TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Implementación de una API GraphQL sobre el COVID-19

Curso 2020/2021

Alumno/a:

Sergio Guillén González

Director/es:

Manuel Torres Gil

1 INTRODUCCIÓN Y OBJETIVOS

1.1 PRESENTACIÓN

En este proyecto se pretende implementar una API GraphQL[1] utilizando el framework Spring sobre los casos de Coronavirus a nivel global, con especial interés en los países de España e Italia. Esta API clasifica los casos de Coronavirus en tres tipos: confirmados, muertes y altas.

La fuente de datos que se utilizará para este proyecto es la API *Coronavirus COVID-19* (de aquí en adelante la API fuente), esta maneja una gran cantidad de datos con acceso a múltiples endpoints[2] para consulta de datos referentes a casos confirmados, muertes y altas provocados por la enfermedad desde que hay registros.

En este proyecto accederemos a los endpoints de la fuente utilizando Spring Boot y Java. Se implementarán las entidades que a su vez representarán las tablas que conforman la base de datos para crear una API con soporte GraphQL.

La base de datos se creará a partir de los datos en formato JSON[3] que recibiremos como respuesta de la API, para luego implementar las queries y mutaciones que utilizaremos en GraphQL desde Postman[4], además se implementará el uso de JSON Web Token (JWT)[5] para controlar el acceso a los endpoints de la API, y así evitar que usuarios no deseados accedan a ella.

La API dispondrá también de endpoints que recibirán los datos de la consulta a través de la URL para facilitar que los usuarios que no disponen de conocimientos sobre GraphQL también puedan usarla.

Estos endpoints llamarán internamente a los endpoints que funcionan con GraphQL pero de forma interna es decir, cuando un usuario utilice un endpoint introduciendo los datos de la consulta mediante la URL, la API resolverá la petición llamando al correspondiente endpoint que la resuelve en GraphQL.

Con esta API, podremos consultar datos referentes a los casos de COVID-19 a nivel global y por país, para luego utilizar una librería de visualización de datos llamada MetricsGraphics, con la que podremos generar de forma automática gráficas para estudiar el avance del COVID-19 en los países citados anteriormente.

1.2 MOTIVACIÓN

La motivación de este trabajo es la creación de un servicio de valor añadido que aproveche las ventajas de utilizar el lenguaje de consultas GraphQL. En este caso se ha implementado una API que contiene datos sobre el COVID-19 y que permite obtener dichos datos de forma personalizada además de aprovechar el resto de funcionalidades de GraphQL.

[1] Lenguaje de consulta y manipulación de datos para APIs basado en grafos.
[2] Punto de acceso a la API, URL que responde a una petición.
[3] Acrónimo de JavaScript Object Notation, es un formato de texto utilizado en el intercambio de datos.
[4] Herramienta utilizada para el testeo de APIs.
[5] Cadena de texto que se utiliza para verificar la identidad de un usuario.

A lo largo de este TFG se expondrá todo lo referente a la creación de la API, cubriendo desde la descarga de software para utilizar, hasta la generación de gráficas para visualización de resultados, los puntos a destacar son:

- Descarga y configuración del Entorno de Desarrollo (IDE) Spring Tools.
- Configuración y manejo de una base de datos SQL que esté conectada directamente a la API.
- Implementación de las entidades que conformarán la base de datos en Java.
- Implementación de un esquema GraphQL.
- Implementación de queries y mutaciones en GraphQL y su uso desde diversas herramientas de testeo de APIs cómo Postman.
- Implementación de una API GraphQL sencilla de ejemplo.
- Creación de endpoints de acceso a la API.
- Implementación de un JSON Web Token para evitar que usuarios no deseados interactúen con la API.
- Uso de la librería MetricsGraphics para generar gráficas referentes a la evolución de la pandemia.

1.3 PLANIFICACIÓN

Periodo de tiempo	Actividad realizada	Duración
Septiembre- Octubre	• Formación sobre el lenguaje de consultas GraphQL. • Instalación de Spring Tools y creación del proyecto Maven	30 horas
Octubre	• Creación de un proyecto de ejemplo con Spring Boot • Instalación de la base de datos h2 • Instalación y utilización de GraphiQL para resolver queries	60 horas
Noviembre	• Formación sobre herramientas para testing de APIs • Formación sobre Autenticación JWT • Implementación token JWT	55 horas
Diciembre	• Instalación y uso de Postman con GraphQL • Instalación de Insomnia • Implementación de queries y mutaciones en ambos programas	35 horas
Enero	• Búsqueda de un ejemplo para resolverlo en el proyecto • Búsqueda de una API con información pública y periódica sobre el COVID-19 • Creación de API para el proyecto definitivo	40 horas

Febrero	• Configuración de la base de datos h2 • Tratamiento de datos de la API de origen • Testing de la API usando Postman	40 horas
Marzo-Abril	• Uso de la librería MetricsGraphics para mostrar resultados • Redacción de la memoria TFG	50 horas

Figura 1.3-1 Planificación temporal.

1.4 ESTRUCTURA DE LA MEMORIA

La memoria TFG consta de los siguientes capítulos:

1. Introducción: en este capítulo se justifica la elección de este tema para el desarrollo del TFG junto a la planificación temporal del proyecto.

2. Tecnologías y herramientas: consiste en la enumeración de las tecnologías y herramientas que se han utilizado para llevar a cabo este proyecto.

3. Proyecto de ejemplo: se relata la implementación de una API GraphQL sencilla protegida con JWT, con ejemplos de consultas GraphQL en Postman.

4. Funcionamiento de la API: consiste en un ejemplo de consulta a la API sobre el COVID-19 usando Postman, junto a la generación de gráficas con MetricsGraphics.

5. Conclusiones: exposición de las conclusiones del proyecto.

6. Referencias: la bibliografía consultada para el presente trabajo.

Por último, se adjunta un índice de todas las figuras que conforman el trabajo.

2 TECNOLOGÍAS Y HERRAMIENTAS UTILIZADAS

En este capítulo sirve como introducción a las tecnologías y herramientas utilizadas a lo largo del proyecto, acercando al lector de una forma sencilla a los nuevos frameworks y a la utilización de herramientas diseñadas para el testeo de APIs y queries (Postman, Insomnia y GraphiQL).

Esta parte es esencial, ya que la mayor parte del tiempo que se va a dedicar implementando la API será utilizando estas herramientas. Durante este capítulo, se cubrirá todo lo relacionado con el proceso de instalación e implementación de queries y scripts para automatización de procesos. Por último, se incluye un apartado en el que se muestran las conclusiones del capítulo.

2.1 SPRING FRAMEWORK

Spring es un framework[6] open source[7], que se utiliza para la implementación de aplicaciones *Java*, su primera aparición fue bajo la licencia *Apache 2.0* en Junio de 2003, pero su versión 1.0 no aparecería hasta Marzo de 2004.

Figura 2.1-1 Logotipo de Spring.

La primera versión fue escrita por Rod Johnson, quien lo lanzó junto a la publicación de su libro *Expert One-on-One J2EE Design and Development* [1] (Octubre 2002). La versión 1.2.6 de Spring Framework obtuvo reconocimientos en *Jolt Awards* y *Jax Innovation Awards* en 2006.

Tras el lanzamiento de su primera versión en 2004, surgió una versión 2.0 en 2006, seguida de la versión 2.5 en noviembre de 2007, Spring 3.0 en diciembre de 2009 y dos años más tarde, Spring 3.1.

En enero de 2013 se anunciaba el desarrollo de la versión 4.0, siendo la versión actual es la 5.1.6. [2].

2.1.1 ¿Qué es Spring? Introducción a Spring Boot

[6] *Entorno de trabajo*: es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular.
[7] Modelo de desarrollo de software basado en la colaboración abierta.

Spring se divide en módulos, donde cada uno de estos módulos proporciona ciertas funcionalidades al proyecto. En la Figura 2.1-2 puede verse cómo están distribuidos los módulos de Spring.

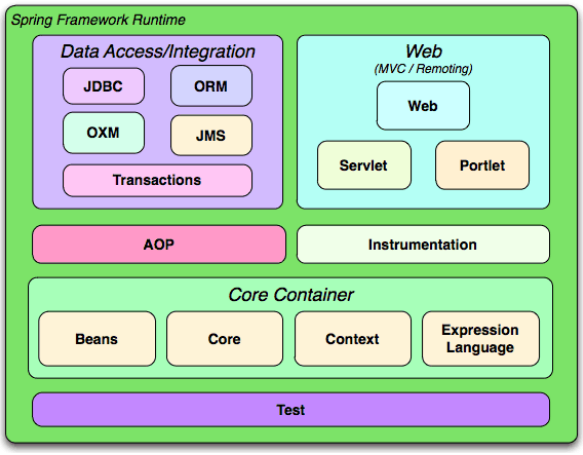

Figura 2.1-2 Estructura de módulos de Spring.

- Core Container*:* este es el módulo principal de Spring, encargado de proporcionar la inyección de dependencias[8], eventos, validación y data binding.
- Data Access: ofrece abstracciones sobre JDBC, mapeadores y transacciones.
- Programación Orientada a Aspectos (AOP)*:* este módulo ofrece soporte para aspectos.
- Instrumentación: proporciona soporte para la instrumentalización de clases.
- Web*:* permite la creación de controladores Web.
- Test: contiene framework de testeo, con soporte JUnit[9] para testear lo que sea necesario.
- Spring Boot soporta Hibernate[10], Java Persistence API (JPA), Java Data Objects (JDO), además, permite el mapeo de estructuras de una base de datos relacional sobre una estructura lógica de entidades para agilizar el desarrollo de aplicaciones.

La configuración inicial de Spring es bastante tediosa, de aquí nace Spring Boot que simplifica este proceso [3].

Existe la posibilidad de enlazar un gestor de base de datos a nuestra aplicación que hemos creado usando Spring Boot, como por ejemplo H2 Database Engine, que es una base de datos en memoria.

El primer paso para crear una aplicación utilizando este framework es crear un proyecto de inicio o utilizar Spring Initializr[11], donde especificaremos qué dependencias y qué gestor de

[8] Patrón de diseño orientado a objetos
[9] Conjunto de librerías utilizadas para la creación de pruebas unitarias (tests) en Java.
[10] Herramienta de mapeo objeto-relacional para Java. Permite el mapeo de atributos entre una base de datos y los objetos de una aplicación.
[11] Herramienta web para definir la estructura de un proyecto de Spring Boot.

dependencias se van a utilizar. El resultado de esto será un fichero comprimido que contiene la ruta de carpetas del proyecto.

2.1.2 Gestor de dependencias: Maven o Gradle

Al utilizar Spring Initializr o al crear el proyecto de forma convencional desde *Eclipse*[12], hay que escoger qué gestor de dependencias se va a utilizar.

Por un lado, la gestión de dependencias de Maven queda reflejada en un archivo xml[13]. Este archivo estará formado por un conjunto de etiquetas. Para añadir una dependencia, simplemente habrá que modificar la etiqueta <*dependencies*> añadiendo dentro de ella una nueva <*dependency*>. En la Figura 2.1-3 se muestra un fragmento del archivo de configuración xml.

```
19  <dependencies>
20  <dependency>
21      <groupId>javax.xml.bind</groupId>
22      <artifactId>jaxb-api</artifactId>
23      <version>2.1</version>
24  </dependency>
25   <dependency>
26      <groupId>io.jsonwebtoken</groupId>
27      <artifactId>jjwt</artifactId>
28      <version>0.2</version>
29  </dependency>
```

Figura 2.1-3 Gestión de dependencias con Maven.

Una vez añadidas las dependencias, se debe actualizar el proyecto para que Maven detecte las nuevas dependencias añadidas y así poder usarlas.

Por otra parte, Gradle mejora algunos aspectos de Maven. Una de las principales diferencias con Maven es que este gestor de dependencias no utiliza xml, si no que utiliza DSL para ello, como puede observarse en la Figura 2.1-4.

```
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'application'

mainClassName = 'hello.HelloWorld'

repositories {
    mavenCentral()
}

jar {
    baseName = 'gs-gradle'
    version =  '0.1.0'
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    compile "joda-time:joda-time:2.2"
    testCompile "junit:junit:4.12"
}
```

Figura 2.1-4 Gestión de dependencias con Gradle.

El gestor de dependencias elegido dependerá de la complejidad del proyecto a abordar. Si el proyecto no requiere una gran personalización, Maven es más que suficiente, pero si el proyecto requiere de comportamientos en función de variables en tiempo de construcción, entonces Gradle puede ser una buena elección [4].

Una vez creado el proyecto y elegido el gestor de dependencias, ya podremos empezar a implementar las clases en Java y hacer uso de todas las funcionalidades que ofrece Spring como

[12] Entorno de desarrollo que se utilizará en este proyecto.
[13] Siglas de eXtensive Markup Language, es un lenguaje basado en el uso de etiquetas.

el uso de anotaciones para mapear objetos, uso de repositorios JPA y enlazar una base de datos al proyecto.

2.1.3 Alternativas a Spring

Existe una amplia variedad de frameworks que funcionan con Java. Algunas de las alternativas más populares son:

- Struts: fue creado por Apache basado en el modelo MVC[14], pensado para personas que están comenzando en el mundo del desarrollo web. Su nueva versión Struts 2 permite el uso de nuevas tecnologías como JavaScript [15] ampliando así la funcionalidad de nuestras aplicaciones Java.

- Google Web ToolKit: GWT es un framework para desarrollar aplicaciones en Java. Su punto fuerte es que puede convertir el código Java en código JavaScript. Es ideal para nuevos desarrolladores ya que facilita la creación de interfaces de usuario sin tener mucho conocimiento sobre lenguajes de scripting y permite la integración con entornos de desarrollo como Eclipse.

- Dropwizard: fue pensado para implementar todo de forma sencilla y rápida, teniendo la capacidad de poder crear aplicaciones web RESTful con un alto rendimiento y fiabilidad. Incluye una gran cantidad de librerías como Guava, Metrics o Jackson, entre otras que incluye Java. Es de código abierto y puede configurarse fácilmente con Eclipse.

Finalmente, se ha optado por el uso del framework *Spring* debido a su facilidad de integración con Eclipse, la gestión de dependencias con Maven, su soporte para GraphQL y la gran comunidad y documentación disponible que hay en Internet [5].

2.1.4 Integración de Spring Boot con Eclipse (Spring Boot Suite 4)

Para comenzar a utilizar Spring Boot en el entorno de desarrollo de Eclipse desde cero, la forma más sencilla es visitar la web de Spring Tools.

Figura 2.1-5 Web de Spring Tools.

Para descargarlo, simplemente seleccionamos nuestra versión de S.O. y comenzará a descargarse, además si estamos utilizando otro IDE que no sea Eclipse, como Visual Studio

[14] Patrón de diseño Modelo-vista-controlador.
[15] Lenguaje de programación interpretado orientado a objetos.

Code o Theia, Spring Tools también ofrece un enlace de descarga para integrarlo con dichos IDEs.

Una vez descargado, se ejecuta y se descomprime su contenido en una carpeta que el propio instalador crea (*sts-4.8.0.RELEASE* en mi caso). Para ejecutarlo, abrimos *SpringToolsSuite4.exe* y ya tendremos nuestro IDE Eclipse con Spring Tools listo para funcionar.

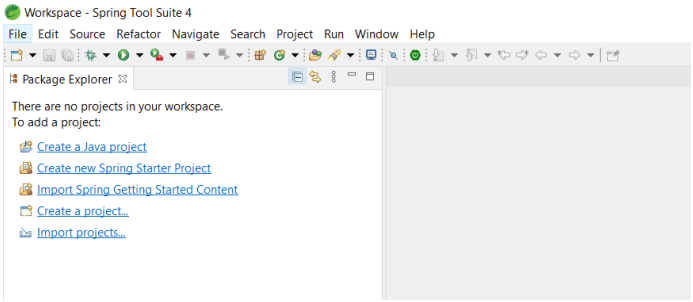

Figura 2.1-6 Interfaz de Spring Boot Suite 4.

Si ya tenemos una versión de Eclipse que estamos utilizando y no deseamos tener que instalar esta versión, existe también una forma alternativa para instalarlo; para ello en nuestra interfaz de *Eclipse* nos dirigimos a *Help* > *Eclipse Marketplace*.

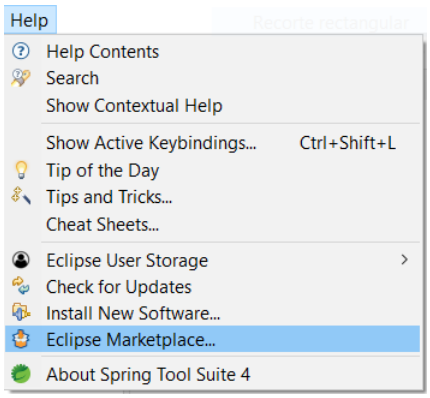

Figura 2.1-7 Ruta de Eclipse Marketplace.

Una vez en *Marketplace*, haciendo uso del buscador obtenemos *Spring Tools 4* (Figura 2.1-8).

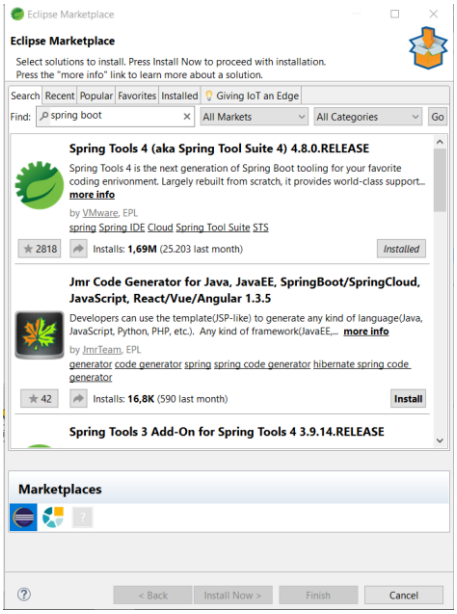

Figura 2.1-8 Integración de Spring Tools 4 con Eclipse.

Tras aceptar los términos de uso, Eclipse se reiniciará, y una vez se cargue, ya estará todo listo para utilizar Spring Tools Suite 4.

2.2 JAVA

Java es un lenguaje de programación además de un sistema informático que fue desarrollado por Sun Microsystems en 1995, y en 2010 la empresa fue adquirida por Oracle.

Figura 2.2-1 Logotipo de Java.

Java no fue creado originalmente para la red Internet, sino que Sun Microsystems comenzó a desarrollarlo con el objetivo de crear un lenguaje, independiente de la plataforma y el sistema operativo en el que se ejecute.

El proyecto original, denominado *Green*, comenzó apoyándose en C++. A medida que el proyecto avanzó el equipo creador de Green comenzó a afrontar dificultades relacionadas con la portabilidad.

Para evitar dichas dificultades se comenzó a desarrollar un nuevo lenguaje, y en 1991 Green fue renombrado a *Oak*, para luego en 1993 volver a cambiar de nombre por el de *First Person Juc* donde Sun Microsystems invirtió una gran cantidad de presupuesto y esfuerzo humano para vender dicha tecnología sin cosechar mucho éxito.

A mediados de 1993 nació *Monsaic*, que fue el primer navegador para la Web, lo que provocó un aumento en el interés por Internet. Fue entonces cuando se rediseñó el lenguaje y Oak pasó a llamarse *Java*.

Sun Microsystems lanzó el entorno JDK 1.0 en 1996, convirtiéndose en la primera especificación formal de la plataforma, y desde entonces se ha continuado con el lanzamiento de nuevas versiones de este.

2.2.1 ¿Qué es Java? Introducción a Java

Java es un lenguaje de programación y entorno para la ejecución de programas escritos en este mismo lenguaje. Mientras que los compiladores tradicionales convierten el código fuente en instrucciones a nivel máquina, el compilador Java traduce el código fuente Java en instrucciones interpretadas por la Máquina Virtual Java (JVM) (Figura 2.2-2).

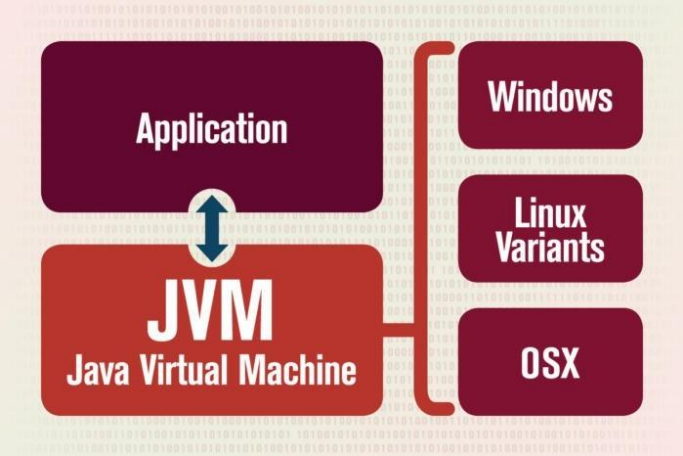

Figura 2.2-2 Funcionamiento de JVM.

Este lenguaje es totalmente portable a una gran variedad de plataformas hardware y sistemas operativos, utiliza muchos conceptos sintácticos de C y C++. A este último añade propiedades de gestión automática de memoria y soporte a nivel de lenguaje para aplicaciones multihilo.

Sin embargo, Java es más fácil de aprender y de utilizar que C++, ya algunas características de C++ como la herencia múltiple y el uso de punteros han sido suprimidos.

La implementación de la JVM es muy eficaz y hace que los programas escritos en Java se puedan ejecutar tan rápido como los programas escritos en C++. Esto, junto a sus fortalezas como lenguaje en Internet, convierte a Java en un lenguaje idóneo para desarrollos en sistemas cliente/servidor.

Java es un lenguaje orientado a objetos, esto quiere decir que cualquier cosa que se nos pase por la mente puede ser modelada como tal. Un coche, una persona, un animal, todo puede ser modelado como un objeto. Un programa escrito en Java se caracteriza por la manipulación y construcción de objetos [6].

La programación orientada a objetos se basa en cuatro pilares (o fundamentos), que son:

- *Abstracción*: esta propiedad es esencial en la programación orientada a objetos. El ser humano gestiona la complejidad mediante la abstracción, así cuando pensamos en una *bicicleta*, no la vemos como un conjunto de elementos, sino como un objeto definido y con un comportamiento determinado. La abstracción nos permite hacer uso de esta *bicicleta* sin tener que preocuparnos sobre cómo funciona internamente.

- *Encapsulamiento*: es el proceso de ocultar los detalles de un objeto que no contribuyen a su funcionamiento. Esto quiere decir que lo que se encuentra dentro de una clase se encuentra oculto; el usuario nunca necesitará conocer el interior de la clase. Si el usuario desea interactuar con el objeto *bicicleta*, tendrá que hacerlo mediante el uso de los métodos Get/Set del objeto *bicicleta* (Ver Figura 2.2-3).

```
3  public class Bicicleta {
4      private String id;
5      private String marca;
6      private String modelo;
7      private int marchas;
8
9      public Bicicleta(String id, String marca, String modelo, int marchas) {
10         super();
11         this.id = id;
12         this.marca = marca;
13         this.modelo = modelo;
14         this.marchas = marchas;
15     }
16
17     public String getId() {
18         return id;
19     }
20     public void setId(String id) {
21         this.id = id;
22     }
23     public String getMarca() {
24         return marca;
25     }
26     public void setMarca(String marca) {
27         this.marca = marca;
28     }
```

Figura 2.2-3 Métodos Get/Set de Bicicleta.

- *Herencia*: es el proceso mediante el cual un objeto obtiene (hereda) las propiedades de otro. Supongamos que la interfaz *vehículo* tiene como atributos matricula, marca y modelo. Una clase que herede de *vehículo* (como por ejemplo la clase *coche*), heredará los atributos de la clase *vehículo* y tendrá otros atributos propios de la clase *coche* como por ejemplo el número de puertas y la potencia.

Lo mismo ocurriría con la clase *moto*, heredaría los atributos de *vehículo* y tendría atributos propios de la clase *moto* como el peso o la marcha en la que se encuentra, además de los métodos propios de una *moto*. (Ver Figura 2.2-4).

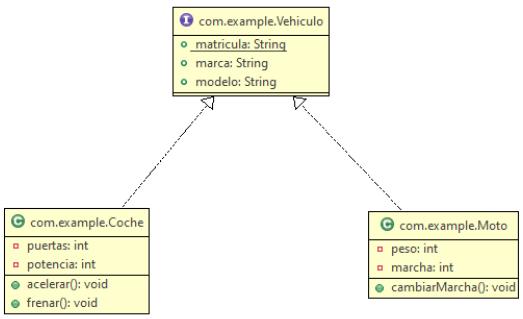

Figura 2.2-4 Concepto de *herencia*.

- *Polimorfismo*: es la capacidad que tienen los objetos de una clase en devolver una respuesta u otra en función de los parámetros que se utilizan al ser llamados.

Durante más de dos décadas, Java ha acumulado una gran cantidad de librerías que contienen código reutilizable, por lo que podremos ahorrar una gran cantidad de tiempo utilizándolas.

2.3 GRAPHQL

GraphQL es un lenguaje de consulta utilizado para comunicar clientes y servidores, creado por Facebook en 2012, y actualmente usado en muchas apps para smartphones, sitios webs y APIs. Este lenguaje se usa en varias aplicaciones conocidas, como la propia app de Facebook, Pinterest, Github o Twitter, entre otras.

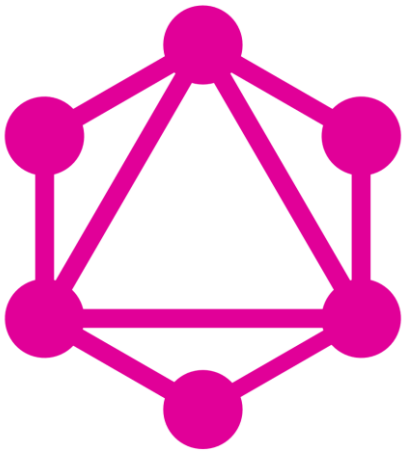

Figura 2.3-1 Logotipo de GraphQL.

Proporciona una aproximación para el desarrollo de APIs web y ha sido contrastado con REST y otras arquitecturas de servicio web. Permite definir la estructura de datos requerida, la cual será devuelta por el servidor, impidiendo que excesivas cantidades de datos sean devueltas.

Los servidores de GraphQL estan disponibles para múltiples lenguajes, como Javascript, Perl, Python, Ruby, Java… [7].

2.3.1 Principios de GraphQL

GraphQL se presenta como alternativa (o complemento) a REST[16] para crear una interfaz de servicio basado en la comunicación utilizando protocolos web HTTP y JSON. Las principales características de GraphQL son:

- Utilización de un esquema en el que se definen los recursos y tipos de datos que maneja la API.
- El usuario especifica que datos quiere exactamente del servidor, evitando el temido overfetching[17] y enviando solo los datos que el usuario solicita.
- Hace uso de un único endpoint *(/graphQL)* y solo acepta peticiones de tipo POST.
- Es una herramienta que no depende de la fuente de datos, esto quiere decir que los datos pueden obtenerse desde un fichero JSON, una base de datos o incluso un archivo de texto plano.

[16] Estilo de arquitectura software distribuidos en Internet.
[17] Efecto que se produce al llamar a un endpoint y este devuelve una gran cantidad de datos que luego no se utilizan.

- Ofrece soporte para una amplia variedad de lenguajes, como Java, Javascript o Ruby, entre otros.

Un servicio que funciona con GraphQL, recibe queries y las resuelve devolviendo los datos solicitados.

2.3.2 Tipos de datos

Este lenguaje de consulta se basa en la utilización de esquemas para la definición de los tipos de datos que soporta la API. Dentro del esquema se definen:

- *Tipos* (*Types*): son objetos o entidades que contienen una serie de atributos.
- *Queries*: son el punto de entrada para realizar las consultas.
- *Mutaciones* (*Mutations*): es un tipo especial de punto de entrada para realizar modificaciones como crear, eliminar o editar elementos. Un sistema que implemente GraphQL pero que no contemple la modificación de elementos puede prescindir del uso de mutaciones.
- *Inputs*: se utilizan para la resolución de queries y mutaciones, consiste en la definición de datos que el sistema espera por parte del usuario.
- *Interfaces*: es un tipo abstracto que especifica una serie de campos que se deben incluir si un objeto implementa dicha interfaz.
- *Enumeration Types* (o simplemente *Enums*): son un tipo especial de valor escalar que está limitado a un conjunto de valores permitidos.

Cuando definimos un objeto, debemos especificar los campos de dicho objeto, lo que nos lleva a los tipos de datos escalares que soporta GraphQL, que son los siguientes:

- *Int*: es un valor entero de 32 bits.
- *Float*: es un valor de coma flotante de doble precisión.
- *String*: es una cadena de texto UTF-8[18].
- *Boolean*: puede ser verdadero o falso.
- *ID*: es un tipo que representa un identificador único, que suele ir acompañado de "!" indicando que no puede cobrar el valor *null*.

Además de estos tipos de datos, GraphQL también soporta la creación de tipos personalizados cuya validación y tratamiento dependen de cómo los implementemos [8].

2.3.3 Ejemplo de uso de GraphQL

[18] Formato de codificación de caracteres Unicode que utiliza símbolos de longitud variable.

Una vez explicado el funcionamiento de los esquemas y los tipos que GraphQL soporta, a continuación, se muestra, a modo de ejemplo, un esquema GraphQL donde se define el objeto tipo *CountryState* con los campos que componen el objeto (Figura 2.3-2).

```
type CountryState{
    idcs: ID!,
    country:String,
    countryCode:String,
    province: String,
    city : String,
    cityCode: String,
    lat: Float,
    lon : Float,
    cases: Int,
    status: String,
    date: String
}

input FechasInput{
    lat: Float,
    lon: Float,
    status: String,
    date: String,
    date2: String
}
type Query{
    obtenerDatosLatLonQL(fechasInput: FechasInput) : [CountryState]
```

Figura 2.3-2 Ejemplo de esquema GraphQL.

Además de la definición del objeto *CountryState*, también se define la entrada de datos que se espera por parte del usuario (*FechasInput)*. Dentro de *Type Query* se definen los puntos de acceso, donde se espera un input del tipo *FechasInput*, el cual devolverá una lista de objetos tipo *CountryState*.

Haciendo uso de la herramienta Postman, accediendo por el punto de acceso */graphql* podemos implementar una query para la consulta sobre las muertes ocasionadas por el Coronavirus entre dos fechas propuestas en un país en concreto o en función de la latitud y longitud en la que se encuentra, tal y como se muestra en la Figura 2.3-3.

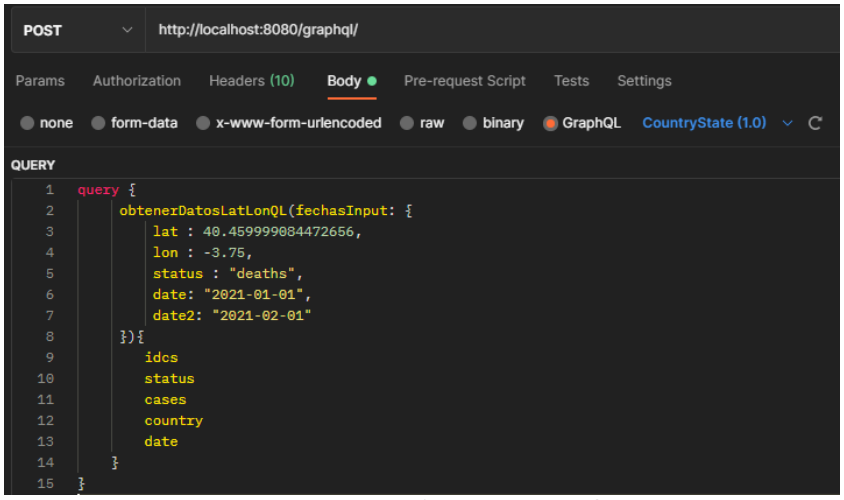

Figura 2.3-3 Muertes por Coronavirus en España entre enero y febrero de 2021 con GraphQL.

Tras especificar la entrada del usuario (*fechasInput*), indicamos cuáles son los datos que queremos que el servidor devuelva, esto es, el identificador, el tipo de caso, el número de casos, el nombre del país y la fecha. El resultado es devuelto en formato JSON tal y como se muestra en la Figura 2.3-4.

Figura 2.3-4 Respuesta del servidor.

Cabe destacar que si no se especifica en el esquema la entrada que se espera por parte del usuario (*fechasInput*), la API funcionaría de forma correcta, pero tendríamos que establecer múltiples puntos de acceso para las distintas consultas que soporta la API, lo cual no tendría sentido si queremos aprovechar todas las ventajas que ofrece GraphQL.

2.3.4 Alternativas a GraphQL. Ventajas e inconvenientes

Las APIs más populares de hoy día utilizan REST, sin embargo, GraphQL se presenta como un fuerte candidato a sustituir REST debido a sus cualidades.

Principalmente cuando utilizamos REST accedemos por una URL o punto de acceso para leer o escribir un recurso, sin embargo, cuando necesitamos trabajar con múltiples recursos esto se traduce en manipular varios puntos de acceso al mismo tiempo, encadenando llamadas de manera secuencial, obteniendo como respuesta todos los datos, sin opción a definir qué es exactamente lo que buscamos.

Con GraphQL se solventa este problema ya que cada vez que se usa el punto de acceso */graphql* especificamos cuales son los datos que necesitamos, pues existe la posibilidad de que no siempre necesitemos toda la información que nos ofrece un punto de acceso.

GraphQL no solamente nos permite definir qué datos queremos, sino también cómo los queremos obtener. Así, por ejemplo, si queremos obtener ciertos datos en un orden en específico, podemos hacerlo, lo que facilitaría su posterior tratamiento.

Una API REST implementa el almacenamiento en caché evitando así que el cliente busque esos recursos, mientras que GraphQL deja esa responsabilidad a los clientes [9].

En la parte de manejo de errores, podemos saber el estado de la respuesta mediante los códigos de estado HTTP, mientras que en GraphQL obtenemos un código 200 —suponiendo que todo funcione correctamente—; en caso de error obtendríamos un fallo en el procesamiento de la consulta (Figura 2.3-5).

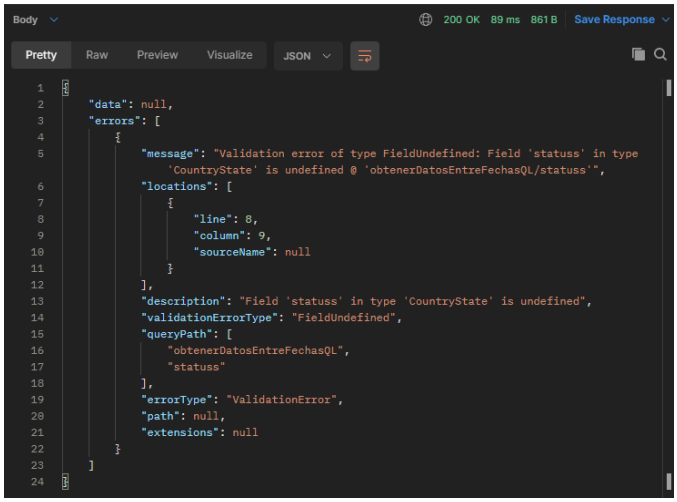

Figura 2.3-5 Ejemplo de fallo en el procesamiento de la consulta.

2.4 H2 DATABASE

2.4.1 ¿Qué es H2 Database? Introducción a H2 Database

H2 Database es un motor de base de datos de código abierto implementado en Java, cuyo desarrollo comenzó en 2004, pero realmente no se publicó hasta un año después. Su autor original fue Thomas Mueller, la misma persona que desarrolló Hypersonic SQL.

Alguno de los motivos por los que he decidido utilizar este motor, es debido a que al estar implementado en Java, podemos integrarlo fácilmente para usarlo en nuestra API [10].

Figura 2.4-1 Pasos para acceder a la base de datos H2.

Otra de las ventajas que ofrece H2 es que soporta tanto el modo embebido como el modo en memoria, acelerando enormemente las operaciones realizadas con ella. Cuando la configuremos podemos establecer un enlace de entrada desde nuestro navegador, en el que mediante un usuario y una contraseña podemos acceder directamente al gestor de base de datos y operar mediante comandos SQL sobre ella.

2.4.2 Uso de la consola H2

Para utilizar la consola H2 accedemos a http://localhost:8080/h2-console/, y mostrará una interfaz similar a esta:

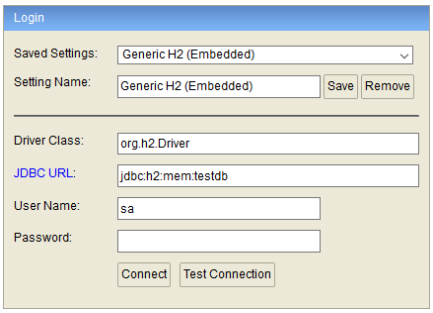

Figura 2.4-2 Login de H2 console.

Al introducir el usuario y la contraseña, ya estaríamos dentro de la base de datos.

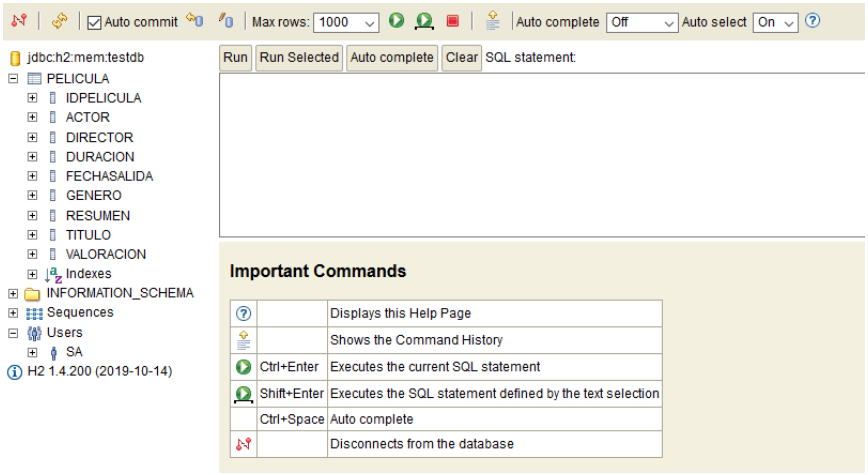

Figura 2.4-3 Interfaz de H2 console.

Una vez dentro podemos ver el contenido de las tablas de nuestra base de datos, donde podemos ejecutar sentencias SQL en el cuadro de texto y obtendremos respuesta cuando le demos a *Run*.

2.5 EL PROTOCOLO HTTP

El protocolo HTTP[19] define los métodos de petición para indicar que es lo que se desea hacer con un recurso determinado.

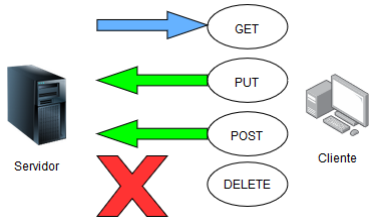

Figura 2.5-1 GET POST PUT DELETE

La definición de dichos métodos es la siguiente:

- *HTTP GET:* obtiene un recurso del servidor.
- *HTTP POST:* crea un recurso del servidor.
- *HTTP PUT:* actualiza un recurso del servidor.

[19] Acrónimo de Hypertext Transfer Protocol.

- *HTTP DELETE:* elimina un recurso del servidor

Aunque parezca que la diferencia entre POST y PUT es que uno se utiliza para crear un recurso y el otro para actualizarlo, esto no es así realmente, pues ambos pueden crear un recurso y actualizarlo.

Realmente PUT reemplaza un recurso en la URL que nosotros especifiquemos, si no existiera lo crea, mientras que POST es algo más genérico, simplemente envía datos a la URL para que los maneje como ella quiera.

Existen más métodos aparte de los citados arriba, como por ejemplo OPTIONS que establece las opciones de comunicación para un recurso, o PATCH para aplicar modificaciones parciales a un recurso, pero realmente los que más se utilizan son los que ya se han descrito anteriormente [11].

2.6 POSTMAN

2.6.1 ¿Qué es Postman?

Postman es una herramienta que se usa para el desarrollo de APIs en equipo, soporta una amplia variedad de lenguajes como REST, SOAP o GraphQL, permite la creación de distintos espacios de trabajo, además de monitorear los tiempos de respuesta de nuestra API y documentarla.

Esta herramienta es gratuita, y para empezar a trabajar con ella simplemente hay que crearse una cuenta que vinculará todo nuestro trabajo a una dirección de correo, por lo que si tenemos que cambiar de equipo nuestro trabajo quedará vinculado a la cuenta.

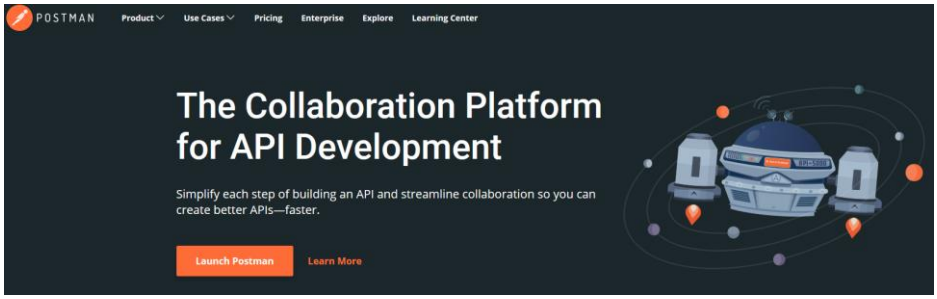

Figura 2.6-1 Página de inicio de Postman.

Para continuar trabajando tendremos que instalar Postman en otro dispositivo y loguearnos con la dirección de correo, esta herramienta también dispone de una versión web que carga desde nuestro navegador por lo que no sería ni necesario su instalación.

2.6.2 Instalación de Postman

Para instalar Postman tenemos que dirigirnos a la dirección *https://www.postman.com/downloads/* y descargar la versión más reciente

Figura 2.6-2 Descargar Postman.

La versión web de Postman es exactamente igual que la versión App, así que da igual la que usemos, yo voy a descargar Postman App, como podemos observar tiene soporte multiplataforma así que no habrá ningún problema a la hora de instalarlo.

Para instalarlo simplemente hacemos doble click en el archivo de instalación, y aparecerá una imagen indicando que la instalación se está llevando a cabo.

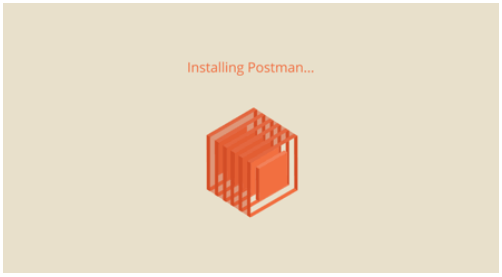

Figura 2.6-3 Instalación de Postman.

Tras esto la instalación habrá terminado y Postman se ejecutará de forma automática.

2.6.3 Primeros pasos en Postman: Creando una cuenta y un Workspace

Aunque no es obligatorio, es recomendable crear una cuenta de la que solo nos pedirán un email, ya que con ella podremos tener acceso a nuestro trabajo desde cualquier otro dispositivo. Para ello, pinchamos en *Create Account* y seguimos los pasos.

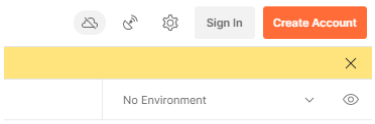

Figura 2.6-4 Creando una cuenta de Postman.

Una vez creada la cuenta, nos logueamos con ella y ya podremos crear nuestro Workspace[20], para ello pinchamos en Workspace y a *New Workspace*.

[20] Es el espacio de trabajo que contendrá todo lo relacionado con el desarrollo de la *API*

Figura 2.6-5 Creación de un nuevo Workspace.

Escribimos un nombre para el Workspace, una descripción y en visibilidad lo configuramos en Personal si vamos a trabajar en solitario, o si vamos a trabajar con un equipo seleccionamos *Team*.

Una vez creado el Workspace ya podemos comenzar a trabajar con Postman, en la barra de tareas izquierda tendremos una columna con múltiples opciones, donde cabe destacar:

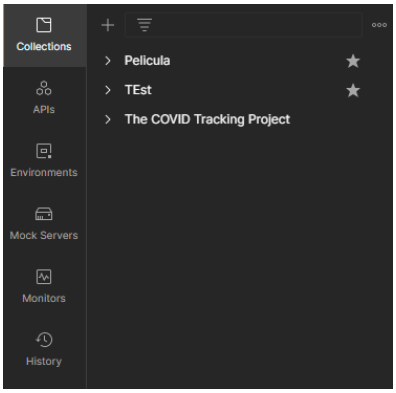

Figura 2.6-6 Interfaz de Postman.

- *Collections:* Aquí será donde almacenaremos todas nuestras carpetas con las diferentes requests.

- *APIs:* Aquí almacenaremos los esquemas de nuestras APIs, en este caso de la API de Películas que será el proyecto de ejemplo.

- *Environments:* Se utiliza para crear variables para utilizarlas en nuestras requests, esto nos será muy útil a la hora de autenticar las requests de forma automática más adelante.

- *Mock Servers:* Esta opción nos permite utilizar endpoints y obtener respuesta sin tener ningún servidor montado.

- *Monitors:* Se utiliza para monitorear que nuestra *API* se comporta de forma correcta, haciendo pruebas periódicamente y midiendo los tiempos de respuesta.

- History: Nos servirá para ver el historial de todas las request que hemos hecho, es una forma rápida de acceder a las requests si no queremos navegar por las carpetas de Collections.

2.6.4 Cargando la API en Postman

En la barra de tareas vamos a APIs, pinchamos en el icono + y creamos una nueva API, nos pedirá algunos parámetros como el nombre, la versión y el tipo de esquema que vamos a usar, la configuración es la siguiente:

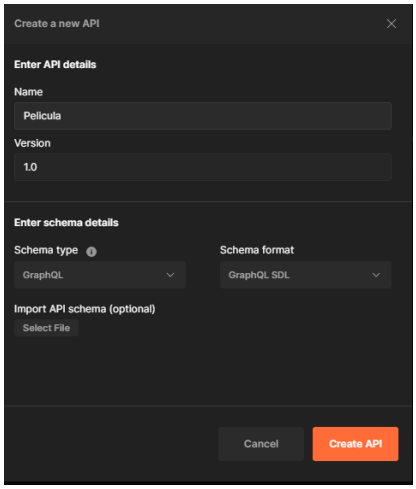

Figura 2.6-7 Creando la API de Películas en Postman.

También podemos importar el archivo directamente, ambas opciones funcionan de la misma manera. Cuando la creemos se nos abrirá una nueva interfaz y pulsamos la opción *Define* para empezar a implementar nuestra API, vamos a usar la misma que cargamos desde Eclipse.

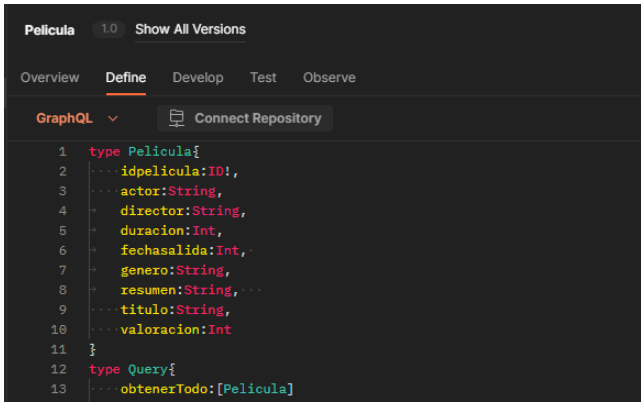

Figura 2.6-8 Definiendo la API Película en Postman.

Una vez hecho esto, se almacenará en el apartado APIs de la barra de tareas y tendremos acceso siempre por si queremos modificar algo de ella, ahora vamos a las Collections para crear las requests.

2.6.5 Creando Collections para organizar las Queries.

En la barra de tareas vamos a Collections y pinchamos en + para añadir una nueva, en mi caso crearé una que se llamará Película, donde nos pedirá el nombre:

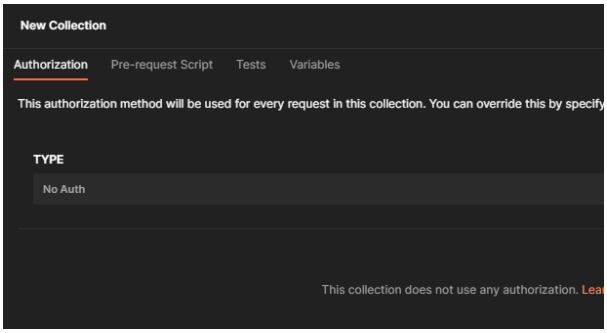

Figura 2.6-9 Creando una nueva Collection en Postman.

En nuestra Collection tendremos varias pestañas, estas son:

- *Authorization:* Si tenemos un método de autorización, como puede ser JWT, podemos configurarlo en esta pestaña para que todas las requests que hagamos desde la Collection Película se puedan autorizar. Para utilizar JWT, debemos crear una request para autenticarnos indicando un nombre de usuario y una contraseña, el servidor nos devolverá un token JWT que debemos pegar manualmente en cada request que queramos realizar. En el apartado 2.6.7 muestro una forma de que este proceso de autenticación se realice de forma automática desde Postman.

- *Pre-Request Script:* Esto es una caja de texto en la que podremos escribir un Script para que se ejecute antes de cada request de esta Collection.

- *Tests:* Son tests que se ejecutarán después de cada ejecución de una request de esta Collection.

- *Variables:* Permite la creación de variables específicas de esta Collection para utilizarla en las requests.

Una vez explicado todo, vamos a crear una serie de carpetas dentro de nuestra Collection, estas serán:

- *Mutations:* Donde almacenaremos las request para crear, eliminar o modificar una película.

- *Queries:* Donde se almacenarán las requests que van a utilizar GraphQL para obtener los datos.

- *URL:* Donde se guardarán las requests que pasarán por URL los parámetros necesarios para la query.

- *seguridadTest:* Aquí se guardará principalmente la request para autenticarse utilizando el token JWT.

El resultado de crear dichas carpetas tendrá un resultado similar a este:

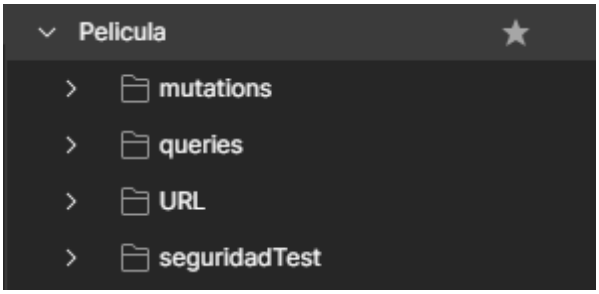

Figura 2.6-10 Creación de carpetas en Postman.

2.6.6 Autenticación JWT

Para mayor seguridad vamos a autenticarnos utilizando JWT, para ello vamos a crear una carpeta que llamaremos *seguridadTest* y dentro vamos a crear el siguiente endpoint:

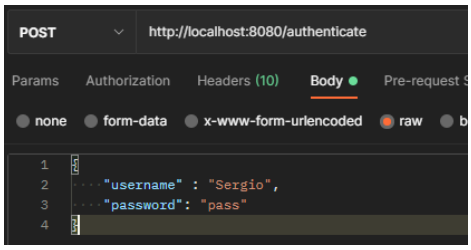

Figura 2.6-11 Creación de endpoint /authenticate.

Dentro del body del endpoint vamos a especificar el usuario y contraseña que se esperan, al ejecutar esta request Java se encargará de devolvernos el token JWT:

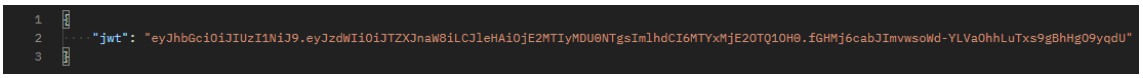

Figura 2.6-12 Resultado de /authenticate.

Si ahora intentamos ejecutar cualquier endpoint de nuestra API no obtendremos respuesta del servidor, esto es porque para cada request el servidor está esperando una etiqueta del tipo *Authorization* con el JWT.

Ahora lo que tenemos que hacer es ir a *Headers*, crear la etiqueta *Authorization* y dentro pegar el token JWT:

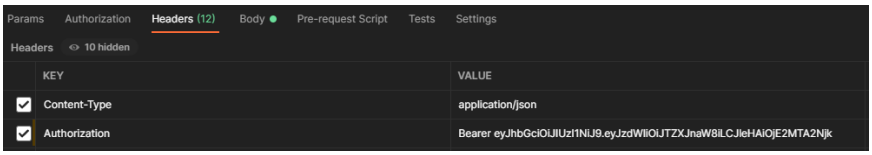

Figura 2.6-13 Etiqueta Authorization con el token JWT.

Una vez que creemos la etiqueta y peguemos dentro la cadena de JWT ya obtendríamos respuesta por parte del servidor.

Este proceso habría que hacerlo cada vez que queramos realizar un request, hay que tener en cuenta que el token JWT tiene una validez limitada, si dejamos pasar mucho tiempo habría que volver a hacerlo.

Esto quiere decir que tendríamos que estar usando el endpoint */authenticate* prácticamente todo el rato si queremos comprobar que todo funciona correctamente, lo cual puede ser bastante tedioso.

No obstante, Postman soporta *Environments* y *variables globales*, por lo que en el siguiente apartado vamos a ver cómo configurar *Postman* para que solamente tengamos que ejecutar el endpoint /authenticate una única vez y que nuestras requests se autentifiquen ellas solas de forma automática sin que tengamos que estar copiando y pegando manualmente la cadena JWT.

2.6.7 Automatizando el proceso de autenticación con Postman

Para automatizar el proceso de autenticación, nos dirigimos a la request que se encarga de obtener el token JWT, y vamos a la pestaña *Tests* donde tendremos que escribir el siguiente código:

Figura 2.6-14 Uso de Tests para la autenticación automática.

Con este código estamos creando una variable global *jwt_token* donde se va a almacenar la cadena de texto que teníamos que introducir en la etiqueta *Authorization*.

En la interfaz de Postman en la zona superior de la derecha, podemos ver un símbolo de un ojo, si le pinchamos podremos ver las variables globales que hay definidas en Postman (si queremos que esta cobre el valor de la cadena JWT, habría que ejecutar */authenticate*):

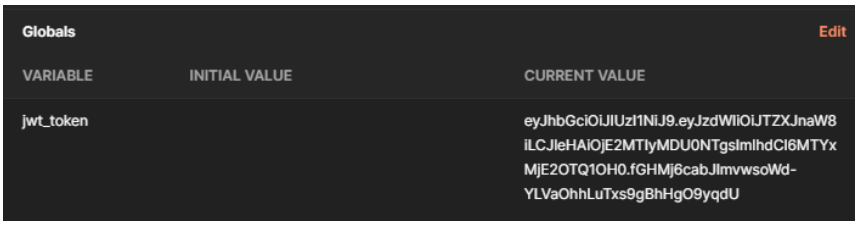

Figura 2.6-15 Creación de variables globales en Postman.

Como he explicado arriba, si queremos que la variable tome el valor de la cadena JWT habría que ejecutar la request */authenticate*, esto solo habría que hacerlo la primera vez.

El siguiente paso es editar cada una de las request, salvo la request */authenticate*, donde se abrirá una pestaña llamada *Authorization*, hay que introducir la siguiente configuración:

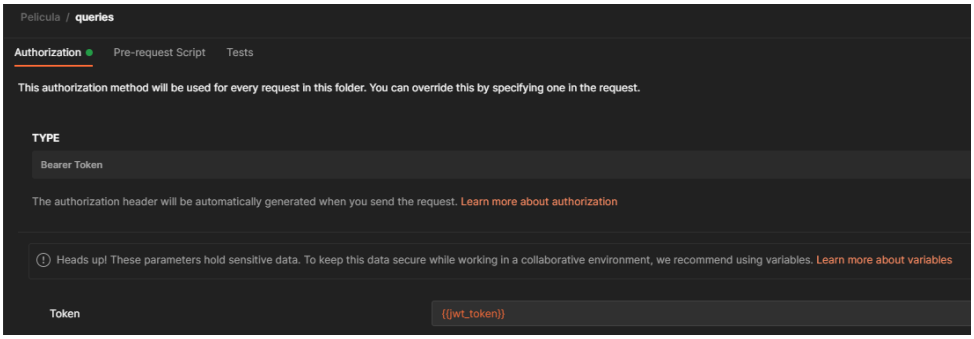

Figura 2.6-16 Figura Autenticación automática con Postman.

Con esta configuración estamos especificando que vamos a usar un *Bearer Token*, y que la variable que lo contiene es *jwt_token*, que es nuestra variable global que creamos previamente en el script de *Tests*, una vez hecho esto la autenticación se realizaría de forma automática y no tendríamos que copiar y pegar la cadena JWT.

2.6.8 Documentación en Postman

Postman nos ofrece la posibilidad de documentar nuestra API, para ello en la barra de tareas la derecha tenemos que pinchar en el icono del documento, donde se nos abrirá la pestaña de documentación:

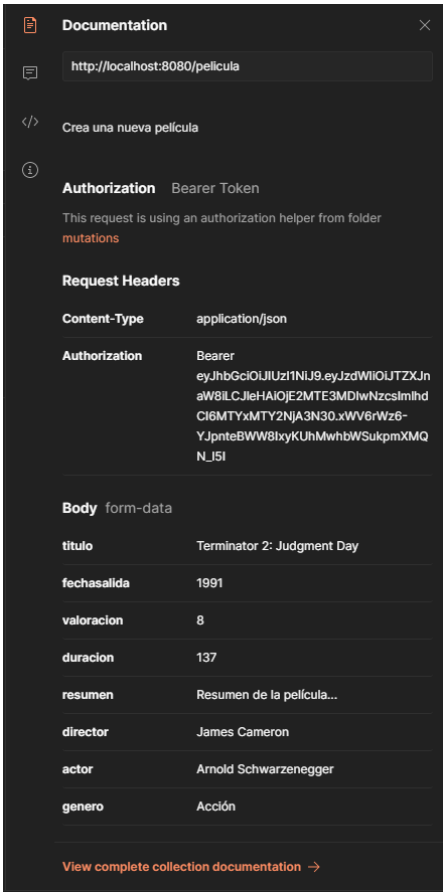

Figura 2.6-17 Abrir documentación en Postman.

Ahora hacemos click donde dice *View complete collection documentation*, donde podremos documentar cada una de las request que tenemos organizadas en carpetas.

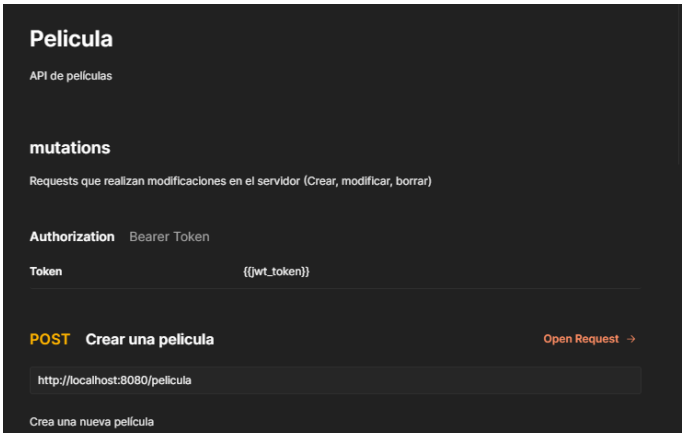

Figura 2.6-18 Documentando la API Películas.

2.7 INSOMNIA

2.7.1 ¿Qué es Insomnia?

Insomnia es una herramienta para desarrollar, testear y debuggear APIs, disponemos de dos versiones de Insomnia para usar:

- Insomnia Designer: Es una herramienta que se centraen el desarrollo colaborativo de APIs.

- Insomnia Core: Es el cliente clásico de Insomnia, para realizar requests e inspeccionar la respuesta de los endpoints.

En este proyecto se va a usar la versión Core.

2.7.2 Instalación de Insomnia Core

Para instalar Insomnia Core nos dirigimos a https://insomnia.rest/download/#windows , Insomnia es multiplataforma por lo que si utilizamos un sistema operativo distinto a Windows, podremos instalarlo de igual manera.

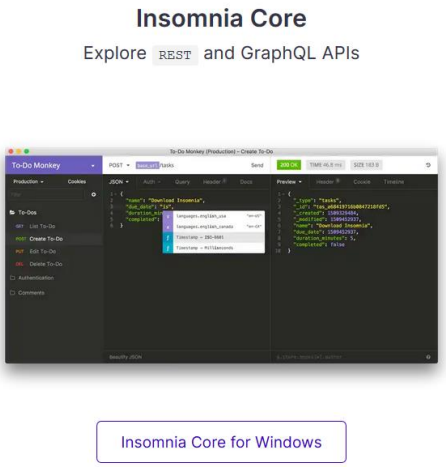

Figura 2.7-1 Descargando Insomnia Core.

Cuando lo tengamos descargado, simplemente iniciamos el instalador siguiendo los pasos y lo tendremos instalado.

2.7.3 Primeros pasos en Insomnia Core: Creando una cuenta.

Al terminar la instalación se abrirá de forma automática, para crear una cuenta vamos a donde pone Insomnia y a login:

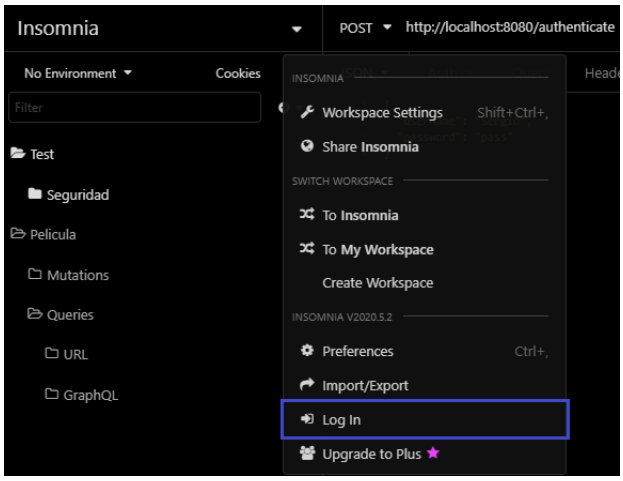

Figura 2.7-2 Creando una cuenta en Insomnia Core.

Esto nos redigirá a una ventana en el navegador donde podremos loguearnos si ya tenemos cuenta, si es la primera vez que usamos Insomnia aparecerá un cuadro de diálogo sugiriendo crear una cuenta, le pinchamos y la creamos.

2.7.4 Creando la carpeta del proyecto

Para crear una carpeta tenemos que pulsar en el signo + donde dice Insomnia:

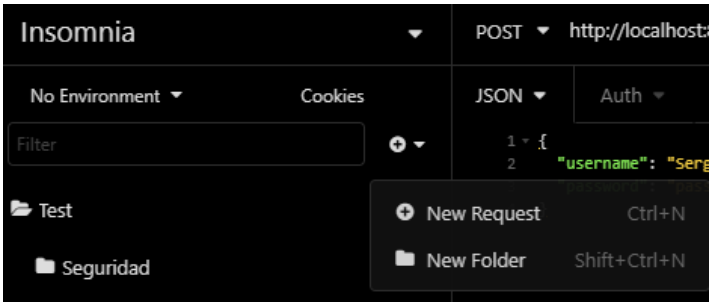

Figura 2.7-3 Creando carpetas en Insomnia.

Vamos a seguir una organización de carpetas similar a la que seguimos en Postman, cuando creemos las carpetas el resultado debe ser como el que muestro a continuación:

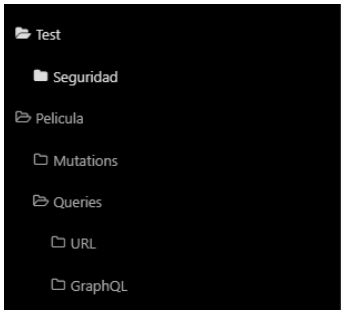

Figura 2.7-4 Estructura de carpetas en Insomnia.

Una vez creadas las carpetas, tendríamos que crear las requests de la misma manera que hicimos con Postman. En la carpeta *Seguridad* tenemos que crear la request para poder autenticarnos.

2.7.5 Automatizando el proceso de autenticación con Insomnia

Para facilitar la autenticación de nuestras requests, necesitamos instalar un plugin en Insomnia tal y como voy a mostrar a continuación.

El plugin se llama AWS cognito token, y para instalarlo tenemos que ir a *Application* > *Preferences* > *Plugins*:

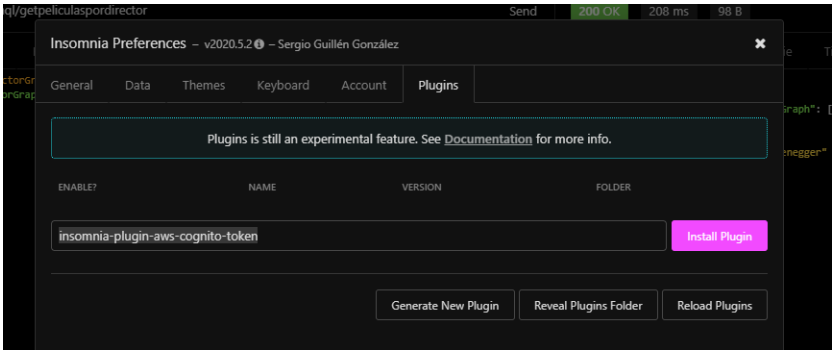

Figura 2.7-5 Instalación del Plugin AWS cognito Token.

Una vez hecho esto pulsamos en *Install Plugin* y ya lo tendremos instalado. Para configurarlo vamos a *No Environment* > *Manage Environments* > *Sub Environments* > *Private Environment*, aquí vamos a crear las variables globales para definir el usuario y la contraseña.

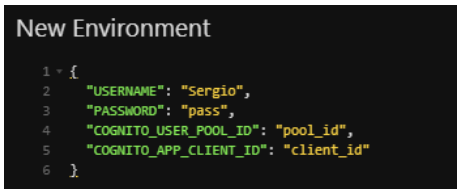

Figura 2.7-6 Creación de private environment.

Ahora tendremos que configurar el plugin:

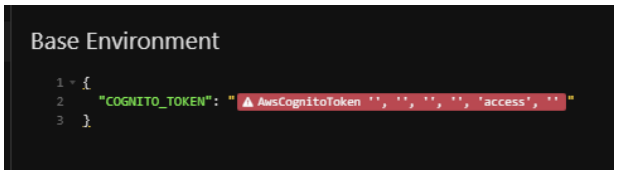

Figura 2.7-7 Configurando AWS cognito Token.

Si hacemos click en la advertencia de color rojo, se nos abrirá un formulario para ir rellenando con datos como el usuario y la contraseña, desafortunadamente al terminar la configuración el plugin no funciona correctamente, se ha intentado volver a configurar pero no se ha conseguido hacer funcionar debido a un problema con la nueva versión del plugin.

Aun así es posible trabajar con Insomnia, pero cada vez que se realice una request será necesario copiar y pegar manualmente el token JWT para autenticarnos.

2.7.6 Documentando con Insomnia

Para documentar hacemos click derecho en una request, y vamos a settings, entonces se nos abrirá un menú como el que voy a mostrar ahora donde podemos añadir una descripción de que es lo que hace cada request:

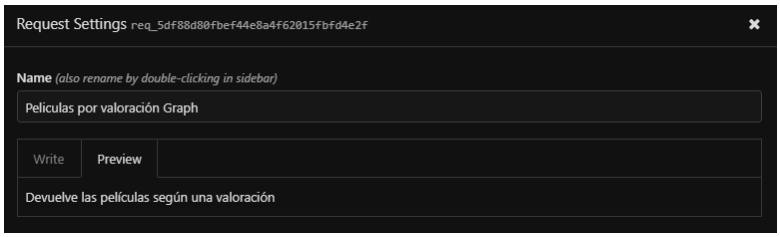

Figura 2.7-8 Documentando Requests con Insomnia.

2.8 GRAPHIQL

2.8.1 ¿Qué es GraphiQL?

GraphiQL al contrario de Postman o Insomnia, no es una herramienta para testear *APIs* como tal, sino que es una herramienta para testear queries y mutaciones en GraphQL desde nuestro navegador, no requiere de instalación y tiene una interfaz bastante simple y fácil de entender.

Al comienzo de este proyecto se ha utilizado para testear que las queries funcionasen de forma correcta, para hacerlo funcionar simplemente tendremos que añadir una dependencia a nuestro pom.xml, si se ha creado el proyecto haciendo uso de Spring Initializr no hace falta añadir nada porque ya viene añadida de serie, la dependencia en cuestión es:

```
<dependency>
        <groupId>com.graphql-java</groupId>
        <artifactId>graphiql-spring-boot-starter</artifactId>
        <version>5.0.2</version>
    </dependency>
```

Figura 2.8-1 Dependencias necesarias para utilizar GraphiQL.

Para acceder a GraphiQL, escribimos en el navegador *localhost:8080/graphiql* y se nos abrirá una interfaz similar a esta:

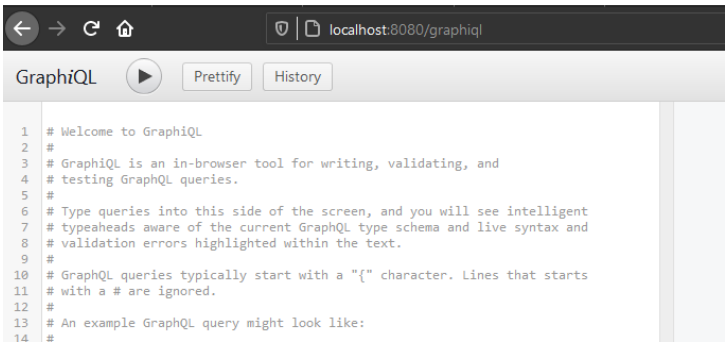

Figura 2.8-2 Interfaz de GraphiQL.

Como podemos observar la interfaz es muy simple, donde tenemos un cuadro de texto para escribir nuestras queries, un botón para ejecutarla, un botón *Prettify* para volver a escribir automáticamente la última query que había escrita y un botón de historial.

En adición a esto, a la derecha existe otro botón llamado *Docs* que si lo pulsamos abrirá un menú desplegable donde podemos consultar los tipos de queries y mutaciones que soporta nuestro esquema.

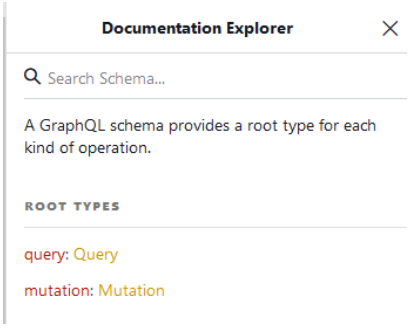

Figura 2.8-3 Menú desplegable de GraphiQL.

GraphiQL toma nuestro esquema directamente del directorio *src/main/resources/graphql* de nuestro proyecto, por lo que no hace falta indicarle donde lo tenemos guardado, automáticamente lo carga desde ahí.

Si por ejemplo abrimos *Query* o *Mutation* dentro de *Documentation Explorer*, obtenemos una vista rápida de cada una de las queries que hay listas para consultar.

Figura 2.8-4 Menú desplegable Query.

Esto mismo ocurre si lo hacemos con mutaciones, GraphiQL tiene función de autocompletado por lo que simplifica bastante el proceso de escribir las queries.

Cuando implementamos las queries a veces es complicado saber qué es lo que hace dicha query, pero si las comentamos en el esquema utilizando "#", GraphiQL lo mostrará en el menú de *Documentation Explorer* al lado de cada query, por lo que facilita la documentación.

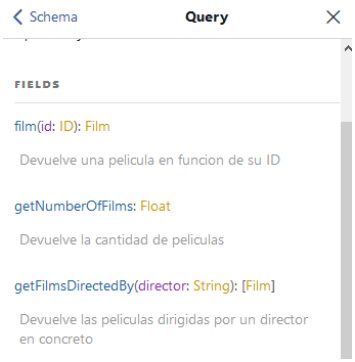

Figura 2.8-5 Descripción de Queries en GraphiQL.

La descripción de las queries también funciona mientras la implementamos, cuando hagamos autocompletar nos mostrará la descripción.

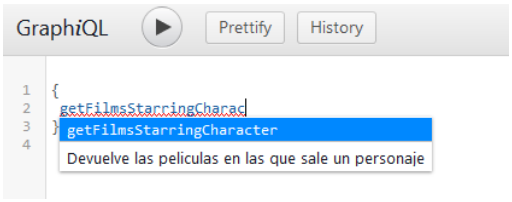

Figura 2.8-6 Función de autocompletar con descripción en GraphiQL.

Un aspecto de GraphiQL que no me gusta respecto a otras herramientas de testeo de APIs (aunque GraphiQL solamente nos sirva para testear las queries, y no los endpoints por ejemplo), es que no hay forma de organizar las queries y mutaciones en carpetas u organizarlas de otra forma.

Sin embargo, dispone de un historial que muestra todas las queries que se han implementado, aun así hubiera sido preferible que tuviera una carpeta para almacenarlas o una forma mejor de gestionarlas como ya hacen otras herramientas como Postman o Insomnia.

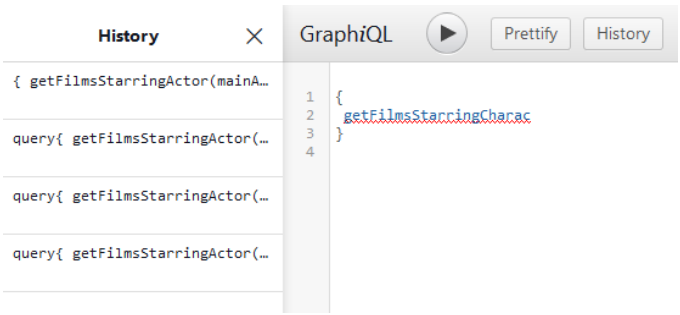

Figura 2.8-7 Mostrar historial de GraphiQL.

Aun así, GraphiQL es una buena herramienta si quieres testear de forma rápida las queries y no quieres instalar ninguna otra herramienta, tiene sus limitaciones pero si realmente solo deseas testear queries y documentarlas brevemente, cumple su función de forma aceptable.

2.9 CONCLUSIONES

En este capítulo se ha explicado las tecnologías y herramientas utilizadas a lo largo del proyecto, cabe destacar los siguientes aspectos:

Por una parte, Postman ha sido la herramienta con la que más cómodo me he sentido, esto se debe a que a diferencia de GraphiQL permite organizar las queries y mutaciones en carpetas además de automatizar la autenticación JWT gracias al soporte de variables globales y environments, lo que me ha ahorrado mucho tiempo evitando que por cada request tuviera que copiar manualmente el token para poder acceder a los endpoints.

Por otra parte, Insomnia es una gran herramienta que tiene todas las funcionalidades que tiene Postman, con una interfaz intuitiva y fácil de entender incluso si nunca has usado Insomnia anteriormente.

Sin embargo, es una plataforma de pago si se desea trabajar en equipo, mientras que Postman ofrece esta posibilidad de forma gratuita. Todo esto sumado al hecho de que no he conseguido configurar de forma correcta el plugin para la autenticación automática JWT le da una clara ventaja a Postman ya que esta funcionalidad ahorra muchísimo tiempo a la hora de depurar endpoints.

Durante el tiempo que he estado usando GraphiQL, sobre todo al comienzo del proyecto, ha facilitado mucho el testeo de queries y mutaciones de forma rápida. Sin embargo, GraphiQL no es una herramienta tan completa como Postman o Insomnia, ya que no dispone de ninguna manera de organizar las queries y mutaciones por carpetas, ni tampoco permite el testeo de endpoints.

Por todas las razones expuestas anteriormente, creo que la herramienta más completa y que recomiendo utilizar es Postman.

3 CAPÍTULO 3. PROYECTO DE EJEMPLO

En este apartado vamos a crear un proyecto de ejemplo sencillo, donde implementaremos una API que soporte GraphQL, con ayuda de Java crearemos los objetos que consideremos necesarios y con las anotaciones podremos crear las tablas de la base de datos H2.

Los pasos para crear el proyecto de ejemplo son los siguientes:

- Definir los paquetes y estructura del proyecto, siendo imprescindible la definición de entidades.
- Definir una interfaz que extienda de JPARepository, para poder guardar y modificar los objetos que creemos.
- Definir un esquema GraphQL afín a las entidades que hemos creado previamente en Java.
- Crear la base de datos H2 y conectarla a nuestro proyecto para poder usarla más adelante.
- Implementar queries y mutaciones.
- Implementar un JSON Web Token para añadir seguridad al proyecto.

3.1 INTRODUCCIÓN AL EJEMPLO.

Nuestro ejemplo será una API de consulta de películas, un proyecto sencillo que estará formado por una única tabla en nuestra base de datos, y por lo tanto solamente tendremos que crear una entidad en Java, dicha entidad tendrá los siguientes atributos:

- `idpelicula`: será el identificador de cada película, único en la base de datos.
- `actor`: es el actor principal que aparece en la película.
- `director`: el director de la película.
- `duracion`: es la duración de la película en minutos.
- `fechasalida`: representa la fecha de salida de la película.
- `genero`: es el género al que pertenece la película
- `resumen`: es un resumen de la película.
- `titulo`: el título de la película.
- `valoracion`: es una nota de 0 a 10 de la película

En base a estas especificaciones, comenzamos a crear el proyecto.

3.2 CREACIÓN DE UN PROYECTO DE EJEMPLO USANDO SPRING BOOT (SPRING BOOT INITIALIZR)

La generación del proyecto puede realizarse desde Spring Initializr (SI), o importar ya uno que haya sido creado previamente. Para crear uno desde cero accedemos a su web [12].

El proceso de generación de proyecto puede realizarse también desde Spring Tool Suite (STS) creando un nuevo *Spring Starter Project* la configuración es la que se muestra en la Figura 3.2-1.

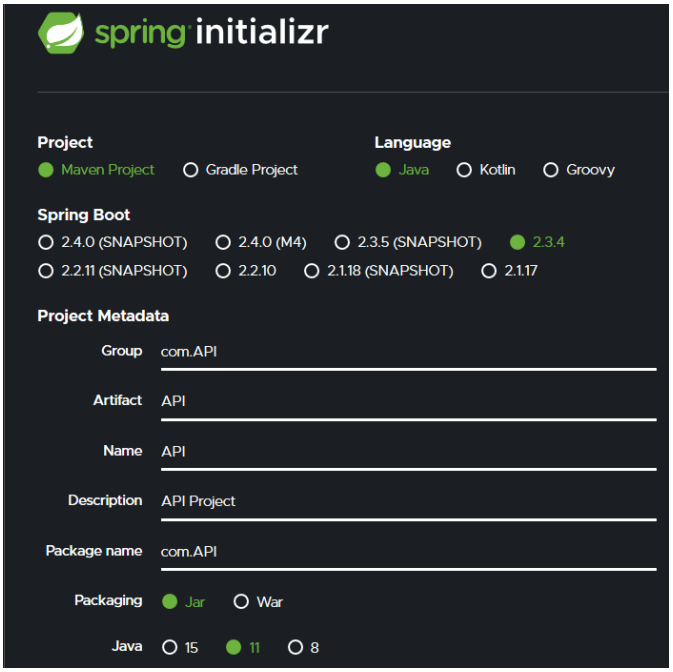

Figura 3.2-1 Creación del proyecto usando Spring Initializr.

Esta configuración que introducimos creará la ruta de carpetas del proyecto, cabe mencionar que podemos crear un proyecto Maven o Gradle, para este proyecto se ha decidido utilizar como ya se dijo en el apartado 2.1.2.

Al crear el proyecto Maven, la configuración del proyecto puede configurarse desde el archivo *pom.xml*, que servirá para añadir más dependencias al proyecto.

A continuación, escogemos la versión de Spring Boot sobre la vamos a trabajar y rellenamos los datos referentes a la estructura de nuestro proyecto, como los nombres de los paquetes y la versión de Java a usar.

3.3 ARCHIVO DE DEPENDENCIAS POM.XML

De forma adicional podemos añadir dependencias a otras librerías desde SI, estas referencias se almacenan en el archivo de configuración pom.xml, este archivo se localiza en la raíz del proyecto.

En la Figura 3.3-1 añado las referencias Spring Web para utilizar Tomcat para ejecutar el servicio, H2 database como gestor de base de datos y Spring Data JPA para hacer uso de los repositorios, que nos permitirán gestionar las películas de la base de datos.

Si en una fase más avanzada del proyecto necesitásemos añadir referencias nuevas, tendríamos que agregarlas al archivo de configuración xml.

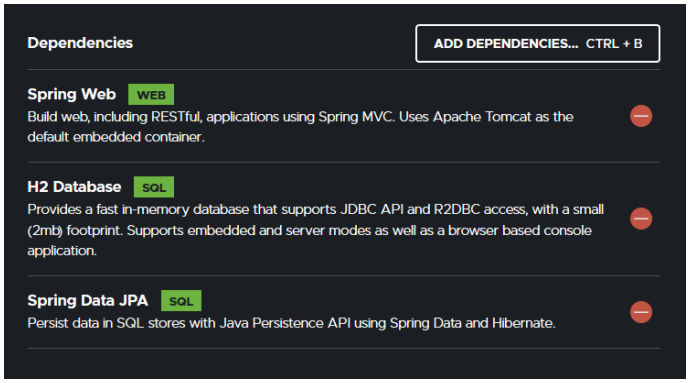

Figura 3.3-1 Añadir dependencias al proyecto Maven.

Una vez llegados a este punto, solo tendremos que darle al botón *Generate* y la página nos proporcionará un archivo comprimido con la estructura de carpetas del proyecto, que tan solo tendremos que abrir desde STS.

Para ello descomprimimos el archivo y desde STS vamos a *File > Open projects from File System* para abrirlo, tal y como se muestra en la Figura 3.3-2

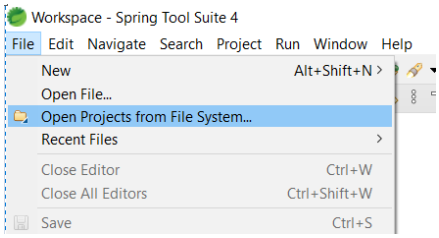

Figura 3.3-2 Importar proyecto de ejemplo con Spring Tools Suite 4.

Se abrirá una ventana como la de la Figura 3.3-3, simplemente seleccionamos la ruta del proyecto.

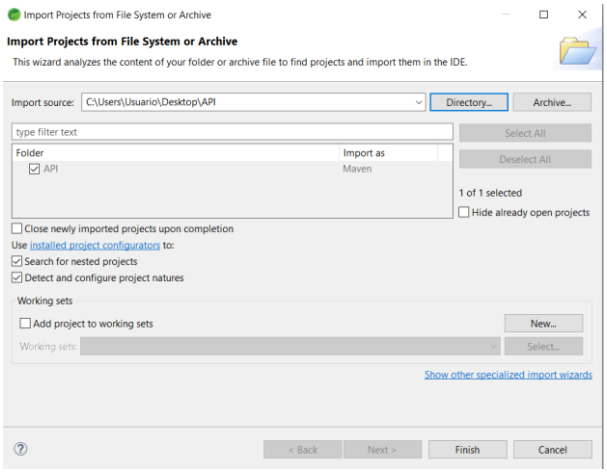

Figura 3.3-3 Terminando de importar el proyecto de ejemplo.

Cuando termine de importarse, el proyecto de ejemplo se habrá generado correctamente y su estructura es la siguiente:

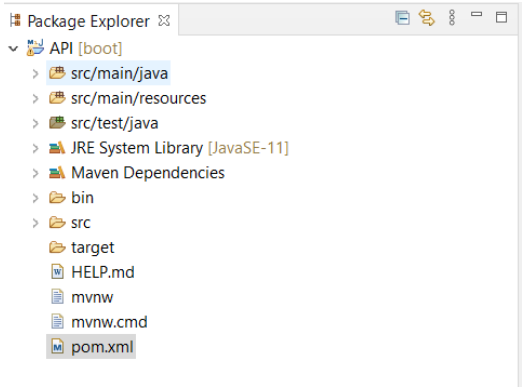

Figura 3.3-4 Estructura del proyecto de ejemplo.

• La ruta principal de nuestro proyecto será API, dentro de la ruta *src* encontraremos todo el código además de un script SQL para inicializar la base de datos si lo consideramos necesario

• En *main/java* estarán los paquetes con las clases Java

• En *main/resources* es la ruta del archivo SQL y de *application.properties* que nos servirá para activar la consola de H2 database y la ruta de la base de datos. Dentro debemos crear la carpeta GraphQL que contendrá el esquema en el que se basará la API.

• Nuestro proyecto incluye JRE y las dependencias Maven de nuestro proyecto, estas pueden modificarse en el archivo pom.xml que se localiza en la raíz del proyecto.

Al abrir pom.xml encontraremos información como la versión de xml, de Java e incluso la de Spring Boot que se está utilizando, para luego encontrar las etiquetas <*dependencies*> para añadir dependencias manualmente, tal y como se muestra en la Figura 3.3-5.

```
<?xml version="1.0" encoding="UTF-8"?>
<project                                    xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.API</groupId>
  <artifactId>API</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>API</name>
  <description>API Project</description
```

Figura 3.3-5 Contenido de pom.xml.

Para añadir una dependencia, simplemente la colocamos dentro de *<dependencies>*, toda dependencia que nueva que se añada al proyecto debe empezar por *<dependency>* y terminar por *</dependency>*, tal y como se muestra en la Figura 3.3-6.

```
        </dependency>
         <dependency>
            <groupId>com.graphql-java</groupId>
            <artifactId>graphql-java-tools</artifactId> <!--  write schema with GraphQL schema language, -->
            <version>5.2.4</version>
        </dependency>
        <dependency>
            <groupId>com.graphql-java</groupId>
            <artifactId>graphiql-spring-boot-starter</artifactId> <!--user interface with which we can be used to test our GraphQL queries   -->
            <version>5.0.2</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
```

Figura 3.3-6 Etiqueta Dependencies de pom.xml.

Llegados a este punto ya tendremos la base del proyecto creada, ahora vamos a implementar las clases Java harán uso de etiquetas para la creación de *entidades* y el contenido de la base de datos en base a las especificaciones de la introducción.

3.4 CREACIÓN DE CLASES EN JAVA

Para comenzar vamos a definir los paquetes que almacenarán las clases Java, comenzando por el paquete *com.API.entity* que contendrá la clase Película con sus atributos, tal y como se muestra en la Figura 3.4-1.

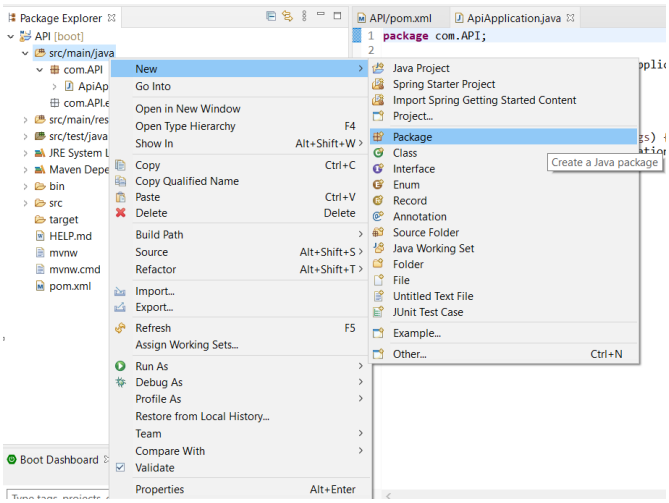

Figura 3.4-1 Creación del paquete Entity.

Las entidades son los objetos Java que forman parte del proyecto, vamos a definir la clase Película que debe tener un identificador único que vaya incrementándose de forma automática cada vez que se llame al método constructor de la clase, tal y como se muestra en la Figura 3.4-2.

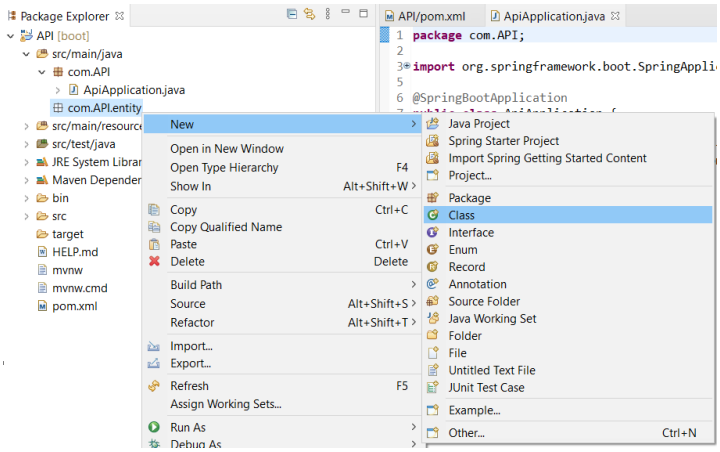

Figura 3.4-2 Creación de clase Película.

Para indicar que el objeto Película es una tabla de la base de datos utilizamos `@Table` , es necesario indicar previamente que la clase es a su vez una entidad mediante la etiqueta `@Entity` tal y como se muestra en la Figura 3.4-3.

```
@Entity
@Table (name = "pelicula")
public class Pelicula {
    @Id
    @Column(name = "idpelicula", nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idpelicula;
    @Column(name = "actor")
    private String actor;
    @Column(name = "director")
    private String director;
    @Column(name = "duracion")
    private int duracion;
    @Column(name = "fechasalida")
    private int fechasalida;
    @Column (name = "genero")
    private String genero;
    @Column(name= "resumen")
    private String resumen;
    @Column(name = "titulo")
    private String titulo;
    @Column(name = "valoracion")
    private int valoracion;
```

Figura 3.4-3 Uso de anotaciones en Película.

Para indicar el contenido de las columnas de la tabla Película utilizamos `@Column` por cada uno de los atributos de la clase, en el caso de `idpelicula` hay que utilizar `@GeneratedValue` para indicar que va a ser el identificador de la película (clave primaria).

Es necesario implementar varios métodos constructores en función de las necesidades del proyecto, estos son:

- Un constructor que cree un objeto a partir de todos sus atributos excepto `idpelicula` ya que este se incrementa de forma automática tal y como se muestra en la Figura 3.4-4. Este método es el que se utilizará en una ejecución normal del programa.

```
public Pelicula(String actor, String director, int duracion, int fechasalida, String genero, String resumen,
        String titulo, int valoracion) {
    super();
    this.actor = actor;
    this.director = director;
    this.duracion = duracion;
    this.fechasalida = fechasalida;
    this.genero = genero;
    this.resumen = resumen;
    this.titulo = titulo;
    this.valoracion = valoracion;
}
```

Figura 3.4-4 Método Constructor de objeto Película sin ID.

- Un constructor para crear un objeto *Película* vacío, este se utilizará para crear una película de forma rápida para cuestiones de testeo (Figura 3.4-5).

```
public Pelicula () {

}
```

Figura 3.4-5 Método Constructor de objeto Película vacío.

- Otro constructor con el identificador de la película, también para realizar pruebas y no tener que introducir todos los parámetros de la película cada vez que necesitemos testear algo.

```
public Pelicula(int idPelicula) {
    this.idPelicula = idPelicula;
}
```

Figura 3.4-6 Constructor de objeto Película solo con ID.

- Es necesario la implementación de métodos get/set para obtener y modificar los atributos de las entidades.

```
public int getIdPelicula() {
    return idPelicula;
}

public void setIdPelicula(int idPelicula) {
    this.idPelicula = idPelicula;
}
```

Figura 3.4-7 Métodos Get/Set.

Con esto ya habrá concluido la creación de las tablas de la base de datos, para modificar el contenido de esta vamos a utilizar JpaRepository, que hará posible la manipulación de *entidades.*

Para ello creamos una interfaz que extienda de JpaRepository tal y como se muestra en la Figura 3.4-8:

```
public interface PeliculaRepository extends JpaRepository<Pelicula, Integer>{


}
```

Figura 3.4-8 Creación de interfaz PeliculaRepository.

3.5 Patrón Repositorio e Inyección de Dependencias

JPARepository es un módulo de Spring Data que permite definir repositorios del tipo que nosotros consideremos. Es el encargado de gestionar las operaciones frente a una tabla de una base de datos, como añadir, eliminar o modificar elementos de la misma.

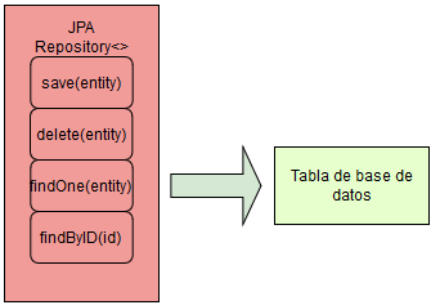

Figura 3.5-1 Funcionamiento de JPA Repository.

Este soporta la creación de clases genéricas, esto quiere decir que podemos crear repositorios del tipo que nosotros consideremos y aplicar las operaciones que nos ofrece JPA a nuestro proyecto.

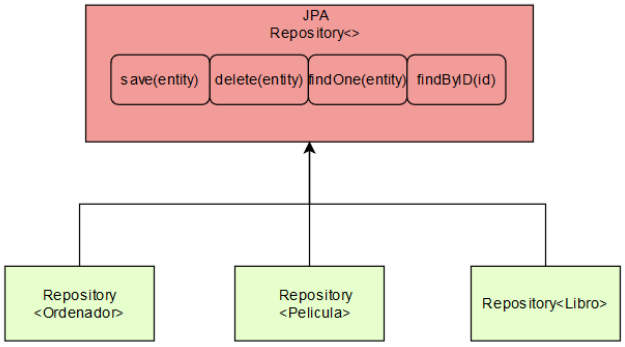

Figura 3.5-2 Extensión de JPA Repository.

Una vez que tenemos creada la clase genérica, podemos crear nuestro propio repositorio del tipo correspondiente y hacer uso de los métodos que interactúan con la base de datos [13].

Por otra parte, la inyección de dependencias es otro patrón de diseño que tiene como finalidad obtener un código más desacoplado, facilitando la modificación de partes del sistema más adelante en caso de que fuera necesario. Este puede realizarse de diversas formas:

- Mediante el uso de un método constructor: Cada entidad de nuestro proyecto tiene una serie de atributos, en Java podemos construir objetos mediante el uso de constructores donde pasando una serie de parámetros al constructor crea un objeto con dichos parámetros.

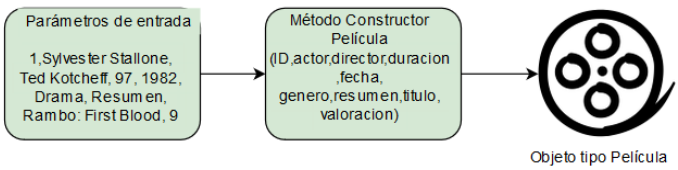

Figura 3.5-3 Inyección de dependencias mediante constructor.

- Mediante el uso de Setters: Cada entidad posee métodos get/set para obtener y modificar atributos de los objetos respectivamente.

- Mediante el uso de anotaciones en Spring: Spring permite el uso de anotaciones, y usando la anotación `@Autowired`.

3.6 INSTALACIÓN Y CONFIGURACIÓN DE BASE DE DATOS H2.

Antes de crear la clase con el objeto PeliculaRepository, necesitamos una base de datos para que nuestro repositorio pueda acceder a ella y modificar o guardar datos.

Para la integración de H2 en el proyecto, tendremos que añadir una dependencia a nuestro archivo pom.xml, tal y como se muestra en la Figura 3.6-1:

```
59        <dependency>
60            <groupId>com.h2database</groupId>
61            <artifactId>h2</artifactId>
62            <scope>runtime</scope>
63        </dependency>
```

Figura 3.6-1 Añadir dependencias para H2 Database.

Es necesario la introducción de unos parámetros de configuración dentro de *application.properties (src/main/resources/application.properties)*, donde definiremos la ruta de la base de datos, activaremos la *consola* H2, y estableceremos la *URL* para acceder desde nuestro navegador, tal y como se muestra en la Figura 3.6-2:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Figura 3.6-2 Configuración de Application.properties.

La base de datos estará vacía por defecto, para añadir la tabla Película a la base de datos implementaremos un script SQL que introducirá algunas películas para poder testear la API, este debe ir en la ruta *src/main/resources*, tal y como se muestra en la Figura 3.6-3:

```
INSERT INTO PELICULA (actor, director, duracion, fechasalida, genero, resumen, titulo, valoracion)
VALUES ('Humphrey Bogart', 'Michael Curtiz',102, 1942, 'Romance', 'Resumen de la pelicula...','Casablanca', 8);

INSERT INTO PELICULA (actor, director, duracion, fechasalida, genero, resumen, titulo, valoracion)
VALUES ('Neve Campbell', 'Wes Craven',111, 1996, 'Terror', 'Resumen de la pelicula...','Scream', 7);

INSERT INTO PELICULA (actor, director, duracion, fechasalida, genero, resumen, titulo, valoracion)
VALUES ('Arnold Schwarzenegger', 'James Cameron',108, 1984, 'Accion', 'Resumen de la pelicula...','The Terminator', 9);

INSERT INTO PELICULA (actor, director, duracion, fechasalida, genero, resumen, titulo, valoracion)
VALUES ('Jamie Lee Curtis', 'John Carpenter',91, 1978, 'Terror', 'Resumen de la pelicula...','Halloween', 8);

INSERT INTO PELICULA (actor, director, duracion, fechasalida, genero, resumen, titulo, valoracion)
```

```
VALUES ('Gary Lockwood', 'Stanley Kubrick',142, 1968, 'Ciencia ficcion', 'Resumen de la pelicula...','Odisea del espacio', 7);

INSERT INTO PELICULA (actor, director, duracion, fechasalida, genero, resumen, titulo, valoracion)
VALUES ('Sylvester Stallone', 'Ted Kotcheff',97, 1982, 'Drama', 'Resumen de la pelicula...','Rambo: First Blood', 9);
```

Figura 3.6-3 Archivo SQL con las películas.

Una vez terminado, lo colocamos dentro de *src/main/resources* y cada vez que ejecutemos el programa la base de datos leerá el archivo SQL y creará la tabla Película con el contenido del Script.

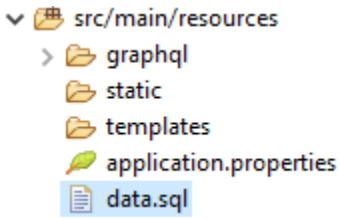

Figura 3.6-4 Ruta de data.sql.

Con esto concluye la parte de la configuración de la base de datos, ahora vamos a crear un esquema que contenga la entidad Película con los mismos atributos que hemos definido en Java.

3.7 CREANDO UN SCHEMA GRAPHQL

Como ya sabemos GraphQL se apoya en el uso de un esquema que se encarga de definir la *API*, para que el proyecto pueda leer el esquema es necesario añadir las dependencias de la Figura 3.3-6

Estas dependencias integran GraphQL con *Java*, además de añadir un endpoint */graphiql* con el que podremos testear las queries y mutaciones de forma rápida.

Si bien la mejor herramienta para implementar queries en este lenguaje es Postman¸GraphiQL hace un buen trabajo sin necesidad de instalar nada.

Para implementar el esquema, creamos un Type Pelicula tal y como se muestra en la Figura 3.7-1:

```
type Pelicula{      //Definimos el tipo Pelicula
    idpelicula:ID!, //Tiene que ser de tipo ID y no puede ser null
    actor:String,   //ahora escribimos el resto de
    director:String,//atributos que faltan para completar
    duracion:Int,   //la clase Película.
    fechasalida:Int,
    genero:String,
    resumen:String,
    titulo:String,
    valoracion:Int
}
```

Figura 3.7-1 Creación del objeto Película en GraphQL.

Además tendremos que implementar las queries y mutaciones de GraphQL, recordamos que:

- Las queries se encargarán de recuperar información del servidor (peticiones GET o POST si utilizan GraphQL).

- Las mutaciones serán las encargadas de crear, modificar o borrar datos del servidor (peticiones PUT, POST, DELETE).

Para definir las queries hay que definir un Type Query tal y como se muestra en la Figura 3.7-2, donde especificamos el nombre de la query junto al parámetro que espera y especificando qué es lo que devuelve, para este proyecto una query siempre devolverá una lista de películas.

```
type Query{
    obtenerTodo:[Pelicula]
    obtenerPeliculasConValoracion(valoracion:Int):[Pelicula]
    obtenerPeliculasPorDirector(director:String):[Pelicula]
    obtenerPeliculasPorTitulo(titulo:String):[Pelicula]
    obtenerPeliculasPorActor(actor:String):[Pelicula]
    obtenerPeliculasPorDuracion(duracion:Int):[Pelicula]
    obtenerPeliculasPorFecha(fechasalida:Int):[Pelicula]
    obtenerPeliculasAntesDe(fechasalida:Int):[Pelicula]
    obtenerPeliculasDespuesDe(fechasalida:Int):[Pelicula]
    obtenerPeliculasPorGenero(genero:String):[Pelicula]
}
```

Figura 3.7-2 Creación de Type Query.

Lo mismo ocurre con las mutaciones:

```
type Mutation{
    crearPelicula(titulo:String, fechasalida:Int,
   valoracion:Int, duracion:Int, resumen:String,
    director:String, actor:String, genero:String):Pelicula,

    borrarPelicula(idpelicula:ID): String,

    modificarPelicula(idpelicula: ID,titulo:String, fechasalida:Int,
   valoracion:Int, duracion:Int, resumen:String,
    director:String, actor:String, genero:String):Pelicula
}
```

Figura 3.7-3 Creación de Type Mutation.

Una vez definido el esquema guardamos el archivo con extensión graphqls y lo guardamos en la ruta *src/main/resources/graphql*, en caso de que dicha carpeta no exista, la creamos tal y como se muestra en la Figura 3.7-4.

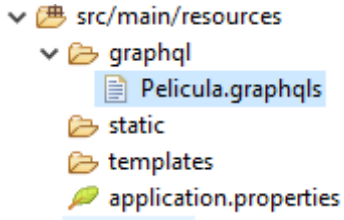

Figura 3.7-4 Ruta del esquema GraphQL.

Una vez hecho esto hay que resolver las queries y mutaciones desde Java.

3.8 DEFINIENDO LAS QUERIES EN EL ESQUEMA

En el apartado anterior ya definimos la estructura de la API con un esquema, pero todavía no hemos definido las queries que soportan GraphQL. Es por eso que vamos a añadir algunas extra que soporten este lenguaje, tal y como se muestra en la Figura 3.8-1:

```
type Query{
   obtenerTodo:[Pelicula]
   obtenerPeliculasConValoracion(valoracion:Int):[Pelicula]
   obtenerPeliculasConValoracionGraph(valoracion:Int):[Pelicula]
   obtenerPeliculasPorDirector(director:String):[Pelicula]
   obtenerPeliculasPorDirectorGraph(director:String):[Pelicula]
...
}
```

Figura 3.8-1 Definición de Queries GraphQL en el esquema.

Una vez hecho esto pasamos a implementarlas en Java.

3.9 DEFINIENDO LAS QUERIES EN JAVA

Lo primero es crear el directorio que va a almacenar las queries y mutaciones en el proyecto, estas se guardarán en la ruta *com.API.resolver*, que contendrá una clase para administrar las queries y otra para las mutaciones, tal y como se muestra en la siguiente figura:

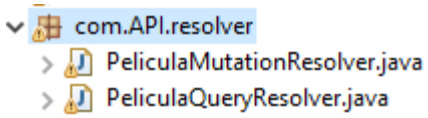

Figura 3.9-1 Creación de QueryResolver y MutationResolver.

Antes de implementar las queries podemos acceder a la base de datos y obtener todo el contenido de ella mediante la consola H2, esto nos servirá para tener una referencia de qué películas tenemos almacenadas, tal y como se muestra en la Figura 3.9-2:

select * from pelicula;

IDPELICULA	ACTOR	DIRECTOR	DURACION	FECHASALIDA	GENERO	RESUMEN	TITULO	VALORACION
1	Humphrey Bogart	Michael Curtiz	102	1942	Romance	Resumen de la pelicula...	Casablanca	8
2	Neve Campbell	Wes Craven	111	1996	Terror	Resumen de la pelicula...	Scream	7
3	Arnold Schwarzenegger	James Cameron	108	1984	Accion	Resumen de la pelicula...	The Terminator	9
4	Jamie Lee Curtis	John Carpenter	91	1978	Terror	Resumen de la pelicula...	Halloween	8
5	Gary Lockwood	Stanley Kubrick	142	1968	Ciencia ficcion	Resumen de la pelicula...	Odisea del espacio	7
6	Sylvester Stallone	Ted Kotcheff	97	1982	Drama	Resumen de la pelicula...	Rambo: First Blood	9

(6 rows, 6 ms)

Figura 3.9-2 Contenido de la base de datos.

Es importante tener en cuenta que la clase que va a resolver las queries (*PeliculaQueryResolver*), implemente *GraphQLQueryResolver*, ya que si no hacemos esto cuando realicemos pruebas desde Postman llamando a GraphQL obtendremos un error que nos impedirá continuar.

A continuación, explico los métodos que se van a utilizar de JPARepository para recuperar y modificar datos de la base de datos:

- findAll(): devolverá toda la colección de películas.
- findByID(id): devolverá la entidad película que coincida con el ID.
- getOne(id): devolverá una referencia a la entidad película, esto nos será de utilidad a la hora de modificar algún atributo de la película.
- delete(pelicula): borra la película.
- save(pelicula): guarda una película.

Si bien en la clase que resuelve las queries solamente vamos a utilizar findAll() y findByID(), el resto nos serán útiles para la gestión de las mutaciones más adelante. La implementación es la siguiente:

```
30    @RequestMapping(value = "/pelicula={pid}", method = RequestMethod.GET)
31    public Optional<Pelicula> getPelicula(@PathVariable("pid") int pid){
32        return pr.findById(pid);
33    }
```

Figura 3.9-3 Implementación en Java de getPelicula().

`@RequestMapping(value = "/pelicula={pid}", method = RequestMethod.GET)` quiere decir que este método espera que le llegue el parámetro que hemos definido como `pid` a través de la URL, como el objetivo de este método es obtener datos del servidor pero no modificarlos, es una petición GET.

`@PathVariable("pid")`se utiliza para especificar que el parámetro que le llega por la URL será tratado internamente como una variable entera `int pid`. Simplemente devolvemos la llamada a `findById` con el ID que nos llega por la URL.

Aunque todavía no hemos llegado a la parte de Postman, voy a mostrar una captura con el funcionamiento de dicho método:

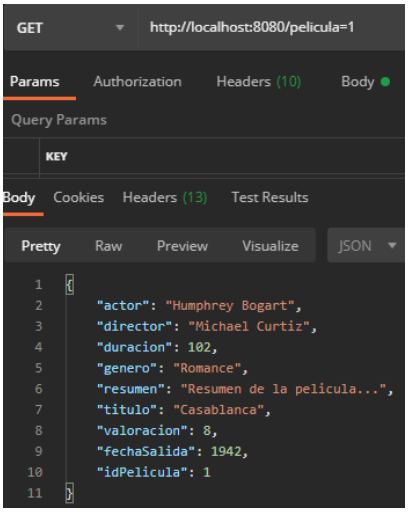

Figura 3.9-4 Resultado de getPelicula.

En este caso le hemos pasado por la URL el ID 1, que se corresponde con la película *Casablanca*, y nos devuelve toda la información sobre la película.

Ahora voy a enseñar cómo implementar el método para obtener las películas según su actor principal, la gran mayoría de los métodos de esta clase son muy parecidos a este así que explicaré el funcionamiento de este y ya habremos concluido esta parte. La implementación es la siguiente:

```
124    public List<Pelicula> obtenerPeliculasPorActor(@PathVariable("actor") String actor){
125        while(actor.contains("_")) {
126            int index = actor.indexOf('_');
127            actor = actor.substring(0, index) + ' ' + actor.substring(index+1);
128        }
129        List<Pelicula> l = new ArrayList<Pelicula> ();
130        for (Pelicula p : pr.findAll()) {
131            if(p.getActor().equals(actor))
132                l.add(p);
133        }
134        return l;
135    }
```

Figura 3.9-5 Implementación del método obtenerPeliculasPorActor().

`@RequestMapping` y `@PathVariable` funcionan de la misma forma que el método anterior, este método devolverá una lista de objetos de tipo película, el bucle while se encarga de sustituir las "_" por espacios en blanco. Esto se debe a que en una URL se espera recibir el nombre completo del actor separado por "_".

Luego se crea una lista para guardar los resultados y recorro la lista que devuelve el método `findAll()`, si el actor coincide con el actor que me llega por la URL entonces lo añado a mi lista solución, para luego devolverla. El resultado de la query es el siguiente:

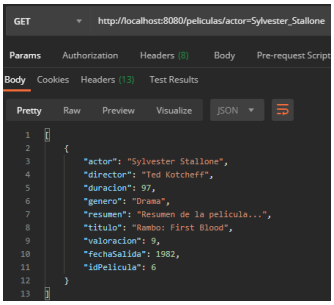

Figura 3.9-6 Resultado de obtenerPeliculasPorActor.

El resto de métodos se resolverían de forma similar, creamos una lista para guardar la información, recorremos la lista que devuelve `findAll()` y comprobamos si el parámetro que nos llega por URL coincide con alguna de las películas.

3.10 DEFINIENDO LAS MUTACIONES EN JAVA

En la Figura 3.9-1 ya se ha creado la clase *PeliculaMutationResolver*, dentro de esta clase se encontrarán métodos para la creación y modificación de las películas. En esta clase hay que especificar que implemente *GraphQLMutationResolver*, ya que si no lo hacemos GraphQL no devolverá respuesta cuando lo usemos en Postman. A continuación muestro el método que se encarga de crear las películas:

```
25    @RequestMapping(value = "/pelicula", method = RequestMethod.POST)
26    public Pelicula crearPelicula(String titulo, int fechasalida, int valoracion,
27            int duracion, String resumen, String director, String actor, String genero) {
28        Pelicula p = new Pelicula( actor,  director,  duracion,  fechasalida,  genero,  resumen,
29                titulo,  valoracion);
30        return pr.save(p);
31    }
```

Figura 3.10-1 Método POST para crear películas.

En este método se especifica el endpoint de acceso para crear una nueva película (POST), simplemente creamos un objeto de tipo película con los atributos que llegan mediante un formulario y guardamos el objeto en el repositorio. En la Figura 3.10-2 se puede ver un ejemplo de uso en Postman.

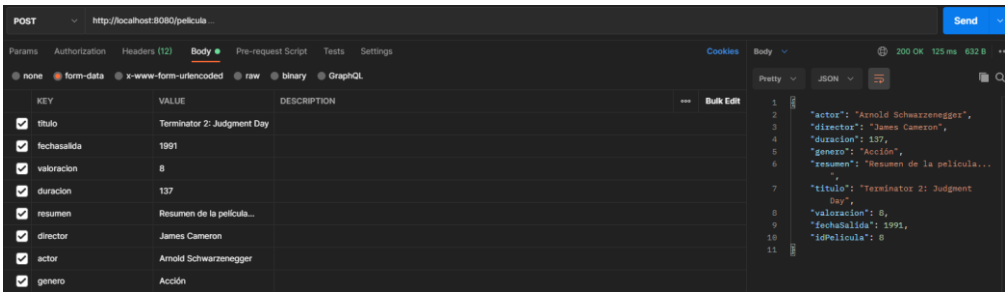

Figura 3.10-2 Creación de una nueva película.

Para eliminar una película hacemos uso del método delete(id), al borrarse mostrará un mensaje indicando que se ha borrado correctamente, tal y como se muestra en la Figura 3.10-3.

```
34    @RequestMapping(value = "/peliculas/{idpelicula}", method = RequestMethod.DELETE)
35    public String borrarPelicula(@PathVariable int idpelicula) {
36        Pelicula p = pr.getOne(idpelicula);
37        pr.delete(p);
38        return "borrado";
39
40    }
```

Figura 3.10-3 Método DELETE para borrar películas.

Si ahora accedemos a ese endpoint desde Postman y como identificador le pasamos el 2, que se corresponde con la película *Scream¸* borrará dicha película.

Para comprobarlo, podemos hacer uso del endpoint */peliculas* que nos devuelve todas las películas almacenadas en la base de datos, o bien podemos acceder a H2 console y comprobarlo desde ahí (en la Figura 3.9-2 puede consultarse el estado inicial de la base de datos), tal y como se muestra en la siguiente captura:

```
SELECT * FROM PELICULA;
```

IDPELICULA	ACTOR	DIRECTOR	DURACION	FECHASALIDA	GENERO	RESUMEN	TITULO	VALORACION
1	Humphrey Bogart	Michael Curtiz	102	1942	Romance	Resumen de la pelicula...	Casablanca	8
3	Arnold Schwarzenegger	James Cameron	108	1984	Accion	Resumen de la pelicula...	The Terminator	9
4	Jamie Lee Curtis	John Carpenter	91	1978	Terror	Resumen de la pelicula...	Halloween	8
5	Gary Lockwood	Stanley Kubrick	142	1968	Ciencia ficcion	Resumen de la pelicula...	Odisea del espacio	7
6	Sylvester Stallone	Ted Kotcheff	97	1982	Drama	Resumen de la pelicula...	Rambo: First Blood	9

(5 rows, 2 ms)

Figura 3.10-4 Método DELETE para eliminar una película.

Como puede observarse la película *Scream* cuyo identificador era 2, ya no existe en la base de datos.

Para finalizar la clase mutaciones, vamos a implementar un método para modificar una película, este método espera un identificador a través de la URL y mediante un formulario se modifican el resto de atributos de la película, tal y como se muestra en la siguiente captura:

```
60    @RequestMapping(value = "/pelicula/{idpelicula}", method = RequestMethod.PUT)
61    public Pelicula modificarPelicula(@PathVariable int idpelicula , String titulo, int fechasalida, int valoracion,
62            int duracion, String resumen, String director, String actor, String genero ) {
63        Pelicula p = new Pelicula(idpelicula, actor, director, duracion, fechasalida, genero, resumen, titulo, valoracion);
64        if(pr.existsById(idpelicula)) {
65            pr.getOne(idpelicula).setActor(p.getActor());
66            pr.getOne(idpelicula).setDirector(director);
67            pr.getOne(idpelicula).setDuracion(duracion);
68            pr.getOne(idpelicula).setFechaSalida(fechasalida);
69            pr.getOne(idpelicula).setGenero(genero);
70            pr.getOne(idpelicula).setResumen(resumen);
71            pr.getOne(idpelicula).setTitulo(titulo);
72            pr.getOne(idpelicula).setValoracion(valoracion);
73        }
74        return p;
75    }
```

Figura 3.10-5 Método PUT para modificar una película.

Para comprobar que funciona, desde Postman vamos a modificar la película cuyo ID es 1 *(Casablanca)* y vamos a modificar su valoración de 8 a 10, en la siguiente captura se muestra cómo hacerlo:

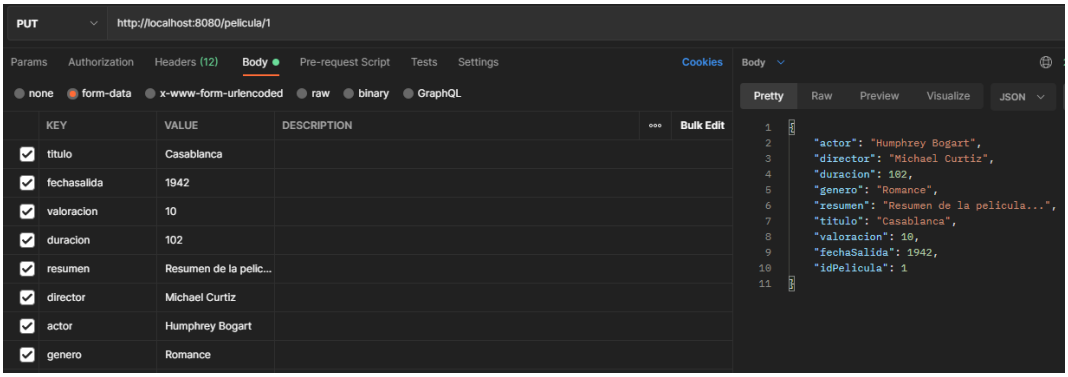

Figura 3.10-6 Modificar película desde Postman.

Ahora vamos a usar el endpoint */película=1* que nos devolverá la película *Casablanca* para comprobar que la modificación se ha realizado con éxito:

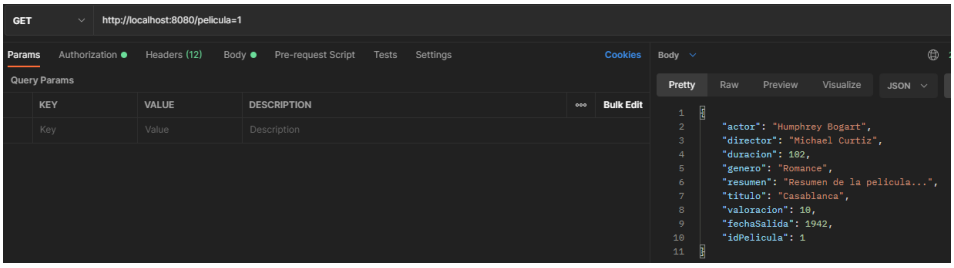

Figura 3.10-7 Resultado de modificar una película.

3.11 GRAPHQL CON POSTMAN

Para utilizar GraphQL con Postman es necesario haber cargado previamente el esquema de la API, y crear las carpetas donde se van a almacenar las queries.

Además de esto es recomendable automatizar el proceso de autenticación, ya que si no lo hacemos ahora cada vez que hagamos uso de una query habrá que añadir el token manualmente, todo esto se explica en el apartado 2.6.3.

Las queries implementadas con GraphQL se encuentran en la carpeta *Queries* de Postman, desde ahí seleccionamos una y marcamos la casilla que pone GraphQL para activar el soporte, tal y como se muestra a continuación:

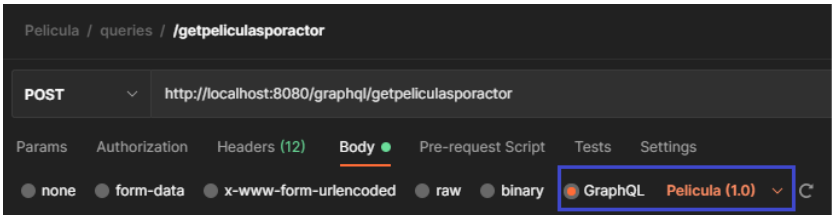

Figura 3.11-1 Utilizar GraphQL en request y seleccionar API.

Una vez hecho esto ya podremos implementar la query, en la Figura 3.11-2 devuelve el identificador de la película, junto a su título y fecha de salida a partir del nombre del actor principal:

Figura 3.11-2 Uso de GraphQL para obtener películas por actor.

En la siguiente captura, se muestra cómo obtener el identificador, el género al que pertenece y el título de la película en función de un director:

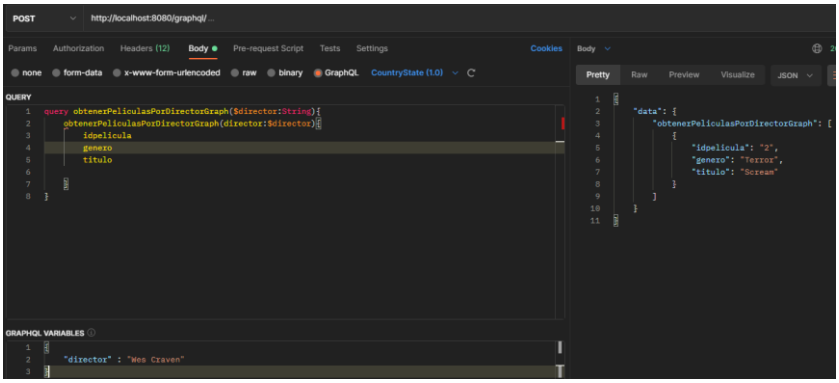

Figura 3.11-3 Uso de GraphQL para obtener película según el director.

También podemos utilizar las mutaciones para modificar las películas, en la siguiente captura se muestra un ejemplo de creación de una película con GraphQL:

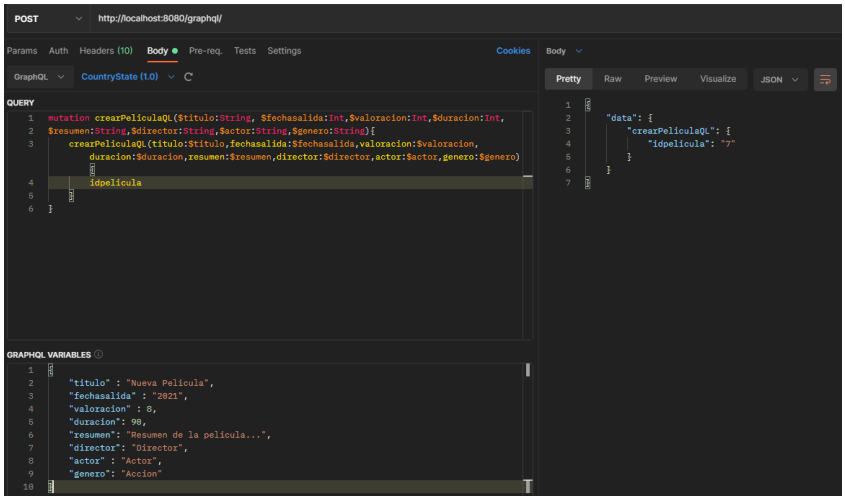

Figura 3.11-4 Uso de GraphQL para crear una película.

Todas estas queries tienen algo en común, y es el uso de el mismo endpoint para obtener o modificar los datos, con esto concluye la parte de GraphQL.

3.12 AUTENTICACIÓN JSON WEB TOKEN

Un JSON Web Token (JWT) es una forma rápida y segura de autenticar usuarios en aplicaciones, su funcionamiento es simple, el usuario especifica un usuario acompañado de una contraseña (esto puede realizarse a través de un formulario de logueo o incluso mediante un archivo JSON), el servidor comprueba que las credenciales que el usuario ha introducido son correctas y si lo son, devuelve un token.

Este token es una cadena de texto alfanumérica que le servirá al servidor para validar al usuario en cuestión, un token tiene el siguiente aspecto:

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJTZXJnaW8iLCJleHAiOjE2MTcyMjAxNDcsImlhdCI
6MTYxNzE4NDE0N30.5OdJ2KQtpanWBVtq08ihhxIIDk7nvUzj_rsBfVqUWHs
```

Figura 3.12-1 Aspecto de un JSON Web Token.

Una vez que el token es devuelto, el usuario debe incluirlo en un *Header* cada vez que realice una request, donde el servidor se encargará de comprobar y validar dicho token, El token no dura para siempre, pues dependiendo de la implementación caduca al cabo de un tiempo.

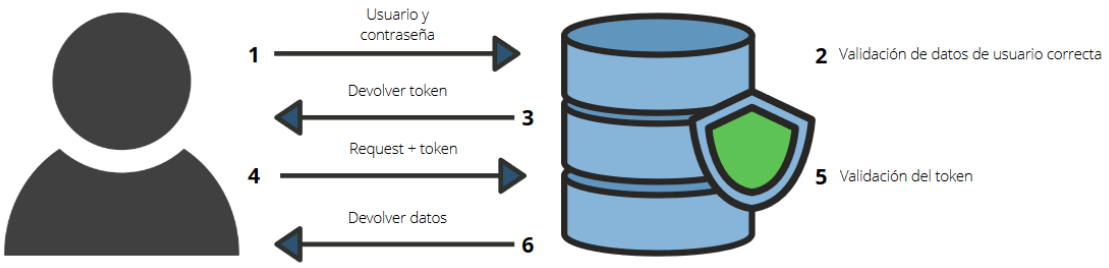

Figura 3.12-2 Funcionamiento de JWT.

El contenido del token se divide en tres secciones por puntos, su significado es el siguiente:

- Header*:* almacena el tipo de token y el algoritmo de encriptamiento.
- Payload*:* contiene los datos que identifican el usuario.
- Firma*:* contiene la firma digital que garantiza que el token no ha sido modificado.

Para que la API haga uso de JWT, hay que definir el valor del usuario y la contraseña con el que el user va a loguearse, junto al tipo de algoritmo con el que se firmará el token.

Antes de implementar las clases en Java es necesario añadir dos dependencias al archivo de configuración, estas son:

```
<dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.2</version>
    </dependency>
   <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
```

Figura 3.12-3 Añadir dependencias para JWT.

Una vez hecho esto, dentro de la ruta *src/main/java* crearemos un paquete llamado *com.TFG.autenticacion* donde almacenaremos las clases referentes a la implementación del *JWT*.

3.13 IMPLEMENTACIÓN DE JWT

El primer paso para la implementación del JWT es definir una clase que contenga los datos de acceso que se esperan por parte del usuario, esta clase debe implementar la interfaz *UserDetailsService* del módulo de seguridad de Spring, el resultado de la implementación debe ser tal y como se muestra a continuación:

```
10 @Service
11 public class MyUserDetailsService implements UserDetailsService{
12
13     @Override
14     public UserDetails loadUserByUsername(String userName) throws UsernameNotFoundException{
15         return new User("Sergio", "pass", new ArrayList<>());
16     }
17
18
19
20 }
```

Figura 3.13-1 Implementación de MyUserDetailsService.

En esta clase especificamos que el usuario que se espera es "*Sergio*" y la contraseña es "*pass*", ahora hay que definir otra clase para tener acceso al usuario y la contraseña mediante el uso de get/set, el resultado de la implementación es el siguiente:

```java
public class AuthenticationRequest implements Serializable {


    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    //need default constructor for JSON Parsing
    public AuthenticationRequest()
    {

    }

    public AuthenticationRequest(String username, String password) {
        this.setUsername(username);
        this.setPassword(password);
    }
}
```

Figura 3.13-2 Implementación de AuthenticationRequest.

Una vez implementada *AuthenticationRequest*, hay que implementar *AuthenticationResponse* que proporcionará acceso al token JWT:

```java
public class AuthenticationResponse implements Serializable {

    private final String jwt;

    public AuthenticationResponse(String jwt) {
        this.jwt = jwt;
    }

    public String getJwt() {
        return jwt;
    }
}
```

Figura 3.13-3 Implementación de AuthenticationResponse.

Ahora hay que implementar una clase que se encargue de hacer todo lo explicado en la Figura 3.13-2, esta clase deberá especificar una clave secreta y el tipo de algoritmo con el que se firmará el token. Esta clase se llamará JwtUtil y va a contener los siguientes métodos:

- `extractUsername(String token)`: extrae el usuario a partir del token.
- `extractExpiration(String token)`: extrae la fecha de expiración del token.
- `extractClaim(String token, Function<Claims, T> claimsResolver)`: extrae las claims llamando a `extractAllClaims`
- `extractAllClaims(String token)`: se encarga de establecer la clave para firmar el token.
- `isTokenExpired(String token)`: comprueba si el token ha expirado.
- `generateToken(UserDetails userDetails)`: genera el token.
- `createToken(Map<String, Object> claims, String subject)`: crea el token especificando el tiempo de validez del token y el tipo de algoritmo con el que se firma.

- `validateToken(String token, UserDetails userDetails)`: comprueba que el token sigue siendo válido.

La implementación de dicha clase se muestra a continuación:

```java
@Service
public class JwtUtil {

    private String SECRET_KEY = "secret";

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    public Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }
    private Claims extractAllClaims(String token) {
        return Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
    }

    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        return createToken(claims, userDetails.getUsername());
    }

    private String createToken(Map<String, Object> claims, String subject) {

        return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY).compact();
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
    }
}
```

Figura 3.13-4 Implementación de JwtUtil.

Es necesario implementar otra clase que, por cada request que el usuario realice compruebe que dentro del *Header Authorization* se encuentra el token, y que mediante la utilización de los métodos implementados en la clase JwtUtil compruebe la validez del token. La clase en cuestión es la siguiente:

```java
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired
    private MyUserDetailsService userDetailsService;

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
            throws ServletException, IOException {

        final String authorizationHeader = request.getHeader("Authorization");

        String username = null;
        String jwt = null;

        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
            jwt = authorizationHeader.substring(7);
            username = jwtUtil.extractUsername(jwt);
        }


        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

            UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

            if (jwtUtil.validateToken(jwt, userDetails)) {

                UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities());
                usernamePasswordAuthenticationToken
                        .setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
            }
        }
        chain.doFilter(request, response);
    }

}
```

Figura 3.13-5 Implementación de JwtRequestFilter.

Y para terminar, hay que implementar una última clase en la que podamos especificar que endpoints están protegidos mediante JWT y cuáles no, la clase es la siguiente:

```java
@EnableWebSecurity
class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private UserDetailsService myUserDetailsService;
    @Autowired
    private JwtRequestFilter jwtRequestFilter;

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(myUserDetailsService);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }

    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {

        httpSecurity.csrf().disable()
        .authorizeRequests().antMatchers("/h2-console/**").permitAll().antMatchers("/graphiql/**").permitAll()
        .antMatchers("/authenticate").permitAll().anyRequest().authenticated().and().authorizeRequests().
        and().exceptionHandling().and().sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS);



        httpSecurity.headers().frameOptions().disable();
        httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

    }

}
```

Figura 3.13-6 Implementación de WebSecurityConfig.

Esta clase extiende de *WebSecurityConfigurerAdapter*, y dentro del método `configure`, podemos especificar qué endpoints requieren autenticación JWT y cuáles no, con la configuración que se muestra en la figura anterior permite acceder a la consola h2 y a GraphiQL sin autenticación JWT, si quisiéramos proteger dichos endpoints con JWT simplemente habría que modificar dicho método.

Con esto ya funcionaría la autenticación JWT, ahora cada vez que el usuario realice una request deberá haberse autenticado previamente utilizando el endpoint */authenticate*, que devolverá el token y deberá agregar en el Header de cada petición dicho token. Este apartado ya se ha cubierto en el Capítulo 2 Tecnologías y Herramientas utilizadas (Apartado 2.7.9) [14].

3.14 CONCLUSIONES SOBRE EL PROYECTO DE EJEMPLO

En este apartado se ha cubierto todo lo relacionado con la creación de una API GraphQL sencilla utilizando Spring Initializr con un sistema de seguridad basado en tokens.

La implementación de esta API ha servido como ejemplo para trasladar todo lo aprendido en este apartado a la creación de una API que trabaja con datos reales sobre el Coronavirus, cubriendo todos los aspectos desde la implementación de clases en Java, hasta la implementación de queries pasando por la instalación de una base de datos.

4 CAPÍTULO 4. FUNCIONAMIENTO DE LA API

En este apartado se harán uso de los conceptos aprendidos durante el desarrollo del proyecto de ejemplo para implementar una API GraphQL sobre los casos de Coronavirus a nivel global, y para los países de España e Italia.

Los datos se tomarán de la API de origen, esta dispone de documentación [15] que contiene información sobre todos los puntos de acceso a ella, con ejemplos de cómo consultar y la respuesta que se debe esperar por parte de la API.

Los temas que se tratarán en este capítulo serán el uso de los endpoints de la API de origen para la obtención de datos sobre la evolución de los casos de la enfermedad durante un periodo de tiempo.

Se crearán las clases Java que representarán las entidades y se almacenarán los datos en la base de datos H2 a partir de la lectura de ficheros en formato JSON que previamente se habrán obtenido al atacar los endpoints de la API de origen.

Se implementará el endpoint */graphql* para la obtención de datos utilizando este lenguaje de consultas además de mostrar un ejemplo de ejecución del programa.

Por último, se introducirán los datos obtenidos en las consultas en MetricsGraphics para generar gráficas sobre la evolución de los casos.

4.1 IMPLEMENTACIÓN DE ENTIDADES

Para crear las entidades que representarán las tablas de la base de datos, seguimos el mismo procedimiento que se siguió en el Apartado 3.4, la base de datos va a almacenar dos tablas.

La tabla Summary contiene datos sobre los contagios a nivel global, estos datos son obtenidos del endpoint *Summary* de la API de origen. La segunda tabla obtendrá los datos del endpoint *By Country Live*, y mostrará datos acumulativos referentes a los casos de COVID de España e Italia.

A continuación muestro como se ha implementado la entidad Summary con los datos globales:

```
10 @Entity
11 @Table(name = "Summary")
12 public class Summary {
13
14     @Id
15     @Column(name = "Id", nullable = false)
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private int idsum;
18     @Column(name = "newConfirmed")
19     private int newConfirmed;
20     @Column(name = "totalConfirmed")
21     private int totalConfirmed;
22     @Column(name = "newDeaths")
23     private int newDeaths;
24     @Column(name = "totalDeaths")
25     private int totalDeaths;
26     @Column(name = "newRecovered")
27     private int newRecovered;
28     @Column(name = "totalRecovered")
29     private int totalRecovered;
30     @Column(name = "date")
31     private String date;
```

Figura 4.1-1 Creación de entidad Summary.

La entidad que va a representar los datos obtenidos del endpoint *By Country Live* se va a llamar *CountryState*:

```java
@Entity
@Table (name = "CountryState")
public class CountryState {
    @Id
    @Column(name = "Id", nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idcs;
    @Column(name = "Country")
    private String Country;
    @Column(name = "CountryCode")
    private String CountryCode;
    @Column (name = "Province")
    private String province;
    @Column(name = "City")
    private String city;
    @Column(name = "CityCode")
    private String cityCode;
    @Column (name = "Lat")
    private float Lat;
    @Column (name = "Lon")
    private float Lon;
    @Column (name = "Cases")
    private int Cases;
    @Column(name = "Status")
    private String Status;
    @Column(name = "Date")
    private String Date;
```

Figura 4.1-2 Creación de entidad CountryState.

Aparte del mapeo de atributos de ambas entidades, estas clases contienen un par de métodos constructores para la creación de objetos de ese tipo en concreto, además de métodos get/set tal y como se definieron para el proyecto de ejemplo.

4.2 USO DE ENDPOINTS DE LA API DE ORIGEN

El objetivo de atacar a los endpoint de la API de origen es la obtención de datos para almacenarlos en la base de datos, y crear una nueva API que muestre los datos utilizando GraphQL.

A lo largo de este proyecto se va a atacar a tres endpoints de la API de origen, dichos endpoints son:

- GET Countries: Se hará uso de este endpoint para obtener una lista de todos los países de los que contiene información la API, estos países se almacenarán en una estructura de datos y se recorrerá para atacar a los endpoints de los países de los que se desea obtener información. En la Figura 4.2-1 se muestra un ejemplo de la respuesta que cabe esperar de dicho endpoint:

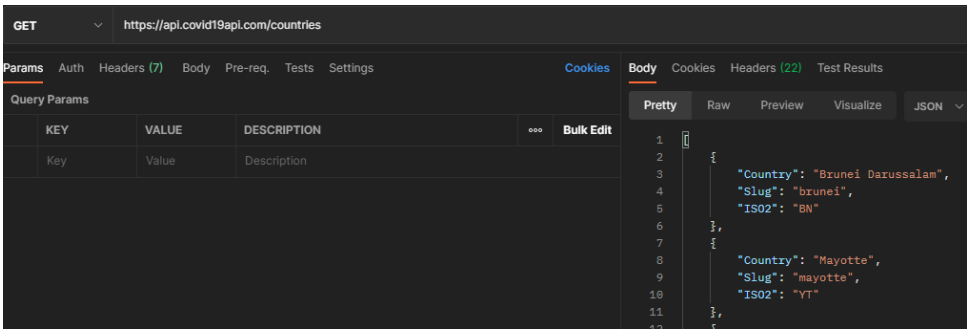

Figura 4.2-1 Obtención de países.

- GET By Country Live: Este punto de acceso devolverá información sobre los casos relativos a un país en concreto, para acceder a este será necesario especificar el nombre del país separado por "-"(el contenido de *Slug* de la Figura 4.2-1), un *status* (confirmed, deaths o recovered), junto a dos fechas de referencia sobre las que se quieren obtener

los datos. En la siguiente captura se muestra un ejemplo de la respuesta que cabe esperar:

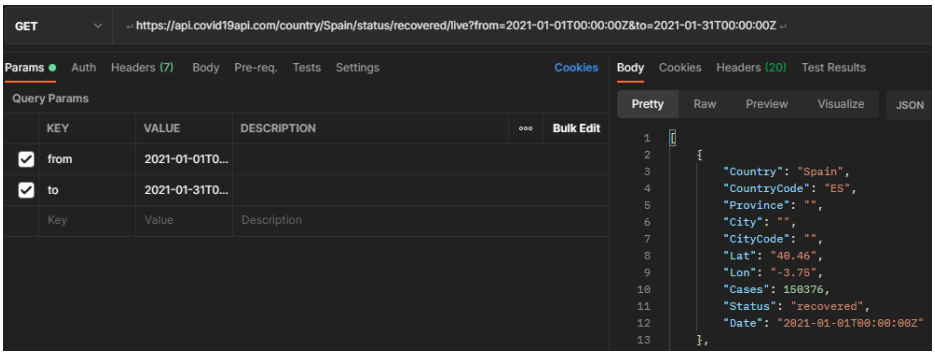

Figura 4.2-2 Obtención de casos de un país.

- GET Summary: Devuelve un resumen a nivel global sobre los nuevos confirmados, muertes y recuperados entre otros datos. A continuación se muestra una captura con los resultados que se obtienen de dicho endpoint:

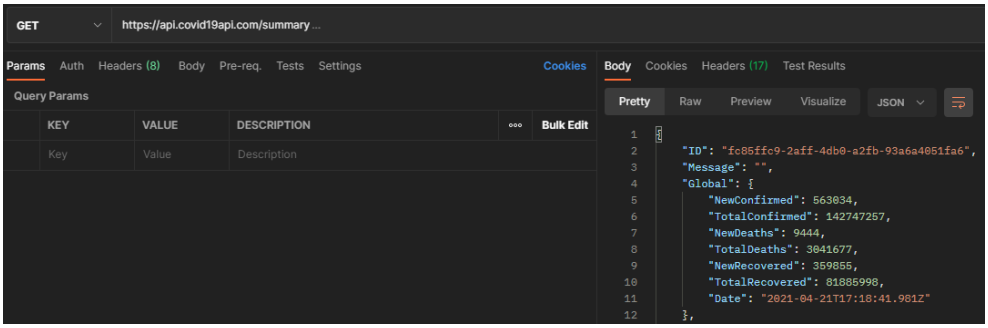

Figura 4.2-3 Obtención de resumen global.

A partir de la información obtenida de estos endpoint se va a implementar una API utilizando lo aprendido en el capítulo anterior.

4.3 IMPLEMENTACIÓN DE LA API

Se va a implementar una API mostrará datos globales sobre la enfermedad (Summary) y también de España e Italia (By Country Live), el endpoint Countries se ha utilizado para obtener datos de todos los países, estos datos se almacenan dentro de la ruta *com.TFG.COPIASEGURIDAD*.

Dentro de esa ruta se encuentra información de todos los países desde el 22 de enero de 2020 hasta el 1 de febrero de 2021, en un principio el enfoque del proyecto era que la API mostrase datos sobre todos los países individualmente, pero finalmente se tomó la decisión de que solo fueran España e Italia.

Esto se debe a que en función de cómo son introducidos los datos para cada país en el endpoint By Country Live, estos pueden tener campos extra como por ejemplo *Province* o *City*, en el caso de España los datos no están divididos por provincias ni por ciudades (el valor del campo es null), pero en el caso de Francia o de Reino Unido sí que se divide por provincias.

Por lo que si se quisiera que nuestra API mostrase los datos de todos los países habría que contemplar que algunos de ellos estarían divididos por provincias y otros no, lo que implicaría

la creación de entidades extra que tuvieran esos campos que algunos países no tienen, lo que aumentaría la complejidad del proyecto considerablemente.

Además de la implementación de nuevas entidades, habría que tomar en consideración que la fuente de datos no siempre está en funcionamiento, ya que debido a la saturación se cae constantemente, lo que dificulta obtener de forma continua y consistente datos sobre todos los países.

La API de origen es capaz de detectar cuando se está haciendo uso abusivo de uno o varios endpoints, lo cual se traduce en un baneo de la IP que está realizando las peticiones, es por esto, sumado al hecho de que no siempre está disponible y teniendo en cuenta que hay países con campos extra que finalmente el proyecto se ha enfocado en España e Italia.

El esquema GraphQL está formado por dos Types, el primero es *CountryState* que va a almacenar los datos del endpoint *By Country Live*, y el segundo es el contenido del endpoint *Summary*:

```
type CountryState{
    idcs: ID!,
    country:String,
    countryCode:String,
    province: String,
    city : String,
    cityCode: String,
    lat: Float,
    lon : Float,
    cases: Int,
    status: String,
    date: String
}

type Summary{
    idsum: ID!,
    newConfirmed: Int,
    totalConfirmed: Int,
    newDeaths : Int,
    totalDeaths : Int,
    newRecovered : Int,
    totalRecovered : Int,
    date : String

}
```

Figura 4.3-1 Definición de Types.

Este esquema definirá las queries que el usuario puede realizar, pero en este caso no se definirá ninguna mutacion, pues no se contempla que el usuario pueda añadir o modificar datos de la API.

```
type Query{
    obtenerDatosDeUnMes(country:String,status:String,date:String) : [CountryState]
    obtenerDatosEntreFechasQL(csdd: CSDD) : [CountryState]
    obtenerDatosEntreFechas(country:String,status:String,date:String,date2:String) : [CountryState]
    obtenerDatosDeUnDiaQL(country:String,status:String,date:String) : [CountryState]
    obtenerDatosDeUnDia(country:String,status:String,date:String) : [CountryState]
    obtenerDatosLatLonQL(fechasInput: FechasInput) : [CountryState]
    obtenerSummaryEntreFechasQL(date:String, date2:String) : [Summary]
}
```

Figura 4.3-2 Definición de Queries.

En la ruta *src/main/resources/graphql* se encuentra el esquema de la API. Como ya sabemos, GraphQL trabaja con un único endpoint de acceso donde el usuario realiza una petición POST y GraphQL se encarga de resolverla y devolver los datos que el usuario ha solicitado, es por esto que se ha decidido incluir en el esquema las entradas que se esperan por parte del usuario, tal y como se muestra en la siguiente captura:

```
input LLSDD{
    lat: Float,
    lon: Float,
    status: String,
    date: String,
    date2: String
}

input CSDD{
    country: String,
    status: String,
    date: String,
    date2: String
}

input DD{
    date1: String,
    date2: String
}
```

Figura 4.3-3 Definición de Inputs.

En Java también se han definido dichos Inputs, se encuentran dentro de la ruta *src/main/java/com.TFG.inputs*.

4.3.1 Tratamiento de datos provenientes de la API.

El primer endpoint al que accede el programa es a */Countries*, cuando obtiene los datos de dicho endpoint lo almacena en el archivo *países.txt* de la ruta *com.TFG.archivos*.

```
public ArrayList<String> extraerURLPaises(String direccionPaises) throws IOException, ParseException  {

    StringBuffer texto = new StringBuffer();

    try {
        URL url = new URL(direccionPaises);

        try {

            url.openConnection();
            InputStreamReader in = new InputStreamReader((InputStream) url.getContent());
            BufferedReader buff = new BufferedReader(in);
            String linea = "";

            while(linea != null) {
                linea = buff.readLine();
                if(linea != null ) {
                    texto.append(linea + "\n");
                }
            }
            buff.close();

        }catch(IOException e) {
            System.out.println("Error IOException");
        }
    }catch(MalformedURLException e) {
        System.out.println("Error MalformedURLException");
    }
    FileWriter myWriter = new FileWriter(absolutePath);
    myWriter.write(texto.toString());
    myWriter.close();
    return null;

}
```

Figura 4.3-4 Método para obtener información sobre los países.

Del archivo de países interesa el contenido de *Slug* Figura 4.2-1, se utilizará un `ArrayList<String>` que almacenará los slugs de cada país:

```
//RELLENAR ARRAY DE PAISES
public ArrayList<String> crearPaises() throws FileNotFoundException{
    String [] items = null;
    String linea = "";
    Scanner scan  = null;
    paises = new ArrayList<String> ();
    scan = new Scanner(new File(absolutePath));
    while(scan.hasNextLine()) {
        linea = scan.nextLine().trim();
        if(linea.isEmpty() || !linea.startsWith("\""+"Slug"+"\"")) continue;
        items = linea.split(":");
        paises.add(items[1].replace(",", "").trim());
    }
    scan.close();
    return paises;

}
```

Figura 4.3-5 Crear ArrayList de países.

Para poder acceder al endpoint By Country Live, hay que especificar un país (el slug almacenado en el ArrayList de países), un estado y dos fechas, por esto mismo se ha decidido

hacer uso de una estructura de datos que almacene los slugs, para luego recorrerla con un bucle y atacar a la URL para obtener los datos de los países:

```
340     public void obtenerTodo(String direccion) throws IOException, InterruptedException {
341         for (String p : paises) {
342             String a = p.replaceAll("\"","");
343             //Descargar muertes
344 
345             System.out.println("Iniciando descarga de "+a);
346             Thread.sleep(5*1000);
347             extraerURL(direccion.replace("PAIS", a).replace("ESTADO", "deaths"));
348             System.out.println("Descarga de "+a+ " "+ "deaths" + " completada");
349             //Descargar confirmados
350 
351             Thread.sleep(15*1000);
352             extraerURL(direccion.replace("PAIS", a).replace("ESTADO", "confirmed"));
353             System.out.println("Descarga de "+a+ " "+ "confirmed" + " completada");
354             //Descargar altas
355 
356             Thread.sleep(15*1000);
357             extraerURL(direccion.replace("PAIS", a).replace("ESTADO", "recovered"));
358             System.out.println("Descarga de "+a+ " "+ "recovered" + " completada");
359 
360         }
361 
362     }
```

Figura 4.3-6 Obtención de datos de todos los países.

Una vez descargados los ficheros con los datos de los países, se listan los ficheros de la ruta *com.TFG.archivos***,** y se leen utilizando un *Scanner* tal y como se muestra en la siguiente captura:

```
237 public void cargarArchivosBD(String ruta) throws IOException {
238 
239     File carpetaArchivos = new File(ruta);
240     String [] lista = carpetaArchivos.list();
241     if(lista == null || lista.length == 0) {
242         System.out.println("<ERROR cargarArchivosBD> La ruta especificada no contiene elementos. La ruta es: "+ "ruta");
243     }else {
244         for(int i = 0; i<lista.length;i++) {
245             File file = new File(ruta+lista[i]);
246             String [] items = null;
247             String linea = "";
248             Scanner scan  = null;
249             System.out.println(lista[i]);
250             scan = new Scanner(file);
251             System.out.println("Leyendo archivo: "+lista[i]);
252             while(scan.hasNextLine()) {
253                 linea = scan.nextLine().replace("\"", "").replace("},", "}\n").
254                         replace("[{Country:","").replace("CountryCode:","").replace("Province:,","").
255                         replace("City:,", "").replace("CityCode:,", "").replace("Lat:", "").replace("Lon:","").
256                         replace("Cases:", "").replace("Status:", "").replace("Date:", "").replace("{Country:","").
257                         replace("}","").replace(",]", "").
258                         trim();
259                 if(linea.isEmpty() || linea.startsWith("[]")) continue;
260                 items = linea.split(",");
261                 if(items.length == 7) {
262 
263                     CountryState cs = new CountryState(items[0], items[1],"" ,"" ,Float.parseFloat(items[2]), Float.parseFloat(items[3]),
264                             Integer.parseInt(items[4]), items[5], items[6].replace("}", "").replace("T","-T"));
265                     csr.save(cs);
266                 }else if (items.length == 8) {
267                     CountryState cs = new CountryState(items[0], items[1],items[2] ,"" ,Float.parseFloat(items[3]), Float.parseFloat(items[4]),
268                             Integer.parseInt(items[5]), items[6], items[7].replace("}", "").replace("T","-T"));
269                     csr.save(cs);
270                 }else if (items.length == 9) {
271                     CountryState cs = new CountryState(items[0], items[1],items[2] ,items[3] ,"" ,Float.parseFloat(items[4]), Float.parseFloat(items[5]),
272                             Integer.parseInt(items[6]), items[7], items[8].replace("}", "").replace("T","-T"));
273                     csr.save(cs);
274                 }else if (items.length == 10) {
275                     CountryState cs = new CountryState(items[0], items[1],items[2] ,items[3] ,items[4] ,Float.parseFloat(items[5]), Float.parseFloat(items[6]),
276                             Integer.parseInt(items[7]), items[8], items[9].replace("}", "").replace("T","-T"));
277                     csr.save(cs);
278                 }
279             }
280 
281             scan.close();
282         }
283         System.out.println("Archivos CountryState cargados en BD exitosamente");
284 
285     }
286 
287 }
```

Figura 4.3-7 Creación de objetos y carga en la base de datos.

Si bien en una ejecución normal del programa no se hace uso del método de la Figura 4.3-6 por los motivos expuestos anteriormente, sí que se utilizaron para obtener los datos de la ruta *com.TFG.COPIASEGURIDAD*.

La base de datos debe almacenar los datos provenientes del endpoint By Country Live (casos por país) y Summary (resumen global), por lo que tendremos dos tablas cuyas columnas se corresponden con cada uno de los datos que recibimos en el archivo JSON.

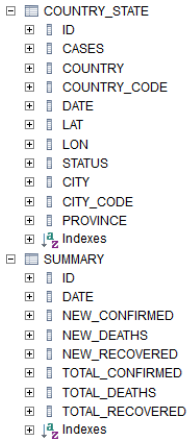

Figura 4.3-8 Estructura de las tablas de la base de datos.

El programa se conectará a la API fuente realizando tres peticiones por cada uno de los países de los que queramos obtener los datos (confirmados, muertes y altas) y almacenará los archivos en una ruta predefinida. En el caso de los archivos obtenidos de By Country Live se almacenarán en *com/TFG/archivos* y para Summary se guardarán en *com/TFG/summary*.

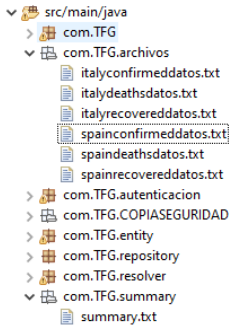

Figura 4.3-9 Directorio de los datos.

El motivo por el cual se almacenan los datos es debido a que el programa obtendrá los datos de los endpoints una única vez, los guardará en un directorio y luego leerá esos archivos para rellenar las tablas de la base de datos. Esto quiere decir que no es necesario disponer de una conexión a Internet para que el programa funcione, ya que este funciona en local al tener ya los archivos descargados previamente.

4.3.2 Ejemplo de ejecución del programa

Cuando ejecutemos el programa, obtendremos por consola algunos mensajes de referencia indicando el estado del programa, por cada uno de los archivos leídos y cargados en la base de datos el programa mostrará un mensaje, y al terminar nos avisará de que se han cargado de forma correcta.

Figura 4.3-10 Resultado de ejecución del programa.

Una vez ejecutado, si accedemos a la base de datos H2 podremos comprobar que los datos se han cargado de forma correcta.

92	1953509	Italy	IT	2021-01-28T00:00:00Z	41.8699989318847 66	12.5699996948242 19	recovered
93	1973388	Italy	IT	2021-01-29T00:00:00Z	41.869998931884766	12.569999694824219	recovered
94	1990152	Italy	IT	2021-01-30T00:00:00Z	41.869998931884766	12.569999694824219	recovered
95	2010548	Italy	IT	2021-01-31T00:00:00Z	41.869998931884766	12.569999694824219	recovered
96	2024523	Italy	IT	2021-02-01T00:00:00Z	41.869998931884766	12.569999694824219	recovered
97	1928265	Spain	ES	2021-01-01T00:00:00Z	40.459999084472656	-3.75	confirmed
98	1928265	Spain	ES	2021-01-02T00:00:00Z	40.459999084472656	-3.75	confirmed
99	1928265	Spain	ES	2021-01-03T00:00:00Z	40.459999084472656	-3.75	confirmed
100	1958844	Spain	ES	2021-01-04T00:00:00Z	40.459999084472656	-3.75	confirmed

Figura 4.3-11 Datos cargados correctamente en la base de datos.

Ahora podemos abrir Postman para realizar las consultas que consideremos.

4.4 USO DE POSTMAN PARA LA CONSULTA DE DATOS

La API permite consultar los datos utilizando GraphQL, y en el caso de que el usuario no sepa utilizar este lenguaje también puede introducir los datos a través de la URL y obtener una respuesta de la API en formato JSON.

Esto es posible debido a que cuando realizamos una petición GET, esta internamente llama al endpoint que resuelve la petición utilizando GraphQL y devuelve los datos.

Una vez que los datos han sido cargados en la base de datos, podemos iniciar Postman y realizar peticiones para ver los resultados, pero antes voy a explicar el formato que deben seguir los endpoints para obtención de datos.

- La dirección para obtener datos de la tabla *CountryState*, que contiene los datos por país es *http://localhost:8080/country=PAÍS/status=STATUS/from=YYYY-MM-DD/to=YYYY-MM-DD* en *PAÍS* tendremos que escribir el nombre del país que queramos consultar, en *STATUS* los tipos de casos que queremos que devuelva (confirmed, deaths o recovered) y entre que dos fechas queremos los datos en formato yyyy-mm-dd. A continuación muestro una captura obteniendo los datos para los contagios en España, entre los meses de enero y febrero de 2021:

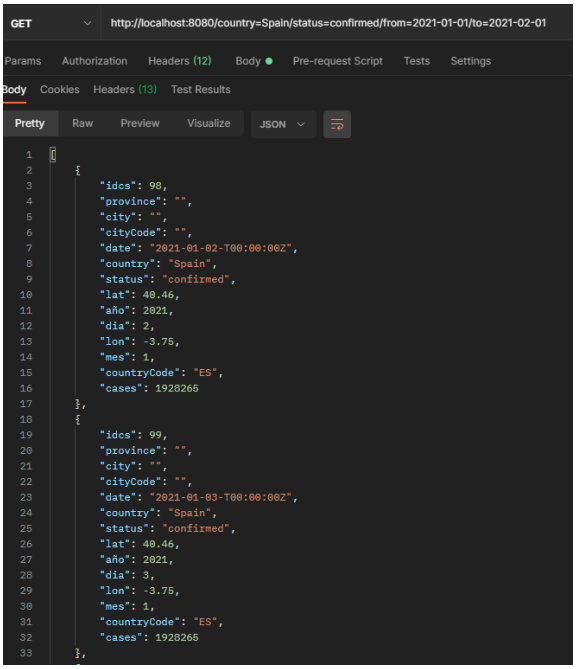

Figura 4.4-1 Confirmados en España entre Enero y Febrero de 2021.

También es posible obtener los datos en función de la latitud y la longitud de un país, además hay que añadir un status y dos fechas de referencia:

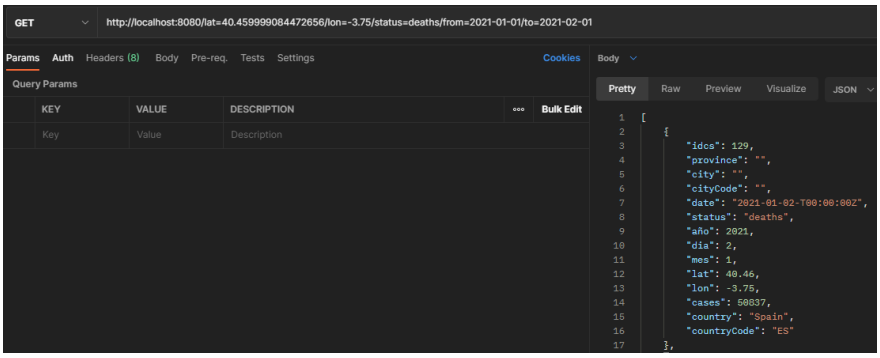

Figura 4.4-2 Obtener datos de un país en función a la latitud y la longitud

- Ahora vamos a realizar la misma petición de la Figura 4.4-1 pero accediendo desde el endpoint que soporta GraphQL. Como ya sabemos, para realizar una petición GraphQL esta debe ser POST, el endpoint que se corresponde con dicha petición es `http://localhost:8080/graphql` y aquí muestro una captura de su funcionamiento:

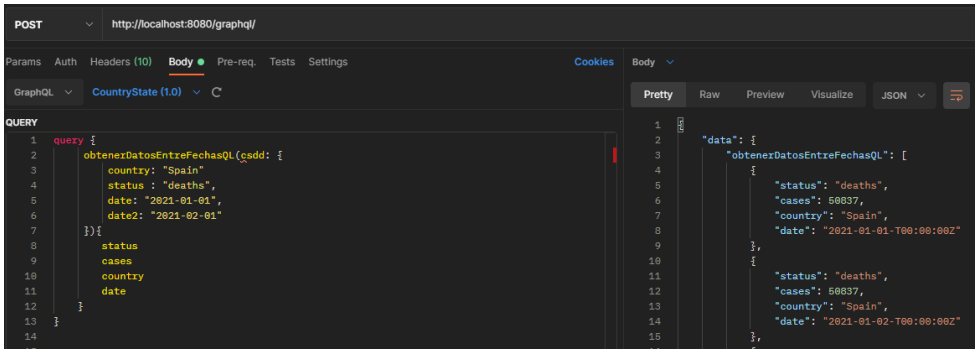

Figura 4.4-3 Query en GraphQL para la obtención de muertes en España entre Enero y Febrero de 2021.

En la captura anterior se puede observar la query en GraphQL junto a los datos que solicitamos, que son el ID, el tipo de caso, el número de casos, el país y la fecha junto a su resultado, aquí muestro su resolución en *Java*:

```
@RequestMapping(value = "/country={country}/status={status}/from={date}/to={date2}", method = RequestMethod.GET)
public List<CountryState> obtenerDatosEntreFechas(@PathVariable("country") String country, @PathVariable("status")String status,
        @PathVariable("date") String date, @PathVariable("date2") String date2)  {
    CSDD csdd = new CSDD(country,status,date,date2);
    return obtenerDatosEntreFechasQL(csdd);

}

public List<CountryState> obtenerDatosEntreFechasQL(CSDD csdd){

    String [] entrada = csdd.getDate().split("-");
    int eA = Integer.parseInt(entrada[0]); int eM = Integer.parseInt(entrada[1]); int eD = Integer.parseInt(entrada[2]) -1;
    String [] entrada2 = csdd.getDate2().split("-");
    int eA2 = Integer.parseInt(entrada2[0]); int eM2 = Integer.parseInt(entrada2[1]); int eD2 = Integer.parseInt(entrada2[2]) +1;
    List<CountryState> l = new ArrayList<CountryState>();
    Date d = new Date(eA,eM,eD);
    Date d2 = new Date(eA2,eM2,eD2);
    for (CountryState cs : csr.findAll()) {
        if(cs.getCountry().equals(csdd.getCountry())&& cs.getStatus().equals(csdd.getStatus())) {
            Date o = new Date(cs.getAño(),cs.getMes(),cs.getDia());
            if(d.before(o) && d2.after(o)) {
                l.add(cs);
            }
        }

    }

return l;

}
```

Figura 4.4-4 Obtener datos de CountryState entre fechas en Java.

- Si ahora deseamos obtener los datos de un país en función de la latitud y la longitud, accedemos por el mismo endpoint */graphql* tal y como muestro a continuación:

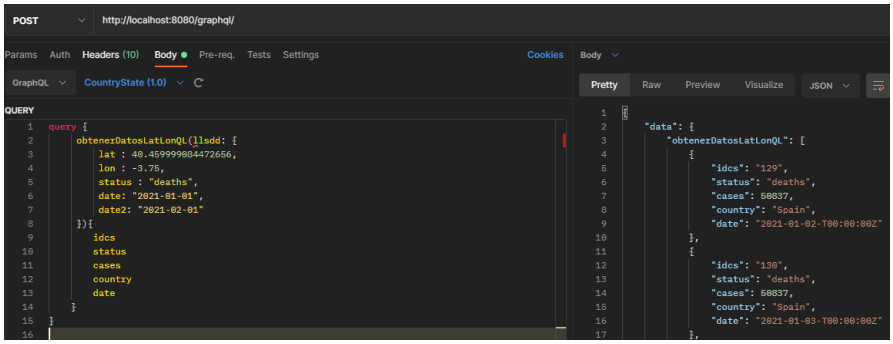

Figura 4.4-5 Query en GraphQL para obtener las muertes entre enero y febrero de 2021 según la latitud y longitud.

En la siguiente captura se muestra la resolución en Java:

```
@RequestMapping(value = "/lat={lat}/lon={lon}/status={status}/from={date}/to={date2}", method = RequestMethod.GET)

public List<CountryState> obtenerDatosLatLon(@PathVariable("lat") float lat, @PathVariable("lon") float lon,
        @PathVariable("status") String status,@PathVariable("date") String date, @PathVariable("date2") String date2){
    LLSDD llsdd = new LLSDD(lat,lon,status,date,date2);
    return obtenerDatosLatLonQL(llsdd);
}

public List<CountryState> obtenerDatosLatLonQL(LLSDD llsdd){
    String [] entrada = llsdd.getDate().split("-");
    int eA = Integer.parseInt(entrada[0]); int eM = Integer.parseInt(entrada[1]); int eD = Integer.parseInt(entrada[2]);
    String [] entrada2 = llsdd.getDate2().split("-");
    int eA2 = Integer.parseInt(entrada2[0]); int eM2 = Integer.parseInt(entrada2[1]); int eD2 = Integer.parseInt(entrada2[2]);
    List<CountryState> l = new ArrayList<CountryState>();
    Date d = new Date(eA,eM,eD);
    Date d2 = new Date(eA2,eM2,eD2);
    for (CountryState cs : csr.findAll()) {
        if(cs.getStatus().equals(llsdd.getStatus()) && cs.getLat() == llsdd.getLat()&& cs.getLon() ==llsdd.getLon() ) {
            Date o = new Date(cs.getAño(),cs.getMes(),cs.getDia());
            if(d.before(o) && d2.after(o)) {
                l.add(cs);
            }
        }

    }

return l;

}
```

Figura 4.4-6 Obtener datos de CountryState según latitud y longitud en Java.

En caso de que queramos obtener más datos de la tabla CountryState, lo único que tendríamos que hacer es en el recuadro *Query* añadir los que queramos y volver a ejecutar.

- Para la obtener datos de la tabla Summary, podemos introducir los datos mediante la *URL* tal y como muestro a continuación:

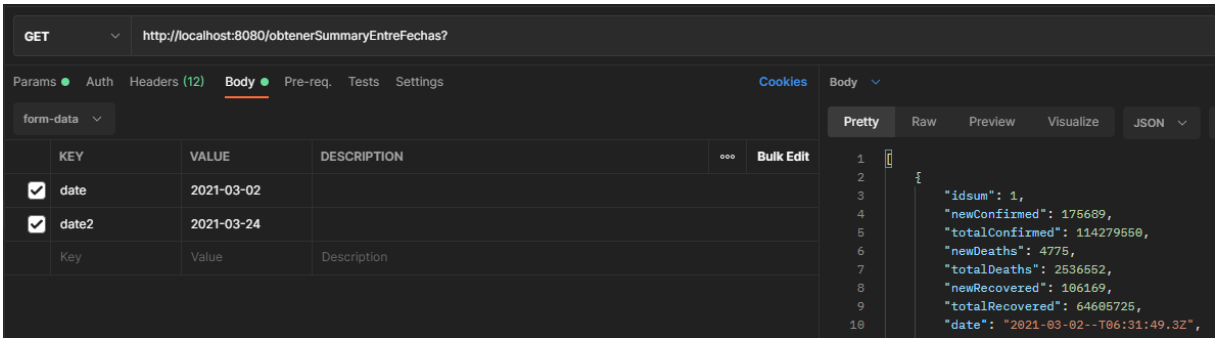

Figura 4.4-7 Obtener datos de Summary entre dos fechas

- O podemos acceder al endpoint */graphql* especificando la query tal y como se muestra en la siguiente captura:

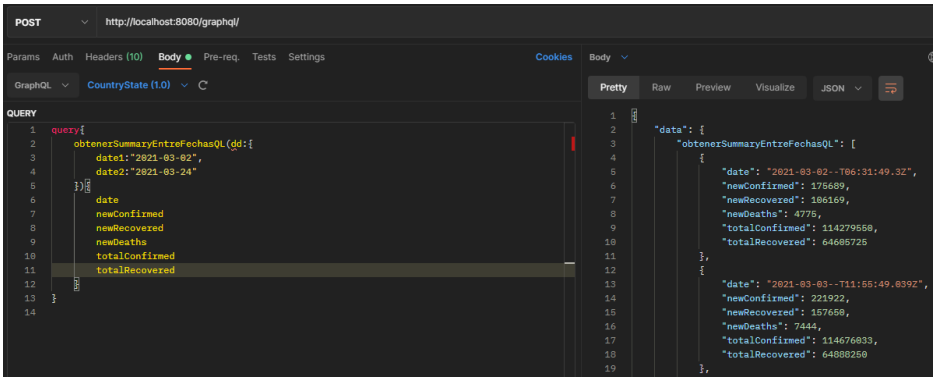

Figura 4.4-8 Query en GraphQL para obtener los nuevos confirmados, nuevos recuperados junto al total de confirmados y total de recuperados de la tabla Summary.

La implementación en Java es tal y como muestro a continuación:

```
@RequestMapping(value = "/obtenerSummaryEntreFechas/from={date}/to={date2}", method = RequestMethod.GET)
public List<Summary> obtenerSummaryEntreFechas(@PathVariable("date")String date, @PathVariable("date2")String date2){
    DD dd = new DD(date,date2);
    return obtenerSummaryEntreFechasQL(dd);

}

@RequestMapping(value = "graphql/obtenerSummaryEntreFechasQL", method = RequestMethod.POST)
public List<Summary> obtenerSummaryEntreFechasQL(DD dd){
    String [] entrada = dd.getDate1().split("-");
    int eA = Integer.parseInt(entrada[0]); int eM = Integer.parseInt(entrada[1]); int eD = Integer.parseInt(entrada[2]) -1;
    String [] entrada2 = dd.getDate2().split("-");
    int eA2 = Integer.parseInt(entrada2[0]); int eM2 = Integer.parseInt(entrada2[1]); int eD2 = Integer.parseInt(entrada2[2])+1;
    List<Summary> l = new ArrayList<Summary>();
    Date d = new Date(eA,eM,eD);
    Date d2 = new Date(eA2,eM2,eD2);
    for (Summary s : sr.findAll()) {
            Date o = new Date(s.getAño(),s.getMes(),s.getDia());
            if(d.before(o) && d2.after(o)) {
                l.add(s);
            }
    }
return l;

}
```

Figura 4.4-9 Obtener Summary entre fechas desde Java.

Los datos que nos devuelve sobre los casos de COVID podemos introducirlos en MetricsGraphics para generar gráficas y poder estudiar más a fondo cómo evoluciona la enfermedad.

4.5 UTILIZACIÓN DE METRICSGRAPHICS PARA GENERAR GRÁFICAS

MetricsGraphics es una librería para la visualización de datos de forma simple, soporta una amplia variedad de gráficas como gráficas lineales, de dispersión, histogramas o de barras. Para

trabajar con esta herramienta, utilizamos los archivos JSON que devuelve la API y luego especificamos los parámetros para generar las gráficas.

4.5.1 Evolución de los casos globales

En este apartado vamos a trabajar con los datos de la tabla Summary (globales) para generar gráficas y ver cómo evolucionan los casos de Coronavirus globalmente.

Lo primero es obtener los datos de la tabla Summary desde Postman. Para generar las gráficas vamos a necesitar el contenido de *newConfirmed*, *newRecovered*, *newDeaths* y la fecha, que va a ser del 2 al 24 de Marzo.

En la siguiente captura muestro como obtener dichos datos utilizando GraphQL, el resultado de la petición debe ser similar a la siguiente figura:

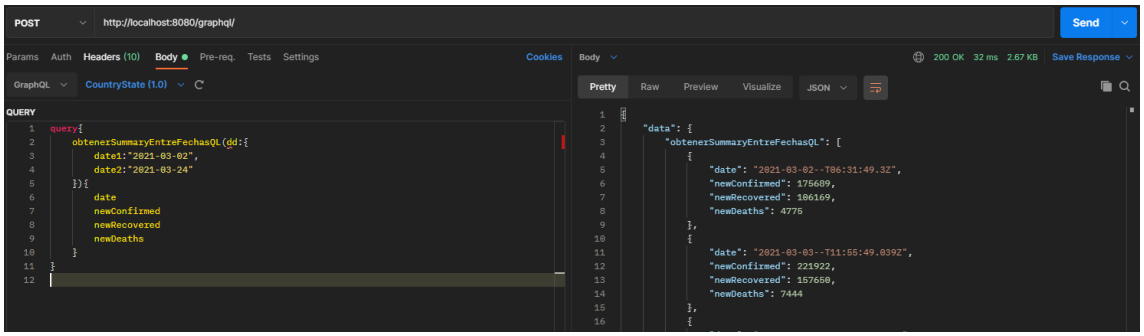

Tabla 4.5.1-1 Obtener nuevas muertes, nuevos confirmados y nuevas altas globales con GraphQL.

A continuación cargamos el archivo en MetricsGraphics, y generamos una gráfica utilizando el siguiente *script*:

```
MG.data_graphic({
  title: "Nuevos Confirmados Globales (Del 2 al 24 de Marzo)",
  description: "Desde el 2 al 24 de Marzo",
  data: JSON.parse(document.querySelector('.data textarea').value),
  markers: [{'dia': 9, 'label': 'Día con menor número de casos'}],
  width: 600,
  height: 400,
  target: ".result",
  x_accessor: "dia",
  x_label:"Días de Marzo",
  y_label: "Número de casos",
  y_accessor: "newConfirmed",
  chart_type:'line',

});
```

Figura 4.5-1 Ejemplo de Script para generar gráficas con MetricsGraphics.

El resultado de ejecutar dicho *script* es la Figura 4.5-2, que muestra la evolución de casos globales confirmados entre el 2 y el 24 de Marzo, donde se puede observar que el día con menor número de casos fue el Martes 9 con un total de 175,328 casos y el día con mayor número de casos fue el Miércoles 24 con un total de 325,641 nuevos confirmados.

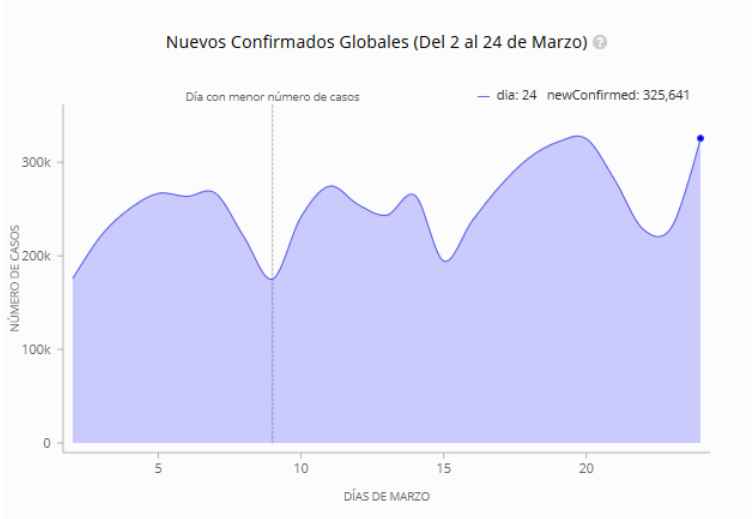

Figura 4.5-2 Nuevos Confirmados Globales.

Para mostrar el número de nuevas muertes simplemente modificamos el parámetro *y_accesor* indicando que representa atributo *newDeaths*. En la Figura 4.5-3 puede observarse la evolución del número de muertes globales, teniendo su máximo el Jueves 4 con un total de 8.064 casos y un mínimo el Lunes 8 con 3.396 casos registrados.

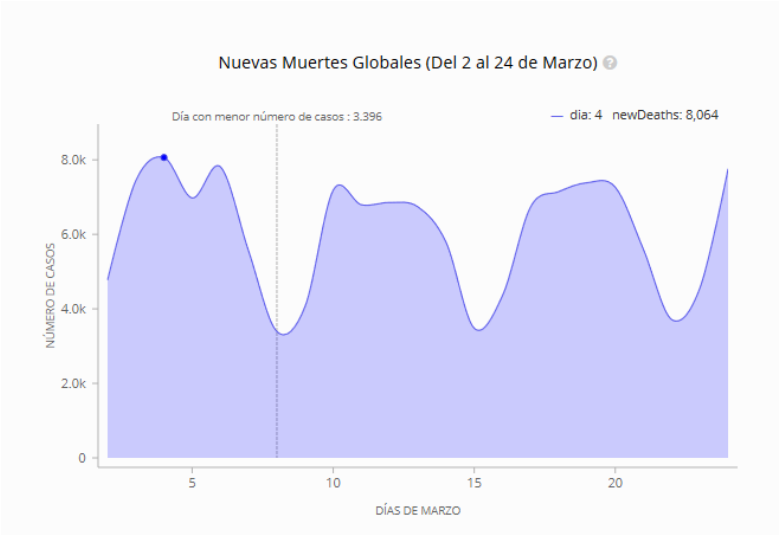

Figura 4.5-3 Nuevas Muertes Globales.

Si modificamos el script para mostrar los casos recuperados, modificamos el parámetro *y_accesor* por el atributo *newRecovered*, el resultado es la Figura 4.5-4 que muestra el número de recuperaciones entre el 2 y el 24 de Marzo, donde el Martes 23 se alcanzó el máximo número de recuperaciones (196,159) y el Lunes 15 el mínimo (102,497).

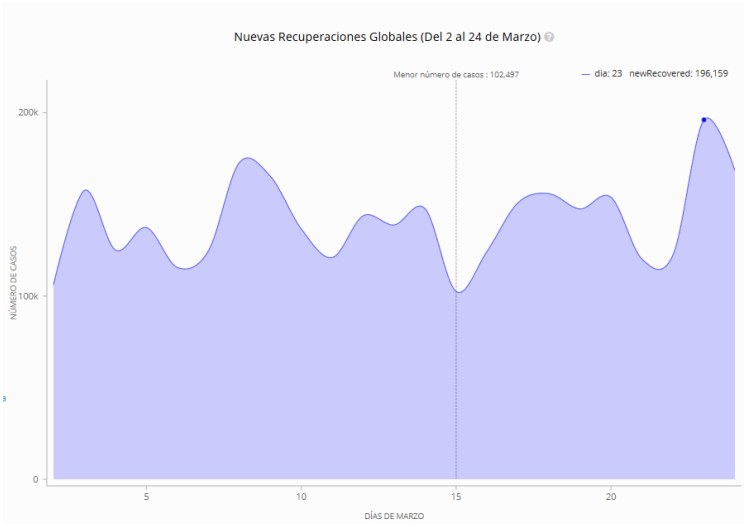

Figura 4.5-4 Nuevos Recuperados Globales.

4.5.2 Evolución de los casos en España

El contenido de la tabla CountryState almacena datos acumulativos sobre los diferentes casos de Coronavirus en un país determinado, esto quiere decir que las gráficas que se van a generar para los países de España e Italia van a ser una línea ascendente para los tres tipos de casos.

Esto se debe a que el número de confirmados, de muertes y de recuperados siempre va a ir en aumento al ser un valor acumulativo.

Los datos a partir de los cuales van a generarse las gráficas son del mes de Enero, para obtenerlos es necesario especificar el nombre del país, el tipo de caso, y las fechas entre las que solicitamos los datos en el endpoint */graphql*.

Para hacer esto, ejecutamos la siguiente query en GraphQL desde Postman:

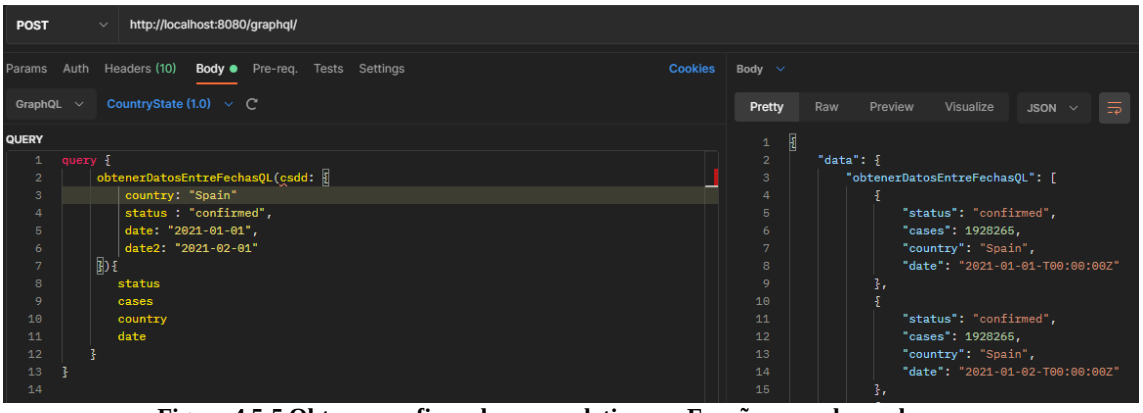

Figura 4.5-5 Obtener confirmados acumulativos en España para el mes de enero.

Ahora introducimos el archivo JSON en MetricsGraphics y ejecutamos el siguiente script:

```
MG.data_graphic({
  title: "Evolución de casos confirmados en España durante Enero de 2021",
  description: "Casos confirmados en España durante Enero de 2021",
  data: JSON.parse(document.querySelector('.data textarea').value),
  markers: [{'dia': 1, 'label': 'Día 1: 1.928.265 casos'}],
  width: 800,
  height: 600,
  target: ".result",
  x_accessor: "dia",
  x_label:"Días de Enero",
  y_accessor: "cases",
  y_label: "Número de casos"
});
```

Figura 4.5-6 Script para generar gráfica de confirmados en España.

El resultado de ejecutar el script anterior es la Figura 4.5-7 donde se puede apreciar la evolución del número de confirmados en España durante el mes de Enero, donde ha habido un total de 814.854 contagios solo durante ese mes.

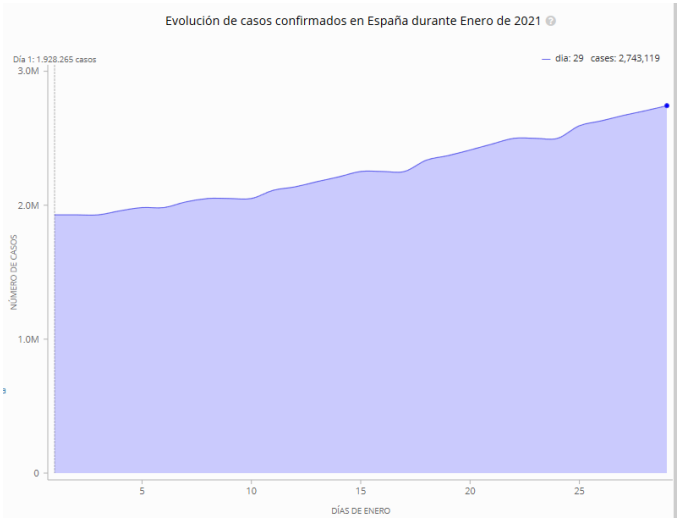

Figura 4.5-7 Evolución de los casos confirmados en España durante enero de 2021.

Para obtener los casos de muertes en España simplemente modificamos la query de la Figura 4.5-5 Obtener confirmados acumulativos en España para el mes de eneroespecificando el Status *deaths*, introducimos el archivo JSON en MetricsGraphics y volvemos a ejecutar el script. El resultado es la Figura 4.5-8 donde se puede observar la evolución de las muertes en España durante Enero de este año, acumulando un total de 7.482 muertes en ese mes.

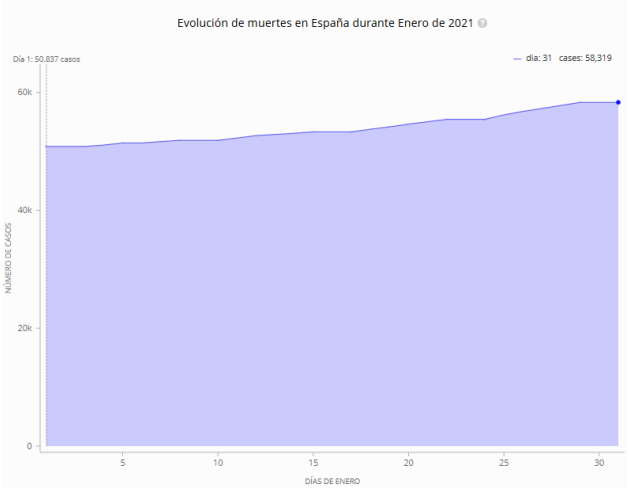

Figura 4.5-8 Evolución de los casos de muertes en España durante enero de 2021.

Para obtener las altas en España modificamos la query de la Figura 4.5-5 especificando *recovered* en *Status*, obtenemos el archivo JSON y lo introducimos en MetricsGraphics para ejecutar una vez más el *script*.

Antes de mostrar la gráfica referente a los casos de recuperados en España, si observamos los archivos JSON que devuelve la API de origen, nos daremos cuenta de que el número de recuperaciones no avanza en el mes de enero.

Esto se debe a un error a la hora de introducir los datos en la API de origen, y este error se remonta al 18 de mayo de 2020 hasta el mes de mayo de 2021. El error en cuestión es que a partir de dicha fecha se registra un total de 150.376 recuperaciones y este número en lugar de ir incrementando con el paso del tiempo, se repite constantemente.

Por este motivo, en lugar de mostrar la gráfica referente al mes de enero de 2021, voy a mostrar la gráfica que representa los recuperados registrados durante el mes de abril de 2020, donde se puede observar un aumento en los casos recuperados, ya que si mostrase la gráfica del mes de enero lo que podríamos observar es una línea paralela al eje de coordenadas X, indicando que no ha habido ninguna recuperación, pues el valor se repite para cada uno de los días de dicho mes.

En la siguiente gráfica se puede observar la evolución durante el mes de abril de 2020 de los casos recuperados, donde se registró un total de 89.403 casos recuperados.

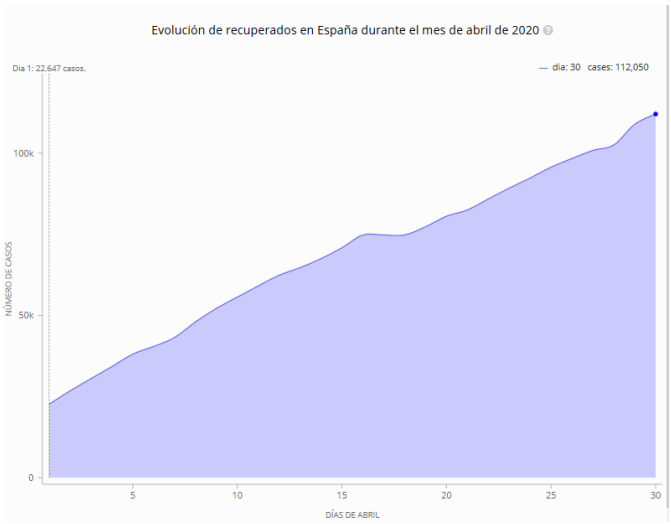

Figura 4.5-9 Evolución de recuperados en España durante abril de 2020.

4.5.3 Evolución de los casos en Italia

Para obtener los datos de Italia desde Postman, accedemos al endpoint */graphql* y escribimos la query especificando los datos que deseamos, el nombre del país, Status *confirmed* junto a la fecha de inicio y la fecha de fin, tal y como muestro en la siguiente captura:

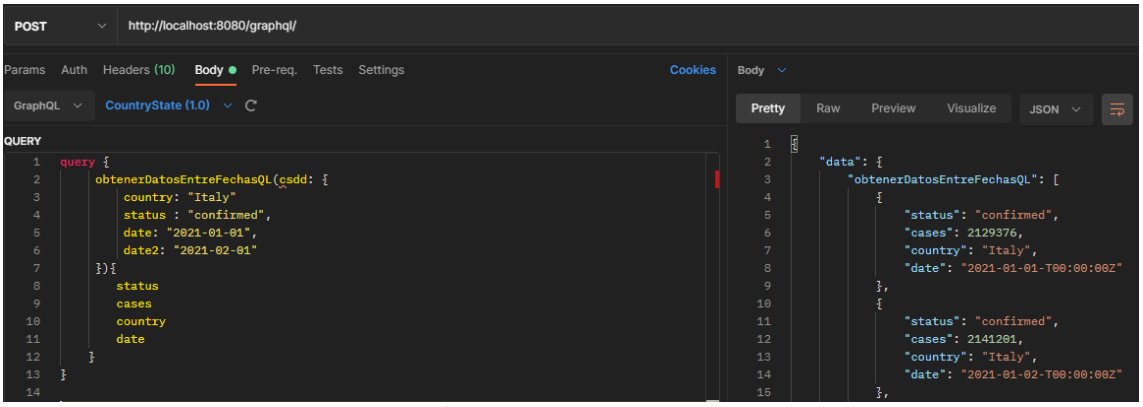

Figura 4.5-10 Obtener confirmados acumulativos de Italia en el mes de enero de 2021.

Una vez obtenidos los datos para Italia, introducimos el archivo JSON en MetricsGraphics y ejecutamos el siguiente script:

```
MG.data_graphic({
  title: "Evolución de confirmados en Italia durante Enero de 2021",
  description: "Casos confirmados en Italia durante Enero de 2021",
  data: JSON.parse(document.querySelector('.data textarea').value),
  markers: [{'dia': 1, 'label': 'Día 1: 2.129.376 casos'}],
  width: 800,
  height: 600,
  target: ".result",
  x_accessor: "dia",
  x_label:"Días de Enero",
  y_accessor: "cases",
  y_label: "Número de casos"
});
```

Figura 4.5-11 Script para generar gráfica de confirmados en Italia.

Si ejecutamos el script el resultado es la Figura 4.5-12 donde se muestra la evolución de los confirmados en Italia durante 2021, habiendo un total de 423.656 nuevos confirmados durante esta fecha.

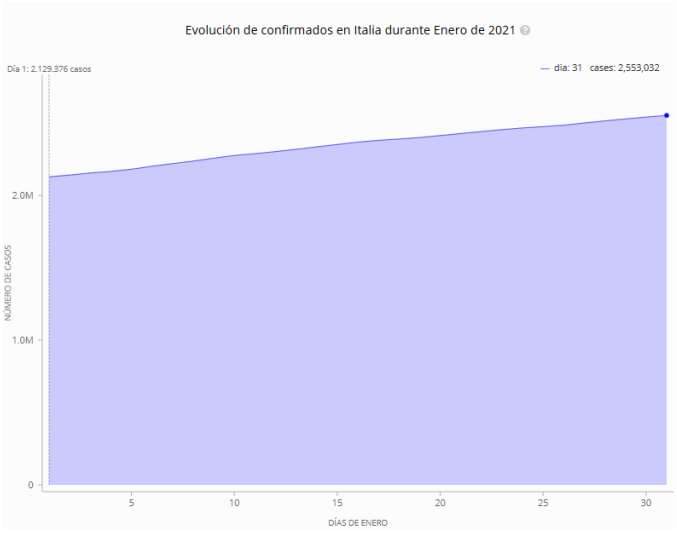

Figura 4.5-12 Evolución de casos confirmados en Italia durante Enero de 2021.

Para obtener los datos de las muertes, modificamos el valor de Status a *deaths* de la query y la volvemos a ejecutar (Figura 4.5-10). El resultado obtenido es la Figura 4.5-13 donde se muestra la evolución de las muertes en Italia durante el mes de Enero de 2021, con un total de 13.895 muertes durante ese mes.

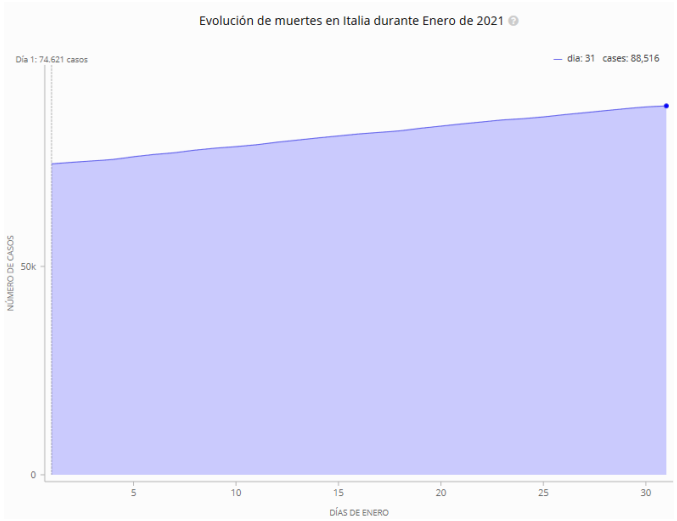

Figura 4.5-13 Evolución de muertes en Italia durante Enero de 2021.

Y por último, para obtener los casos recuperados modificamos Status a *recovered* y volvemos a ejecutarla, el resultado es la Figura 4.5-14 donde se puede observar la evolución de casos recuperados en Italia durante el mes de abril de 2020, con un total de 59.098 recuperados.

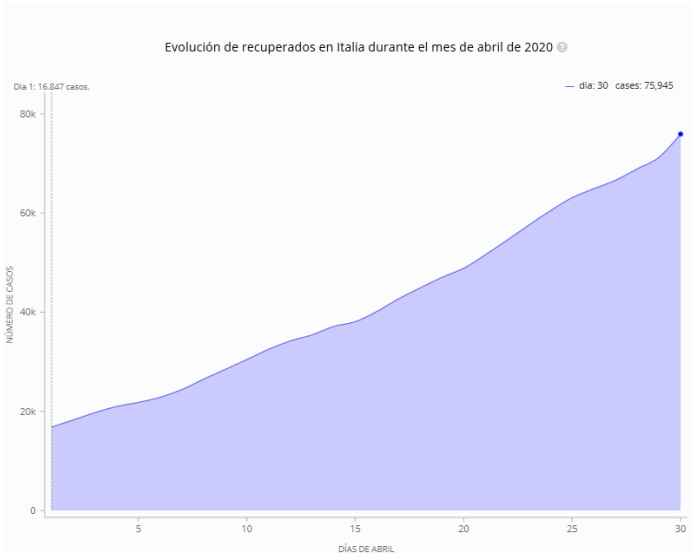

Figura 4.5-14 Evolución de recuperados en Italia durante abril de 2020.

4.6 CONCLUSIONES

Tras introducir los archivos JSON en MetricsGraphics y generar las gráficas, se puede observar que en la gráfica de la Figura 4.5-2 Nuevos Confirmados Globales los días en los que se registran menos casos son al comienzo de la semana (lunes o martes por norma general), mientras que los días en los que se registra un mayor número de confirmados son los fines de semana.

Respecto a la gráfica de la Figura 4.5-3 Nuevas Muertes Globales podemos concluir que los días en los que menos muertes se han registrado han sido los lunes, a partir de ese día se puede observar cómo aumentan de forma muy significativa las muertes al acercarse el fin de semana.

Si observamos la gráfica de la Figura 4.5-4 Nuevos Recuperados Globales**,** el número de recuperados varía de una semana a otra por ejemplo, de la semana del 2 al 8 de Marzo se puede

observar un máximo de recuperaciones en el lunes día 8 y se registra un mínimo el sábado día 6.

Si vemos cómo se comporta la curva en la semana del 9 al 15 se puede ver claramente cómo desciende el número de recuperados hasta llegar a su mínimo con un total de nuevos recuperados de 102,497 el lunes 15.

En la semana del 16 al 22 de Marzo se puede ver cómo aumenta el número de nuevos recuperados hasta llegar al fin de semana que es cuando vuelve a caer.

Respecto a las gráficas referentes a España e Italia, podemos concluir que el número de confirmados aumenta en España a un ritmo mayor que lo hace en Italia, sin embargo, Italia se lleva la peor parte si hablamos de muertes registradas a lo largo de Enero, pues casi duplica el número de muertes registradas en España.

5 CONCLUSIONES FINALES DEL PROYECTO

Tras concluir el desarrollo de este proyecto, se ha aprendido a desarrollar una API GraphQL con Spring Boot desde cero, cubriendo todos los aspectos de esta desde la creación de esquemas, pasando por el mapeado de entidades en Java y la creación de una base de datos, hasta creación de endpoints y su documentación usando Postman.

Este proyecto también ha servido como una introducción al mundo de las APIs, su funcionamiento y cómo podemos obtener y modificar datos de ellas. Cabe destacar los siguientes aspectos del proyecto:

Se ha investigado el mundo de las nuevas tecnologías relacionadas con los frameworks más usados hoy día, se ha investigado sobre la integración de un motor de base de datos en una API Java.

Se ha aprendido a utilizar varias herramientas para el testeo de APIs como Postman o Insomnia, además de implementar un sistema de seguridad basado en tokens para limitar el acceso a la API.

Respecto a GraphQL se ha implementado una API sobre el Coronavirus basada en un esquema predefinido, además de definir queries y mutaciones para la interacción con ella.

Se ha investigado y utilizado el patrón repositorio para operar sobre las tablas de la base de datos, además de implementar un ejemplo sencillo de una API sobre películas.

Se han aplicado conocimientos adquiridos durante el Grado relacionados con el uso de estructuras de datos y lectura/escritura de ficheros.

Mediante el uso de la herramienta MetricsGraphics se ha mostrado la evolución de la enfermedad globalmente y en especial para los países España e Italia durante un periodo de tiempo.

En general este ha sido un proyecto orientado para aquellas personas que deseen adentrarse en el mundo de las APIs, haciendo especial mención a GraphQL de una forma accesible para cualquier persona que desee implementar una pueda hacerlo siguiendo esta guía, cubriendo

todos los aspectos del desarrollo y añadiendo la generación de gráficas utilizando MetricsGraphics.

6 REFERENCIAS

[1] R. Johnson, Expert One-To-One, Wrox Press, 2002.

[2] «Wikipedia,» 16 01 2020. [En línea]. Available: https://es.wikipedia.org/wiki/Spring_Framework. [Último acceso: 26 04 2021].

[3] R. Pahino, «CampusMVP. ¿Qué son Spring Framework y Spring Boot?,» 31 03 2020. [En línea]. Available: https://www.campusmvp.es/recursos/post/que-son-spring-framework-y-spring-boot-tu-primer-programa-java-con-este-framework.aspx. [Último acceso: 29 03 2021].

[4] G. Vázquez, «www.chackray.com, Gradle VS Maven: Definiciones y diferencias principales,» [En línea]. Available: https://www.chakray.com/es/gradle-vs-maven-definiciones-diferencias/. [Último acceso: 26 04 2021].

[5] R. Jiménez, «www.openwebinars.com, Los 7 mejores frameworks de Java de 2020,» 28 08 2018. [En línea]. [Último acceso: 26 04 2021].

[6] I. Z. M. Luis Joyanes Aguilar, «Programación en Java 2,» de *Algorítmos, Estructuras de Datos y Programación Orientada a Objetos*, Madrid, Mc Graw Hill, 2002, p. 725.

[7] «Wikipedia. GraphQL,» [En línea]. Available: https://es.wikipedia.org/wiki/GraphQL. [Último acceso: 26 04 2021].

[8] «Página de inicio de GraphQL,» [En línea]. Available: https://graphql.org/learn/. [Último acceso: 09 04 2021].

[9] J. M. Lema, «Diferencias entre API REST y GraphQL,» 01 10 2019. [En línea]. Available: https://enmilocalfunciona.io/diferencias-api-rest-y-graphql/. [Último acceso: 30 03 2021].

[10] «Historia de H2 Database,» [En línea]. Available: https://www.h2database.com/html/history.html. [Último acceso: 26 04 2021].

[11] D. Lázaro, «www.diego.com.es Métodos HTTP,» 2018. [En línea]. Available: https://diego.com.es/metodos-http. [Último acceso: 26 04 2021].

[12] «Spring Initializr,» [En línea]. Available: https://start.spring.io/. [Último acceso: 2021 04 09].

[13] C. Á. Caules, «Arquitectura Java: Introducción a Spring Data y JPA,» 2016 07 29. [En línea]. Available: https://www.arquitecturajava.com/introduccion-spring-data-y-jpa/. [Último acceso: 2021 04 20].

[14] I. Oliveto, «www.medium.com Securing applications with JWT Spring Boot,» 19 07 2019. [En línea]. Available: https://medium.com/wolox/securing-applications-with-jwt-spring-boot-da24d3d98f83. [Último acceso: 26 04 2021].

[15] «Documentación de la API de origen,» [En línea]. Available: https://documenter.getpostman.com/view/10808728/SzS8rjbc#27454960-ea1c-4b91-a0b6-0468bb4e6712. [Último acceso: 09 04 2021].

[16] «Spring Framework,» [En línea]. Available: https://spring.io/projects/spring-framework. [Último acceso: 29 03 2021].

[17] «Página de inicio de MetricsGraphics,» [En línea]. Available: https://metricsgraphicsjs.org/. [Último acceso: 2021 04 09].

[18] «Página de descarga de Postman,» [En línea]. Available: https://www.postman.com/. [Último acceso: 2021 04 09].

[19] «Página de descarga de Insomnia,» [En línea]. Available: https://insomnia.rest/. [Último acceso: 09 04 2021].

[20] «Página de descarga de Spring Tools 4,» [En línea]. Available: https://spring.io/tools. [Último acceso: 09 04 2021].

[21] «Página de descarga de Java,» [En línea]. Available: https://www.java.com/es/download/. [Último acceso: 09 04 2021].

7 ÍNDICE DE FIGURAS

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

En este proyecto se pretende implementar una API con soporte GraphQL sobre los casos de Coronavirus a nivel global, y haciendo especial hincapié a los países de España e Italia. GraphQL es un lenguaje de consultas basado en esquemas que pretende hacerle sombra a las APIs REST convencionales. El proyecto se ha realizado utilizando el framework Spring en Java, donde se implementa una API sencilla de ejemplo con soporte GraphQL para luego extrapolar todo lo aprendido durante su implementación a una API de las mismas características sobre el COVID-19. Además de la implementación de la API se hará uso de MetricsGraphics para la generación de gráficas para un futuro estudio de la evolución de la pandemia.

GRADO EN INGENIERÍA INFORMÁTICA, 2020/2021