat:Distribution",
  "dcat:accessURL":
"https://www.zaragoza.es/sede/servicio/equipamiento/basic/campos-de-futbol.json?srsname=utm30n_etrs89",
  "dct:format":{
    "@id":
"https://datos.gob.es/catalogo/l01502973-campos-de-futbol/resource/07895a12-b2ab-4a94-b03a-7a7b00f74d5b/format"
  },
  "dct:identifier": "https://www.zaragoza.es/sede/servicio/catalogo/formato/4242",
  "dct:title":{
    "@language": "es",
    "@value": "json"
  }
}
```

Código 16 Ejemplo declaración del recurso Distribution en Json

Entidad Dataset

En el Código 19 se muestra un ejemplo del recurso Dataset estructurado en un fichero JSON, donde se incluyen varias de sus propiedades, explicadas con detalle en las tablas 3 y 4. En primer lugar, se definen sus identificadores mediante las propiedades `@id` y `dct:identifier`. Por otro lado, se declaran las palabras clave o keywords, las descripciones y los títulos, indicando, además, el lenguaje asociado a cada uno. Otros metadatos son la categoría, representada por `dcat:theme`, las fechas de emisión (`issued`) y de última modificación (`modified`), definidas como `DateTime`, las localizaciones espaciales relativas al recurso (`spatial`), la licencia, declarada como una URL al portal correspondiente y los lenguajes, en este caso, únicamente español ("es"). Además de estos, se referencian a otros recursos a través de las propiedades `dct:accrualPeriodicity` y `dct:publisher`. En cuanto al publicador, se define como `skos:Concept`. En el ejemplo, la notación del recurso es "L01502973", lo que corresponde con el código asociado a la entidad pública cuyo nombre lo indica la propiedad `prefLabel`. Por otro lado, `AccrualPeriodicity` hace referencia a un recurso de tipo `Frequency`, que referencia a su vez a otro de tipo `DurationDescription` tal y como se muestra en el Código 18. El recurso de este tipo contiene los datos relativos a un periodo de tiempo, en este caso, anual. Para más información sobre estas propiedades se puede consultar el anexo A.2.3.

```json
{
  "@id": "http://datos.gob.es/recurso/sector-publico/org/Organismo/L01502973",
  "@type": "skos:Concept",
  "skos:notation": "L01502973",
  "skos:prefLabel": "Ayuntamiento de Zaragoza"
}
```

Código 17 Ejemplo declaración del recurso publisher en Json

```json
{
  "@id": "https://datos.gob.es/catalogo/l01502973-campos-de-futbol/Frequency",
  "@type": "dct:Frequency",
  "rdf:value":{
    "@id": "https://datos.gob.es/catalogo/l01502973-campos-de-futbol/DurationDescription"
  }
}
```

```json
{
  "@id": "https://datos.gob.es/catalogo/l01502973-campos-de-futbol/DurationDescription",
  "@type": "time:DurationDescription",
  "time:years": {
    "@type": "xsd:decimal",
    "@value": "1"
  }
}
```

Código 18 Ejemplo declaración del recurso Frequency y DurationDescription en Json

```
{
  "@id": "https://datos.gob.es/catalogo/l01502973-campos-de-futbol",
  "@type": "dcat:Dataset",
  "dcat:distribution":{
      "@id":
"https://datos.gob.es/catalogo/l01502973-campos-de-futbol/resource/e2dbfba7-2a21-4079-a07d-
aaea45b84c26"
  }
  "dcat:keyword":[
    {
      "@language": "es",
      "@value": "fÃºtbol"
    },
    {
      "@language": "es",
      "@value": "equipamientos"
    },
  ],
  "dcat:theme": {
    "@id": "http://datos.gob.es/kos/sector-publico/sector/deporte"
  },
  "dct:accrualPeriodicity": {
    "@id": "https://datos.gob.es/catalogo/l01502973-campos-de-futbol/Frequency"
  },
  "dct:identifier": "https://www.zaragoza.es/sede/servicio/catalogo/1040",
  "dct:issued": {
    "@type": "xsd:dateTime",
    "@value": "2014-01-16T00:00:00+01:00"
  },
  "dct:language": "es",
  "dct:publisher": {
    "@id": "http://datos.gob.es/recurso/sector-publico/org/Organismo/L01502973"
  },
  "dct:title": {
    "@language": "es",
    "@value": "Campos de FÃºtbol"
  }
}
```


Código 19 Ejemplo fichero JSON con Dataset

A.5.2 Fichero CSV

En el Código 20 se muestra parte de un fichero CSV extraído del portal de datos abiertos del Gobierno de España. Como se puede observar cada columna (en mayúsculas) representa uno o varios metadatos y cada fila corresponde con un dataset concreto, en este caso, por simplicidad solo se ha y una fila con un dataset y la cabecera con las columnas. Por ejemplo, "ETIQUETAS" hace referencia a 'keywords' en el modelo GraphQL, "FECHA DE CREACIÓN" a issued y "DISTRIBUCIONES" a los diferentes metadatos y propiedades propias de una entidad distribución. En caso de no existir valor para una columna concreta, como pasa, por ejemplo, con `NORMATIVA`, se coloca una cadena vacía entre las ',' (línea nueve de la imagen). En el caso de la propiedad "DISTRIBUCIONES" cada campo de dicha entidad se identifica mediante etiquetas, como ocurre con "[ACCESS_URL]" o "[MEDIA_TYPE]".


```
URL,IDENTIFICADOR,TÍTULO,DESCRIPCIÓN,TEMÁTICAS,ETIQUETAS,FECHA DE CREACIÓN,FECHA DE ÚLTIMA MODIFICACIÓN,FRECUENCIA DE ACTUALIZACIÓN,
IDIOMAS,ÓRGANO PUBLICADOR,CONDICIONES DE USO,COBERTURA GEOGRÁFICA,COBERTURA TEMPORAL,VIGENCIA DEL RECURSO,RECURSOS RELACIONADOS,
NORMATIVA,DISTRIBUCIONES
https://datos.gob.es/catalogo/abcb6be9-8e4c-48c0-9fc8-423cdaafddd7,,[es]Memoria de la Agencia Tributaria,
[es]La Agencia Estatal de Administración Tributaria|en|The State Tax Administration Agency",Hacienda,
[es]AEAT//Agencia Estatal de Administración Tributaria//Impuestos,2013-10-02T00:00:00+0200,2022-12-30T00:00:00+0100,
[TYPE]http://www.w3.org/2006/time#years[VALUE]1,es//ca//gl//en,Agencia Estatal de Administración Tributaria,
https://sede.agenciatributaria.gob.es/Sede/gobierno-abierto/reutilizacion-informacion/condiciones-reutilizacion.html,España,
2012-01-01T00:00:00+0100-2021-12-31T00:00:00+0100,2023-12-20T00:00:00+0100,,,
[TITLE_es]Memoria de la Agencia Tributaria 2021[ACCESS_URL]https://sede.agenciatributaria.gob.es/Sede/informacion-institucional/memorias/
memoria-2021.html[MEDIA_TYPE]HTML//[TITLE_es]Memoria de la Agencia Tributaria (todos los ejercicios)[ACCESS_URL]https://sede.agenciatributaria.gob.es/Sede/informacion-institucional/memorias.html[MEDIA_TYPE]HTML
```


Código 20 Cabecera y una línea del fichero CSV obtenido del portal de datos del Gobierno de España

Anexo B Visión detallada del diseño

Este apartado del anexo explica la aplicación desde un punto de vista en alto nivel. Se describe la estructura y módulos que la componen, los esquemas que rigen los modelos DCAT y el diseño de la base de datos, a partir del esquema entidad-relación.

B.1 Estructura de la aplicación: la arquitectura hexagonal

Como se ha explicado en la sección de diseño, la arquitectura planteada para la aplicación es la hexagonal. Su composición es un núcleo donde se encapsulan las funcionalidades y las reglas de negocio, una capa de adaptadores primarios y secundarios que reciben solicitudes del mundo exterior y las transforman en el formato comprensible por el núcleo o interactúan con servicios externos o dependencias, como bases de datos, respectivamente. Y entre medias una capa de puertos, que definen, mediante interfaces, cómo el núcleo de la aplicación interactúa con los adaptadores. En la figura 19 se identifica esta composición en este proyecto.

Para este proyecto se puede identificar como lógica del negocio los métodos resolvers (**ResolverServices**) y los encargados del proceso ETL (**HarvesterServices**), junto con otros servicios secundarios que apoyan a los anteriores (**SecondaryServices**) y las entidades de la base de datos (**DomainEntities**). Además, en la lógica se haya el esquema **GraphQL** y las posibles consultas que los adaptadores Netflix DGS y Apollo respectivamente usarán para generar el correspondiente código. Como puertos se pueden identificar las interfaces de los repositorios para gestionar el acceso a la base de datos (**PersistencePort**), interfaces de los servicios del proceso de harvesting (**HarvesterPort**) y las de los resolvers (**ResolversPort**). En lo referido a adaptadores se identifica por un lado el adaptador de persistencia para el acceso a la base de datos gestionada por Hibernate (**PersistenceAdapter**), los adaptadores correspondientes a Apollo Kotlin, identificado como **ApolloKotlinAdapter** y Netflix DGS (**NetflixDGSAdapter**). Este recibe las solicitudes GraphQL del cliente, ya sea mediante una interacción con la UI o a través de la API, y las dirige al componente correspondiente en la funcionalidad. Por último, se encontraría también el adaptador encargado de acceder a las fuentes de datos externas (**HarvesterAdapter**), aunque en una primera aproximación estas fuentes de datos serán ficheros ya descargados localmente.

El resultado no llega a ser una arquitectura hexagonal al 100% pues, como se ha mencionado, las entidades de la base de datos (**DomainEntities**) incluidas en el núcleo, se encuentran anotadas para indicar al ORM sobre qué tabla y columnas de la base de datos mapear la entidad y sus propiedades y relaciones, luego el núcleo no está aislado de detalles técnicos. Por tanto, surge el dilema de que no se deberían proveer dichas entidades con estas anotaciones técnicas en el núcleo, ni implementarla en el adaptador, pues el núcleo dejaría de tener acceso a ella. La mejor opción sería crear una clase con las anotaciones y sin lógica en el adaptador y mapear la clase de núcleo con esta [58], no obstante, no se ha implementado.

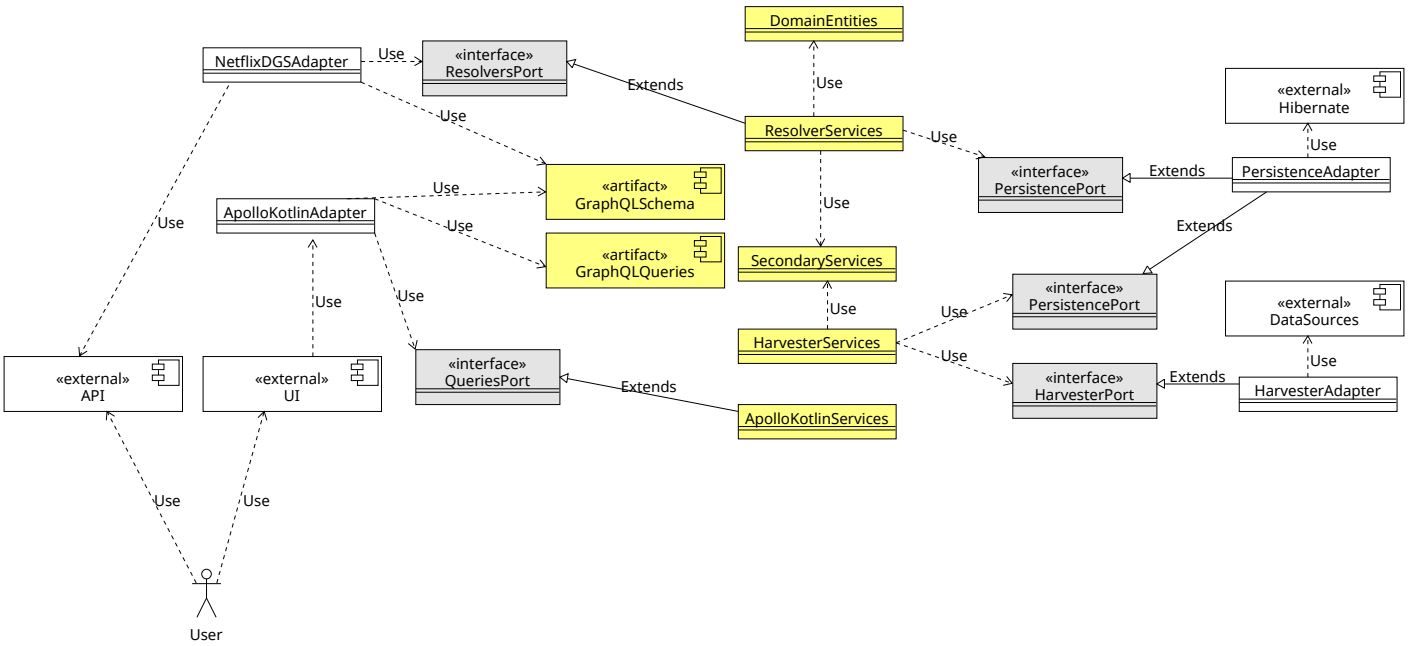


Figura 19 Diagrama de diseño de la aplicación

B.2 Mapeo DCAT al esquema GraphQL

Como se ha visto en el anexo A.2.1, existen dos relaciones de generalización o herencia. Por un lado, se identifica una relación de generalización entre el recurso Resource y los recursos DataServices y Dataset. Por otro lado, este último mantiene otra relación de herencia con Catalog y con DatasetSeries. Con ello se puede deducir una única generalización en la que un recurso Resource se especifica en Dataset, DatasetSeries, DataService y Catalog. No obstante, al comprobar las relaciones del recurso Catalog, se hace una separación en cuanto a los recursos que puede contener, distinguiendo entre Resource, Catalog, DataService y Dataset. Esto, unido al hecho de que un servicio de datos sirve recursos que no pueden ser DataServices, permite deducir un punto intermedio en la herencia que distinga entre recursos contenidos en el catálogo (resourcesInCatalog) y datasets contenidos en el catálogo (datasetsInCatalog), de forma que en este último no se incluya el DataService. Con esto se simplifican las consultas y la definición de los campos del esquema, como se puede apreciar en la definición GraphQL del recurso Catalog en el Código 21.

```
type Catalog implements ResourceInCatalog
& DatasetInCatalog & Resource{
  id: ID!
  title: [LangString!]
  language: [String!]
  isPrimaryTopicOf: [CatalogRecord!]
  isServedBy: [DataService!]
  inCatalog: [Catalog!]
  resources: [ResourceInCatalog!]
  datasets: [DatasetInCatalog!]
  records: [CatalogRecord!]
}
```

Código 21 Definición del recurso Catalog en GraphQL

B.3 Diseño de la Base de Datos

En este apartado se explica, en primer lugar, el diseño realizado de la estructura lógica de la base de datos mediante el esquema Entidad-Relación, donde se representan las relaciones entre las distintas entidades. A continuación, a través del esquema relacional, se describe como se ha implementado la estructura mencionada mediante tablas y relaciones en la base de datos.

B.3.1 Modelo Entidad-Relación

En la figura 20 se muestra el esquema entidad-relación con las entidades y atributos principales sin ningún tipo de transformación, como se explicará en el siguiente apartado. Lo primero que destaca es la relación de especialización que tiene la entidad **`ResourceEntity`** con las entidades que representan los recursos datasets, dataServices, catalogs y datasetSeries. Como en el esquema GraphQL (ver la sección 3.1 del documento) desarrollado a partir de DCAT se menciona otra especialización de los recursos en recursos en el catálogo y datasets en el catálogo, aquí se utiliza el discriminador de la relación `type`. Esta relación se da porque la mayor parte de los atributos de las entidades hijas son compartidos y, además, es una forma de organizar los recursos y facilitar su acceso. Forman por tanto una jerarquía exclusiva y total, pues no se contemplan registros de la entidad padre que no guarden relación con las hijas. A continuación, se diseñan las entidades **`CatalogRecordEntity`** y **`Distribution-Entity`**. La primera representa la clase catalog record, que describe recursos contenidos en un catálogo, por ello se forman dos relaciones. Una relación muchos-a-muchos con `CatalogEntity` pues un catalog record puede estar registrado en varios catalog y a su vez un catalog registrar varios de ellos (relación **`hasCatalogs`** que corresponde con `records` en el modelo GraphQL) y otra muchos-a-uno con `ResourceEntity`, pues cada catalog record registra un recurso, mediante la relación **`primaryTopic`**. Por otro lado, la entidad `DistributionEntity` representa la clase distribución del modelo GraphQL y forma una relación entre la entidad de los data services y la de los datasets, creando la tripleta **`DataServiceEntity`** - **`DistributionEntity`** – **`DatasetSeriesEntity`**. Para ello se utilizan las relaciones **`accessService`** y **`distributions`**. La primera especifica las distribuciones disponibles para el dataset, definiendo una relación muchos-a-muchos entre `DatasetEntity` y `DistributionEntity`. La segunda relaciona `DistributionEntity` con `DataService-Entity` e indica que el servicio de datos que proporciona acceso a la distribución del conjunto de datos. Para representar que un servicio proporciona acceso a los datasets en los catálogos se utiliza la relación **`servedDatasets`** entre la entidad **`DataServiceEntity`** y **`ResourceEntity`**, y dado que un servicio de datos no provee otros servicios de datos, esta relación no se dará para los recursos cuyo tipo sea dataservice. Para finalizar, los dataset pueden organizarse en series que comparten ciertas propiedades, por ello la relación **`inSeries`** indica el dataset series del que el dataset es parte creando una relación muchos-a-muchos entre las entidades correspondientes.

Como se puede apreciar, muchos atributos son multivaluados, como ocurre con **`languages`** o **`keywords`**. A la hora de crear las tablas de la base de datos van a sufrir una transformación que originará una nueva tabla, al igual que ocurre con las relaciones muchos-a-muchos. Esto se explica a continuación.

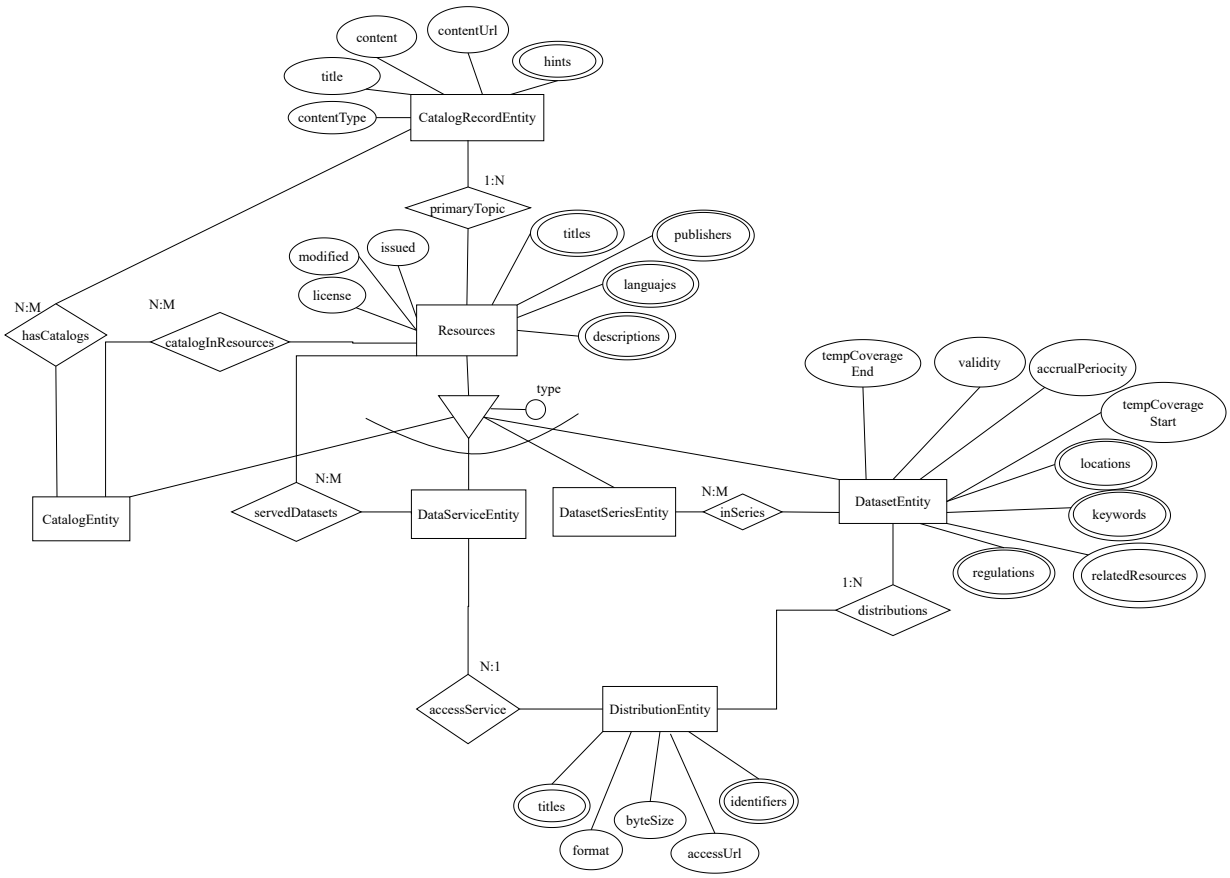


Figura 20 Esquema Entidad-Relación sin transformaciones y con atributos básicos

B.3.2 Modelo relacional

Como se puede ver en la figura 21, el modelo relacional representa la jerarquía de especialización de los recursos en diferentes tablas, cada una con sus propios atributos. Se ha representado así para visualizarlo mejor, pero a la hora de implementar la base de datos real, pese a considerar esta opción, se ha decidido mantener una única tabla con todos los atributos comunes y no comunes. Por ejemplo, contiene el atributo común `issued`, pero también el propio de los datasets, `accrual_periodicity`, cuyo valor será nulo para aquellos recursos que no sean de este tipo (`type`).

Otras transformaciones o decisiones tomadas han sido, por un lado, la creación de claves artificiales para las tablas que representan los resources, catalog records y distributions, y, por otro lado, la transformación de los atributos multivaluados, creando una nueva relación cuya clave primaria está compuesta por la clave primaria de la entidad origen y el atributo multivaluado [40]. Un ejemplo es el caso de los atributos `languages` y `keywords`. Del primero se origina la entidad **`LanguageEntity`** y del segundo, **KeywordEntity**, que mantiene una relación muchos-a-uno con `LanguageEntity`, de ahí que la clave primaria del language se incluya en la tabla de keywords como clave ajena.

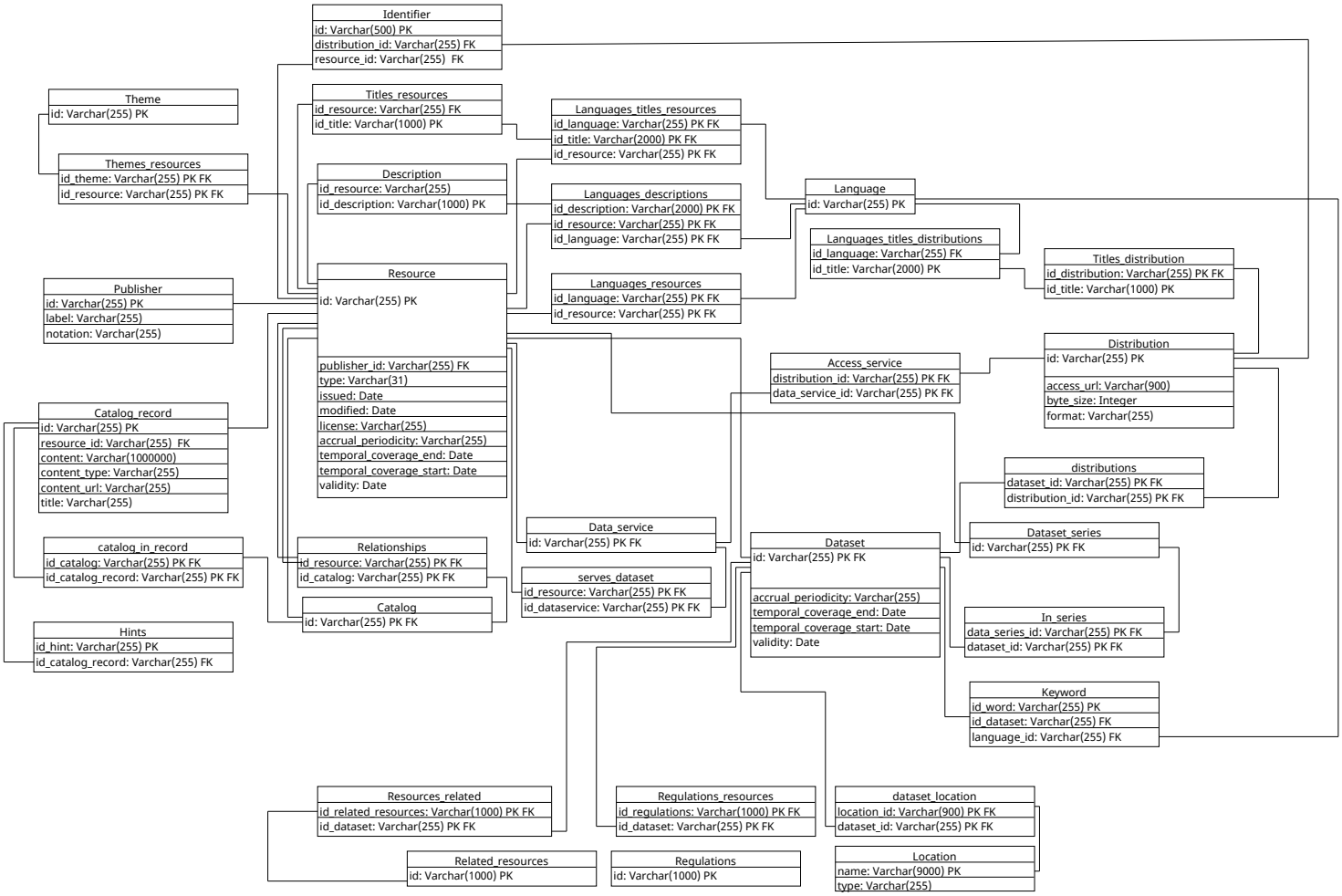


Figura 21 Esquema relacional completo

Anexo C Detalles de implementación

Este apartado del anexo detalla todos los aspectos relacionados con la fase de implementación del proyecto, desde ficheros de configuración, procesamiento de fuentes de datos e implementación de la base de datos hasta detalles técnicos relacionados con el cliente y las tecnologías utilizadas.

C.1 Kotlin Multiplatform

Kotlin Multiplatform permite compartir código entre diferentes plataformas de forma eficiente. Las plataformas que conforman la aplicación son: el lado del cliente (`JsMain`), el lado del servidor (`JvmMain`) y el código compartido en `CommonMain`. Además, dispone de soporte para pruebas en el entorno JVM (`JvmTest`). Dado que hasta que no se comenzó a implementar el cliente no se usaba este enfoque, fue necesario adaptar el fichero **build.gradle.kts** cuando comenzó a aplicarse, agregando el plugin correspondiente[5] y separando las dependencias en diferentes áreas (`dependencies`). A continuación, se debe definir la extensión `kotlin` con la configuración del **JVM** y del **JS**, donde se aplican los plugin correspondientes y se implementan diferentes tareas (`tasks`). Las dependencias para cada plataforma se definen en el contenedor `sourceSets` como muestra el Código 22.

```
sourceSets{
  val commonMain by getting{
  dependencies {}
  }
}
```

Código 22 Definición de dependencias en el bloque SourceSets del Gradle

La plataforma `CommonMain` contiene las clases, interfaces, funciones, objetos y demás elementos que no dependen de una plataforma en particular y se comparten entre todas. Otra plataforma es `JvmMain` y hace referencia al conjunto de elementos o fuentes específicas de la plataforma JVM, en este caso lo conforma la parte propia del servidor Kotlin-SpringBoot y NetflixDgs: base de datos, repositorios, servicios, adaptadores y elementos que usen la librería estándar de Java. Además, Kotlin Multiplatform proporciona soporte para pruebas unitarias y de integración en este entorno a través de `JvmTest`. Por último, `JsMain` hace referencia a la plataforma JavaScript y permite ejecutar código Kotlin directamente en un entorno de navegador web. Aquí se incluyen, por ejemplo, las funciones que interactúan con el DOM de la página web o las que utilizan las librerías de JavaScript. En el proyecto se incluye la parte Kotlin-React del cliente y llamadas Apollo Kotlin al servidor entre otros. Para poder lanzar el cliente se han definido dos tareas que configuran el webpack y el runner, mostradas en el Código 23.

```
tasks.getByName<Copy>("jvmProcessResources"){
    dependsOn(tasks.getByName("jsBrowserWebpack"))
}

tasks.getByName<JavaExec>("run"){
    classpath(tasks.getByName<Jar>("jvmJar"))
}
```

Código 23 Tareas del Gradle utilizadas en la configuración del webpack y el runner para lanzar el cliente

[5] `kotlin("multiplatform")`

La primera garantiza la coherencia entre los recursos generados y los necesarios para la ejecución en la plataforma JVM y el entorno del navegador. La tarea **jvmProcessResources** asegura que los recursos necesarios estén disponibles para la ejecución del código en JVM y **jsBrowserWebpack** se relaciona con el proceso de empaquetado y compilación de archivos en el entorno del navegador. La segunda permite que los **artefactos JS** (archivos resultantes de la compilación, empaquetado o transformación de código JavaScript y otros recursos relacionados) generados por jvmJar se puedan encontrar y servir. Para realizar la implementación y configuración correspondiente se han investigado varios enlaces y repositorios referenciados en la bibliografía: [41,42,43,44,45].

C.2 Detalles de implementación de la Base de Datos

A continuación se detalla cómo se lleva a cabo la configuración de los dos tipos de bases de datos que se han planteado utilizar en el proyecto, y aspectos técnicos sobre la implementación de las entidades y el ORM JPA.

C.2.1 Configuración

Esta sección realiza una explicación de las sentencias necesarias para configurar las bases de datos H2 de Hibernate y PostgreSQL en el fichero **application.properties**.

Fichero de configuración para H2

En el fichero se configura la conexión a la base de datos H2, se especifica como crear las tablas y como deben tratarse las entidades de la base de datos al arrancar la aplicación. Dado que H2 se ha utilizado para pruebas volátiles con datos no persistentes, es necesaria una configuración que permita cargar los datos cada vez que se lance la aplicación y destruya las tablas al finalizarla.

En el Código 24 se muestra el fichero de configuración con las siguientes sentencias:

- `server.port=8081:` Establece el puerto en el que el servidor ejecuta la aplicación, en este caso, 8081.
- `spring.sql.init.mode=always:` Especifica que los scripts SQL de inicialización que contienen, por ejemplo, sentencias para la inserción de datos, deben ejecutarse al iniciar la aplicación.
- `spring.h2.console.enabled=true:` Habilita la consola de administración de H2.
- `spring.h2.console.path=/h2:` Indica la ruta mediante la que se accede a la consola de administración de H2 a través del navegador.
- `spring.datasource.url=jdbc:h2:mem:memDb;DB_CLOSE_DELAY=-1:` Establece la URL de conexión a la base de datos H2. En este caso, se utiliza una base de datos en memoria denominada "**memDb**". El campo `DB_CLOSE_DELAY=-1` indica que la base de datos no se cerrará automáticamente cuando se cierre la conexión.
- `spring.datasource.driverClassName=org.h2.Driver:` Especifica la clase del controlador JDBC empleado en la conexión a la base de datos H2.

- `spring.datasource.username=sa:` Declara nombre de usuario utilizado para la autenticación.

- `spring.datasource.password=:` Declara, si existe, la contraseña de usuario utilizada para la autenticación.

- `spring.jpa.database-platform=org.hibernate.dialect.H2Dialect.:` Define el dialecto de base de datos que debe emplear Hibernate. En este caso, el de H2. Un dialecto en Hibernate es una implementación concreta de una serie de funciones o características propias de una base de datos. Por ejemplo, cada base de datos puede tener su propia sintaxis SQL, de forma que es importante que Hibernate las conozca para poder interactuar correctamente con ellas.

- `spring.jpa.properties.hibernate.globally_quoted_identifiers=true:` Declara que Hibernate debe tratar los elementos de la base de datos (columnas o nombres de tablas entre otros) con comillas dobles.

- `spring.jpa.properties.hibernate.format_sql=true:` Indica a Hibernate que formatee las consultas SQL generadas para que sean más legibles.

- `spring.jpa.hibernate.ddl-auto=update:` Declara el modo en el que es manejado el esquema de la base de datos por Hibernate al iniciar la aplicación. En este caso, se actualizará automáticamente. Si el esquema ye existe, Hibernate verifica que diferencias existen entre el esquema actual y las entidades definidas en el código y realiza las modificaciones necesarias para que se ajuste a la nueva estructura. Otros valores que pueden asignarse son: `create`, que crea desde cero en cada inicio el esquema, `create-drop`, que crea el esquema al iniciar la aplicación y lo elimina al finalizar, `validate`, que genera un error si hay discrepancias entre el esquema y las entidades definidas en el código y `none`, que desactiva cualquiera de las operaciones anteriores sobre el esquema.

- `logging.level.org.springframework.web=DEBUG, logging.level.org.hibernate=ERROR:` Especifican los niveles de registro (log) para los paquetes de la `aplicación correspondientes a springframework e hibernate. En` este caso, se establece el nivel **DEBUG** para el primero, de forma que se registran mensajes de depuración relacionados con el código de Spring Framework en relación con la web, y **ERROR** para el segundo, de forma que solo se registran mensajes de error en relación con Hibernate.

```
server.port=8081

spring.sql.init.mode=always
pring.h2.console.enabled=true
spring.h2.console.path=/h2

spring.datasource.url=jdbc:h2:mem:memDb;DB_CLOSE_DELAY=-1
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto = update

logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

Código 24 Ejemplo application.properties para H2

Fichero de configuración para PostgreSQL

Al igual que ocurre con H2, para poder utilizar PostgreSQL se necesita una configuración que establezca la conexión, credenciales o como debe tratar el esquema de la base de datos. En el Código 25 se muestra el fichero empleado en el proyecto.

- `spring.jpa.defer-datasource-initialization`: Si el valor asignado es `true`, esta propiedad pospone la inicialización de la fuente de datos hasta que se realice una operación que la requiera, como puede ser una consulta. Esto implica una reducción del tiempo de arranque y una mayor eficiencia de recursos.

- `spring.datasource.url:` Se especifica la URL de la base de datos a la que se conectará la aplicación. En este caso, la base de datos se denomina "postgres" y su tipo es PostgreSQL. Se accede a través de localhost en el puerto 54320.

- `spring.datasource.driver-class-name`: Declara el nombre de la clase del controlador (el driver) de la base de datos. En este caso, es el controlador de PostgreSQL.

- `spring.datasource.data`: Indica el script que debe ejecutarse tras inicializar la base de datos. En un primer momento se utilizó para verificar el correcto almacenamiento de los datos a través de operaciones de tipo INSERT contenidas en un fichero SQL

El resto de las propiedades se explican en el punto anterior.

```
server.port=8081

spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always

spring.datasource.url=jdbc:postgresql://localhost:54320/postgres
spring.datasource.username=postgres_tfg
spring.datasource.password=
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.data=classpath:data.sql

logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

Código 25 Ejemplo application.properties para PostgreSQL

C.2.2 Entidades y repositorios

Las entidades de la base de datos cuyo diseño se detalla en el anexo B.3 se implementan utilizando las anotaciones propias de Spring Framework en código Kotlin. Para poder operar sobre los datos almacenados, JPA dispone de varios métodos predefinidos y facilita la implementación y declaración de otros acordes a la base de datos.

Implementación de entidades y relaciones de base de datos

Para indicar que una clase Kotlin se comporta como una entidad de la base de datos se utiliza anotación `@Entity` lo que indica, además, que se deberá mapear a una tabla de la base de datos (`@Table`). El paquete utilizado es Jakarta, que forma parte de la especificación JPA y se encarga del mapeo objeto-relacional, a través de una serie de anotaciones.

Un ejemplo de entidad es la mostrada en el Código 26. Esta entidad es ResourceEntity, la clase padre de las entidades específicas de datasets, dataset series, data services y catalogs. Dado que es una superclase se implementan como `open` tanto la clase como sus métodos y, además, se establece la columna que actúa como discriminador en la jerarquía a través de la anotación `@DiscriminatorColumn`. En las entidades hijas, por ejemplo, en los datasets, se asigna el valor para el discriminador: `@DiscriminatorValue("dataset")`. Las columnas (`@Column`) de la entidad se corresponden con las propiedades del tipo GraphQL y dentro de la clase Kotlin se implementan también las relaciones, de las que se identifican 3 tipos:

Tabla 10 Anotaciones para implementar relaciones entre entidades JPA

Anotación	Descripción
@OneToMany	Relación uno a muchos. Mediante mappedBy se especifica el nombre del atributo en la entidad relacionada encargado de mapear la relación inversa. En ella no aparece este parámetro, en su lugar se utiliza la anotación @JoinColumn, aunque también puede utilizarse en esta (ver tabla 11).
@ManyToOne	Relación muchos a uno. Suele acompañarse con @JoinColumn (ver tabla 11).
@ManyToMany	Relación muchos a muchos. Mediante mappedBy se especifica el nombre del atributo en la entidad relacionada encargado de mapear la relación inversa, en la que esta anotación no contará con dicho atributo, si no que utilizará @JoinTable (ver tabla 11).

Otras anotaciones que suelen acompañar a las de la tabla 10 son las mostradas en la tabla 11:

Tabla 11 Anotaciones utilizadas en consonancia con las indicadas en Tabla 10

Anotación	Descripción
@JoinTable	Establece la tabla de unión que creada para almacenar la relación muchos a muchos.
@MapsId	Se utiliza para mapear una relación de muchos a uno y establecer una clave primaria compartida entre dos entidades.
@JoinColumn	Especifica la columna de unión para las relaciones de muchos a uno o uno a uno.

La clave primaria de la entidad se anota con `@Id`, aunque puede tratarse de una clave compuesta (es el caso de los atributos multivaluados) y se utiliza para ello `@EmbeddedId`. La variable que actúa como clave compuesta se implementa como una clase anotada con `@Embeddable` cuyas propiedades corresponden a las columnas que componen la clave primaria. Mediante `@MapsId` se declara la clave compartida en la relación con la entidad de la que se deriva el atributo en cuestión. En el Código 27 se ve un ejemplo.

```
@Entity
@DiscriminatorColumn(name = "tipo", discriminatorType = DiscriminatorType.STRING)
@Table(name = "resource")
open class ResourceEntity{
    @Id
    open lateinit var id: String

    @Column(name = "tipo", nullable = false, insertable = false, updatable = false)
    open var type: String? = null

    @OneToMany(mappedBy ="resource")
    open lateinit var identifiers  :Collection<IdentifierEntity>

    @ManyToMany(mappedBy ="resources")
    open lateinit var theme:MutableCollection<ThemeEntity>

    @ManyToMany
    @JoinTable(
        name = "relationships",
        joinColumns =[JoinColumn(name = "id_resource")],
        inverseJoinColumns =[JoinColumn(name = "id_catalog")],
    )
    open lateinit var catalogResources: MutableCollection<CatalogEntity>

    @ManyToOne
    @JoinColumn(name = "publisherId", referencedColumnName = "id", nullable = true)
    open var publisher: PublisherEntity? = null
}
```

Código 26 Ejemplo de @Entity

```
@EmbeddedId
lateinit var id: TitleResourceId
```

```
@Embeddable
class TitleResourceId: Serializable{
    @Column(name="id_title",nullable=false,length = 11000,columnDefinition = "varchar(11000) INDEX")
    lateinit var titleId:String
    @Column(name="id_resource",nullable=false)
    lateinit var resourceId:String
}
```

Código 27 Ejemplo de implementación de clave primaria compartida

Operaciones CRUD con métodos del ORM JPA

Los repositorios ofrecen una serie de métodos encargados de manipular y acceder a las bases de datos. Para implementar un repositorio de datos de una entidad concreta, como es el caso de **ResourceEntity** (Código 28), se utiliza la anotación `@Repository` sobre una interfaz que

extiende las librerías correspondientes. En el proyecto se utiliza **JpaRepository,** pero también puede utilizarse **CrudRepository**. La ventaja de la primera es que extiende a la segunda y ofrece operaciones más complejas.

En cuanto a los métodos que ofrece JpaRepository se encuentran los predefinidos como `findAll`, `findById`, `save` o `delete`, que no requieren ninguna implementación (aunque pueden sobrescribirse). Pero también es posible definir nuevos a partir de una consulta SQL, mediante la anotación `@Query` y `@Transactional`, que asegura la consistencia de los datos o crear métodos personalizados a partir de la sintaxis JPA y las propiedades de la clase entidad. Por ejemplo, `findDatasetByInseriesIdIn` localiza los dataset que pertenecen a una serie de datasetSeries (valor que se pasa como parámetro para que lo detecte la cláusula `In`). Para hacer uso de estos métodos simplemente se inyecta la dependencia del repositorio correspondiente y se invoca al método concreto. Otro ejemplo es el mostrado en la tabla 12, donde se consultan los elementos de la tabla `resource` cuya columna `id` es igual al valor de la variable `idS` y su columna `type` es diferente al indicado por `typeS`.

```
@Repository
interface ResourceRepository : JpaRepository<ResourceEntity, String>{
    fun findByIdAndTypeNot(idS: String, type: String): ResourceEntity?

    fun findServesDatasetByDatasetServiceId(id: String): Collection<ResourceEntity>

    @Query(
        value="select r.\"id\",r.\"issued\",r.\"modified\",r.\"tipo\",r.\"license\",
        r.\"accrual_periodicity\",r.\"temporal_coverage_end\",
        r.\"temporal_coverage_start\",r.\"validity\",r.\"publisher_id\" from \"description\" as des,
        \"resource\" as r where r.\"id\" =des.\"id_resource\" and
        r.\"tipo\" = 'dataset' and des.\"id_description\" IN :descriptions", nativeQuery = true)
    @Transactional
    fun findByDescriptionsIn(@Param("descriptions") descriptions: Collection<String>, page: Pageable): Page<DatasetEntity>

}
```

Código 28 Ejemplo @Repository JPA

Tabla 12 Ejemplo de operación CRUD en SQL y en JPA

`SELECT * FROM resource WHERE id = :idS AND type <> :typeS`
`fun findByIdAndTypeNot(idS: String, typeS: String): ResourceEntity?`

C.3 Visión detallada del proceso ETL

El proceso ETL se ha realizado sobre ficheros CSV y cargando los datos en una base de datos PostgreSQL. No obstante, previamente se contemplaron otras opciones, que junto con la que se eligió finalmente, se detallan a continuación.

C.3.1 Primera aproximación

La versión final utiliza el formato CSV para extraer los diferentes metadatos y relaciones de los datasets contenidos en el portal de datos del Gobierno de España y volcarlos en una base de datos PostgreSQL. No obstante, previamente se barajaron otras opciones.

En una primera aproximación se utilizó como base de datos **H2** de Hibernate. Se trata de un motor de base de datos relacional, ligero, capaz de funcionar completamente en memoria que, al embeberse, no requiere de una configuración compleja (ver anexo C.2.1). Su estado puede comprobarse a través de una consola local (figura 22) verificando así que las tablas creadas y los datos almacenados son los correctos. Como inconveniente, hay que mencionar que la consola no permite realizar operaciones sobre los datos o las tablas si no es mediante

sentencias SQL. El acceso a la consola, aunque es configurable, se lleva a cabo por defecto mediante una URL con el siguiente formato: **http://localhost:<port>/h2**.

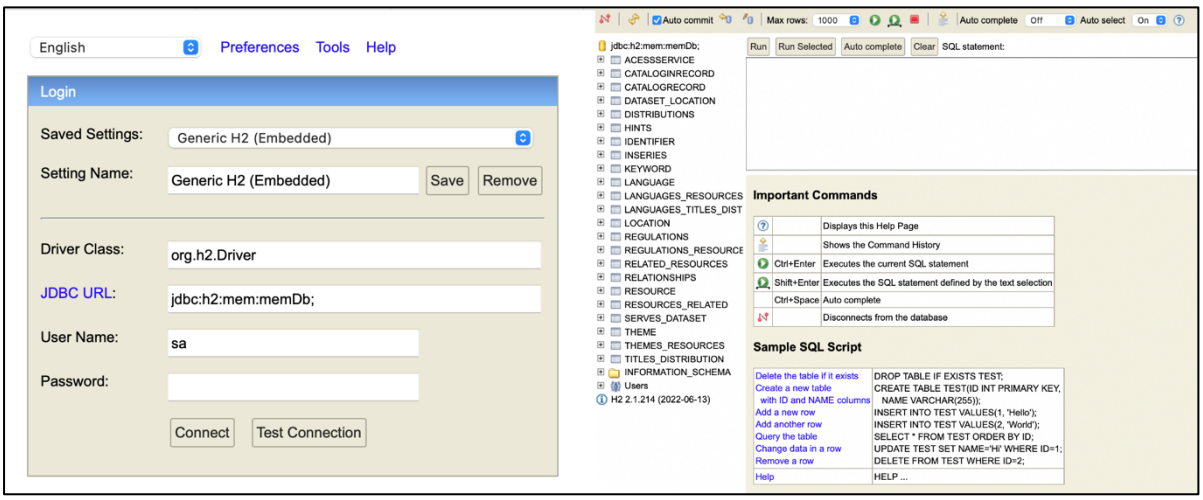


Figura 22 Consola H2 de Hibernate

En cuanto a las fuentes de donde se extraen los datos para, posteriormente, procesarlos y almacenarlos, se utilizaron datos ficticios predefinidos en un fichero SQL, donde, mediante sentencias de tipo `INSERT` se cargaban en la base de datos al arrancar el sistema, validando así que las entidades y las relaciones cumplen con los objetivos establecidos y siguen el modelo GraphQL planteado y además, de esta forma es posible validar, en un entorno controlado donde se conocen todos los datos existentes, que las consultas GraphQL son correctas. A continuación, se probaron con datos reales del portal, a pequeña escala, es decir, en vez de cargar todos los conjuntos existentes, como ocurre con CSV, solo se carga un dataset cada vez. Para ello se utilizó el formato JSON, cuyo proceso ETL se explica a continuación.

C.3.2 Procesamiento de ficheros JSON

Una primera aproximación para la carga de datasets reales fue utilizar los ficheros con formato JSON que se pueden descargar para cada dataset en el portal del Gobierno de España. En ellos se encuentran los distintos metadatos y relaciones que describen dicho conjunto. A continuación se explica el procesamiento que se les aplica para extraer aquellos elementos que resultan valiosos para este proyecto.

Proceso de extracción

Este proceso se encarga de extraer el contenido del fichero JSON que resulte útil de acuerdo con el modelo GraphQL. La tarea la llevan a cabo una serie de servicios representados con la anotación **`@Service.`**

El primer paso es cargar el contenido del fichero. Para ello se utiliza una interfaz que ofrece Spring a través del paquete **`org.springframework.core.io`**, denominada **`Resource Loader`**. A continuación, se lee el contenido en formato Bytes y se transforma a un objeto JSON. De este objeto interesan las propiedades **`@type`** (indican, por ejemplo, que ese objeto contiene los metadatos correspondientes a un dataset o una distribución) y con ellas se genera un mapa cuya clave es el objeto JSON correspondiente al tipo y el valor la propiedad del tipo mencionada (se muestra en el Código 29). Esto es así pues pueden repetirse varios valores de **`@type`**, como es el caso de las distribuciones.

El siguiente paso es procesar estos objetos JSON. Aquí conviene destacar que en ocasiones una propiedad puede aparecer como un objeto, como un String o como un array JSON. En

función del tipo, se debe realizar un procesamiento u otro, como se muestra en el Código 31, invocándose funciones cuya labor es la misma: extraer los metadatos o propiedades que interesen del recurso y almacenarlas en una estructura de datos, pero adaptándose a cada tipo de elemento. La lógica es simple: en una cláusula `WHEN` se comprueba el valor de la propiedad, como puede ser `theme`, siguiendo el ejemplo. En caso de ser un array se recorre y se extraen sus elementos, que son almacenados en una colección de cadenas de caracteres. En caso de ser un objeto no hace falta el bucle que lo recorre, simplemente se extrae el valor correspondiente, como muestra el Código 32. Puede darse también el caso de que el valor sea de tipo String, en tal caso se recupera el valor de la propiedad y se introduce en la colección del mapa directamente:

```
"notation" -> fields["Notation"] = fieldsId
```

Hay casos, en los que un mismo recurso o propiedad puede aparecer de diferentes formas, como es el caso de `dcat:theme`, reflejado en el Código 30, que se define como array y como objeto. También hay metadatos comunes o de igual nombre en varios recursos, es el caso de `@id`, común para **Dataset**, **Distribution**, **Concept**, **IMT** y **PeriodOfTime**. En estos casos, en una cláusula `WHEN` se identifica el tipo y se define la clave del mapa donde se guarda el valor como una colección. Además, si es un Dataset o una Distribution el valor de `@id` debe almacenarse también para la clave `identifier` de cada uno, tal cual se hace en el Código 33.

```
override fun getJSONArray(url: String): Map<JSONObject, String> =
    JSONObject(String(resourceLoader.getResource(url).inputStream.readAllBytes()))
        .getJSONArray("@graph")
        .toList()
        .map {
            o -> o to o["@type"] as String
        }
        .toMap()
```

Código 29 Función de carga del contenido del fichero JSON

```
"dcat:theme":[
  {
    "@id": "http://datos.gob.es/kos/sector-publico/sector/medio-rural-pesca"
  },
  {
    "@id": "http://datos.gob.es/kos/sector-publico/sector/economia"
  }
]
```

```
"dcat:theme": {
    "@id": "http://datos.gob.es/kos/sector-publico/sector/medio-ambiente"
}
```

Código 30 Ejemplo de una misma propiedad registrada como JSON Array y como JSON Object

```
fun processJsonElement(
    element: Any?,
    key: String,
    type: String,
    mapModel: MutableMap<String, Collection<String>>,
    rango: String?
):MutableMap<String, Collection<String>> = when (element){
        is JSONObject ->jsonModelServices.processObjectByKey(key,element,type,mapModel,rango)
        is JSONArray ->jsonModelServices.processArrayByKey(key,element,type,mapModel)
        is String ->jsonModelServices.processStringByKey(key,element,type,mapModel)
        else -> mutableMapOf()
}
```

Código 31 Función para procesar recursos JSON

```
"theme" ->{
    val fieldsTheme = mutableListOf<String>()
    for (i in 0 until array.length()) fieldsTheme.add(i, array.getJSONObject(i).getString("@id"))
    fields["Theme"] =fieldsTheme
}
"theme" -> fields["Theme"] = mutableListOf<String>(objeto.getString("@id"))
```

Código 32 Dos formas de procesar la propiedad dct:theme según sea Object o Array JSON

```
when (k){
    "@id"->{
        val key = when(type){
            "Dataset" ->"DatasetId"
            "Distribution" ->"DistributionId"
            "Concept" ->"PublisherId"
            "IMT" ->"ImtId"
            "PeriodOfTime" ->"PeriodOfTimeId"
            else ->null
        }
        if(key != null) fields[key] = fieldsId
        if(key == "DistributionId" || key == "DatasetId"){
            val identifiers = fields[key + "entifiers"]?.plus(fieldsId)
            fields[key + "entifiers"] = identifiers ?:fieldsId
        }
    }
```

Código 33 Ejemplo procesamiento campo '@id'

Proceso de transformación

El siguiente paso en el proceso ETL es estructurar los metadatos y propiedades extraídas con el fin de crear las instancias correspondientes que cargar en la base de datos Para esto se declaran una serie de clases de datos que albergarán los metadatos extraídos. Dado que estas clases deben tratarse como una misma, pues se quiere un mapa que contenga valores de metadatos asociados a una distribución, a un dataset, un publicador..., se define una interfaz que es implementado por dichas clases. De esta forma se obtiene una relación como la presentada en la figura 23. Aquí se ve la interfaz común ModelJsonMapping y las distintas estructuras de datos como puede ser `DatasetJsonMapping`, con metadatos propios de un dataset o `DistirbutionJsonMapping`, mostrado en el Código 34, con metadatos propios de una distribución. El resultado del procesamiento es un mapa compuesto por estos modelos

como clave y el tipo de cada uno como valor, pues, por ejemplo, pueden existir varios tipos **`Distribution`** con lo que el tipo no puede actuar como clave.

```
data class DistributionJsonMapping(
        val id: String,
        val accessUrl: String?,
        val titlesText:Collection<String?>,
        val titlesLang:Collection<String?>,
        val byteSize: Int?,
        val identifier:Collection<String?>,
        val format: String?,
    ):ModelJsonMapping
```

Código 34 Ejemplo modelo de datos para Distribution en procesamiento JSON

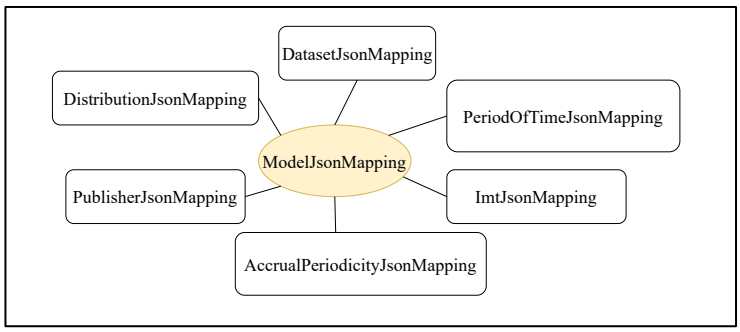


Figura 23 ModelJsonMapping diagrama

Proceso de carga

Durante el proceso de carga se recorre el mapa resultante del paso anterior extrayendo y transformando, si fuese necesario, para cada entidad los recursos y metadatos necesarios, con el fin de crear una instancia de ella y sus relaciones correspondientes mediante los métodos ofrecidos por JPA y los definidos en el módulo de repositorios (anexo C.2.2).

C.3.3 Procesamiento de ficheros CSV

En Kotlin existe una herramienta ofrecida por el usuario doyaaaaaken en GitHub que agiliza y simplifica el procesado de ficheros CSV. Se trata de la librería **kotlincsv.dsl.csvReader.** Esta librería ofrece la función `open(ips: InputStream)` para leer el contenido de un fichero pasado por parámetro como `InputStream`. Al igual que ocurre con los ficheros JSON, el fichero debe cargarse con el método `getResource` de la interfaz de Spring **ResourceLoader**. La ventaja de utilizar `csvReader` es que dentro de la función que lee el fichero, se puede estructurar su contenido en una clase de datos Kotlin en función de las columnas del CSV. En este caso la estructura es el objeto `DatasetCSVModel` y la función devuelve una lista con todos los elementos (ver el Código 35). Por cada línea del fichero CSV se crea el objeto **DatasetCSVModel** y se devuelve el conjunto de todos los modelos creados a través de una lista. Cada campo del CSV sufre una transformación para adaptarse al modelo y facilitar se recuperación en los servicios encargados de la carga en Base de Datos. Como ejemplo, de procesamiento se presenta el elemento "ETIQUETAS" del CSV, que representa el metadato keywords, mediante la función del Código 36. La secuencia de transformación se muestra en la tabla 13. Como se ve, se introducen palabras clave ("[[SPLIT_LAST]]") para identificar los campos y almacenarlos finalmente en un mapa clave-valor (previamente separadas por "[[SPLIT_LAST]]").

Tabla 13 Pasos en el procesamiento de metadatos del fichero CSV mediante etiquetas

Paso	Transformación
1	[ca]2012//Històric[en]Port//Ship[es]Barco
2	{ca=2012//Històric, en=Port, es=Barco}
3	[[SPLIT_FIRST]]ca[[SPLIT_LAST]]2012//Històric[[SPLIT_FIRST]]en[[SPLIT_LAST]]Port//Ship[[SPLIT_FIRST]]es[[SPLIT_LAST]]Barco
4	ca[[SPLIT_LAST]]2012//Històric en[[SPLIT_LAST]]Port//Ship es[[SPLIT_LAST]]Barco

Si el número de elementos es muy grande almacenar todos los elementos en una lista y luego crear las distintas entidades puede ser poco eficiente y en tal caso lo que se hace es crear el objeto y almacenarlo en la base de datos, siguiendo un proceso similar a los JSON mediante servicios y corrutinas.

```
fun processCsv(inputStream: InputStream): List<DatasetCSVModel> = csvReader().open
(inputStream) {
        readAllWithHeaderAsSequence().map {
            DatasetCSVModel(
                it["URL"],
                fieldOrNull(it["IDENTIFICADOR"]),
                titleDescription(it["TÍTULO"]),
                titleDescription(it["DESCRIPCIÓN"]),
                fieldSplit(it["TEMÁTICAS"]),
                keywords(it["ETIQUETAS"]),
                localDateTimes(it["FECHA DE CREACIÓN"]),
                localDateTimes(it["FECHA DE ÚLTIMA MODIFICACIÓN"]),
                frequency(it["FRECUENCIA DE ACTUALIZACIÓN"]),
                fieldSplit(it["IDIOMAS"]),
                fieldOrNull(it["ÓRGANO PUBLICADOR"]),
                fieldOrNull(it["CONDICIONES DE USO"]),
                fieldOrNull(it["COBERTURA GEOGRÁFICA"]),
                periocity(it["COBERTURA TEMPORAL"]),
                localDateTimes(it["VIGENCIA DEL RECURSO"]),
                relatedResourcesAndRegulations(it["RECURSOS RELACIONADOS"]),
                relatedResourcesAndRegulations(it["NORMATIVA"]),
                distributions(it["DISTRIBUCIONES"]),
            )
        }.toList()
    }
```

Código 35 Función de extracción y transformación de recursos en un fichero CSV

```
fun keywords(value: String?): Map<String, String>? =
      if(!value.isNullOrBlank()){
          value?.replace("\\[(\\w{2})\\]".toRegex(),"[[SPLIT_FIRS]]\$1[[SPLIT_LAST]]")?.replaceFirst("[[SPLIT_FIRS]]", "")?.split(
"[[SPLIT_FIRS]]")?.associate{
              val (left, right) = it.split("[[SPLIT_LAST]]")
              left to right
          }
      } else null
```

Código 36 Función de transformación del recurso keywords obtenido de un fichero CSV

C.3.4 Ejecución del proceso ETL con GraphQL

El proceso ETL para uno o varios datasets se puede lanzar a través de una mutación GraphQL cuya declaración se muestra en el Código 37. La operación `createCatalogRecord` va a crear un nuevo registro de catálogo, el dataset asociado y el resto de las relaciones y clases correspondientes. En función del formato indicado en `contentType` y accediendo a los metadatos ya sea a través de una ruta a la fuente destino (`contentUrl`) o procesando la

cadena de texto indicada por `content` se cargan los metadatos correspondientes al dataset y todos los recursos relacionados, como pueden ser las distribuciones.

```
type Mutation{
  createCatalogRecord(input: CatalogRecordInput):CatalogRecordOutput
}

input CatalogRecordInput{
  inCatalog: ID
  catalogRecordId: ID
  contentType: String
  content: String
  contentUrl:String
  hints:[String!]
}

type Error{
  message: String
}

union CatalogRecordOutput = CatalogRecord | Error
```

Código 37 Mutación GraphQL relacionada con el proceso ETL

C.4 Detalles de implementación del módulo Cliente

El cliente está implementado en Kotlin y React. Debido a que React está escrito en JavaScript, se necesita una manera de comunicarse entre el mundo JavaScript y el mundo Kotlin, por ello se utilizan los Kotlin Wrappers que se explican a continuación. Además, en esta sección se explica con mayor detalle Apollo Kotlin, su configuración y las tareas de generación de código que ofrece.

C.4.1 Kotlin Wrappers

Los Kotlin Wrappers se utilizan para crear una interfaz en lenguaje Kotlin que permite a los desarrolladores interactuar con bibliotecas y frameworks JavaScript, como es el caso de React. A continuación, se explican los principales Kotlin Wrappers, recogidos en la tabla 14.

Tabla 14 Kotlin Wrappers

Kotlin Wrapper	Descripción
kotlin-react	Permite crear y manipular componentes de React mediante código Kotlin en lugar de JavaScript.
kotlin-react-dom	Se emplea junto con kotlin-react para interactuar con el DOM (Document Object Model) en una aplicación web. Ofrece diversas funcionalidades para renderizar componentes de React en el navegador.
kotlin-react-router-dom	Consiste en una biblioteca de JavaScript para el enrutamiento en aplicaciones web (SPA). Permite gestionar la navegación y las rutas de la aplicación en Kotlin.
kotlin-redux	Proporciona una interfaz Kotlin para trabajar con Redux, lo que facilita el manejo del estado de la aplicación de forma escalable.
kotlin-emotion	Se trata de un wrapper para Emotion, una biblioteca diseñada para escribir estilos CSS con JavaScript. Permite aplicar estilos a los componentes de React.
kotlin-mui	Integra Material-UI, una biblioteca de estilos y componentes que se han integrado para mejorar el aspecto visual y el diseño de la interfaz.

Kotlin-react

Este wrapper permite crear y manipular componentes de React con Kotlin. Los componentes de React [27] se pueden entender como funciones o clases, que reciben un objeto de

propiedades como parámetro, identificado comúnmente como `props`, y devuelven elementos de React. Las propiedades permiten actualizar acciones automáticas, por ejemplo, se puede recibir una colección de elementos de tipo `String` y devolverlos en una lista de HTML, permitiendo la comunicación entre componentes padres e hijos. En el Código 38 se muestra un ejemplo. El componente se denomina `catalogInfoTest` y se ha definido un `props` personalizado mediante el que se recibe una lista de valores `String` a través de la variable `listOfString`. Para ello se declara una interfaz que implementa la interfaz **Props**. El término `FC` en la función es la abreviatura de **Functional Component** y es un interfaz proporcionado por el wrapper para representar un componente de tipo función. En el ejemplo, además, se utiliza una lista de MUI-Material, para cargar la lista recibida por `props` y al pulsar sobre sus elementos se navega a la ruta correspondiente indicada por la función `handleOnClick`.

```
external interface CatalogInfoTestProps : Props{
  var listOfString: Collection<String>
}

val catalogInfoTest = FC<CatalogInfoTestProps> {prop ->
                                                s
  val navigate = useNavigate()

  var listTestCatalogs by useState(props.listOfString)

  val handleOnClick:  event: MouseEvent<HTMLElement, *>) -> Unit = { event->
      navigate("/catálogs/${event.currentTarget.id}")
  }

  List
      {istTestCatalogs.map{
          ListItemButton
              onClick = handleOnClick
              id = "$it"
              + "$it"
          }

} }
}
```

Código 38 Ejemplo componente React

Kotlin-react-dom

Document Object Model o DOM es una interfaz de programación para documentos HTML y XML que ofrece una representación, estructurada en forma de árbol, del documento, a fin de poder acceder y manipular sus elementos [28]. El DOM divide el documento en una serie de nodos que representan diferentes componentes. Estos forman una estructura jerárquica mediante relaciones padre-hijo o hermano-hermano. Para poder interactuar con estos nodos proporciona una serie de métodos e interfaces que permiten manipular dinámicamente el contenido. En la figura 24 se muestra el papel del DOM con un grafo.

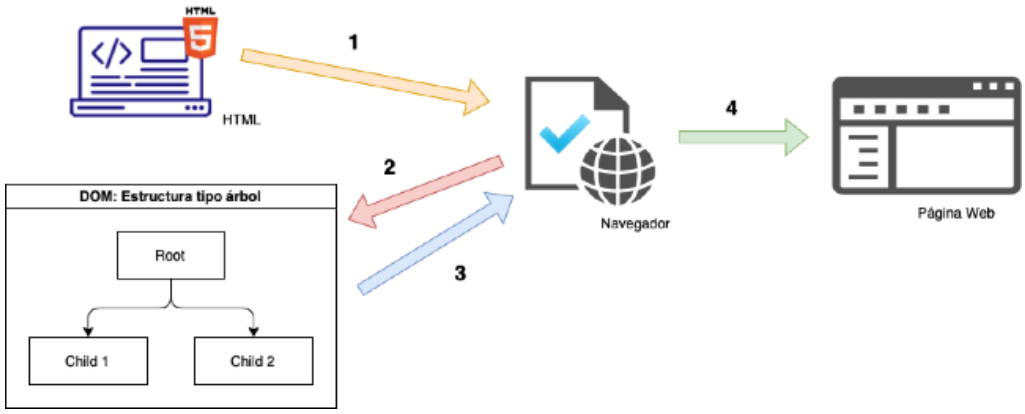


Figura 24 Diagrama sobre el papel del DOM en el normalizado de una página web

El proceso por el que se convierte el código fuente de la página web en una representación visual que puede mostrarse en una página web se conoce como **renderizado** y sigue varios pasos. En primer, lugar el navegador debe interpretar y analizar el código para, a continuación, crear la estructura de nodos anteriormente explicada (DOM). Cada elemento del código (HTML, por ejemplo) se convierte en un nodo dentro de la jerarquía. Tras la construcción del DOM, el navegador hace uso de la estructura resultante para renderizar la página web. Este proceso conlleva determinar cómo mostrar los diferentes elementos de la jerarquía en la pantalla considerando aspectos como los estilos CSS. Finalmente dibuja los pixeles en el área de visualización establecida. En el Código 39 se muestra un ejemplo de código real. En él se localiza el contenedor del DOM donde se renderiza la aplicación (`getElementById`) y a continuación crea la raíz de la aplicación en el contenedor (`createRoot`). Finalmente se indica el elemento que actúa como raíz, en este caso **Application** y se crea una instancia de este.

```
val container = document.getElementById("root") ?:error("Couldn't find root container!")
        createRoot(container).render(
            Application.create{name = "Localiza tu Dataset"}
        )
```

Código 39 Ejemplo de extracción de componente DOM

Kotlin-react-router-dom

Esta biblioteca proporciona funcionalidades de enrutamiento para aplicaciones web. Permite crear rutas, manejar sus parámetros y gestionar la navegación a través de ellas en, por ejemplo, aplicaciones SPA. A la hora de implementar y definir las rutas de la aplicación se debe tener en cuenta que el wrapper ofrece dos tipos principales de enrutamiento: **HashRouter** y **BrowserRouter**. El primero almacena el estado de navegación definiéndolo tras el símbolo '#' en la url: `http://localhost:8081/#/datasets`. Este enfoque es apropiado en los casos en los que el servidor no tiene constancia de las rutas existentes y siempre devuelve la misma página inicial, como es el caso de esta aplicación (Código 40). La principal ventaja que tiene es que el navegador no envía información sobre la ruta al servidor web, lo que simplifica la configuración. Por otro lado, existe el enfoque BrowserRouter, en el que las rutas no se definen a partir del símbolo '#', pues el servidor ya está configurado para responder a las diferentes rutas y devuelve la página HTML correspondiente. Pese a que se obtiene una URL "limpia", tiene el inconveniente de que se debe configurar el servidor de manera que responda siempre con el mismo HTML para cada una de las URL [29].

```
HashRouter{
            Routes{

                Route{
                    path = "/"
                    element = InitPage.create()
                }
                Route{
                    path = "/datasets/:id"
                    element = resourceInfo.create()
                }
            }
        }
```

Código 40 Ejemplo de HashRoutes en el proyecto

Kotlin-redux

Esta biblioteca de Kotlin JS implementa el patrón de **arquitectura Redux** para el manejo del estado en aplicaciones web desarrolladas con Kotlin. Proporciona herramientas para crear el objeto **store**, donde se almacena el estado de la aplicación, definir tanto **acciones** como **reducers** y conectar los componentes de React al store para acceder y actualizar el estado de la aplicación [30].

Redux es un patrón de administración del estado de una aplicación de forma centralizada y predecible. El objeto store es accesible desde cualquier parte de la aplicación y es modificado mediante las acciones, enviadas a través de una función conocida como **action creator**. El reducer es el responsable de aplicar los cambios de acuerdo con las acciones definidas al estado. Se implementa como una función que recibe el estado actual y una acción concreta y devuelve un nuevo estado. Otros elementos de este patrón son **dispatch** y **subscription**. El primero es responsable de enviar una acción al store y el segundo permite a los componentes de la aplicación recibir notificaciones cuando el estado cambia [31]. En una primera aproximación se implementó este patrón para conservar el estado de los filtros de la aplicación, pero resulta demasiado complejo y es mucho más simple definir un contexto común entre los componentes correspondientes, tal y como se explica más adelante en este anexo.

Kotlin-emotion

Esta biblioteca proporciona una interfaz de Kotlin para trabajar con Emotion, una biblioteca de estilos de JavaScript para React. Entre sus características destaca que permite definir estilos en línea de manera dinámica. En este proyecto se decidió no utilizarla puesto que el frontend está implementado con componentes de MUI-Material que traen consigo su propio elemento de definición de estilos (`sx`) y el resto se han declarado en un fichero CSS.

Kotlin-mui

Se trata de una biblioteca de componentes de interfaz de usuario [32,33] basada en **Material Design** [35]. Permite crear interfaces aprovechando componentes y estilos predefinidos, pero también es posible definir estilos propios.

Un ejemplo de componente es `Box`, un contenedor flexible que permite agrupar otros elementos y aplicar estilos. En el Código 41 se proporciona un ejemplo, en el que se utilizan, además, los componentes `Card` (el componente principal para la creación de una tarjeta, esto es, una estructura para mostrar contenido en un área rectangular), `CardContent` (utilizado para agregar contenido dentro de la tarjeta) y `Typography`, empleado para mostrar textos con diferentes estilos y variantes tipográficas. Como se puede observar, en `sx` se declaran los

estilos adaptando la sintaxis CSS. Más componentes y detalles de implementación están disponibles en el repositorio de GitHub de Karakum [34].

```
import mui.material.*
import mui.system.sx
import react.*
external interface CardProps:Props{var datasetInf :DatasetModel}
var CardListCatalog: FC<CardProps>{ props->
    Box{
        Card{
            sx{
                display=Display.flex
                width = Sizes.CardList.Width
            }
            CardContent{
                Typography{
                    +"${props.datasetInfo.title}"
                }
            }
        }
    }
}
```

Código 41 Ejemplo de componentes de Kotlin MUI

C.4.2 Apollo Kotlin

Este framework se centra en la gestión eficiente de la comunicación con servidores GraphQL, permitiendo a los desarrolladores definir consultas y mutaciones de manera intuitiva, optimizando la forma en la que se obtienen y se presentan en la interfaz del usuario. Para utilizarlo en este proyecto, es necesario añadir el plugin y la dependencia de Apollo Kotlin al fichero de configuración Gradle y, a continuación, configurar Apollo para que detecte el esquema GraphQL y los escalares personalizados. Además, se debe indicar el paquete y directorio donde generará el código Kotlin correspondiente [36]. La versión utilizada en el proyecto es la 3.7.4 y el paquete es `com.apollographql.apollo3`. Este paquete ofrece una serie de tareas encargadas de generar el código y los recursos necesarios para trabajar con GraphQL. Entre ellas destacan las siguientes:

Tabla 15 Tareas Gradle de Apollo Kotlin

Tarea	Descripción
convertApolloSchema	Convierte un esquema de Apollo GraphQL de un formato a otro. Existen varios formatos de esquema que se pueden utilizar, como por ejemplo GraphQL SDL (Schema Definition Language) o JSON.
downloadApolloSchema	Descarga un esquema GraphQL desde un servidor remoto y lo guarda localmente.
generateApolloSources	Genera el código Kotlin utilizado para interactuar con el servidor GraphQL. El código generado puede ser tipos de datos, consultas o mutaciones entre otros.
generateServiceApolloSchema	Genera el esquema GraphQL para un servicio específico.
generateServiceApolloSources	Genera el código Kotlin utilizado para interactuar con el servidor GraphQL de un servicio específico.
generateServiceApolloUsedCoordinates	Genera un archivo con las coordenadas de dependencia empleadas por los servicios de Apollo Kotlin.
pushApolloSchema	Carga el esquema GraphQL a un servidor remoto, de forma que puede utilizarse en tiempo de ejecución.

En el contexto de Apollo Kotlin, los **servicios** hacen referencia a una API o Backend concreto para acceder a datos a través de GraphQL. Pueden representar una fuente de datos, un servicio web u otro tipo de servicios que ofrecen datos a través de una interfaz GraphQL. Las

coordenadas identifican una dependencia específica empleada por un servicio, incluyendo información sobre el grupo, el nombre o la versión de la dependencia.

En el Código 42 se muestra el bloque `apollo` con la configuración necesaria para que las tareas descritas en la tabla 15 detecten correctamente el esquema de GraphQL. En él se define la ubicación del archivo con el esquema GraphQL, el nombre del paquete en el que se almacenan los modelos generados y el mapeo de los escalares personalizados a los adaptadores Kotlin. El hecho de mapear los escalares se debe a que Apollo solo genera los tipos básicos por defecto, si por ejemplo un tipo del esquema tiene un campo de tipo LangString, en el modelo Kotlin se muestra `Any`, lo que impide recuperar las propiedades de dicho escalar. Esto se soluciona mapeando los diferentes escalares personalizados que intervienen en el esquema.

```
apollo{
    service("service"){
        packageName.set("com.schema")
        schemaFile.set(file("src/commonMain/graphql/com/schema/schema.graphqls"))
        generateKotlinModels.set(true)
        outputDirConnection{connectToKotlinSourceSet("jsMain")}

        mapScalar("LangString", "commonModels.LangStringAdapterScalar", "commonModels.langStringAdapter")
        mapScalar("Concept", "commonModels.ConceptAdapterScalar", "commonModels.conceptAdapter")
        mapScalar("MediaType","commonModels.MediaTypeAdapterScalar", "commonModels.mediaTypeAdapter")
        mapScalar("Frequency","commonModels.FrequencyAdapterScalar", "commonModels.frequencyAdapter")
    }
}
```

Código 42 Configuración específica de Apollo Kotlin en build.gradle.kts

C.4.3 Detalles adicionales: Estado compartido entre los componentes React

El **contexto** (context) es un concepto utilizado para compartir datos entre componentes sin tener que pasarlos manualmente a través de propiedades. Resulta muy útil en componentes anidados , donde varios de ellos necesitan acceder a los mismos datos. En el ejemplo mostrado en el Código 43 se muestra la implementación de un estado común a través del contexto usando el wrapper kotlin-react. Es parte del código usado en la aplicación para compartir los filtros seleccionados entre distintos componentes, aunque en el ejemplo solo se muestra `InitPage`. Mediante `createContext` se crea un contexto indicando el tipo de este, en este caso un mapa. A continuación, se indica cual es el estado a almacenar y se hacen disponibles a todos los componentes que se incluyan dentro del alcance del contexto, en este caso solo `InitPage`. Desde este, se accede a dicho mapa de valores mediante `useRequiredContext` y podrá modificarse o acceder a cualquier valor del contexto.

```
// Application component
val FilterListContextAll = createContext<StateInstance
<MutableMap<String,MutableMap<String,Collection<String>>>>>()

val Application = FC<ApplicationProps> {props ->
    val filtersMap = FiltersMapKeys()
    val state = useState(filtersMap.filtersSelectedMap)
    var listTestDatasets by useState(mutableListOf<DatasetModel>())
    val (listFiltersTest)=state
    FilterListContextAll(state) {InitPage()}
}

//InitPage component
val InitPage = FC<Props>{
    var selectedFiltersContext by useRequiredContext(FilterListContextAll)
}
```

Código 43 Ejemplo de contexto compartido por los componentes invocados desde el componente Application

C.5 Detalles de implementación del Servidor GraphQL

El servidor se implementa en Kotlin e incluye otros lenguajes como GraphQL para la declaración del esquema y algunas consultas. En esta sección del anexo se explican detalles de la implementación de escalares personalizados en el esquema GraphQL, aspectos técnicos de Netflix DGS, CORS y el entorno de pruebas.

C.5.1 Escalares personalizados en el esquema GraphQL

GraphQL proporciona de forma predeterminada distintos escalares como son `String`, `Float` o `Boolean`. Sin embargo, es posible manejar datos más complejos a través de la definición de escalares personalizados. Por ejemplo, se puede definir un campo de tipo **LocalDateTime** o **LangString**. En el esquema, los escalares personalizados se declaran como `scalar <NombreScalar>`. Una vez declarado en el esquema hay que implementar el adaptador, que traduce del tipo implementado en Kotlin y el esquema GraphQL. En el Código 44 se muestra el ejemplo del escalar **LocalDateTime**. La implementación se lleva a cabo mediante la clase `LocalDateTimeScalar`, etiquetada con @DgsScalar. Esta clase implementa el interfaz **Coercing** que proporciona varios métodos (ver tabla 16) encargados de realizar la conversión desde el tipo de datos personalizado a su representación GraphQL (serialización) y viceversa (deserialización). Cuando se realiza una consulta o mutación en GraphQL que involucre un campo con el escalar LocalDateTime se activa el adaptador. Durante la serialización el método `serialize` realiza la conversión del objeto LocalDateTime a String. Durante la deserialización el método `parseLiteral` o `parseValue` convertirá el valor String en un objeto LocalDateTime. Una vez realizada la conversión el servidor GraphQL utilizará el objeto en la lógica de resolución de campos de forma correcta.

Tabla 16 Métodos de coerción

Método	Descripción
serialize	Convierte un objeto del tipo de dato personalizado en su representación en GraphQL (una cadena de texto por lo general).
parseLiteral	Convierte un valor obtenido de una consulta GraphQL como un literal en un objeto del tipo de dato personalizado
parseValue	Convierte un valor obtenido de una consulta GraphQL como una variable en un objeto del tipo de dato personalizado

Para llevar a cabo esta implementación se han consultado los enlaces [37,38,39].

```kotlin
@DgsScalar(name = "LocalDateTime")
class LocalDateTimeScalar : Coercing<LocalDateTime, String>{ // Coercing Input,Output

    @Autowired
    lateinit var converter: ConvertersAuxiliarEntitiesTo

    var dateFormatter: DateTimeFormatter = DateTimeFormatter.ofPattern(DATE_PATTERN)

    override fun serialize(input: Any): String?{
        if (input is LocalDateTime){
            return input.format(dateFormatter)
        }
        return null
    }
    override fun parseLiteral(input: Any): LocalDateTime{
        if (input is String){
            return LocalDateTime.parse(input, dateFormatter)
        }
        return LocalDateTime.now()
    }
    override fun parseValue(input: Any): LocalDateTime{
        if (input is String){
            return LocalDateTime.parse(input, dateFormatter)
        }
        return LocalDateTime.now()
    }
}
```

Código 44 Ejemplo de adaptador para el escalar GraphQL personalizado 'LocalDateTime'

C.5.2 Netflix DGS

Netflix DGS es framework GraphQL basado en Kotlin que proporciona herramientas para crear y consumir servicios GraphQL en aplicaciones Kotlin. Ofrece herramientas para generar clases y métodos específicos en Kotlin a partir de definiciones GraphQL, lo que facilita la escritura de consultas y la implementación de los resolvers. Además, ofrece anotaciones de clases y métodos de forma que se resuelvan automáticamente las consultas o mutaciones y los campos solicitados.

Generador de código Kotlin

Netflix DGS ofrece la herramienta **dgs graphql codegen**, que genera automáticamente y en tiempo de compilación el código correspondiente a la estructura del esquema GraphQL: tipos, consultas, mutaciones... En el Código 45 se muestra la tarea **GenerateJavaTask** en la que se establece la ubicación del esquema GraphQL, el nombre del paquete del código generado, el lenguaje en el que se genera dicho código y el mapeo de los escalares personalizados y el paquete al que pertenecen. El código se genera en el directorio **build/generated** del proyecto. El directorio **build** se genera automáticamente y contiene los resultados de la construcción del proyecto. Dentro del mismo existen varios subdirectorios con distinta finalidad. El denominado **generated** contiene archivos generados automáticamente por herramientas de generación de código. Un ejemplo del código Kotlin resultante de la conversión del tipo Distribution del esquema se refleja en el Código 46. En él se muestra una clase de datos junto con una serie de anotaciones propias de la biblioteca Jackson, utilizada para la serialización y deserialización de objetos JSON. Los atributos de la clase se definen mediante la anotación `@JsonProperty`.

```
tasks.withType<com.netflix.graphql.dgs.codegen.gradle.GenerateJavaTask>
{
    schemaPaths = mutableListOf(
        "src/jvmMain/resources/schema/schema.graphqls"
        )
    packageName = "com.graphqlDGS.graphqlDGS.model"
    language = "kotlin"
    typeMapping=mutableMapOf(

        "LangString" to "es.unizar.iaaa.tfg.annotations.LangString",
        "Frequency" to "es.unizar.iaaa.tfg.annotations.Frequency",
        "Concept" to "es.unizar.iaaa.tfg.annotations.Concept",
        "NonNegativeInt" to "es.unizar.iaaa.tfg.annotations.NonNegativeInt",
        "MediaType" to "es.unizar.iaaa.tfg.annotations.MediaType",
    )
}
```

Código 45 Tarea de Netflix DGS encargada de la generación de tipos a partir del esquema

```
type Distribution implements
ReferenceWithinExternalContext{
  id: ID!
  title: [LangString!]
  accessUrl:String
  accessService: [DataService!]
  byteSize: NonNegativeInt
  format: MediaType
  identifier: [String!]
}
```

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NONE)
public data class Distribution(
  @JsonProperty("id")
  public val id: String,
  @JsonProperty("title")
  public val title: List<LangString>? = null,
  @JsonProperty("accessUrl")
  public val accessUrl: String? = null,
  @JsonProperty("accessService")
  public val accessService: List<DataService>? = null,
  @JsonProperty("byteSize")
  public val byteSize: NonNegativeInt? = null,
  @JsonProperty("format")
  public val format: MediaType? = null,
  @JsonProperty("identifier")
  public override val identifier: List<String>? = null,
) : ReferenceWithinExternalContext
  public companion object        {
}
```

Código 46 Ejemplo de código generado a partir del esquema GraphQL con dgs graphql codegen

Adaptadores

Netflix DGS ofrece una forma de conectar el código Kotlin al esquema, utilizando una serie de anotaciones que se aplican a clases y métodos. Estas se encuentran explicadas en la tabla 17.

Tabla 17 Anotaciones Netflix DGS

Anotación	Descripción
@DgsComponent	Representa una clase como un componente de DGS, que contiene la lógica para resolver los campos solicitados en una consulta GraphQL.
@DgsQuery	Representa el método anotado como una consulta GraphQL. Contiene la lógica para resolver los campos solicitados en dicha consulta.
@DgsData	Se utiliza en un método dentro de un componente de DGS para especificar que se va a resolver un campo de un tipo de objeto específico. En el ejemplo, resuelve el campo servesDataset de un DataService concreto. Guarda relación con @DgsQuery. Si se ejecuta la consulta dataService con parámetro id igual a “dServ1”, al solicitar su campo servesDataset se resuelve para dicho id.
@InputArgument	Marcar un parámetro en un método anotado con @DgsQuery como un argumento de entrada en una consulta GraphQL
@DgsMutation	Representa el método anotado como una mutación GraphQL. Contiene la lógica para resolver los campos solicitados en dicha mutación.
@DgsScalar	Define un escalar personalizado en GraphQL. Los escalares permiten la conversión entre un tipo de dato personalizado y su representación en GraphQL.

En el Código 47 se muestra un ejemplo de componente de Netflix DGS, en particular, corresponde con el recurso data service. En la clase, se incluye una consulta (`@DgsQuery`) y un metadato que describe el servicio de datos, anotado con @DgsData. De esta forma el motor

de DGS, cuando identifique que la consulta GraphQL emitida solicita el campo `servesDataset` de data service, invocará este método, y al ejecutar la consulta dataService ocurrirá se invocará el método anotado con `@DgsQuery`.

```
@DgsComponent
class DataServiceQueries(
    private val dataServicesServices: DataServicesServices,
){
    @DgsQuery
    fun dataService(@InputArgument id: String?): DataService?{
        if (id == null){return null}
        return dataServicesServices.getDataService(id)
    }
    @DgsData(parentType = "DataService")
    fun servesDataset(dfe: DgsDataFetchingEnvironment): Collection<DatasetInCatalog?>{
        val dserv: DataService = dfe.getSource()
        return dataServicesServices.getServesDataset(dserv.id)
    }
}
```

Código 47 Ejemplo de "adapter" de Netflix DGS

C.6 Configuración de CORS

En el Código 48 se muestra cómo se habilita la funcionalidad de manejo de CORS en una aplicación web basada en Spring. La clase `WebConfig` implementa la interfaz denominada `WebMvcConfigurer`. Esta interfaz proporciona métodos para personalizar varios aspectos del comportamiento de Spring MVC, básicamente permite ajustar como se manejan las solicitudes y respuestas en la aplicación. Un ejemplo de método es `addCorsMappings`, quien configura la política CORS, en este caso, indicando que deben permitirse solicitudes desde cualquier origen (**"/**"**).

```
@Configuration
@EnableWebMvc
class WebConfig:WebMvcConfigurer{
    override fun addCorsMappings(registry: CorsRegistry){
        registry.addMapping("/**")
    }
}
```

Código 48 Configuración de CORS en el servidor GraphQL

C.7 Entorno de pruebas unitarias y de integración

Para configurar el entorno de pruebas de los repositorios con el fin de realizar pruebas de integración con JPA y la base de datos se utiliza la anotación de SpringBoot **@DataJpaTest**. Entre otras cosas facilita la inyección de dependencias (como es el caso de TestEntityManager, una alternativa de EntityManager) y la configuración automática de la base de datos. Para el caso de las consultas GraphQL el entorno se configura con la siguiente anotación: **`@SpringBootTest`** [47].

En estas pruebas se inyecta la dependencia **`DgsQueryExecutor`** [48], una interfaz que proporciona varios métodos para ejecutar una consulta y obtener un resultado evitando lidiar con HTTP. En ambos casos los resultados obtenidos se validan con paquete Assertions de JUnit, por ejemplo, con los métodos `assertEquals` o `assertThat` respectivamente. Además, para el caso de las consultas los resultados se extraen mediante JsonPath gracias a **`GraphQLResponse`**, una clase que encapsula y ofrece métodos para gestionar la respuesta. A la hora de implementar un test en el que se valida la respuesta devuelta por una consulta GraphQL hay tener en cuenta como declarar la consulta o mutación y como pasarle los parámetros si fuesen necesarios. En el Código 49 se puede ver un ejemplo en el que se prueba la mutación `createCR` encargada de la creación de CatalogRecord, el recurso y los elementos correspondientes como distribuciones y data services. El primer paso es declarar la mutación en una variable utilizando `“””_”””trimIndent`, donde las comillas permiten escribir varias cadenas de texto de forma que abarquen varias líneas y el método `trimIndent` elimina el sangrado adicional de las líneas de la cadena [49]. Los nombres de los parámetros de la consulta o mutación deben declararse también como variables, por ejemplo, `inputParam`, pues si se coloca directamente `$input` en la consulta, como un String, no lo reconoce. Los valores asociados a estos parámetros se estructuran en un mapa cuyas claves son cadenas de texto que indican el nombre del parámetro y se pasan junto con la solicitud al método correspondiente del **DgsQueryExecutor**.

En este caso el método es `executeAndGetDocumentContext`, que tras ejecutar la consulta devuelve un **DocumentContext**. Un DocumentContext se puede utilizar para extraer múltiples valores utilizando JsonPath. Para ello se utiliza **GraphQLResponse**. Además, en caso de disponer varios test que se ejecutan secuencialmente, es importante el uso de la anotación `@DirtiesContext` (ver el Código 50), para evitar que al ejecutar varias pruebas que modifican la base de datos los siguientes se vean afectados por los anteriores. De esta forma se indica que el contexto de la aplicación es “sucio” en cierto momento (en el proyecto se considera sucio tras la ejecución de cada método de test) y debe cargarse de nuevo. Esto implica, entre otras cosas, volver a crear la base de datos e insertar los datos de prueba.

```
@DirtiesContext(methodMode = DirtiesContext.MethodMode.AFTER_METHOD)
@Test
fun `Creo CR a partir de json y tiene inCatalog records id su id`(){
    val inputParam = "\$input"

    val query = """
    mutation createCR($inputParam:CatalogRecordInput){
        createCatalogRecord(input:$inputParam){
            ... on CatalogRecord{
                primaryTopic{
                    inCatalog{
                       id
                        records {
                           id
                        }
                    }
                }
            }
        }
    }""".trimIndent()
```

Código 49 Ejemplo test de consulta GraphQL parte 1

```
    val crInput = mutableMapOf<String, Any>(
        "input" to mapOf(
            "inCatalog" to "root",
            "contentType" to "application/json",
            "contentUrl" to urlRecord,
            "catalogRecordId" to "CRNuevo",
            "hints" to listOf("datos.gob.es")
        )
    )
    val createCR = dgsQueryExecutor.executeAndGetDocumentContext(query, crInput)
    val response = GraphQLResponse(createCR.jsonString())
    val records = response.extractValue<Collection<String>>(
        "data.createCatalogRecord.primaryTopic.inCatalog[*].records[*].id"
    )
    assertThat("CRNuevo").isIn(records)
}
```

Código 50 Ejemplo test de consulta GraphQL parte 2

Anexo D Pruebas finales de GraphQL

A continuación, se muestra el ejemplo de dos resultados obtenidos al ejecutar, un endpoint de la API del portal de datos del Gobierno de España (Código 51) y una consulta GraphQL (Código 52). Como se puede observar, la principal diferencia es la simplicidad del segundo resultado, pudiéndose comprobar como solo se obtienen aquellos campos que son requeridos, en lugar de cargar todo y producir una sobrecarga de datos innecesarios. La consulta es la correspondiente al Código 11, presentada en la sección 5.1 de este documento. En este caso, la consulta GraphQL solo solicita el id, título, publicador y formatos de cada dataset disponible.


```
"items": [
    {
        "_about":
"https://datos.gob.es/catalogo/ea0020951-el-laberinto-de-el-estrecho-de-torres-una-pr
  "license": "https://creativecommons.org/licenses/by-nc-sa/3.0/es/",
  "publisher": "http://datos.gob.es/recurso/sector-publico/org/Organismo/EA0020951",
  "distribution": {
      "accessURL": "https://digital.csic.es/handle/10261/264167",
      "format": {
          "type": "http://purl.org/dc/terms/IMT",
          "value": "application/pdf"
      },
      "identifier": "https://doi.org/10.20350/digitalCSIC/14560"    ,
      "title": [
          {
              "_value": "Selección documental",
              "_lang": "es"
          }
      ],
      "type": "http://www.w3.org/ns/dcat#Distribution"
  },
  "identifier": "https://digital.csic.es/handle/10261/264167",
  "issued": "dom, 29 sep 2013 22:00:00 GMT+0000",
  "keyword": [
      {
          "_value": "Edad Moderna",
          "_lang": "es"
      }
  ],
```


Código 51 Ejemplo de parte del resultado JSON devuelvo al acceder al endpoint catalog/dataset

```
{"data":{
    "resourcesByFilter":[
      { "id": "0042bec9-875c-395c-803a-e426d9311053",
        "title":[],
        "publisher":{
          "notation": "L01280066",
          "label": "Ayuntamiento de Alcobendas"
        },
        "distributions":[
          {"format":{
              "type": "text",
              "subtype": "json"
            }
          },
          {"format":{
              "type": "application",
              "subtype": "vnd.ms-excel"
            }
          }
        ]
      },
      {
        "id": "00439443-cf25-36b5-9cff-ddb37dfb5d86",
        "title":[],
        "publisher":{
          "notation": "L01462444",
          "label": "Ayuntamiento de Torrent"
        },
        "distributions":[
          {"id": "c7e5532b-7e44-379d-ac18-2781bc7a22e7",
            "format": null
          }
        ]}
    ]
  }
}
```


Código 52 Resultado parcial JSON de la consulta GraphQL indicada en el Código 11

Anexo E Apariencia final de la UI

La interfaz de usuario, como ya se ha mencionado, consta de tres páginas diferentes. Una que muestra un listado de los recursos de un tipo concreto existentes, en la que se pueden realizar búsquedas o aplicar filtros (figura 25), otra en la que se muestra la información (metadatos y relaciones) de un recurso en específico (figura 27) y finalmente una página que simula una API donde se pueden efectuar consultas predefinidas introduciendo ciertos argumentos como el nombre del publicador o categorías (figura 28). Además, en todas las páginas existe una cabecera con un menú donde se puede navegar entre los recursos disponibles o la API mencionada (figura 26).

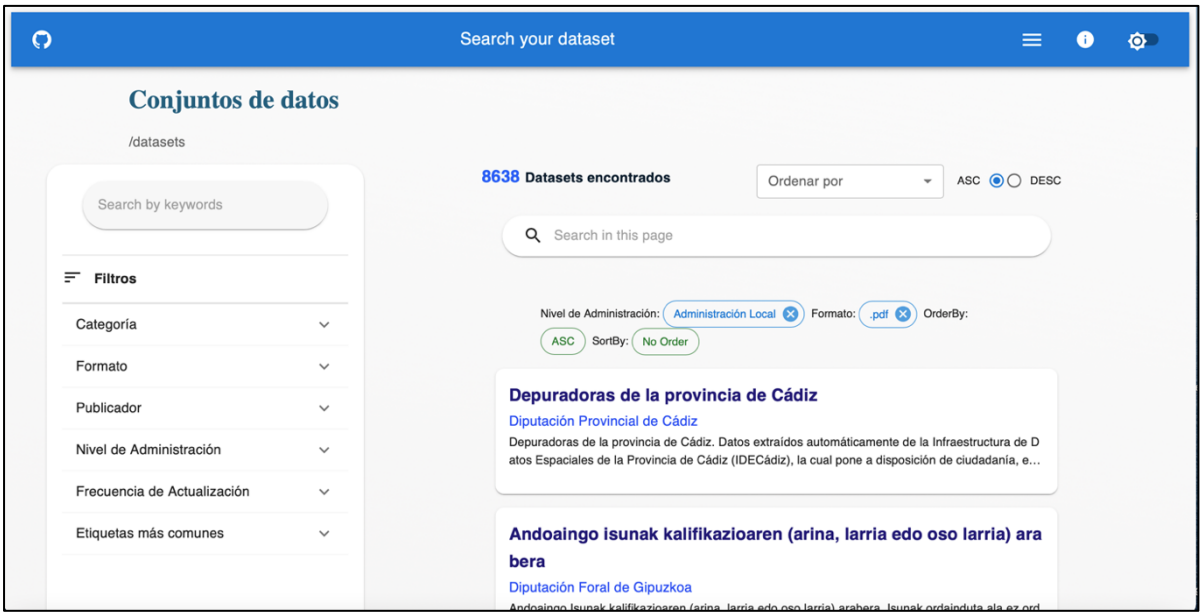


Figura 25 Resultado final: página inicial con filtros

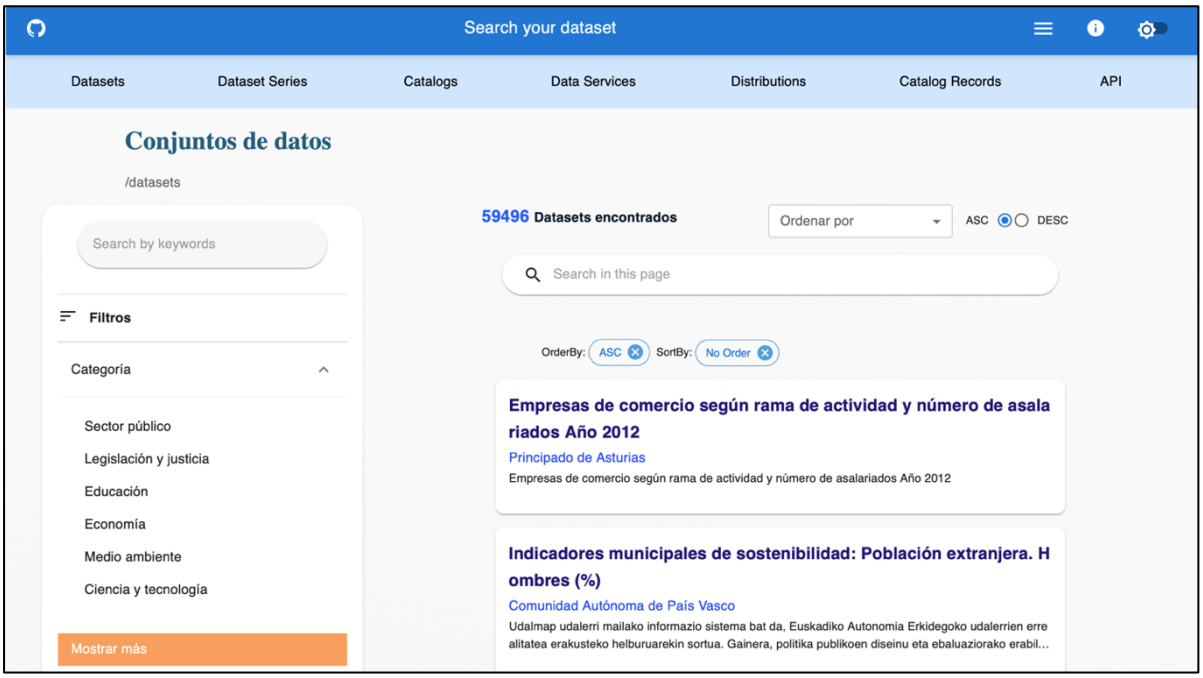


Figura 26 Resultado final: página inicial con menú desplegable

Figura 27 Resultado final: página secundaria

Figura 28 Resultado final: página API

Anexo F Gestión del tiempo

En este anexo se indica el tiempo empleado en la realización del proyecto y como se ha gestionado su desarrollo. Se ha llevado a cabo un control de versiones a través de GitHub, quién, además, facilita la organización del proyecto mediante herramientas como las issues o las wikis. Las primeras permiten realizar un seguimiento de tareas, mejoras o problemas en el proyecto y las wikis, crear documentación colaborativa para el proyecto. Por otro lado, GitHub dispone de un gran número de repositorios públicos, esto es, lugares donde desarrolladoras almacenan código fuente y archivos de sus proyectos. Esto ha resultado de utilidad pues se encuentra mucha documentación y ejemplos que ayudan a resolver ciertos problemas que han surgido en el desarrollo de este proyecto o entender mejor las tecnologías utilizadas.

En cuanto a la gestión del tiempo se ha llevado a cabo un registro de los distintos procedimientos implicados a través de una hoja de cálculo. Las fases se pueden sintetizar como un análisis previo del Trabajo de Fin de Grado, estudiando tecnologías y el problema a resolver. A continuación comenzó el desarrollo de la parte de GraphQL, la que más costó pues Kotlin, GraphQL y DCAT eran conceptos nuevos. Por último se desarrolló el resto del servidor y el cliente y se cargaron los datos del portal. Cada fase se dividió en varias tareas pequeñas y se implantaron objetivos semanales. Hay que tener en cuenta que se ha dedicado tiempo a las pruebas con el fin de agilizar el resto del desarrollo. A continuación se refleja el resultado a través de la tabla 18.

Tabla 18 Gestión del tiempo

Tarea	Descripción	Tiempo (h)
Análisis y estudio del problema y las tecnologías	Análisis del contexto y el problema a resolver, identificando las distintas tecnologías que se van a utilizar, diseño de prototipos.	61
Esquema GraphQL	Tiempo dedicado al estudio del modelo DCAT y la elaboración de un esquema GraphQL en base a dicho modelo.	54
Base de datos	Tiempo dedicado al diseño, configuración, implementación y poblado de la base de datos.	35
Servidor	Tiempo empleado en la implementación del servidor GraphQL y el proceso ETL. Se incluyen pruebas realizadas, errores detectados y soluciones elaboradas.	51
Cliente	Tiempo empleado en la implementación del cliente. Se incluyen pruebas realizadas, errores detectados y soluciones elaboradas.	63
Memoria	Tiempo invertido en la elaboración de este documento	91
Otros	Reuniones, tiempo dedicado a la organización, tutoriales de ayuda.	27
Total	Tiempo total dedicado	382