Proyecto Fin de Grado
Ingeniería de las Tecnologías de Telecomunicación

Servicio para la visualización y procesamiento de datos de monitorización de pacientes almacenados en Neo4J

Autor: Juan Manuel González Orta
Tutor: Jorge Calvillo Arbizu

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2024

Proyecto Fin de Grado
Ingeniería de las Tecnologías de Telecomunicación

Servicio para la visualización y procesamiento de datos de monitorización de pacientes almacenados en Neo4J

Autor:
Juan Manuel González Orta

Tutor:
Jorge Calvillo Arbizu
Profesor Permanente Laboral

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Proyecto Fin de Grado: Servicio para la visualización y procesamiento de datos de monitorización de pacientes almacenados en Neo4J

Autor: Juan Manuel González Orta

Tutor: Jorge Calvillo Arbizu

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

A todos mis seres queridos

Agradecimientos

Finalmente, hemos llegado al punto final. Mamá y Papá, lo he conseguido. Tras muchos años de constancia, dedicación y sacrificio conseguimos poner fin a esta etapa. Gracias no solo por todo el apoyo que me habéis brindado estos años, si no también por guiarme de la mejor de las maneras, no os puedo estar más agradecido. También te agradezco a ti, hermana por la educación y buenos consejos que siempre me has dado y seguirás dándome siempre, lo mismo para Marco y mis sobrinos Lola y Mateo, que algún día leerán esto y se acordarán de mí.

Gracias también a Oliva, mi pareja, que no solo ha estado para celebrar mis éxitos, que por cierto son suyos también, sino también por ayudarme siempre durante mis peores rachas. También me gustaría agradecer y dedicar este trabajo a sus padres Emilio y Marisol y a su hermano Jesús. A todos mis amigos de la infancia, que también formáis parte de esto: Felipe, Mohamed, Lagos, Novoa, Juampi, Quini y Sierra, muchas gracias tanto por estar siempre a mi lado como por compartir vuestra vida conmigo.

Gracias también a Teleco porque sin ti jamás hubiera conocido a mis compañeros Félix, Bruno, Javi, Aldana, Rafa y todos aquellos que me dejo, pero saben quiénes son. Habéis hecho que este camino no solo sea más ameno sino también inolvidable.

Gracias a mis abuelos porque ver vuestra cara de alegría cuando sabíais que ya quedaba menos para terminar me motivaba cada vez más y más a seguir, aunque lastimosamente, abuela, o yeya como a mí me gustaba llamarte, no vaya a poder celebrar contigo este triunfo, es en este momento cuando mando un beso al cielo y aprovecho para expresar que, aunque nunca más podamos estar juntos, siempre te llevaré en mi corazón, desde allá donde estes yeya, sé que me cuidas, sé que siempre me cuidarás.

A todos mis familiares os agradezco también lo que hacéis por mí, a todos mis profesores y profesoras que han formado parte de este proceso educativo especialmente a Tom y Paco pero sobre todo a Jorge, mi tutor de TFG, que desde el primer momento me ha guiado y ayudado sin ningún problema y que sin duda me ha facilitado enormemente el desarrollo de este proyecto, muchísimas gracias.

Afronto este momento con muchísima alegría, pero también siento tristeza pues sin duda estos años han sido maravillosos y me marcarán para siempre. Esto no es el final, sino el principio de una nueva etapa que comienza en mi vida, de eso estoy seguro.

Finalmente me dedicaré a mí mismo unas palabras: enhorabuena por lo que has conseguido, a aquel niño que desde pequeño ya soñaba con ser ingeniero, que no se conformaba con aprobar los exámenes si no con entender los conceptos, que aunque hubiera momentos en los que todo parecía imposible siempre logró levantarse y superar las adversidades, sin duda, te lo mereces.

Juan Manuel González Orta

Cartaya, 2024

Resumen

Actualmente, las Tecnologías de la Información y las Comunicaciones (TIC) han revolucionado diversos sectores, incluyendo el sanitario. La capacidad de monitorizar las constantes vitales de los pacientes en tiempo real es crucial para anticipar y responder a cambios en su salud. Sin embargo, el manejo de estos datos plantea desafíos significativos en términos de almacenamiento, procesamiento y visualización.

En el ámbito sanitario, los datos de un paciente suelen estar distribuidos en múltiples sistemas que no siempre se comunican entre sí de manera eficiente. Esta fragmentación dificulta el acceso rápido y preciso a la información necesaria para una atención óptima. Además, la correcta interpretación de estos datos puede verse complicada sin las herramientas adecuadas de visualización y análisis.

Con el objetivo de abordar estos problemas, este proyecto se centra en el diseño y desarrollo de un servicio que utilice Neo4J, una base de datos orientada a grafos, para el almacenamiento y procesamiento de datos de monitorización de pacientes. Neo4J es especialmente adecuada para gestionar y analizar relaciones complejas entre datos, lo cual recordemos es vital en la monitorización de constantes vitales.

El proyecto propone un sistema compuesto por un backend desarrollado con NestJS que interactúa con Neo4J, y un frontend basado en Angular que proporciona una interfaz intuitiva para los profesionales de la salud. Este enfoque permite no solo un almacenamiento eficiente de los datos, sino también una visualización clara y accesible de estos.

Por tanto, el objetivo de este trabajo es demostrar que las bases de datos orientadas a grafos, junto con tecnologías modernas de desarrollo web, pueden ofrecer una solución avanzada para la monitorización de pacientes, mejorando tanto la calidad de la atención médica como la eficiencia operativa en el sector sanitario.

Abstract

Currently, Information and Communication Technologies (ICT) have revolutionized various sectors, including healthcare. The ability to monitor patients' vital signs in real time is crucial for anticipating and responding to changes in their health. However, managing this data presents significant challenges in terms of storage, processing, and visualization.

In the healthcare sector, a patient's data is often distributed across multiple systems that do not always communicate efficiently with each other. This fragmentation hampers quick and accurate access to the information necessary for optimal care. Moreover, the correct interpretation of this data can be complicated without adequate visualization and analysis tools.

To address these issues, this project focuses on the design and development of a service that utilizes Neo4J, a graph-oriented database, for storing and processing patient monitoring data. Neo4J is particularly well-suited for managing and analyzing complex data relationships, which is vital in monitoring vital signs.

The project proposes a system consisting of a backend developed with NestJS that interacts with Neo4J, and a frontend based on Angular, which provides an intuitive interface for healthcare professionals. This approach not only enables efficient data storage but also ensures clear and accessible visualization of the data.

Therefore, the objective of this work is to demonstrate that graph-oriented databases, combined with modern web development technologies, can offer an advanced solution for patient monitoring, thereby improving both the quality of medical care and operational efficiency in the healthcare sector.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

La mitad está hecha cuando tienen buen principio las cosas.

- Fernando de Rojas -

En este primer capítulo se analizará la motivación de este proyecto, enfocándose tanto en el interés como en la necesidad de su realización, se discutirán además los objetivos que se buscan alcanzar con la realización del mismo así como la metodología y el plan de trabajo adoptados para llevar a cabo el proyecto.

1.1 Motivación y objetivos

El constante avance de la tecnología se ha hecho evidente en todos los campos, y el sector sanitario no es una excepción. El equipamiento usado en los hospitales o incluso la aplicación de inteligencia artificial (IA) para el diagnóstico de imágenes son un ejemplo de ello. En este proyecto vamos en enfocarnos concretamente, en el ámbito de la monitorización de las constantes vitales de los pacientes, también conocido como telemedicina. Monitorizar con precisión haciendo uso de dispositivos dotados para ello permite anticiparse a situaciones que puedan empeorar la salud del paciente y, en caso de que se produzcan, el personal sanitario podrá tomar decisiones con mayor rapidez y administrar tratamiento de forma más eficiente y adecuada [1]. Los dispositivos sensores mencionados anteriormente han emergido como herramientas cruciales que permiten la recopilación de datos de salud en tiempo real, facilitando el seguimiento de los signos vitales y otros indicadores relevantes del paciente. Sin embargo, suelen ser frecuentes los escenarios en los que el volumen y/o la complejidad de estos datos presentan numerosas dificultades, entre ellas:

- Almacenamiento eficiente: es decir, gestionar grandes volúmenes de datos de manera que se puedan almacenar de forma segura y accesible.
- Procesamiento: sin demoras significativas.
- Visualización efectiva: presentar los datos de manera que sean fácilmente interpretables por el personal sanitario.
- Mantenimiento de la precisión: Garantizando que los datos recopilados sean precisos y fiables, minimizando los errores.

El uso de bases de datos tradicionales ofrece una opción para gestionar esta información de manera estructurada.

Sin embargo, debido a las dificultades anteriores no siempre son la herramienta más adecuada. Otro tipo de bases de datos pueden proporcionar unas características más adecuadas a este contexto. Es el caso de las bases de datos basadas en grafos, las cuales permiten no solo almacenar y procesar los datos de manera efectiva, sino también realizar análisis complejos de las relaciones entre diferentes conjuntos de datos.

Sin embargo, actualmente hay pocas aplicaciones de este tipo de tecnologías en el ámbito de la monitorización de pacientes. Este hecho motivó el desarrollo de un servicio que va más allá de simplemente almacenar y procesar datos, ya que también proporciona una interfaz gráfica para visualizarlos entre otros aspectos. Es por tanto que la implementación de un sistema con estas características puede tener el potencial de mejorar la calidad de la atención médica remota a través de nuevas capacidades de almacenamiento, procesamiento y visualización de los datos de monitorización

Esto resalta el objetivo central de este proyecto: diseñar y desarrollar una solución tecnológica que almacene y procese datos de manera efectiva haciendo uso de una base de datos basada en grafos (en la implementación concreta de Neo4J), ofreciendo herramientas para la visualización y análisis de las constantes vitales de los pacientes. Definimos a continuación un desglose en detalle de este objetivo:

- Exploración y comprensión de las bases de datos basadas en grafos como Neo4J.
- Aplicación de los conocimientos adquiridos durante el grado para el desarrollo de una solución a un problema.
- Diseño, desarrollo, implementación y pruebas de un servicio de backend que se encargue de conectarse y obtener los datos almacenados en la base de datos de manera coherente.
- Diseño, desarrollo, implementación y pruebas de una API alojada en el backend, la cual será consumida por el frontend para así obtener los datos a modo de respuesta del servidor de manera coherente.
- Diseño, desarrollo, implementación y pruebas de una interfaz gráfica para el usuario (GUI) que muestre los datos obtenidos de manera visual.

En la Ilustración 1, se muestra el esquema del sistema que vamos a diseñar y desarrollar, mostrándose claramente las tecnologías usadas, si bien es cierto que la implementación e integración de estas se abordará en detalle en el apartado Materiales y métodos de la memoria. Nótese que en la ilustración la comunicación entre el frontend y el backend se lleva a cabo mediante una API REST brindando mucha versatilidad a la solución desarrollada.

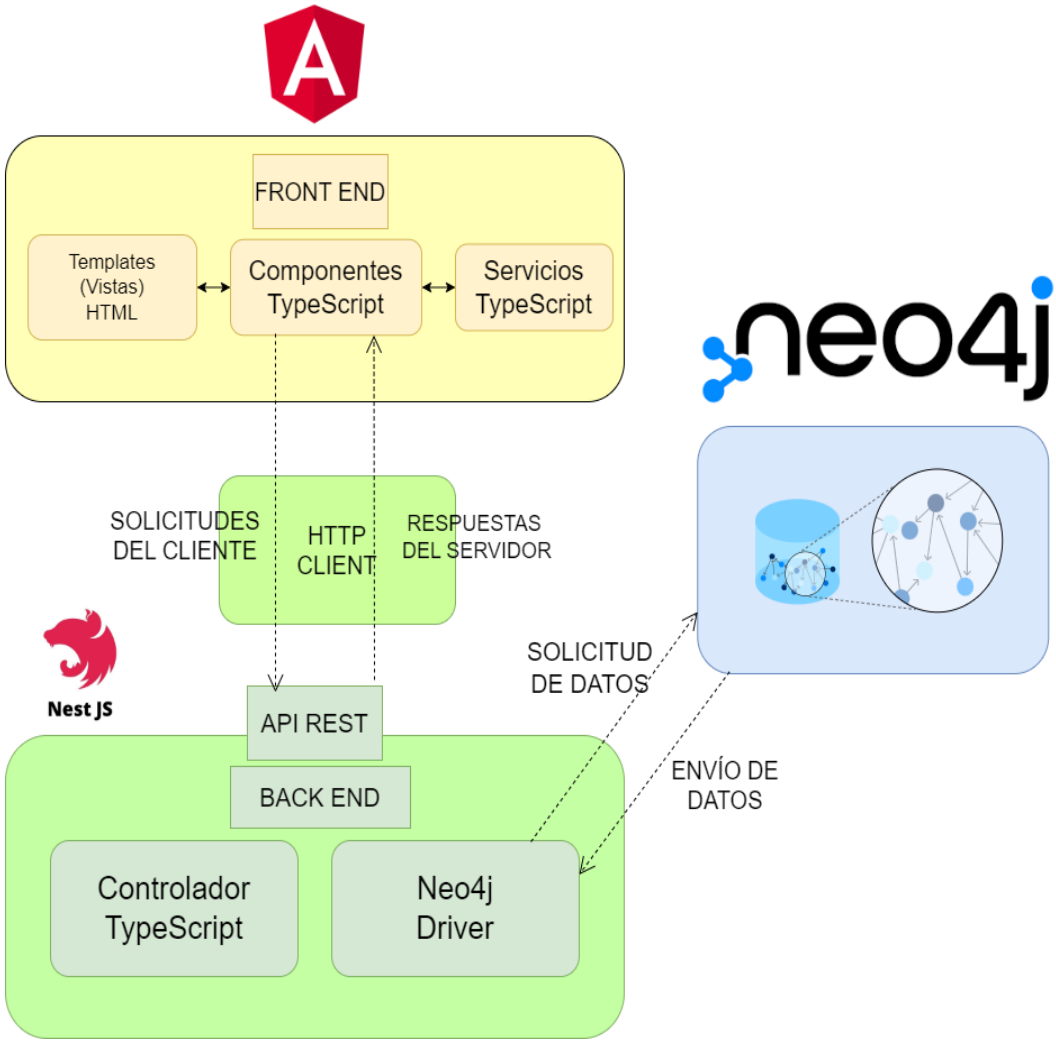

Ilustración 1 : Comunicación entre Front-end, Back-end y base de datos Neo4J

1.2 Metodología empleada

El enfoque metodológico que se ha seguido para la realización del trabajo combina tanto la metodología de desarrollo iterativo como la incremental.

- **Metodología de desarrollo iterativo:** Pues el desarrollo se ha dividido en fases o iteraciones, comenzando con la investigación de tecnologías, pasando por la selección y aprendizaje de las tecnologías elegidas, y luego avanzando hacia el desarrollo real del frontend y el backend habiéndose realizado previamente algunos proyectos de prueba para establecer una sólida familiarización con las herramientas finalmente seleccionadas. Este enfoque ha permitido el avance gradual del proyecto.

- **Metodología de desarrollo incremental:** Pues se ha construido y mejorado continuamente el sistema, agregando funcionalidades y refinando la implementación de este a lo largo del tiempo. En lugar de intentar desarrollar todo el sistema de una sola vez, se ha optado desde un principio por dividirlo en partes más pequeñas y manejables.

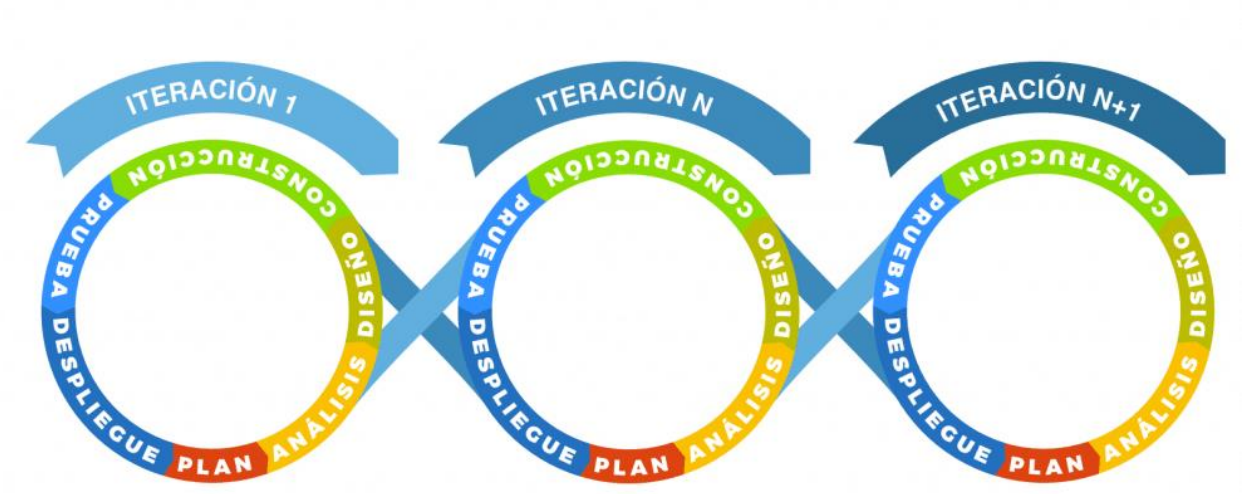

Ilustración 2 : Modelo de desarrollo incremental e iterativo

1.3 Plan de trabajo

Antes de comenzar con el diseño y desarrollo del sistema, es necesario comentar una serie de aspectos relacionados con la comprensión de lo que queremos lograr:

1. Investigación de frameworks y librerías para el desarrollo del frontend. El objetivo de esta tarea es analizar de entre la gran variedad de soluciones que podemos encontrar disponibles y optar por aquella que mejor se adecúe a nuestras necesidades. Teniendo en cuenta aspectos como la capacidad del framework para la elaboración de gráficas intentando evitar aquellas tecnologías que ofrezcan soluciones complejas o que no proporcionen resultados esperados. Por otro lado, la tendencia que seguiremos para la elección de la tecnología que usaremos para el desarrollo del frontend se verá muy ligada sobre todo a la búsqueda de aquellas que presenten actualmente un auge en el mercado. Tras tener claro el frontend a usar deberemos tener en cuenta la existencia de librerías que proporcionen un toque más visual al desarrollo de nuestra página web.
2. Investigación de frameworks para el desarrollo backend. La idea de esta tarea es de entre todas las soluciones posibles que podemos encontrar disponibles, seleccionar aquella que permita una integración sencilla y eficiente con la base de datos Neo4j, este objetivo es crucial pues no todos los framework de backend disponen de drivers u otros elementos que permitan esta tarea.
3. Estudio y comprensión profundos del funcionamiento de las bases de datos basadas en grafos, concretamente de Neo4J pues es la que se usará para el desarrollo del proyecto. Además, durante todo el grado no hemos tenido relación alguna con este tipo de datos y era totalmente necesario una previa familiarización con Neo4J previa a comenzar el proyecto.
4. Estudio y comprensión del lenguaje CYPHER para el manejo de nodos y relaciones, consultas o importación de datos en Neo4J.

Los siguientes puntos responden a una metodología tanto iterativa como incremental de tal forma que los siguientes aspectos se han repetido en ciclos para la implementación final de cada una de las funcionalidades que queremos lograr en el proyecto:

1. Lectura de la documentación de Neo4J para comenzar a probar el funcionamiento de la base de datos y la sintaxis del lenguaje Cypher. En función a los datos que quisiéramos obtener de la base de datos, la consulta ligada a esta tarea será diferente. De modo que era necesario acudir a esta documentación ya que proporciona la información y los ejemplos necesarios para construir todo tipo de consultas de

manera eficiente.

2. Realización de micro certificaciones en la página oficial de Neo4J las cuales facilitaban la comprensión y funcionamiento de las bases de datos orientadas a grafos, al aprendizaje del lenguaje CYPHER y al volcado de datos mediante ficheros CSV. Esta tarea permitió asentar de manera interactiva y eficaz conocimientos sólidos y aplicables al desarrollo del proyecto.
3. Despliegue local de bases de datos Neo4J con el fin de aplicar no solo lo aprendido gracias a las micro certificaciones, sino que también indagar de manera autónoma en las distintas posibilidades que ofrecen tanto Neo4j como CYPHER.
4. Comprobación de las correctas respuestas por parte de Neo4j a las distintas consultas realizadas además de la elaboración de escenarios de prueba que simularían en menor escala el entorno final, del que dispondremos en el proyecto.

2 MATERIALES Y MÉTODOS

El placer más noble es el júbilo de comprender

-Leonardo da Vinci-

En esta sección de la memoria se abordarán los fundamentos teóricos en los que se basa el trabajo, haciendo especial hincapié en la base teórica de las bases de datos basadas en grafos, concretamente en Neo4J. También se llevará a cabo una descripción de las herramientas utilizadas en el desarrollo del proyecto, incluyendo tanto los frameworks usados como el IDE en el que el proyecto ha sido desarrollado.

2.1. Bases de datos orientadas a grafos (BDOG)

2.1.1 Origen

Para comprender la idea de este tipo de bases de datos, también conocidas como BDOG hay que remontarse atrás en el tiempo pues, aunque pueda parecer que las BDOG surgieron hace unos años realmente no es así.

El nacimiento de las bases de datos orientadas en grafos tuvo lugar durante el siglo XVIII en Königsberg, Prusia Königsberg está dividida por el río Pregel en cuatro secciones conectadas entre sí por siete puentes.

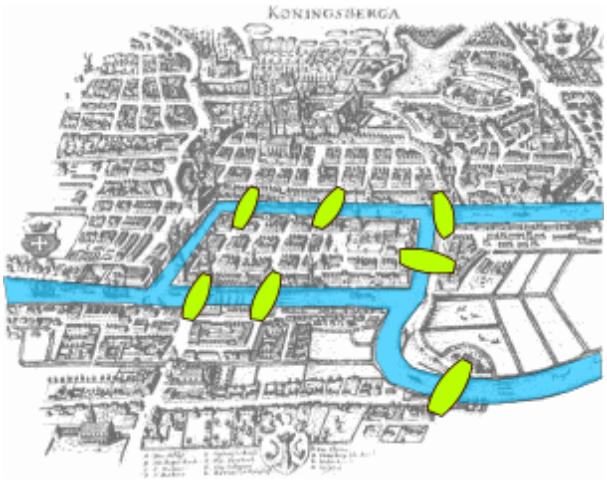

Ilustración 3 : Problema de los siete puentes

Es en este punto donde cómo no un matemático se plantea el problema de si es posible dar un paseo por la ciudad cruzando una sola vez cada uno de los puentes y llegar finalmente al mismo punto de partida, el matemático en cuestión era Leonhard Euler.

Euler llegó a la conclusión de que no existía una ruta con esas características y para demostrarlo el célebre matemático llevo a cabo una abstracción del problema dando pie a la primera noción de grafo. De manera que

reformuló el problema de la siguiente manera: Cada puente será representado mediante una línea que unirá a dos puntos, y cada uno de estos puntos representaba una región diferente. Así, el problema se reduce a decidir si existe o no un camino que comience por uno de los puntos, recorra todas las líneas una sola vez y regrese al mismo punto de partida. Esto se logra separando en nodos cada uno de los terrenos por los que se divide la ciudad y las líneas o arcos que los unen representarían los puentes que conectan entre sí los distintos terrenos entre sí. La representación gráfica quedaría como se muestra en la Ilustración 4.

Ilustración 4 : Simplificación del problema de los siete puentes mediante un grafo

De esta manera Euler determinó que necesariamente los puntos intermedios de un recorrido deben estar necesariamente conectados a un número par de líneas. En efecto, si llegamos a un punto desde alguna línea, la única manera de salir de ese punto es por una línea diferente Esto significa que tanto el punto inicial como el final serían los únicos que podrían estar conectados con un número impar de líneas. Sin embargo, el requisito adicional del problema dice que el punto final de la ruta debe coincidir con el punto inicial, de la misma por lo que no podría existir ningún punto conectado con un número impar de líneas. Si nos volvemos a fijar en la Ilustración 4 es imposible definir un camino con las restricciones impuestas previamente. Euler concluye así la inexistencia de una solución al problema de los siete puentes [2].

Pero no fue un esfuerzo completamente en vano. Aunque los grafos se originaron en las matemáticas, también son una forma muy conveniente de modelar y analizar datos. Si bien los datos que poseemos ciertamente tienen valor, son las conexiones entre los datos las que realmente pueden agregar valor. Crear o inferir relaciones entre sus registros puede generar información real sobre un conjunto de datos.

Han pasado 300 años y estos principios fundamentales se utilizan para resolver problemas complejos que incluyen búsqueda de rutas, análisis de la cadena de suministro y recomendaciones en tiempo real.

2.1.2 Elementos

En cuanto a los elementos principales que componen las BDOG, estos son los nodos o vértices y relaciones o arcos. Estos elementos deben estar presentes en cualquier grafo, no obstante, pueden estar dotados de información adicional, aunque a continuación vamos a explicar la función y significado de cada uno de estos elementos indispensables de cualquier grafo, así como la manera en la que se complementan entre sí.

- **Nodos o vértices:** Círculos del grafo que normalmente representan objetos o entidades. En el ejemplo mostrado anteriormente, los nodos se correspondían con los terrenos en los que Königsberg estaba dividido.

- **Relaciones o arcos:** Líneas que unen a los diferentes nodos del grafo entre sí. Podemos usar estos elementos para describir cómo un nodo está conectado con otro. En el ejemplo anterior podríamos implementar los arcos entre los nodos terreno de la siguiente manera para representar que los distintos terrenos están conectados entre sí a través de puentes.

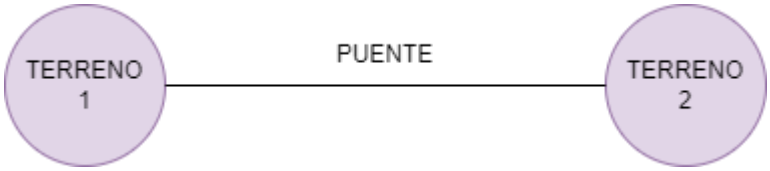

Ilustración 5 : Conexión de dos nodos mediante un arco

Una vez definidos los conceptos de nodo y relación, podemos generar grafos más complejos siguiendo la misma filosofía en los que se muestren no solo un mayor número de nodos, sino que también distintas relaciones entre sí como se muestra en la Ilustración 6.

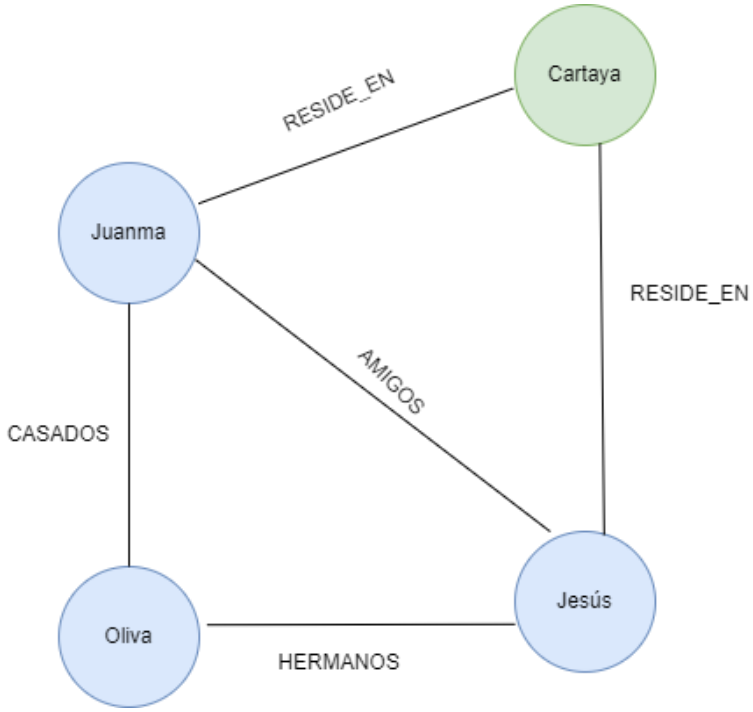

Ilustración 6 : Grafo con distintos tipos de nodos y de relaciones

Del ejemplo anterior podemos destacar varios aspectos:

- Un nodo puede tener establecida más de una relación con otros nodos
- Un nodo puede tener establecida en más de una ocasión la misma relación con otros nodos
- No necesariamente todos los nodos del grafo deben estar relacionados entre sí.

2.1.2.1 Ponderaciones de las relaciones

En los ejemplos mostrados hasta ahora las relaciones eran simétricas, es decir, los grafos eran no dirigidos, pues no podemos identificar un nodo origen y otro destino. Sin embargo, hay veces que plasmar la información en este tipo de grafos puede conllevar a confusión de manera que recurrimos a los grafos dirigidos.

Como ya sabemos, todo grafo consta al menos de dos elementos, nodos y relaciones, pero además estos pueden estar dotados de información adicional. La peculiaridad de los grafos dirigidos es que las relaciones suelen estar

ponderadas por ejemplo con un valor de coste, tiempo, distancia o prioridad, además de partir desde un nodo origen hacia un nodo destino, lo cual aporta a estos grafos una dimensión adicional de información, de manera que las relaciones del mismo tipo, pero en direcciones opuestas pueden tener un significado o aportar información diferente.

Podemos usar la Ilustración 6 para poner un ejemplo de esto, si la relación CASADOS es simétrica, entonces vamos a definir la relación asimétrica o dirigida AMOR. Aunque dos personas pueden amarse, el grado en que lo hacen puede variar drásticamente. Como hemos mencionado, las relaciones direccionales a menudo pueden calificarse con algún tipo de ponderación, en este caso la ponderación va a ser la fuerza. Aquí vemos que la fuerza de la relación AMOR describe cuánto ama una persona a otra y aunque las relaciones que se muestran sean del mismo tipo no aportan la misma información.

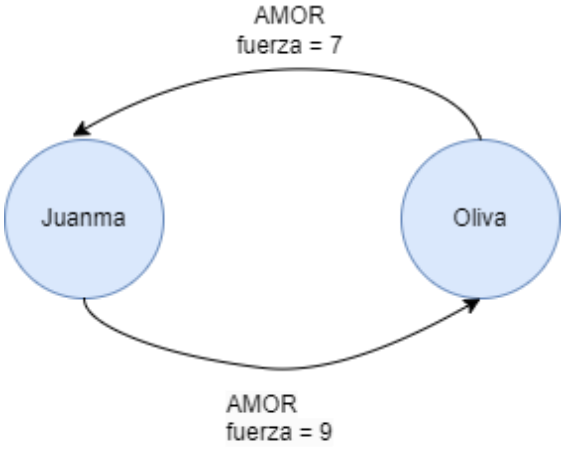

Ilustración 7 : Ejemplo de grafo dirigido con propiedades

Un ejemplo muy común en el que se usan grafos dirigidos es en el cálculo de la mejor ruta desde un origen A hasta un destino B siendo las ponderaciones de las relaciones la distancia entre las distintas ciudades que podemos encontrar por el camino.

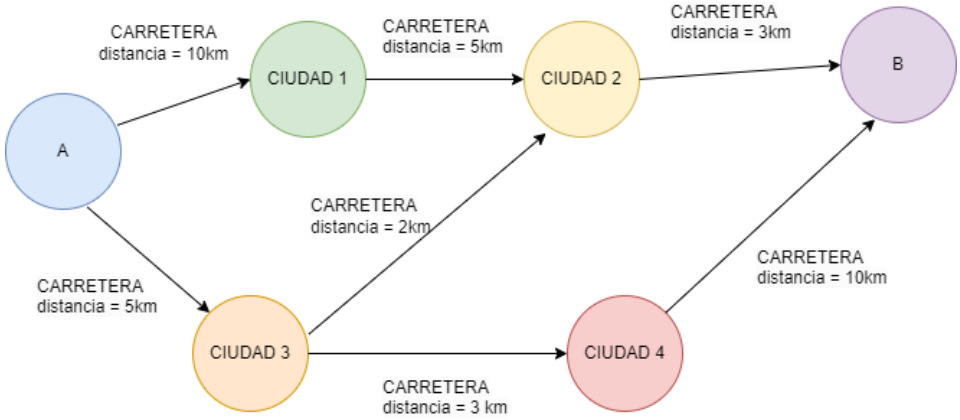

Ilustración 8 : Grafo dirigido para cálculo de mejor ruta

En este ejemplo, la pregunta que podríamos cuestionarnos es: ¿cuál es la ruta en la que recorro una menor distancia desde A hasta B? Usando las relaciones CARRETERA y la ponderación distancia para las relaciones del grafo podemos ver que el viaje más corto será el que siga el recorrido:

A→CIUDAD 3→ CIUDAD 2 → B.

Esta conclusión no hubiera sido posible si el grafo no estuviera dotado de ponderaciones.

El ejemplo anterior es uno de los muchos escenarios en los que podemos encontrar grafos para su representación,

otros podrían ser:

- Transporte y logística
- Operaciones de red y TI (tecnologías de la información)
- Comercio electrónico y recomendaciones en tiempo real.

2.1.2.2 Nodos, etiquetas y propiedades

Recordemos que los nodos son los elementos del grafo que representan los objetos o entidades de este. Podemos utilizar dos elementos adicionales para proporcionar un contexto adicional a los datos:

- Etiquetas: Gracias a estas podemos indicar que un nodo pertenece a un subconjunto de nodos dentro del grafo. Las etiquetas son muy importantes en Neo4j, el porqué es tan importante lo veremos posteriormente, durante el desarrollo de la memoria.
- Propiedades: Son pares de clave - valor y se pueden agregar o eliminar de un nodo según sea necesario. Los valores de las propiedades pueden ser un únicos o también pueden conformar una lista.

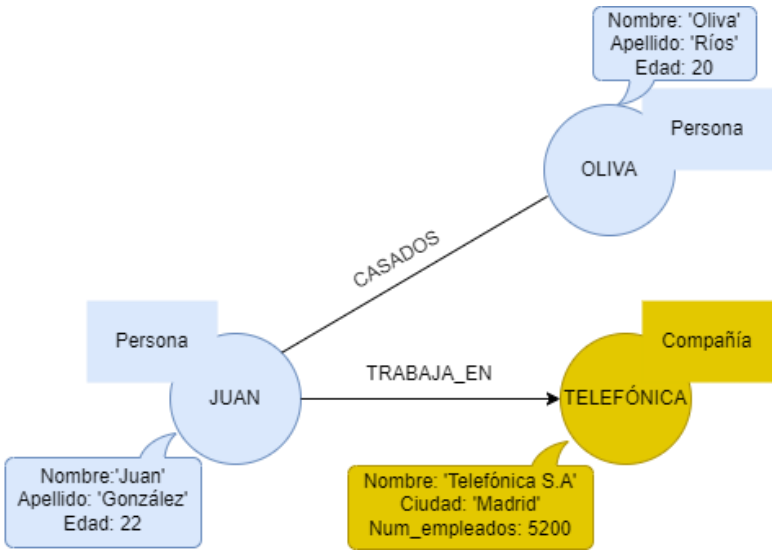

Ilustración 9 : Grafo con etiquetas y propiedades

En el ejemplo anterior se ha hecho uso de etiquetas y propiedades en los distintos nodos, gracias a estos elementos podemos resaltar lo siguiente:

- Los nodos JUAN y OLIVA pertenecen al subconjunto de nodos Persona mientras que el nodo TELEFÓNICA pertenece al subconjunto de nodos Compañía.
- Al agregar las propiedades a los distintos nodos conocemos más información de estos, por ejemplo: el nombre completo del nodo Persona JUAN es Juan González, además, su edad es de 22 años, podemos aplicar lo mismo para el nodo OLIVA o para el nodo TELEFÓNICA.
- No necesariamente las relaciones deben tener una ponderación asociada como ya se mencionó.

2.1.3 Comparativa con bases de datos relacionales (BDR)

Las BDOG han emergido como una solución eficaz para ciertos tipos de problemas que las bases de datos relacionales no pueden manejar de manera eficiente. A continuación, se comparan las BDOG con las BDR.

2.1.3.1 Almacenamiento y relación de los datos

Tanto las bases de datos de grafos como las bases de datos relacionales almacenan elementos de datos con relaciones predefinidas entre ellos. Sin embargo, representan las relaciones de datos de manera muy diferente. Las bases de datos relacionales almacenan datos en formato tabular con filas y columnas. Los datos relacionados también se almacenan en tablas y los puntos de datos se vinculan a la tabla original. Las operaciones relacionadas con las relaciones de datos se vuelven ineficientes, ya que requieren múltiples búsquedas en tablas de datos. Por el contrario, una base de datos de grafos almacena los datos como una red de entidades y relaciones. Utiliza la teoría matemática de grafos para almacenar y realizar operaciones sobre las relaciones de datos. Las bases de datos de grafos son mucho más eficientes en el modelado de relaciones ya que mejoran significativamente el rendimiento de las aplicaciones para casos de uso con interconexiones de datos complejas [3].

2.1.3.2 Modelo de datos

- BDR: La base de datos relacional utiliza tablas de datos que organizan la información en filas y columnas. Las columnas contienen atributos específicos de la entidad de datos, mientras que las filas representan registros de datos individuales. El esquema fijo de las bases de datos relacionales exige que las relaciones entre las tablas se definan con anticipación con claves principales y externas.

Por ejemplo, imaginemos que tenemos una aplicación de red social con perfiles de clientes que puedan ser amigos entre sí. Para modelar los datos necesitaremos dos tablas.

ID_Cliente (PK)	Nombre	Ubicación
C1	María	ESP
C2	Julián	ITA
C3	Pedro	EE. UU.
C4	Jose	ESP

Tabla 1: Clientes de la red social

ID _Cliente (PK)	ID_Amigo (PK)
C1	C2
C1	C3
C2	C4
C2	C1
C3	C1
C3	C4

Tabla 2: Amigos de la red social

Como podemos observar, hay redundancia y duplicación en la Tabla 2 pues las relaciones de amistad se registran varias veces. Por ejemplo: C1 → C2 y C2 → C1; la relación de amistad entre C1 y C2 está almacenada dos veces, una vez desde la perspectiva de C1 y otra desde la perspectiva de C2.

- BDOG: Por otro lado, como ya sabemos una base de datos de grafos utiliza una estructura gráfica con propiedades, relaciones ponderadas o no y nodos para representar los datos. La cual hace que una base de datos de grafos sea útil para representar datos conectados. Ofrece más flexibilidad con respecto a las relaciones y los tipos de datos.

Los datos de la red social del ejemplo anterior ahora se representarían de la siguiente manera:

{ID_Cliente: “C1”

Nombre: “María”

Ubicación:“ESP”

amigos:”C2,C3”}

Ya no encontramos duplicación ni redundancia de los registros de datos mientras se modelan las relaciones.

2.1.3.3 Diferencias clave

Existen muchos más aspectos que diferencian tanto por su función como por su utilidad a las BDOG de las BDR entre los cuales tenemos:

- Operaciones: las BDOG utilizan algoritmos de recorrido que priorizan la profundidad o la amplitud para consultar y recuperar rápidamente datos conectados, siendo especialmente útiles para interconexiones y consultas complejas. En cambio, las BDR emplean SQL para recuperar y manipular datos, permitiendo realizar consultas como SELECT, INSERT, UPDATE y DELETE. Estas bases de datos son eficaces para manejar datos estructurados con relaciones bien definidas entre tablas, destacándose en filtrados complejos, agregaciones y uniones de múltiples tablas.
- Escalabilidad: las BDR generalmente escalan verticalmente mediante la actualización del hardware, lo que tiene limitaciones y altos costos. También pueden escalar horizontalmente mediante la fragmentación, pero esto aumenta la complejidad y puede causar problemas de coherencia. En contraste, las BDOG escalan excelentemente de forma horizontal utilizando particiones, distribuyendo datos en varios servidores y permitiendo el procesamiento paralelo eficiente de consultas, incluso a gran escala.
- Rendimiento: las BDOG ofrecen adyacencia libre de índice, mejorando el rendimiento al permitir navegar entre entidades relacionadas directamente mediante punteros, sin necesidad de índices o tablas de asignación. Esto permite recorridos de relaciones en tiempo constante, independientemente del tamaño de los datos, lo que las hace muy eficientes. En contraste, las BDR dependen de búsquedas de índices y escaneos de tablas para identificar relaciones, lo que consume más tiempo y no ofrece el mismo rendimiento.
- Facilidad de uso: las BDOG son ideales para trabajar con datos conectados y realizar consultas de saltos múltiples usando lenguajes como Cypher (en el caso de Neo4J), que simplifican la sintaxis. Por otro lado, las BDR, pueden complicarse con consultas de saltos múltiples en SQL. Sin embargo, ofrecen robustez y fiabilidad para gestionar datos estructurados, apoyándose en las propiedades ACID para garantizar la validez de los datos. A continuación, se muestra un ejemplo de dos consultas con el mismo objetivo: obtener los nombres de los productos que pertenecen a la categoría "Dairy Products". Una de ellas realizada en Cypher y la otra en SQL, en el ejemplo se puede observar como la consulta realizada en Cypher simplifica significativamente el objetivo.

```
Cypher

MATCH (p:Product)-[:CATEGORY]→(l:ProductCategory)-[:PARENT*0..]
→(:ProductCategory {name:"Dairy Products"})
RETURN p.name

SQL

SELECT p.ProductName
FROM Product AS p
JOIN ProductCategory pc ON (p.CategoryID = pc.CategoryID
AND pc.CategoryName = "Dairy Products")

JOIN ProductCategory pc1 ON (p.CategoryID = pc1.CategoryID)
JOIN ProductCategory pc2 ON (pc1.ParentID = pc2.CategoryID
AND pc2.CategoryName = "Dairy Products")

JOIN ProductCategory pc3 ON (p.CategoryID = pc3.CategoryID)
JOIN ProductCategory pc4 ON (pc3.ParentID = pc4.CategoryID)
JOIN ProductCategory pc5 ON (pc4.ParentID = pc5.CategoryID
AND pc5.CategoryName = "Dairy Products");
```

Ilustración 10 : Diferencia entre consultas realizadas en distintos lenguajes

La diferencia es clara, mientras que en Cypher con tres líneas de código obtenemos el resultado esperado, en SQL necesitamos bastantes más sintaxis.

Esto no siempre es así, hay escenarios en los que es conveniente hacer uso de BDR:

- Cuando es necesario el cumplimiento de ACID y altos niveles de integridad y coherencia de datos, como por ejemplo en las transacciones financieras
- Si tenemos que trabajar con datos altamente estructurados que se ajustan bien al modelo de datos tabulares, como en la administración de recursos empresariales o cuando estos tienen relaciones limitadas

Y otros en los que las BDOG ofrecen mejores resultados:

- Cuando tenemos que trabajar con datos que tienen relaciones complejas, como por ejemplo en las redes sociales, la detección de fraudes, los gráficos de conocimiento y los motores de búsqueda
- Aquellos en los que se necesita un esquema en evolución, ya que las BDOG permiten modificar los nodos y las propiedades sin alterar el resto de la estructura de la base de datos o aquellos casos en los que queremos trabajar con datos interconectados y necesitamos realizar un número elevado de saltos entre las relaciones

2.1.4 Comparativa entre las distintas BDOG que ofrece el mercado

Expondremos a continuación de manera tabular las que son actualmente las principales bases de datos orientadas a grafos disponibles en el mercado obtenidas en [4].

BDOG	Lenguaje de consulta	Características clave
Neo4J	Cypher	ACID, alta disponibilidad, visualización gráfica integrada, soporte para clústeres.
Amazon Neptune	Gremlin, SPARQL	Totalmente gestionado, escalable, altamente disponible, integración con

		otros servicios de AWS.
Azure Cosmos	Cassandra (CQL)	Soporte para transacciones ACID, alta compatibilidad con otras BBDD , indexa automáticamente los datos.
ArangoDB	AQL,Gremlin	Soporte para transacciones ACID, escalabilidad, flexibilidad, soporte para fragmentación.

Tabla 3: Comparativa entre las principales BDOG actuales

2.2. Base de datos Neo4J

Una vez abordado tanto el origen de las BDOG como la comparativa con las BDR y las distintas soluciones que ofrece el mercado, estamos listos para profundizar en la que finalmente hemos seleccionado para el desarrollo de nuestra aplicación web, Neo4J.

Ilustración 11 : Logo Neo4J

Neo4j nace de la mano de "Neo Technology" como una base de datos orientada a grafos de código abierto implementada en Java y Scala. Su desarrollo se inició en 2003 y se puso a disposición del público en el año 2007. Hoy en día Neo4J es usado por una gran cantidad de compañías en una gran variedad de industrias [5]. Es una base de datos de grafos nativa, lo que significa que implementa un modelo de grafos verdadero hasta el nivel de almacenamiento. Los datos se almacenan tal como se dibujarían en una pizarra, lo que significa que estos se almacenan y gestionan directamente en esta estructura, sin necesidad de adaptaciones o transformaciones adicionales.

2.2.1 Eficiencia y velocidad de consultas: la ventaja de la propiedad nativa en Neo4j

La propiedad nativa de Neo4j es una de las características que la distingue de otras bases de datos. Cualquier base de datos es capaz de obtener consultas de 3 o 4 grados de relación, es decir, para obtener el resultado de la consulta sería necesario realizar varios saltos entre relaciones. Pero ¿Y si necesitamos obtener el resultado de una consulta que implique 20 grados de relación y además necesitamos que esta sea rápida? Aquí es donde destaca la propiedad nativa de Neo4j. A diferencia de otras bases de datos, la de Neo4J no necesita calcular las relaciones de los datos durante las consultas, las conexiones ya están guardadas en la base de datos, gracias a esto, las consultas sobre datos altamente conectados son mucho más rápidas.

A esta propiedad también se le conoce como “graph first”, debido a cómo se procesan las operaciones que se realizan en la base de datos, tanto el almacenamiento como las consultas. Cuanto menor sea la sobrecarga de búsqueda de índices mayor será la capacidad de navegar a través de las conexiones de los datos, a esto se le llama adyacencia libre de índice. Esto significa que cada nodo hace referencia directamente a sus nodos adyacentes lo que significa que acceder a relaciones y datos relacionados es simplemente una búsqueda de puntero de memoria.

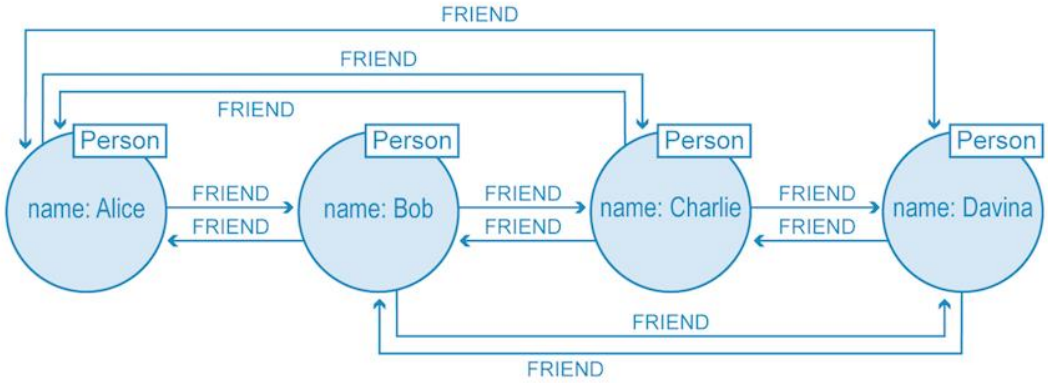

Ilustración 12 : Representación de la adyacencia sin índice en un grafo

A menudo, un procesamiento de grafos que no sea nativo necesita realizar una búsqueda de índices para alcanzar el siguiente elemento. Es por esto por lo que si el grado de relación es elevado el resultado de la consulta será muy lento e inadmisible debido a que, sin adyacencia sin índice, un conjunto de datos de grafos grande será aplastado por su propio peso porque las consultas tardarán cada vez más en realizarse a medida que crece el conjunto de datos.

En la ilustración anterior se busca un solo nivel de conexión en una relación de muchos a muchos. También se refleja en la imagen que cada consulta de un nivel o salto, requieres dos búsquedas de índice. A medida que se buscan más niveles de conexión, la complejidad del procesamiento aumenta exponencialmente.

Por otro lado, tenemos el almacenamiento de grafos el cual se refiere a la estructura subyacente de la base de datos que contiene datos de grafos. Cuando esta estructura se construye específicamente para almacenar datos en formato de grafos, como es el caso de Neo4j, se le conoce como almacenamiento nativo de grafos. Lo que hace que el almacenamiento de grafos sea distintivamente nativo es la arquitectura de la base de datos de grafos desde cero.

Neo4j consigue esto ya que cada capa de su arquitectura, desde el tiempo de ejecución del lenguaje de consulta Cypher hasta la gestión de los archivos en el disco, está optimizada para almacenar y consultar datos de grafos. Ninguna parte de esta estructura depende de tecnologías que no estén diseñadas específicamente para grafos [6].

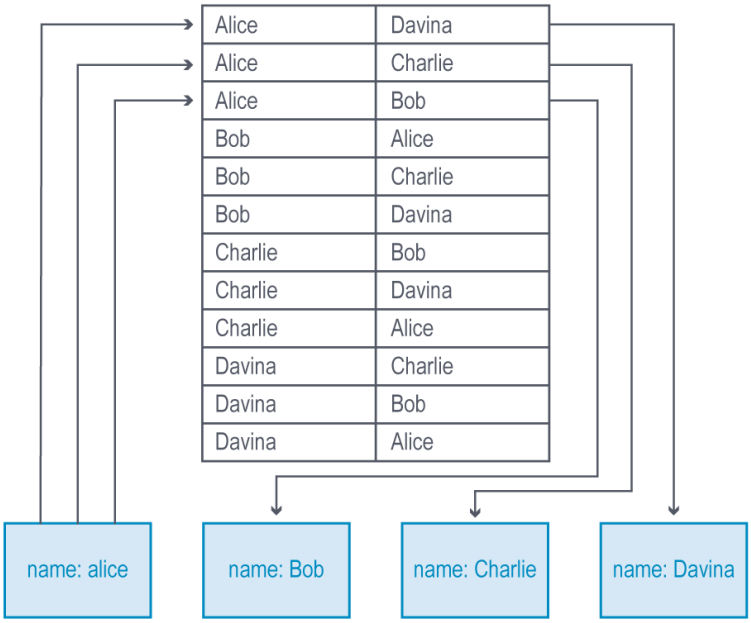

Ilustración 13 : Consulta a base de datos sin adyacencia sin índice

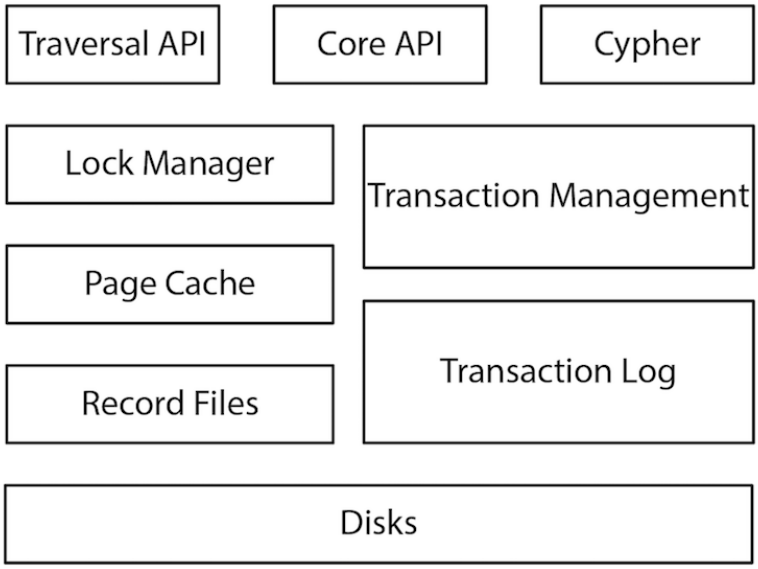

Ilustración 14 : Arquitectura de la base de datos nativa Neo4j

2.2.2 Capacidades adicionales de Neo4j

Además de la eficiencia y velocidad de consultas gracias a su propiedad nativa, Neo4j ofrece una serie de ventajas y capacidades que la convierten en una buena opción para manejar datos altamente conectados.

A continuación, se detallan algunas de estas características adicionales:

- Transacciones ACID (atomicidad, consistencia, aislamiento y durabilidad): lo cual garantiza la integridad de los datos en las transacciones.
- Soporte para clústeres: permitiendo la distribución de la carga de trabajo y mejorando la disponibilidad del sistema.
- Conmutación por error en tiempo de ejecución: garantizando la continuidad del servicio en caso de fallos [7].

Aunque en apartados anteriores ya hayamos realizado una comparación entre las principales BDOG que hay actualmente disponibles, Neo4J además destaca en su sencillez por los siguientes aspectos:

- Cypher es un lenguaje de consulta declarativo similar a SQL usado y desarrollado por Neo4J, con lo cual, está optimizado para este tipo de base de datos.
- Recorridos en tiempo constante en grafos grandes, tanto en profundidad como en amplitud, gracias a la representación eficiente de nodos y relaciones. Permite escalar hasta miles de millones de nodos en hardware moderado.
- Esquema de grafos de propiedades flexible que puede adaptarse con el tiempo, haciendo posible materializar y agregar nuevas relaciones posteriormente, así como propiedades de los nodos para atajar y acelerar cualquier necesidad de cambio.
- Drivers para una gran multitud de lenguajes de programación: Java, JavaScript y Python entre otros [7].

2.2.3 Elementos de Neo4j

Como ya hemos explicado, Neo4j almacena sus bases de datos de la misma manera que esbozaríamos un grafo en una pizarra. Estos grafos están compuestos por nodos conectados entre sí por relaciones. Recordemos que un nodo era aquel elemento que representaba a una entidad en el grafo mientras que las relaciones definían cómo estaban conectados entre sí los nodos. Además, los nodos podían contar con etiquetas y propiedades adicionales para que así estos ofrecieran más información y las relaciones podían además de tener una ponderación asociada, determinar si el grafo era o no simétrico o lo que es lo mismo, si estaba o no dirigido. No obstante, en Neo4j tanto los nodos como las relaciones tienen una serie de peculiaridades.

- Peculiaridades de los nodos en Neo4j:
    1. En Neo4j, un nodo puede tener cero, una o muchas etiquetas.
    2. No es necesario que existan propiedades para cada nodo con una etiqueta particular. Si una propiedad no existe para un nodo, esta se trata como un valor null.
    3. Deben crearse antes que las relaciones que los una, pero no es obligatorio que estén relacionados, es decir, podemos tener nodos sin conectar en nuestro grafo.
- Peculiaridades de las relaciones en Neo4j
    1. Cada relación debe tener una dirección en el grafo. De manera que, podemos diferenciar entre un nodo origen y un nodo destino. Es por esto por lo que los nodos deben crearse antes que la relación que los una. Aunque esta dirección es obligatoria, la relación se puede consultar en cualquier dirección o ignorarse por completo en el momento de la consulta.

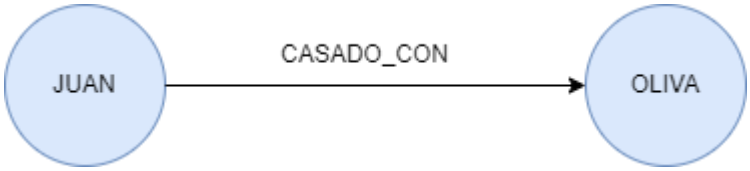

Ilustración 15 : Relación desde nodo origen hacia nodo destino

Si consideramos la imagen anterior, la relación CASADO_CON debe existir y puede proporcionar algún contexto adicional, pero puede ignorarse para los fines de la consulta.

2. Cada relación en un grafo Neo4j debe tener un tipo. Esto nos permite elegir en el momento de la consulta qué parte del gráfico recorreremos.
3. Al igual que ocurre con los nodos, las relaciones también pueden tener propiedades, es decir, ponderaciones.

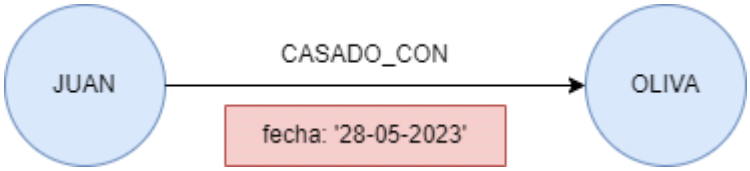

Ilustración 16 : Relación con propiedad o ponderación

2.2.4 El lenguaje Cypher

Cypher es el lenguaje de consulta de grafos declarativos de Neo4j. Fue creado en 2011 por los propios ingenieros de Neo4j como un lenguaje equivalente a SQL para bases de grafos. Cypher permite a los usuarios centrarse en qué recuperar del grafo, en lugar de en el cómo recuperarlo. Como tal, Cypher permite a los usuarios aprovechar todo el potencial de las bases de datos orientadas a grafos al permitir consultas eficientes y expresivas [8].

Ilustración 17 : Logo de Cypher, Neo4j

2.2.4.1 Sintaxis

La sintaxis que usa Cypher es bastante visual. Esto facilita el entendimiento tanto de las consultas como de la creación, modificación o borrado de nodos y relaciones. Lo que hace esto posible es el hecho de que Cypher sigue los principios de ascii-art para su sintaxis:

- Los nodos se representan mediante paréntesis “( )”
- Se puede especificar la etiqueta (o las etiquetas) de un nodo mediante el carácter “:” de la siguiente manera.

(:Persona) , siendo la etiqueta de ese nodo Persona

- Las relaciones se establecen mediante con dos caracteres "-", de la siguiente manera

  (:Persona)- -(:Libro). No obstante si recordamos, las relaciones deben tener una dirección establecida, esto lo conseguimos con los caracteres "<" y ">", el ejemplo anterior quedaría así

  (:Persona)-->(:Libro).

- El tipo de relación se especifica entre corchetes "[ ]" y de la misma manera que las etiquetas de los nodos con el carácter ":" de la siguiente manera [:ESCRITO]. Eso sí, esta relación debe ubicarse entre el carácter "-" de apertura y el carácter "-" de cierre.
- Las propiedades tanto de nodos como de relaciones se deben especificar entre llaves "{ }" siguiéndose una sintaxis JSON, por ejemplo: {título: "El Árbol de la Ciencia"}.

Definida esta sintaxis podríamos generar alguna consulta, por ejemplo, si queremos obtener al nodo Persona que ha escrito el libro El Árbol de la Ciencia, la consulta sería la siguiente:

(:Persona)-[:ESCRITO]-> (:Libro {título: 'El Árbol de la Ciencia})

Como comprobamos, la sintaxis es bastante intuitiva y fácil de recordar. La consulta anterior, podemos representarla gráficamente de la siguiente manera:

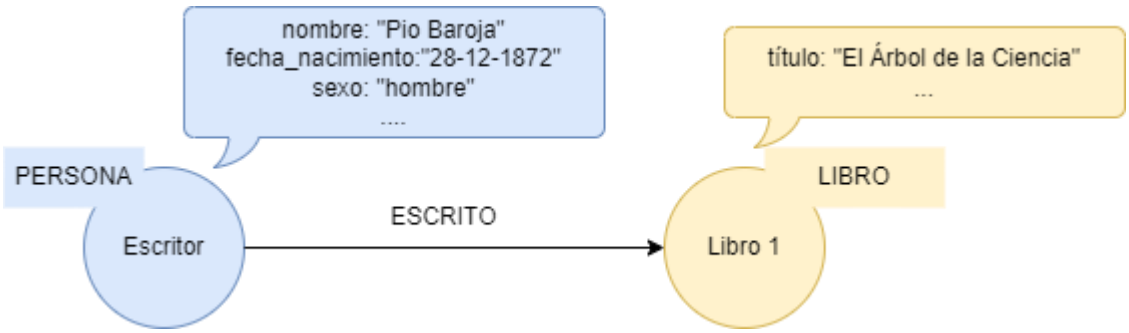

Ilustración 18 : Representación gráfica de la consulta realizada en Cypher

Concretamente el resultado de la consulta serían todas las propiedades del nodo que coincidiera que hubiera escrito el libro "El Árbol de la Ciencia".

Cypher funciona mediante la coincidencia de patrones en los datos. Al igual que en SQL o en otros lenguajes de bases de datos existen palabras reservadas. Por ejemplo, el FROM de SQL se corresponde con el MATCH de Cypher. Otras palabras reservadas que podemos encontrar en Cypher serían:

- **WHERE**: se utiliza para añadir condiciones a un patrón de búsqueda.
- **CREATE**: se utiliza para crear nodos y relaciones en el grafo.
- **MERGE**: Al igual que CREATE se puede utilizar para crear nodos y relaciones en el grafo. Lo que le diferencia de CREATE es que además, asegura que un patrón especificado exista. Si no existiera, lo crearía.
- **DELETE**: se utiliza para eliminar nodos y relaciones del grafo.
- **SET**: se utiliza para actualizar las propiedades de un nodo o relación.
- **REMOVE**: se utiliza para eliminar etiquetas de nodos o propiedades.
- **WITH**: se utiliza para encadenar varias partes de una consulta y pasar resultados intermedios.
- **RETURN:** se utiliza para especificar qué se debe devolver después de una consulta.

Supongamos que queremos obtener todos los nodos PERSONA de un grafo, ir buscándolos uno a uno sería engorroso. Cypher permite la asignación de variables a los nodos, estas deben preceder al carácter ":" de sus etiquetas para que posteriormente podamos retornar todos esos nodos mediante la palabra reservada RETURN que hagan MATCH con las características impuestas en nuestra consulta.

Por ejemplo, si en el siguiente grafo realizamos la consulta: MATCH (p:PERSONA) RETURN p;

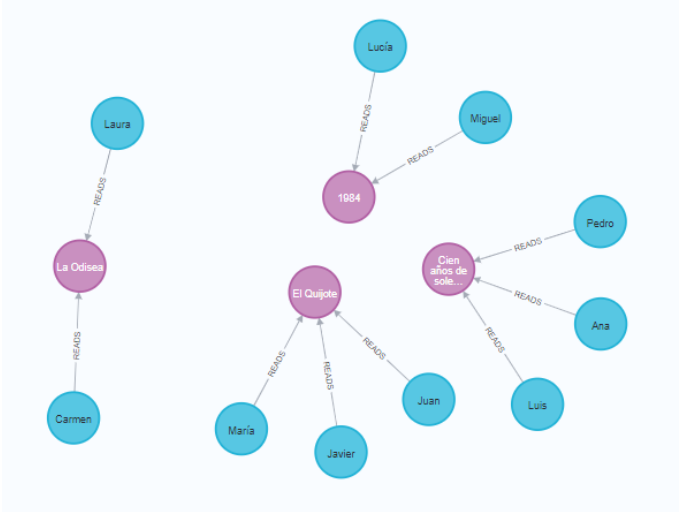

Ilustración 19 : Ejemplo de grafo en Neo4j Desktop

El resultado sería el siguiente:

```
match (p:Persona) return p;
```

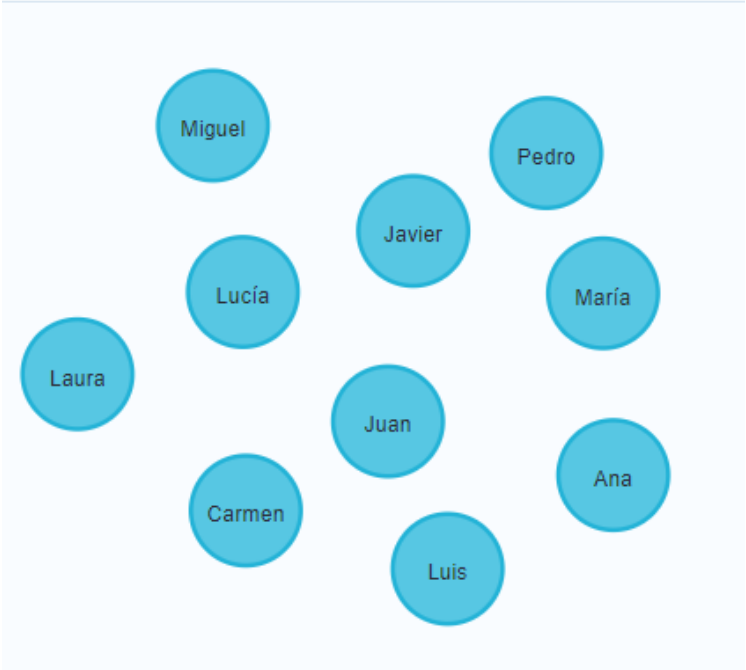

Ilustración 20 : Resultado de la consulta retornando el valor de una variable

Si nos interesara obtener solamente los nodos con la etiqueta PERSONA que hayan leído el libro con título “El Quijote”, la consulta que deberíamos realizar sería la siguiente, nótese el sentido de la relación de izquierda a derecha:

MATCH (p:PERSONA)-[:READS]->(:BOOK{title:"El Quijote"}) RETURN p;

Siendo este el resultado:

Ilustración 21 : Nodos PERSONA que han leído El Quijote

Si observamos la Ilustración 19, podemos comprobar que los nodos PERSONA que han leído “El Quijote”, se corresponden con Juan, Javier y María.

Las consultas realizadas anteriormente son algunos ejemplos sencillos del funcionamiento de Cypher en una base de datos Neo4j desplegada localmente. Mucha más información y contenido audiovisual sobre Cypher y en general sobre Neo4j puede ser consultado en su página web oficial [9].

2.3. FrontEnd – Angular

Una vez abordada la base de datos sobre la que se soportará el proyecto y el lenguaje que usa la misma para el manejo de los grafos, estamos en situación para introducir el resto de las herramientas usadas. Concretamente, en este apartado abordaremos las tecnologías usadas para el desarrollo del frontend de la aplicación. De esta manera, comenzaremos con Angular.

Angular es un framework frontend de código abierto basado en JavaScript, desarrollado y mantenido por Google. Es especialmente útil para crear aplicaciones y páginas web, y es adecuado tanto para versiones de escritorio como para aplicaciones móviles, así como para aplicaciones nativas [10].

Este framework, está escrito en TypeScript. Este lenguaje está muy relacionado con JavaScript, de hecho, ofrece todas y cada una de las funcionalidades de JavaScript y una capa adicional además de estas: el conocido como sistema de tipos de TypeScript. Por ejemplo, JavaScript proporciona tipos primitivos de lenguaje como “string” y “number”, pero no verifica que se hayan asignado de manera consistente. TypeScript sí lo verifica.

Esto significa que cualquier código JavaScript en funcionamiento existente también es código TypeScript. El principal beneficio de TypeScript es que puede resaltar comportamientos inesperados en el código, lo cual reduce la posibilidad de errores [11].

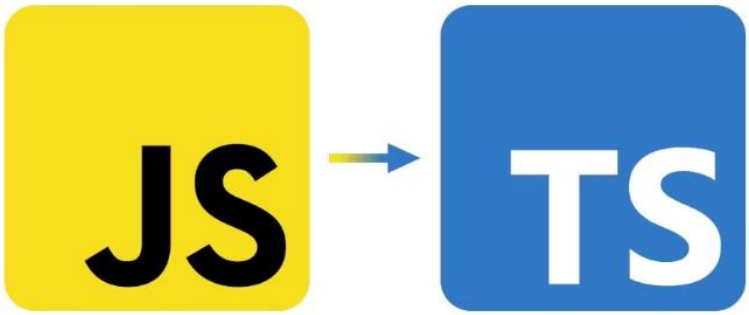

Ilustración 22 : De JavaScript a TypeScript

Además, Angular sigue una arquitectura basada en el patrón de diseño Modelo-Vista-Controlador (MVC), lo que facilita la sincronización entre HTML y TypeScript. Esta característica hace que este framework sea una herramienta muy versátil para el desarrollo web.

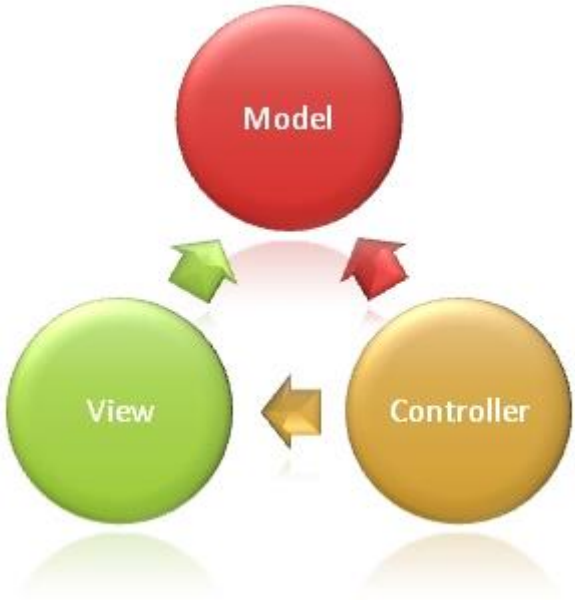

Ilustración 23 : Patrón MVC

Además, Angular proporciona una amplia biblioteca de plantillas, módulos, componentes y metadatos para simplificar el proceso de desarrollo. También está diseñado para automatizar y simplificar los procesos de desarrollo [10].

Entre las principales características de este framework, las cuales pueden ser consultadas en [12] tenemos:

- Multiplataforma
    1. Aplicaciones Web Progresivas (PWA): pues utiliza las capacidades modernas de la plataforma web para ofrecer experiencias similares a las de las aplicaciones. Instalación de alto rendimiento, sin conexión y en pocos pasos.
    2. Nativo: permite crear aplicaciones nativas con estrategias en Cordova, Ionic o NativeScript
    3. Escritorio: pues es posible crear aplicaciones instaladas en el escritorio y acceder a las APIs nativas del sistema operativo.
- Velocidad y rendimiento:
    1. Generación de código: Angular permite la conversión de plantillas en código altamente optimizado para las máquinas virtuales JavaScript de hoy en día

2. Universal: permite ser desarrollado en multitud de servidores para una representación casi instantánea en HTML y CSS.
3. División de código: las aplicaciones desarrolladas en Angular se cargan rápidamente gracias al enrutador de componentes, el cual ofrece una división automática del código para que se cargue solo aquel que sea necesario para representar la vista solicitada.

- Productividad
  1. Plantillas: permite la creación rápida de vistas de interfaz de usuario con una sintaxis de plantilla sencilla y potente.
  2. Angular CLI: ofrece una herramienta de línea de comando en la cual podemos generar componentes, servicios…
  3. Soportado por multitud de IDEs populares.

2.3.1 "Single Page Application" (SPA)

Toda aplicación Angular, es del tipo SPA, esto quiere decir que se trata de una aplicación de una única página web. El funcionamiento de este tipo de aplicaciones tiene una diferencia fundamental respecto a las aplicaciones web tradicionales.

Lo primero que ocurre en una aplicación ya sea SPA o no es que el cliente, hace una petición al servidor para recibir una página web. El servidor recibe esa petición inicial y responde mandándole al cliente el archivo HTML correspondiente que cargará en su navegador. La diferencia fundamental es que en una aplicación web tradicional cada vez que realizamos alguna operación, es decir: pulsamos un botón, enviamos un formulario, desplegamos un menú… esta enviará más peticiones al servidor y en consecuencia este enviará más archivos HTML como respuesta. Es decir, cada operación que realizamos en una página web tradicional se responde con un HTML por parte del servidor, de manera que estamos cargando plantillas constantemente. Por otro lado, en una página web SPA una vez obtenido el HTML como respuesta a la petición realizada inicialmente no se volverá a cargar un HTML entero durante esa ejecución en el navegador, es decir, no tendremos que actualizar la página, si no que esta se irá modificando por zonas.

Este comportamiento es posible gracias al uso de la tecnología AJAX. Cada vez que un usuario realiza una acción en una SPA, se envía una petición mediante AJAX al servidor y este nos responde con un documento JSON [13].

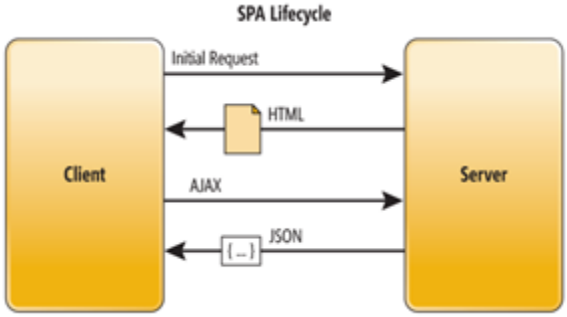

Ilustración 24 : Ciclo de vida de aplicaciones SPA

2.3.2 Estructura y flujo

En este apartado vamos a profundizar en la estructura de una aplicación Angular y el flujo de ejecución que esta sigue cuando es arrancada.

Cuando cargamos una aplicación Angular, esta busca una función conocida como "función principal de la aplicación", la cual está designada con el nombre "main". El contenido de esta es la definición del módulo principal que debe cargar la aplicación tras ser arrancada. Posteriormente, se carga el módulo AppModule,

también conocido como módulo raíz, el cual está definido en el archivo app.module.ts. Este se encarga entre otros aspectos de definir el componente principal que debe cargar para poder ejecutar la aplicación. Por defecto, el componente principal se denomina AppComponent. Este componente cuenta en su interior con:

- La definición de una clase la cual puede contener propiedades, distintos métodos y un constructor.
- La definición de un decorador, es decir, código JavaScript que permite añadir anotaciones, metadatos y especifica el comportamiento que va a tener la clase a la que acompaña, un decorador está formado por: un selector, una plantilla y un estilo CSS.

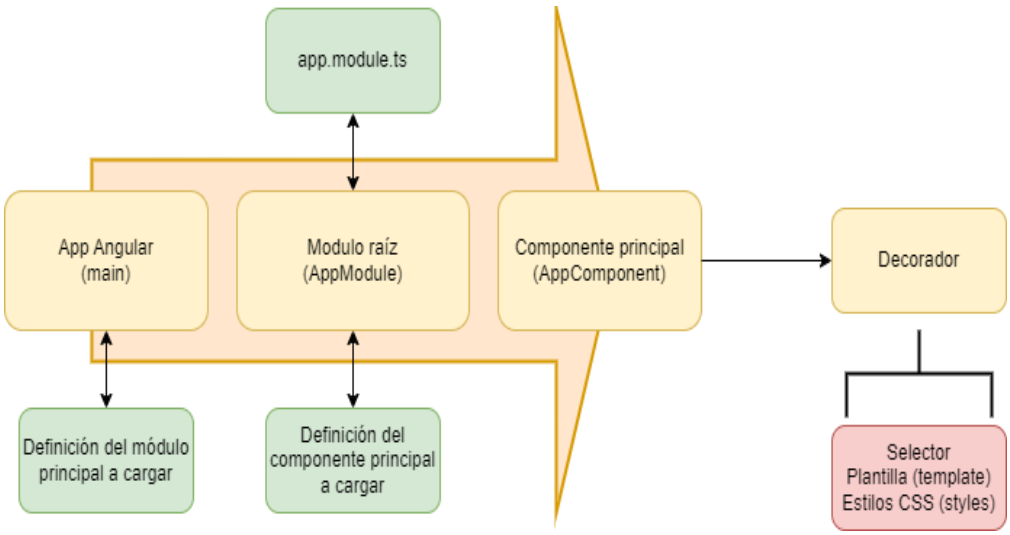

Ilustración 25 : Estructura y flujo de una aplicación Angular

2.3.3 Componentes en Angular

Los componentes son la base fundamental de las aplicaciones en Angular. Son bloques de construcción reutilizables y autónomos que encapsulan la lógica, la plantilla y los estilos de una parte específica de la interfaz de usuario. Es decir, un componente es una combinación de una plantilla HTML, estilos CSS y lógica TypeScript [14].

Ya sabemos que, una vez creada una aplicación Angular, esta cuenta por defecto con un componente principal, pero además de este, podemos crear nuestros propios componentes, cada uno con su clase y su decorador. Por supuesto este puede ser insertado en nuestro documento HTML además del componente por defecto.

Podemos crear incluso un componente (padre) y dentro de este crear más componentes (hijos) para posteriormente insertarlos en nuestras aplicaciones HTML siempre respetando el SPA, las posibilidades que Angular nos ofrece son numerosas y variadas.

Existen dos maneras fundamentales de crear componentes en Angular

- Manualmente.
- Automáticamente mediante línea de comandos (CLI); por ejemplo, mediante la consola del IDE Visual Studio Code (VSC).

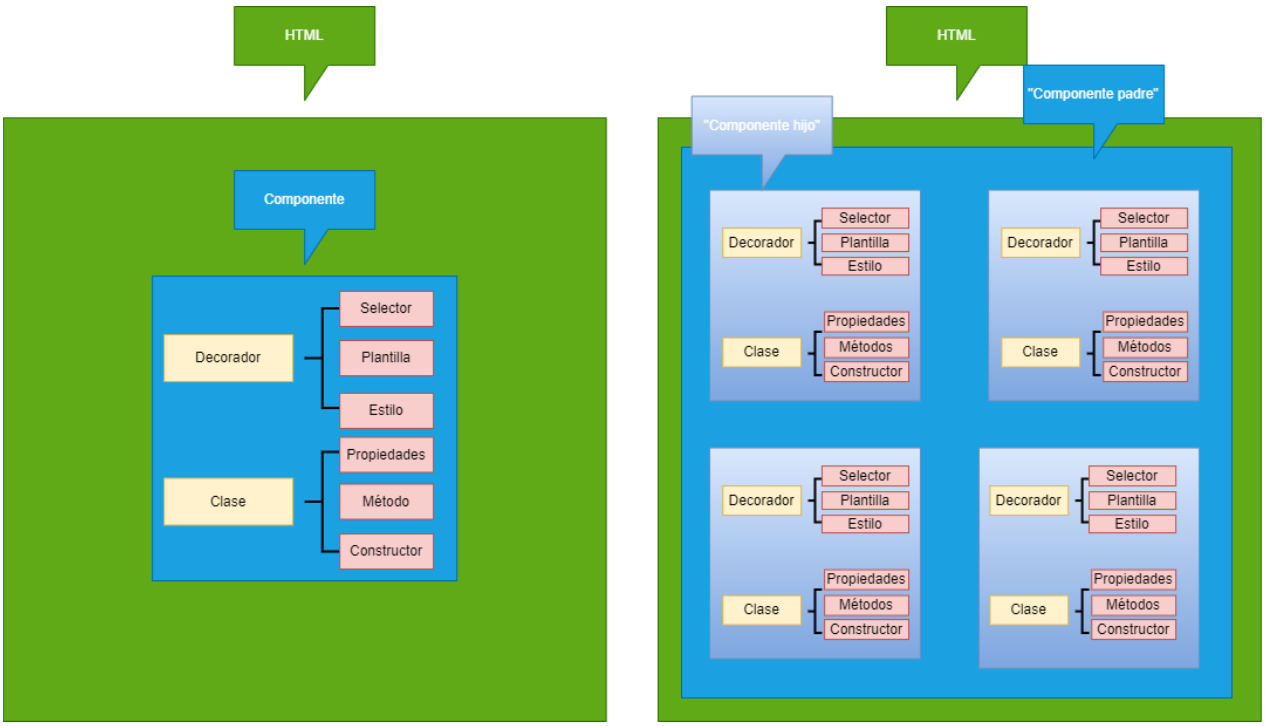

Ilustración 26 : Inserción de componentes Angular en una plantilla HTML respetando el SPA

2.4. BackEnd – Nest.js

Como ya hemos comentado, la parte del backend de la aplicación se desarrolla en Nest (NestJS) [15].

Nest es un framework diseñado principalmente para construir aplicaciones del lado del servidor en Node.js de manera eficiente y escalable. No obstante, también puede ser usado como tecnología para desarrollar un frontend. Utiliza JavaScript progresivo, está construido al igual que Angular con TypeScript y combina elementos de POO (Programación Orientada a Objetos), PF (Programación Funcional) y PRF (Programación Reactiva Funcional).

Si indagamos en las profundidades de Nest, comprobamos que este hace uso de robustos frameworks de servidor HTTP como Express (el predeterminado) y opcionalmente puede ser configurado para usar Fastify.

Nest proporciona un nivel de abstracción sobre estos frameworks comunes de Node.js (Express/Fastify), pero también expone sus APIs directamente al desarrollador. Esto da a los desarrolladores la libertad de usar la miríada de módulos de terceros disponibles para la plataforma subyacente.

Ilustración 27 : Logo de NestJS

En los últimos años, gracias a Node.js, JavaScript se ha convertido en la "lingua franca" del desarrollo web tanto para aplicaciones frontend como backend. Esto ha dado lugar a proyectos asombrosos (como por ejemplo Angular) que mejoran la productividad de los desarrolladores y permiten la creación de aplicaciones frontend rápidas, comprobables y extensibles. Sin embargo, aunque existen muchas bibliotecas, asistentes y herramientas excelentes para Node (y JavaScript del lado del servidor), ninguna de ellas resuelve eficazmente el principal problema de la arquitectura.

Nest proporciona una arquitectura de aplicación lista para usar que permite a los desarrolladores y equipos crear aplicaciones altamente comprobables, escalables, acopladas de manera flexible y fácilmente mantenibles. La arquitectura está fuertemente inspirada en Angular [15].

Hay tres motivos principales por los cuales finalmente se decidió seleccionar este framework en lugar de otro para el desarrollo del backend son los siguientes:

1. Consistencia en el Lenguaje: ya que ambos frameworks están construidos con TypeScript, lo cual simplifica en gran medida el desarrollo de nuestra aplicación reutilizando partes del código, como por ejemplo las interfaces y tipos.
2. Arquitectura Similar: NestJS se inspira en gran medida en la arquitectura de Angular. Esto significa que los conceptos como módulos, decoradores o componentes anteriormente mencionados se aplican de la misma forma.
3. El driver oficial de Neo4j para Node.js es altamente compatible con NestJS, permitiendo una integración sencilla y eficiente.

Otras ventajas que ofrece Nest y que pueden ser consultadas en [15] son las siguientes:

- Extensible: NestJS ofrece una flexibilidad excepcional a través de su arquitectura modular.
- Versátil: gracias a que proporciona una base sólida y bien estructurada para todo tipo de aplicaciones del lado del servidor.
- Progresivo: pues introduce patrones de diseño y soluciones establecidas al ecosistema de Node.js.

2.4.1 Desarrollo de la interfaz REST del backend NestJS

Uno de los objetivos principales que se ha buscado durante el desarrollo del proyecto ha sido desacoplar el frontend del backend a través de una interfaz REST con el objetivo de alimentar no solo nuestro frontend Angular con los datos obtenidos de Neo4j sino que también a cualquier otro sistema que actúe de cliente REST. Ganando así mucha versatilidad y otorgando una mayor entidad al proyecto. De manera que el desarrollo de esta interfaz REST supone la definición de los distintos métodos que estarán disponibles para realizar las distintas consultas a la base de datos.

2.5. Estilos – Bootstrap 5

El lenguaje usado para la elaboración de los estilos en nuestra aplicación web ha sido “Cascading Style Sheets” (CSS).

CSS se utiliza para definir cómo se deben presentar los elementos HTML en términos de diseño, color, fuente, tamaño y posición en la página web. Sin embargo, esta tarea a veces puede ser muy engorrosa y requiere de mucho tiempo. La tarea de diseñar unos estilos visuales y llamativos sin invertir una cantidad elevada de tiempo puede solucionarla Bootstrap.

Bootstrap es una biblioteca de código abierto que ofrece una serie de componentes y estilos predefinidos, lo que permite construir interfaces web de manera rápida y sencilla [16]. Además, es sencilla de integrar en Angular, es este otro de los motivos por el cual hemos decidido usar Bootstrap.

Ilustración 28 : Logo de Bootstrap

Es también gracias a Bootstrap que podemos acceder a una gran cantidad de clases predefinidas para diseñar distintos elementos HTML como podrían ser: botones, formularios o barras de navegación de manera sencilla.

Otra ventaja de Bootstrap es su amplia comunidad de usuarios y desarrolladores, lo que significa que hay una gran cantidad de recursos, documentación disponible en la web.

2.6. Entornos de desarrollo integrado usados - IDEs

Hasta ahora hemos abordado las distintas herramientas en las que se soportará el proyecto, es por ello que estamos en disposición de abordar los diferentes IDEs en los que se ha ejecutado todo el proceso de desarrollo de la aplicación full-stack. Por un lado, hemos usado Visual Studio Code para la elaboración tanto del frontend como del backend. Por otro lado, el despliegue y creación de la base de datos se ha llevado a cabo en la aplicación de escritorio Neo4j Desktop. La puesta en marcha de ambos permite el funcionamiento e integración de todas las partes por las que se compone el servicio. A continuación, profundizaremos en cada uno de ellos destacando sus aspectos más importantes.

2.6.1 Visual Studio Code (VSC)

Como hemos mencionado previamente, tanto el frontend como el backend han sido desarrollados en Visual Studio Code (VSC) [17].

Visual Studio Code (VSC) es un editor de código fuente gratuito y de código abierto desarrollado por Microsoft. Es conocido por su velocidad, flexibilidad y capacidad para manejar una amplia variedad de lenguajes de programación y tareas de desarrollo. A continuación, mostramos las principales características del editor, las cuales han sido obtenidas de [17]:

- Ligero: diseñado para ser altamente eficiente en recursos.
- Soporte para múltiples lenguajes de programación: de forma nativa o a través de extensiones, incluyendo JavaScript y TypeScript.
- Extensiones: Uno de los aspectos más relevantes de este editor son extensiones que permiten agregar funcionalidades específicas según las necesidades. No obstante, no es recomendable cargar al IDE de demasiadas extensiones.

- Terminal integrada: La cual es muy útil y práctica permitiendo ejecutar comandos desde el propio editor. De manera que es posible realizar acciones como el arranque del servidor o la creación de componentes entre otras, facilitando así el desarrollo.

Ilustración 29 : Logo IDE Visual Studio Code (VSC)

La familiarización previa a la realización del proyecto con este IDE ha sido otro motivo por el cual finalmente nos hemos decantado por él. Otro de los aspectos que se podría destacar del editor es que es compatible con una gran variedad de sistemas operativos. Además, su menú "IntelliSense" proporciona autocompletado de código inteligente, resaltado de sintaxis, y sugerencias contextuales, lo cual ha optimizado en gran medida el desarrollo del proyecto.

2.6.2 Neo4j Desktop

Para la creación de nuestra BDOG hemos usado Neo4j Desktop [18]. Neo4j Desktop es una aplicación cliente diseñada para trabajar exclusivamente con Neo4j. Este IDE está destinado a facilitar el aprendizaje y la experimentación con Neo4j de manera local, proporcionando todo lo necesario para desplegar y crear bases de datos de manera sencilla y eficiente. Además, Neo4j Desktop permite la creación ilimitada de estas bases de datos locales [18].

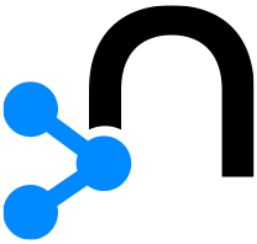

Ilustración 30 : Logo Neo4j Desktop

El motivo principal de haber usado este IDE para nuestra base de datos es que al estar desarrollada por la empresa Neo4j su integración y compatibilidad con esta base de datos es completa. Además, este IDE es muy visual y el despliegue de una base de datos es casi inmediato. También proporciona al igual que Visual Studio Code una consola para realizar modificaciones o consultas a la base de datos en tiempo real.

En cuanto iniciamos la aplicación nos aparece un menú como este:

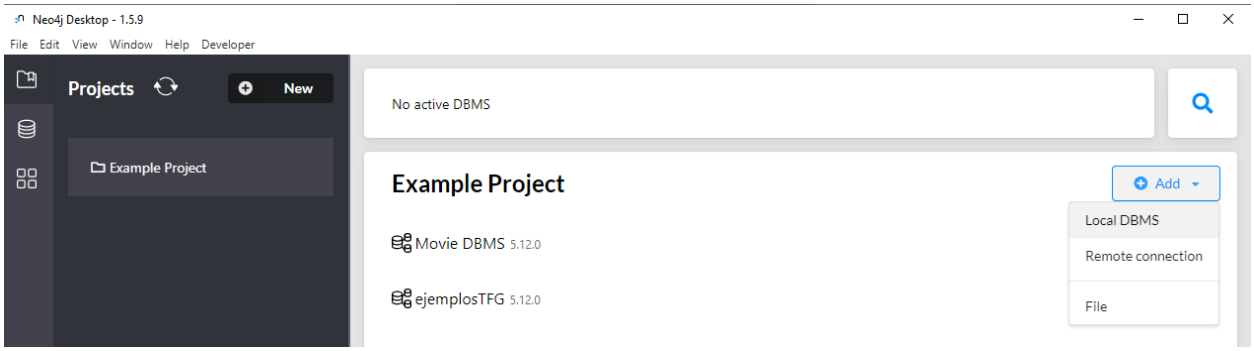

Ilustración 31 : Menú Neo4j Desktop

Como se puede apreciar en la ilustración 31 este es bastante intuitivo y el despliegue de una base de datos local parece casi inmediato. No obstante, abordaremos esta cuestión en apartados posteriores de la memoria.

3 DESARROLLO Y RESULTADOS

Todo lo complejo puede dividirse en partes simples.

- René Descartes -

En este capítulo se abordará el proceso de desarrollo del proyecto. Analizaremos y desglosaremos cada una de las secciones en las que este se divide atendiendo a los elementos que conforman cada una de ellas, se expondrá también la manera en la que las distintas tecnologías interaccionan entre sí, así como las adversidades encontradas durante el proceso de desarrollo. Finalmente, se mostrarán los resultados obtenidos.

En la Ilustración 32, se muestra el esquema del sistema que vamos a diseñar y desarrollar, mostrándose claramente las tecnologías usadas, si bien es cierto que la implementación e integración de estas se abordará en detalle en el apartado Materiales y métodos de la memoria. Nótese que en la ilustración la comunicación entre el frontend y el backend se lleva a cabo mediante una API REST brindando mucha versatilidad a la solución desarrollada.

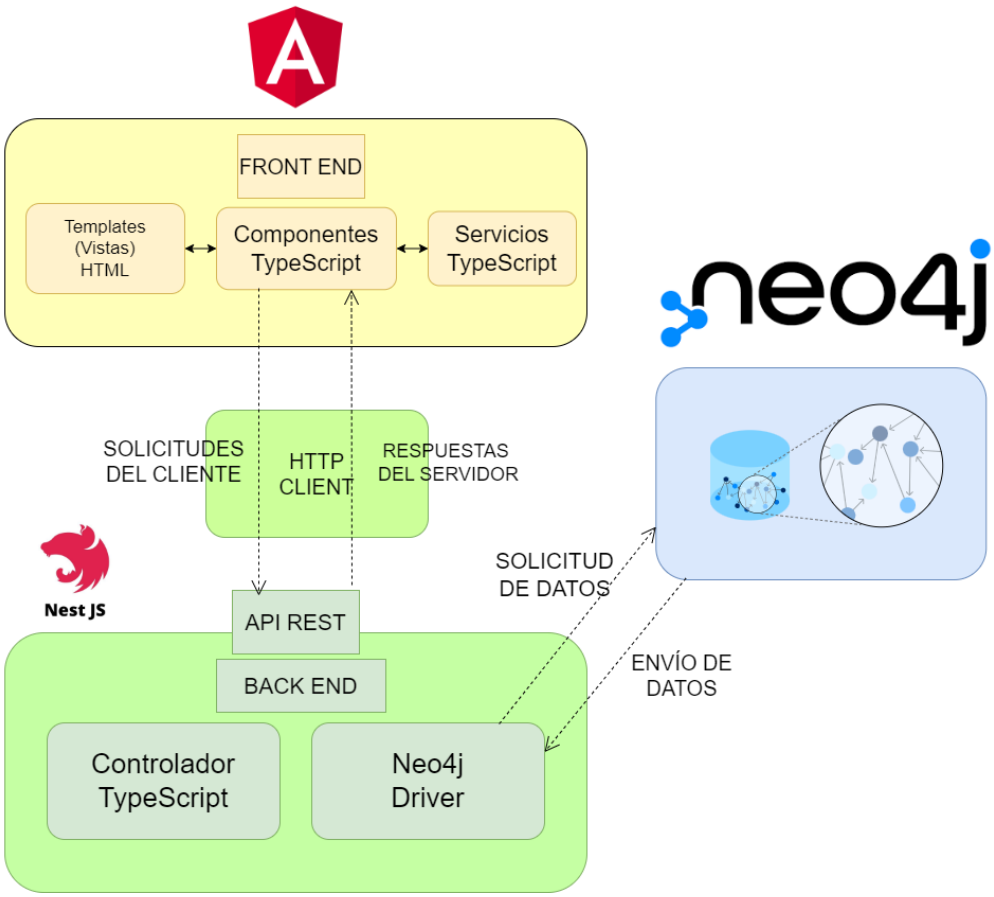

Ilustración 32 : Comunicación entre Front-end, Back-end y base de datos Neo4J

3.1 Neo4j – Almacenamiento de los datos

Como ya sabemos la base de datos orientada a grafos que hemos escogido para el desarrollo del proyecto es Neo4j. De esta manera y como ya se ha mencionado en capítulos anteriores, hemos optado por el uso de una de las aplicaciones de escritorio de las que dispone la empresa Neo4j para el despliegue local de nuestro grafo, Neo4j Desktop, concretamente la versión 1.5.9. Dentro de Neo4j Desktop encontramos diversas herramientas entre las cuales destacaremos Neo4j Browser. Esta herramienta ofrece una consola integrada con el lenguaje CYPHER de manera que nos permite interactuar con nuestras bases de datos Neo4j con capacidades CRUD ("Create", "Read", "Update", "Delete") completas. Además, Neo4j Browser permite ver los resultados obtenidos en una variedad de formatos, incluida la vista gráfica, tabular y JSON. A continuación, explicaremos el proceso de creación y despliegue local de una base de datos Neo4j en Neo4j Desktop.

1. Tal y como se muestra en la ilustración 31, seleccionamos Add > Local DBMS
2. Introducimos la versión que soportará la base de datos, así como las credenciales que deseemos, es crucial recordarlas ya que, si no, no sería posible establecer una conexión futura con el backend. Una vez que pulsemos el botón "Create" ya tendríamos creada y desplegada localmente nuestra base de datos.

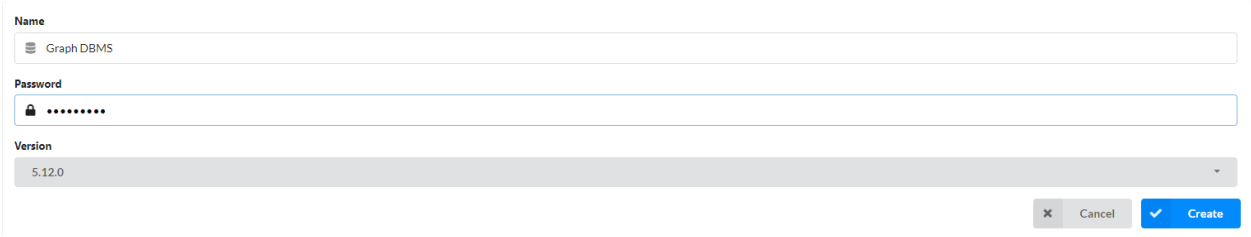

Ilustración 33: Formulario de creación de una base de datos en Neo4j Desktop

Así se vería ahora el menú principal de Neo4j Desktop tras la creación de esta nueva base de datos de ejemplo "Graph DBMS"

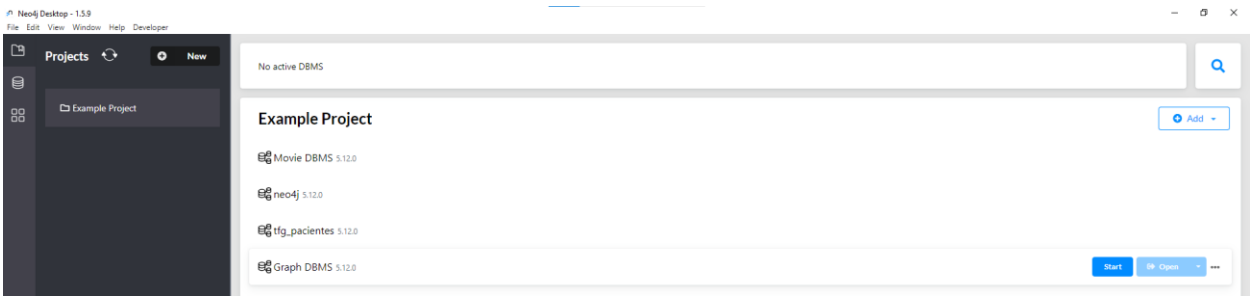

Ilustración 34 : Listado de bases de datos creadas en Neo4j Desktop

En caso de que queramos arrancar la base de datos simplemente pulsamos el botón "Start" y posteriormente se habilitará en botón "Open" que en caso de seleccionar esta opción se nos lanzará a Neo4j Browser que por defecto despliega la base de datos en bolt://localhost:7687. A continuación, se muestra la interfaz de Neo4j

Browser tras la creación de una base de datos.

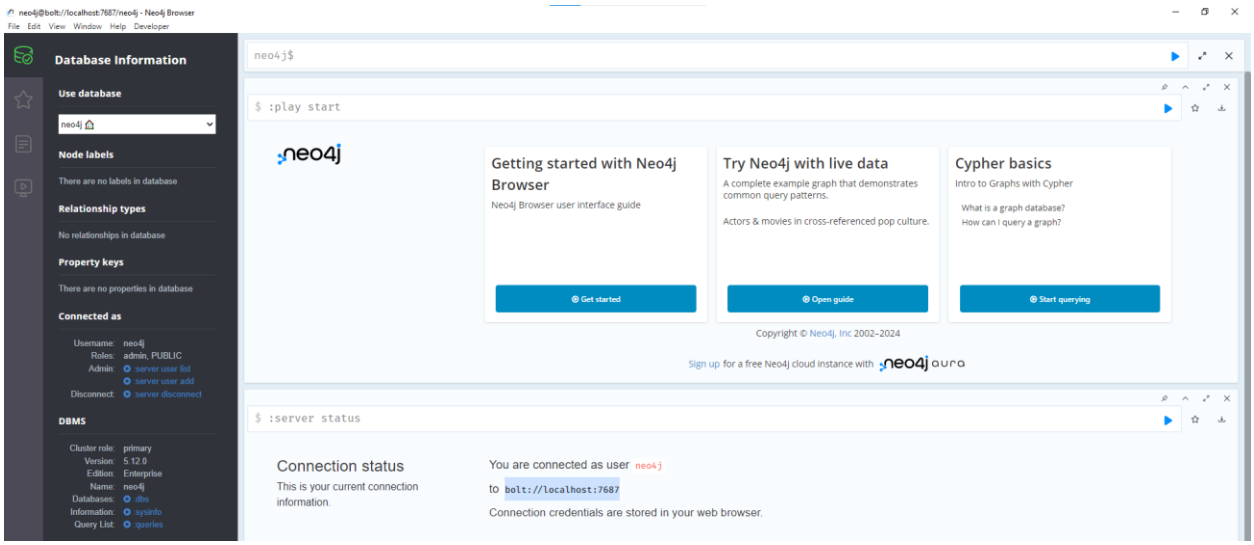

Ilustración 35 : Interfaz gráfica de Neo4j Browser

Como podemos observar en la ilustración 34, la base de datos creada como ejemplo está completamente vacía, en este punto podemos optar por dos opciones:

1. Volcar los datos a la base de datos mediante un fichero csv.
2. Crear los datos manualmente.

3.1.1 Nodos y relaciones

Una vez explicado el procedimiento de creación y despliegue de una base de datos en Neo4j Desktop estamos en disposición de comenzar a explicar cómo se ha llevado a cabo este proceso para el desarrollo del proyecto.

Para dar comienzo al proyecto se hizo uso de la opción 1 para la generación de los datos, es decir, mediante el volcado de un fichero csv, concretamente con la información de los nodos.

El contenido de este fichero csv consistía en un número reducido de nodos, las propiedades de estos y su valor. Estas propiedades serán las mismas durante todo el desarrollo del proyecto, es decir, todos los nodos que se creen posteriormente tendrán el mismo número de propiedades en función al tipo de nodo en cuestión, eso sí, el valor de estas variará.

Comenzaremos plasmando de forma tabular los nodos por los que estará compuesta nuestra base de datos.

NODO	PROPIEDADES	DESCRIPCIÓN
Patient	PacienteID	Identificador del paciente
	PatientName	Nombre del paciente
	Age	Edad del paciente
	Gender	Género del paciente
	BirthDate	Fecha de nacimiento del paciente
	PatientDevice	Identificador del dispositivo relacionado con el paciente
Device	DeviceID	Identificador del dispositivo

	DeviceType	Tipo de dispositivo
	SerialNumber	Número de serie del dispositivo
	DeviceStatus	Estado del dispositivo
Observation	ObservationID	Identificador de la observación
	ObservationDeviceId	Identificador del dispositivo que ha tomado esta observación
	ObservationPatientId	Identificador del paciente al que se refiere esta observación
	ObservationValue	Valor de la observación

Tabla 4 : Tipos de nodo que conforman la base de datos

Mostramos a continuación un pequeño fragmento del lenguaje CYPHER que hemos usado para volcar los datos de los nodos *Patient* desde el fichero csv hacia nuestra base de datos destacando algunos aspectos.

```
LOAD CSV WITH HEADERS FROM 'file:///datosNeo4J.csv' AS fila FIELDTERMINATOR ',' WITH fila WHERE fila.P_ID IS NOT NULL CREATE (:Patient { PatientID: fila.P_ID, Age: fila.P_Age, Gender: fila.P_Gender, BirthDate: fila.P_Birthtime, PatientDevice: fila.P_relatedDevice })
```

- FIELDTERMINATOR: carácter por el cual están separados los distintos campos
- file:///datosNeo4J AS fila: nombre con el que nos referiremos al fichero csv
- WHERE fila.P_ID IS NOT NULL CREATE: si el valor P_ID no es nulo crea el nodo
- :Etiqueta_nodo: {Valor_propiedad: nombre_fichero.Propiedad}

Realizamos el mismo proceso con los nodos *Device* y los nodos *Observation*.

Una vez volcado el fichero csv a la base de datos, ya tenemos un grafo con nodos, pero aún sin relaciones. Por ejemplo, si quisiéramos obtener todos los nodos Device, este sería el resultado gráfico obtenido.

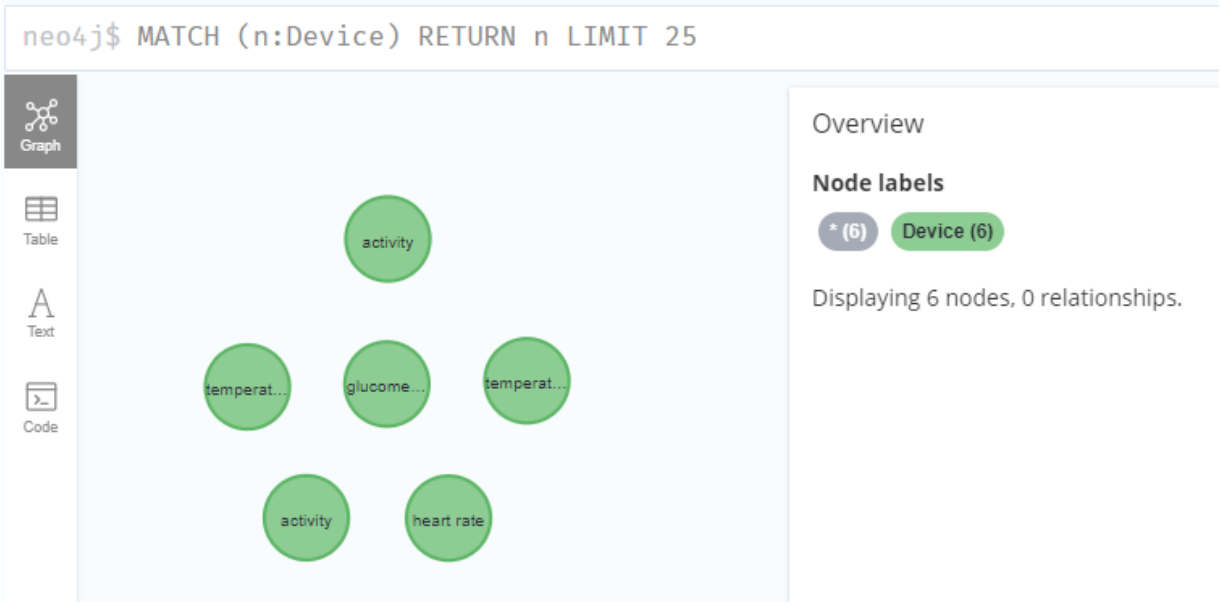

Ilustración 36 : Nodos Device tras el volcado del fichero csv

En cuanto a las propiedades de los nodos, pueden ser consultadas si mantenemos el cursor sobre cualquiera de ellos siendo este el resultado:

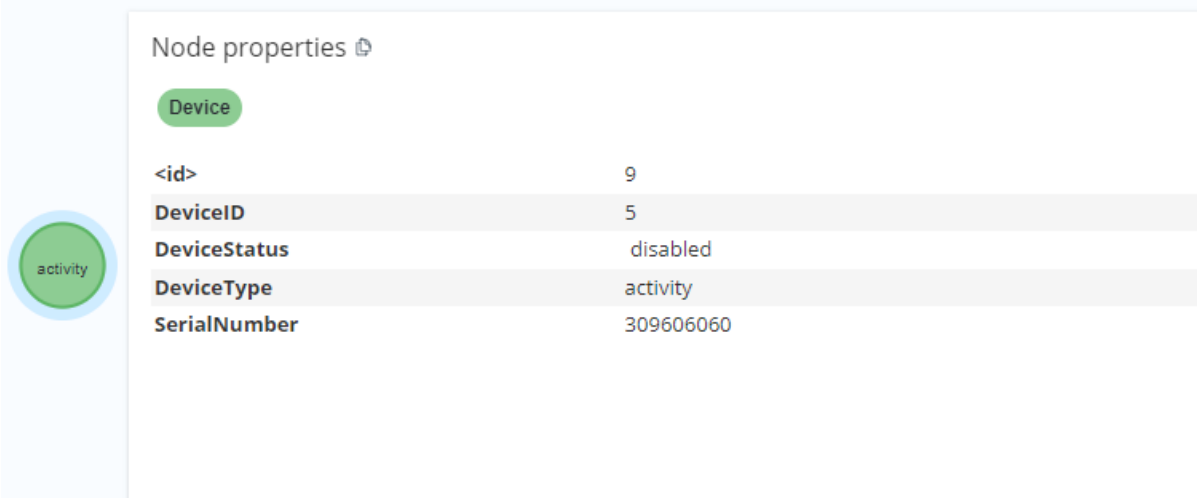

Ilustración 37 : Propiedades de un nodo Device

Si consultamos la tabla 4, podemos comprobar que las propiedades de los nodos Device coinciden con las mostradas en la ilustración 36.

Una vez definidos los tipos de nodos y la información de estos, daremos paso a las relaciones. Estas fueron generadas manualmente. Recordemos que el procedimiento debe ser este, es decir, Neo4j impone que primero deben ser creados los nodos y posteriormente sus relaciones.

A continuación, se muestran representadas en la siguiente tabla las relaciones de las que dispondrá nuestra base de datos.

Tipo de relación	Nodo origen	Nodo destino	Descripción
HAS_DEVICE	Patient	Device	Relaciona a los pacientes con sus dispositivos.
HAS_ASSIGNED	Device	Observation	Relaciona a los dispositivos con sus observaciones.
HAS_OBSERVATION	Patient	Observation	Relaciona a los pacientes con sus observaciones.

Tabla 5 : Tipos de relaciones que conforman la base de datos

Las relaciones expuestas anteriormente permiten mantener ordenada la base de datos. Sin embargo, durante las consultas desde el backend, no es estrictamente necesario utilizar estas relaciones para obtener los resultados deseados. Esto es posible gracias a que la implementación de la base de datos se ha realizado meticulosamente, teniendo en cuenta detalles como la correspondencia entre el valor del atributo PatientID de los nodos *Patient* y el valor ObservationPatientID de las observaciones relacionadas con ese paciente. Es decir, todos los nodos *Observation* de un nodo *Patient* concreto tendrán siempre el mismo valor en el campo ObservationPatientID, el cual se corresponderá con el atributo PatientID del paciente en cuestión.

El siguiente ejemplo muestra un fragmento del lenguaje CYPHER para importar las relaciones HAS_DEVICE.

```
LOAD CSV WITH HEADERS FROM "file:///C:datosNeo4J.csv"
AS fila FIELDTERMINATOR ','
MATCH (p:Patient {PatientID:fila.O_patientID})
MATCH (d:Device {DeviceID:fila.O_deviceID})
```

```
MERGE (p)-[r:HAS_DEVICE]->(o)
```

- FIELDTERMINATOR: carácter por el cual están separados los distintos campos
- file:///datosNeo4J AS fila: nombre con el que nos referiremos al fichero csv
- Finalmente seleccionamos los nodos *Patient*

Se muestran a continuación algunas relaciones HAS_DEVICE que residen en la base de datos.

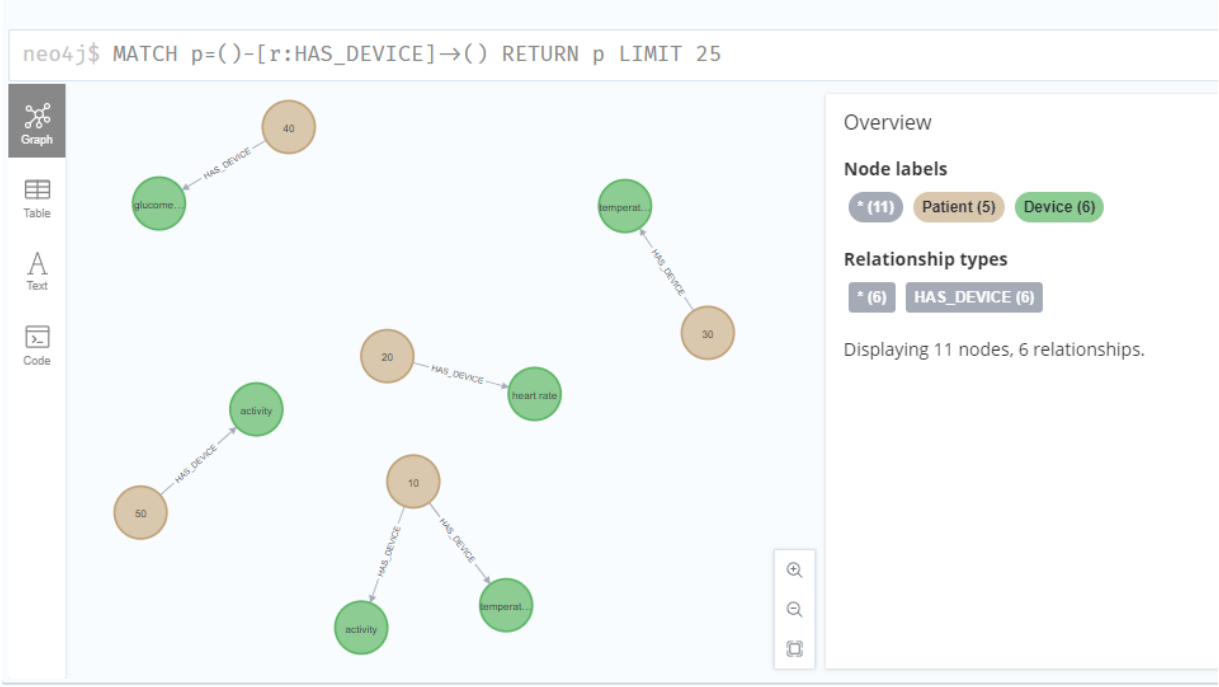

Ilustración 38 : Relaciones HAS_DEVICE del grafo

Podemos destacar de la ilustración 37 que el mismo nodo puede estar relacionado con varios nodos a través de la misma relación, es decir, en este caso el nodo *Patient* con identificador 10 tiene establecida la relación HAS_DEVICE con dos nodos *Device* diferentes, uno de actividad (activity) y otro de temperatura (temperature).

Mostramos a continuación un ejemplo de cómo se relacionarían entre sí un nodo *Patient* con sus nodos *Observation* y *Device* correspondientes a través de las relaciones mostradas en la Tabla 5.

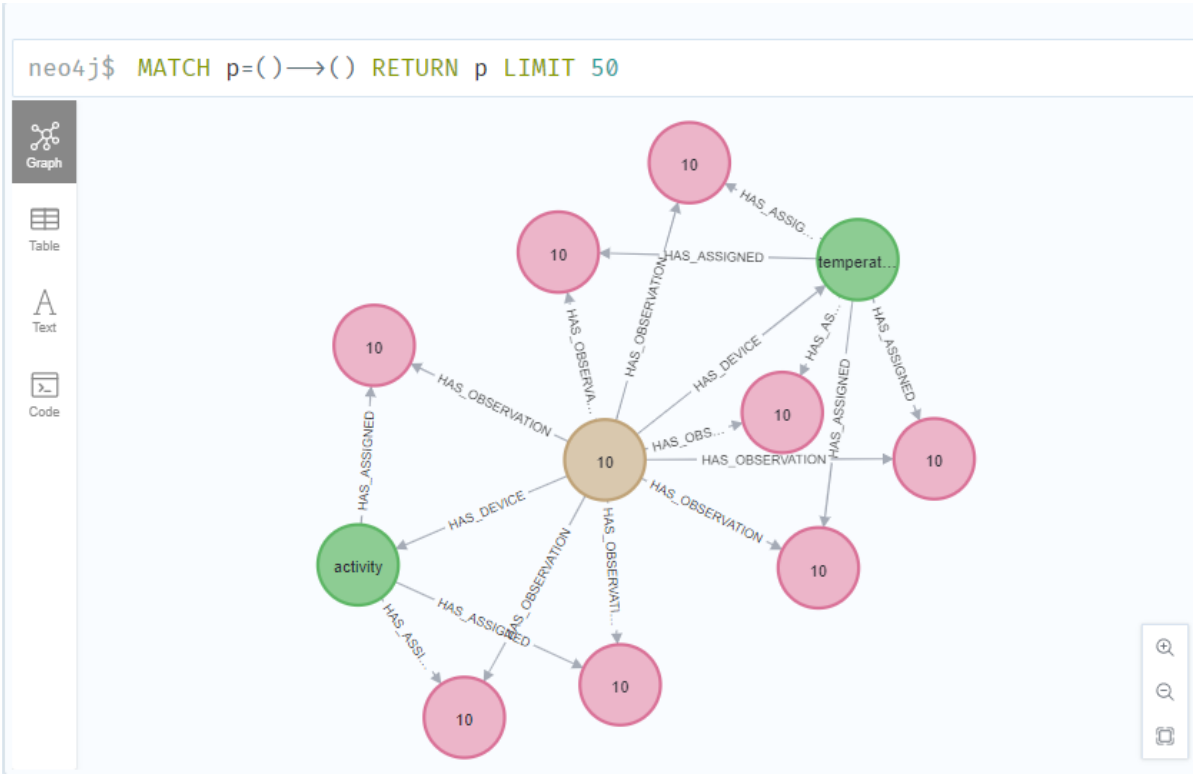

Ilustración 39 : Nodos conectados través de distintas relaciones

Podemos destacar los siguientes aspectos de la imagen anterior:

1. Como comentamos anteriormente, un nodo, puede tener establecidas varias relaciones del mismo tipo con distintos nodos.
2. Un nodo puede tener establecidas varias relaciones y diferentes con otros nodos.
3. Diferenciamos un paciente que tiene asociados dos dispositivos diferentes cada uno de ellos con sus mediciones.

Llegados a este punto es importante destacar un aspecto crucial que se ha tenido en cuenta a la hora del desarrollo de la base de datos. Si recordamos las propiedades de los nodos *Observation* (véase Tabla 4), entre ellas tenemos: ObservationdeviceID y ObservationpatientID. Como ya hemos dicho, se ha implementado la base de datos de manera que sus valores deben corresponderse entre sí, tanto con el ID del nodo *Device* como con el ID del nodo *Patient* que estén asociados, esto permitirá facilitar las consultas en el backend. Por tanto, no solo podremos obtener los nodos *Patient* haciendo uso de las relaciones, sino que también haciendo uso de los parámetros ObservationdeviceID y ObservationpatientID, que como hemos dicho se corresponden con los parámetros DeviceID y PatientID respectivamente.

Mostramos en la siguiente ilustración el estado de la base de datos tras haber incluido tanto todo tipo de nodos como todo tipo de relaciones. Gracias a la capacidad de visualización que ofrece Neo4j Browser podemos diferenciar fácilmente cada trío de nodos *Patient, Observation, Device*. Hay que destacar también que solo se muestra un subconjunto del conjunto total de datos que conforman la base de datos para así diferenciar correctamente los elementos que la componen evitando así una ilustración saturada de elementos y poco legible.

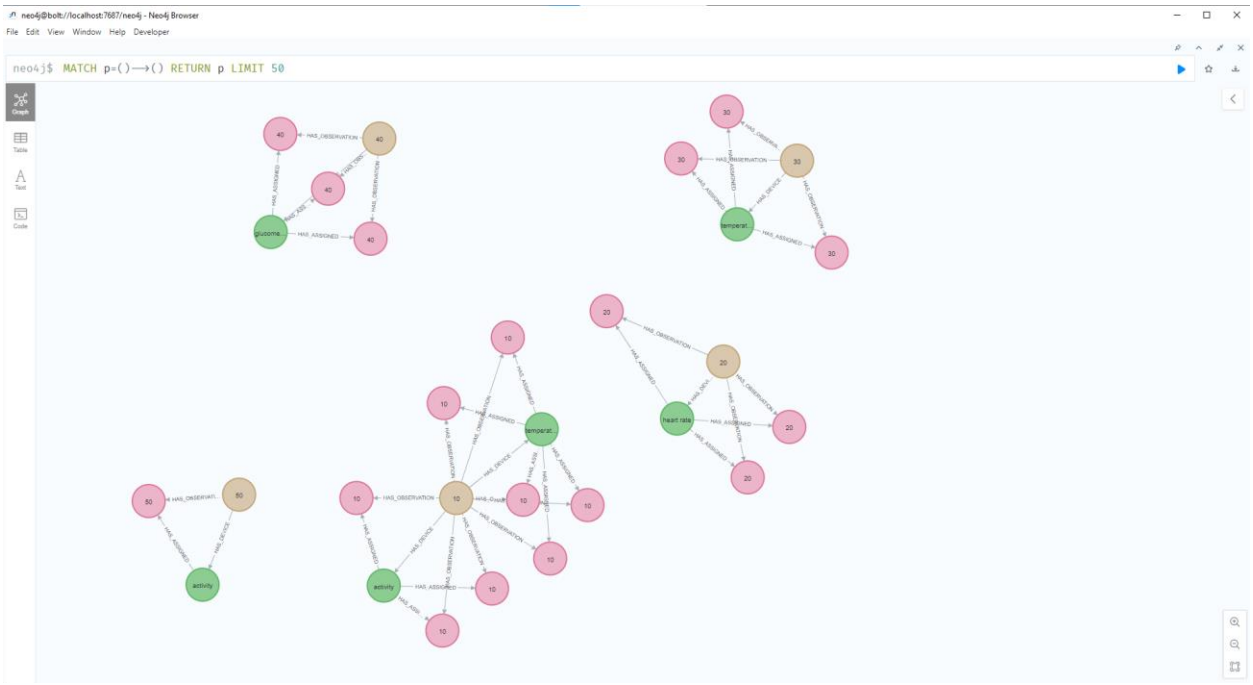

Ilustración 40 : Subconjunto de nodos y relaciones de la base de datos del proyecto

3.2 Desarrollo del backend – NestJS

Una vez definida la base de datos en su totalidad habiendo abordado desde el despliegue de esta hasta los elementos que la componen, podemos sumergirnos en el desarrollo del backend.

Como ya sabemos, este ha sido desarrollado a partir del framework NestJS. La instalación de esta librería se llevó a cabo desde la consola de Windows, aunque nuestro IDE VSCode a través de su propia terminal también permite realizar esta tarea.

Los pasos realizados para crear el proyecto del backend en NestJS fueron los siguientes:

1. Instalación previa de Node.js y npm (Node package manager).
2. Instalación de NestJS CLI a través de la consola de Windows haciendo uso del comando:

   npm i -g @nestjs/cli.

   Esta herramienta es fundamental para poder usar las distintas herramientas que

   ofrece Angular para la creación de aplicaciones.
3. Creación del proyecto, esta vez desde la terminal de Visual Studio Code. Esta tarea es muy sencilla y simplemente debemos introducir el comando:

   nest new nombre_proyecto

   en nuestro caso hemos definido este proyecto con el nombre de “api”.

Tras haber realizado esos pasos, tendremos nuestro proyecto Nest creado, para iniciarlo simplemente introducimos el comando: npm run start:dev. Por defecto, los proyectos en NestJS arrancan en localhost:3000, esta opción puede modificarse desde el fichero main.ts. Sin embargo, nosotros hemos decidido mantener esta configuración ya que no solapaba con ningún otro puerto. Si accedemos desde el navegador a esta URL, el proyecto Nest de nueva creación tiene el siguiente aspecto, dándonos la bienvenida con el conocido “Hello World”.

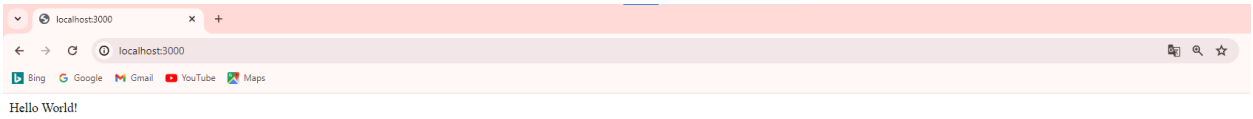

Ilustración 41 : Proyecto de nueva creación en NestJS

Es en este punto donde podemos dar paso a las tareas realizadas para el desarrollo del backend. Estas son:

- Integración de Nest con Neo4j a través de neo4j-driver.
- Definición de una API REST para que esta sea consumida por el frontend, esto conlleva tanto el desarrollo de un controlador que determina los métodos a usar y en que URL deben ejecutarse, es decir, la definición de los distintos endpoints. Así como la definición de un servicio que establezca los métodos que usará el controlador. Estos métodos se basan en la implementación de las distintas consultas y en el correcto almacenamiento de sus respuestas para que estas viajen en un formato correcto en el cuerpo de la respuesta a la petición previamente realizada desde el frontend.

Abordaremos a continuación cada uno de estos aspectos detalladamente.

3.2.1 Integración mediante neo4j-driver

La empresa Neo4j ofrece de manera gratuita esta librería para poder interactuar con bases de datos Neo4j a través de distintos lenguajes de programación entre ellos TypeScript.

Los pasos realizados para implementar el driver en NestJS y conseguir así integrar nuestro backend con la base de datos Neo4j, han sido los siguientes:

1. Instalación del driver en nuestro backend, esta tarea la realizamos desde la terminal que ofrece nuestro IDE, de manera que debemos introducir estando ubicados en la carpeta raíz del proyecto el siguiente comando: npm install neo4j-driver. Además, debemos también instalar las dependencias necesarias.
2. Definición de una interfaz de configuración para el driver, los parámetros que la conformen serán los que posteriormente el módulo principal establecerá para que se permita así la conexión a nuestra base de datos.
3. Definición de la utilidad para el controlador, lo cual implica elaborar una función que cree y verifique la conectividad del driver de neo4j.
4. Creación de un módulo en el cual se creará una instancia del driver y se definirá el servicio que posteriormente se usará para poder obtener los resultados de las distintas consultas que realicemos desde el backend.
5. Creación del servicio que permitirá la interacción desde el backend con neo4j, es decir, en este se definen los métodos que permitirán realizar las consultas, no los métodos que realizan las propias consultas y devuelven los resultados de las mismas. Estos se definirán posteriormente.
6. Configuración del módulo principal de nuestro proyecto Nest para que establezca los valores de la instancia del driver previamente creado por el módulo de Neo4j, es decir, para realizar esta tarea, el módulo principal del proyecto Nest debe hacer uso del módulo Neo4j para acceder a la instancia del driver y así establecer los valores de esta.

Tras realizar estos pasos, tendremos definido el driver que permitirá la interacción de nuestro backend con la base de datos Neo4j, es importante que a la hora de establecer los valores del driver desde el módulo principal recordemos los parámetros con los que se definió la base de datos a la hora de ser desplegada. En el apartado de

la memoria 3.1 se comentaba la importancia de recordar las credenciales de creación ya que posteriormente iban a ser necesitadas en el backend.

3.2.1 Desarrollo de una API REST

El segundo elemento importante que hemos desarrollado en el backend es una API REST con el objetivo de no solo alimentar a nuestro frontend Angular sino a cualquier otro sistema que actúe de cliente REST. La elaboración de nuestra API se ha basado en el desarrollo de dos elementos principales, un servicio y un controlador que lo consume.

3.2.1.1 Servicio de la API REST – AppService

Los servicios son una pieza esencial de las aplicaciones realizadas con el framework NestJS. Están pensados para proporcionar una capa de acceso a los datos que necesitan las aplicaciones para funcionar. Mediante los servicios podemos liberar de código a los controladores y conseguir desacoplar éstos de las tecnologías de almacenamiento de datos que estemos utilizando.

Podemos definirlos como clases, de programación orientada a objetos clasificadas como "provider", un tipo especial de artefactos dentro del framework.

Los providers están preparados para ser inyectados en los controladores de la aplicación, y otras clases si fuera necesario, a través del sistema de inyección de dependencias que proporciona Nest [19]. Para realizar esta tarea, debemos hacer uso del decorador @Injectable() en el servicio, de esta manera el servicio podrá ser inyectado en el controlador y este podrá hacer uso de los métodos definidos en él.

En nuestro caso, hemos definido el servicio app.service.ts. Lo que encontramos en este son los métodos necesarios para gestionar los datos de pacientes, dispositivos y observaciones almacenados en nuestra base de datos Neo4j. A continuación, se detallará el comportamiento y contenido de cada uno de estos métodos:

Método	**getInfoPacientes**
Descripción	Este método se encarga de obtener la información de todos los pacientes almacenados en la base de datos Neo4j
Parámetros de entrada	Ninguno
Consulta	MATCH (p:Patient) RETURN collect(p) AS pacientes
Descripción de la consulta	Recolección de todos los nodos etiquetados como Patient. Se devuelven en una colección.
Salida	Una promesa que se resuelve en un array de objetos de tipo Patient. Promise<{patients:Patient[]}>

Tabla 6 : Método getInfoPacientes

Método	**getInfoDispositivos**
Descripción	Este método se encarga de obtener la información de todos los dispositivos almacenados en la base de datos Neo4j
Parámetros de entrada	Ninguno

Consulta	MATCH (d:Device) RETURN collect(d) AS devices
Descripción de la consulta	Recolección de todos los nodos etiquetados como Device. Se devuelven en una colección.
Salida	Una promesa que se resuelve en un array de objetos de tipo Device. Promise<{devices: Device[]}>

Tabla 7 : Método getInfoDispositivos

Método	**getPacientePorId**
Descripción	Este método se encarga de obtener la información de un paciente específico almacenado en la base de datos Neo4j, utilizando su ID como parámetro de búsqueda.
Parámetros de entrada	(id: string) El ID del paciente que se desea obtener.
Consulta	MATCH (p:Patient {PatientID: $id}) RETURN p
Descripción de la consulta	Encuentra aquel nodo etiquetado como Patient cuyo PatientID coincida con el valor del parámetro id.
Salida	Un objeto que contiene la información del paciente si se encuentra, o null si no existe. Promise<Patient>

Tabla 8 : Método getPacientePorId

Método	**getDispositivoPacientePorId**
Descripción	Este método se encarga de obtener la información de los dispositivos asociados a un paciente específico almacenado en la base de datos Neo4j (recordemos que un paciente puede tener asociado más de un dispositivo), utilizando su ID (el del paciente) como parámetro de búsqueda.
Parámetros de entrada	(id: string) El ID del paciente del que se desea obtener los dispositivos.
Consulta	MATCH (p:Patient)-[:HAS_DEVICE]->(d:Device) WHERE p.PatientID = $id RETURN d
Descripción de la consulta	Encuentra los nodos etiquetados como Device que están asociados con un nodo Patient cuyo PatientID coincida con el valor del parámetro id.
Salida	Un objeto que es un array a su vez de objetos de tipo Device. Promise<{devices: Device[]}>

Tabla 9 : Método getDispositivoPacientePorId

Método	getObservacionesDispositivo
Descripción	Este método se encarga de obtener las observaciones asociadas a un dispositivo específico y a un paciente específico almacenados en la base de datos Neo4j.
Parámetros de entrada	(id_dispositivo: string, id_paciente: string) Los IDs del dispositivo y del paciente cuyas observaciones se desean obtener
Consulta	MATCH(d:Device {DeviceID: $id_dispositivo}) MATCH(o:Observation{ObservationDeviceId: $id_dispositivo,ObservationPatientId:$id_paciente}) WHERE (d)-[:HAS_ASSIGNED]->(o) RETURN collect(o) AS observaciones
Descripción de la consulta	Encuentra los nodos etiquetados como Observation que están asociados con un nodo Device cuyo DeviceID y ObservationPatientId coincidan con los valores de los parámetros id_dispositivo y id_paciente, respectivamente.
Salida	Un objeto que es un array de objetos Observation Promise<{ observaciones: Observation[] }>

Tabla 10 : Método getObservacionesDispositivo

Los métodos mostrados anteriormente de forma tabular son los que ofrecería el servicio al controlador para que este los consumiera. Todos ellos tienen en común dos aspectos:

1. Consumen el servicio definido para que el driver de Neo4j tuviera acceso a la base de datos. Si recordamos, el servicio de Neo4j define los métodos que permiten acceder a la base de datos mediante consultas mientras que los servicios definidos en app.service.ts se encargan de realizar las consultas y devolver una respuesta de las mismas. En otras palabras, es imposible que app.service.ts pueda realizar consultas a la base de datos y devolver los resultados de las mismas si no consume el servicio de Neo4j.
2. Todos hacen uso de las palabras reservadas await y async ya que estamos trabajando con operaciones asíncronas, es decir, la interacción con Neo4j a través de las distintas consultas pueden tardar un tiempo en completarse.
    - async: Se utiliza para definir una función asíncrona la cual siempre devolverá una promesa.
    - await: Se usa para pausar la ejecución de una función async hasta que la promesa que sigue se resuelva. Una vez resuelta, la ejecución de la función continúa con el valor resuelto de la promesa.

3.2.1.2 Controlador de la API REST – AppController

Los controladores son los responsables de manejar las solicitudes entrantes y devolver las respuestas al cliente.

El propósito de un controlador es recibir solicitudes específicas para la aplicación. El mecanismo de enrutamiento controla qué controlador recibe qué solicitudes (aunque en nuestro caso solo disponemos de un único controlador de manera que este mecanismo no está incorporado en la aplicación). Con frecuencia y como es nuestro caso, cada controlador tiene más de una ruta y cada una de ellas pueden realizar diferentes acciones

Para crear un controlador básico se utilizan decoradores, los cuales son funciones que se aplican a clases, métodos o propiedades y modifican su comportamiento [20].

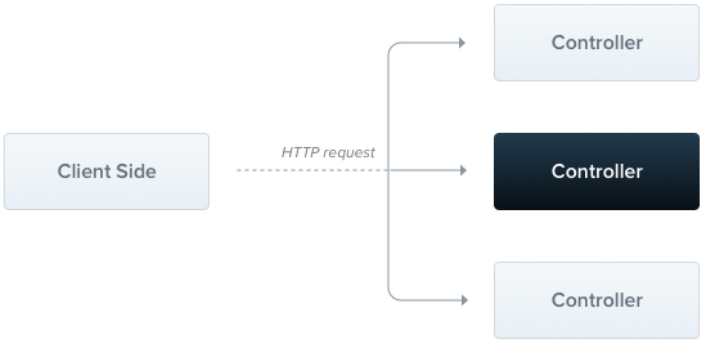

Ilustración 42 : Comunicación entre un cliente y un controlador específico mediante una petición HTTP

En el caso de nuestro único controlador, hemos hecho uso de tres decoradores importados desde el paquete @nestjs/common. Los decoradores en cuestión son los que aparecen en la siguiente tabla. Recordemos que otro ejemplo de decorador era @Injectable.

Decorador	Funcionalidad
@Controller	Define una clase como controlador
@Get	Define los métodos que manejarán solicitudes HTTP GET
@Param	Extraen parámetros de la ruta, parámetros de consulta y el cuerpo de la solicitud.

Tabla 11 : Decoradores usados en el desarrollo del proyecto

Ahora bien, ¿cómo consumimos los métodos el servicio AppService? Como hemos comentado antes, esto era posible gracias al decorador @Injectable, sin embargo, es necesario además importar en el controlador este servicio y además definir una instancia de ese servicio dentro del constructor del propio controlador como se refleja en la siguiente ilustración.

```
5  import { AppService } from './app.service';
6  @Controller()
7  export class AppController {
8    constructor(private readonly appService: AppService) {}
```

Ilustración 43 : Importación del servicio dentro del controlador

Una vez realizado esto, ya podemos consumir todos los recursos que implemente el servicio haciendo uso del objeto appService de tipo AppService.

A continuación, vamos a desglosar cada uno de los métodos definidos en el controlador que permiten consumir los métodos previamente definidos en el servicio, mostrándose tanto los decoradores asociados como los endpoints definidos.

Método	getPacientes

Descripción	Consume el método getInfoPacientes definido en el servicio AppService para obtener la información de todos los pacientes
Decoradores	@Get
Endpoint	GET /pacientes
Parámetros	Ninguno
Respuesta	Una promesa en la que los distintos campos, es decir los pacientes y sus respectivos datos son accedidos a través del objeto message Promise<{ message: any }>

Tabla 12 : Método getPacientes

Método	getPacientesById
Decripción	Consume el método getPacientePorId(id) definido en el servicio AppService para obtener la información de un paciente en concreto.
Decoradores	@Get @Param
Endpoint	GET /pacientes/:id
Parámetros	@Param('id') id: string, lo que hace el decorador es obtener el parámetro id que viaja en la URL, es decir, el parámetro :id que vemos en el endpoint y lo define con el nombre id. Este parámetro es el que se envía al método getPacientePorId para que se lleve a cabo la consulta a la base de datos.
Respuesta	Una promesa que contiene la información del paciente en concreto. Promise<any>

Tabla 13 : Método getPacientesById

Por ejemplo, si queremos acceder a la información del paciente con identificador 10, la URL que se consultará, es decir, el endpoint que se consumirá de la API REST será:

/pacientes/10

Veremos como el cliente consume los endpoints y en consecuencia este comportamiento en apartados posteriores de la memoria.

Método	getDevices
Descripción	Consume el método getInfoDispositivos definido en el servicio AppService para obtener la información de todos los dispositivos
Decoradores	@Get
Endpoint	GET /devices
Parámetros	Ninguno
Respuesta	Una promesa en la que los distintos campos, es decir los dispositivos y sus datos son

	accedidos a través del objeto message Promise<{ message: any }>

Tabla 14 : Método getDevices

Método	getDispositivosDePaciente
Descripción	Consume el método getDispositivoPacientePorId definido en el servicio AppService para obtener todos los dispositivos de un paciente en concreto.
Decoradores	@Get @Param
Endpoint	GET /pacientes/:id/dispositivos
Parámetros	@Param('id') id: string. Lo que hace el decorador es obtener el parámetro id que viaja en la URL, es decir, el parámetro :id que vemos en el endpoint y lo define con el nombre id. Este parámetro es el que se envía al método getDispositivoPacienteId para que se lleve a cabo la consulta a la base de datos.
Respuesta	Una promesa que contiene un objeto de tipo Array de dispositivos en el que encontraremos listados todos los dispositivos del paciente en concreto. Promise<{devices: Dispositivo[]}>

Tabla 15 : Método getDispositivoDePaciente

Por ejemplo, si queremos acceder a los dispositivos del paciente con identificador 10, la URL que se consultará, es decir, el endpoint que se consumirá de la API REST será:

/pacientes/10/dispositivos

Si en lugar de tener identificador 10 el paciente tuviera un identificador con valor 20 la URL a la que se hace la petición será

/pacientes/20/dispositivos

Veremos como el cliente consume los endpoints y en consecuencia este comportamiento en apartados posteriores de la memoria.

Método	getObservacionesDispositivos
Descripción	Consume el método getObservacionesDispositivo definido en el servicio AppService para obtener todas las observaciones de un dispositivo concreto de un paciente en concreto.
Decorador	@Get @Param
Endpoint	GET /pacientes/:id_paciente/dispositivos/:id_dispositivo/observaciones
Parámetros	@Param('id_paciente') id_paciente: string, @Param('id_dispositivo') id_dispositivo: string. Lo que hacen estos decoradores es obtener los parámetro id_paciente e id_dispositivo que viaja en la URL, es decir, el parámetro :id_paciente y el parámetro id_dispositivo que

	vemos en el endpoint y los define con los nombres id_paciente, id_dispositivo respectivamente. Estos parámetros son los que se envían al método getObservacionesDispositivo para que se lleve a cabo la consulta a la base de datos.
Respuesta	Una promesa que contendrá un objeto de tipo Array de Observación, en el cual están listadas todas las observaciones del dispositivo del paciente concreto. Promise<{ observaciones: Observacion[] }>

Tabla 16 : Método getObservacionesDispositivos

Por ejemplo, si queremos acceder a las observaciones del dispositivo con identificador 6 del paciente con identificador 10, la URL que se consultará, es decir, el endpoint que se consumirá de la API REST será:

/pacientes/10/dispositivos/6/observaciones

Si en lugar de esto el identificador del dispositivo fuera el 1 pues se haría una petición a la URL

/pacientes/10/dispositivos/1/observaciones

Veremos como el cliente consume los endpoints y en consecuencia este comportamiento en apartados posteriores de la memoria.

3.3 Desarrollo del frontend – Angular

Tras haber abordado el desarrollo de la base de datos y del backend, podemos dar paso a la parte del cliente. Como ya sabemos, nuestro frontend ha sido desarrollado a partir del framework de JavaScript Angular. La instalación de este framework se llevó a cabo desde la consola de Windows, aunque nuestro IDE VSCode a través de su propia terminal también permite realizar esta tarea.

Los pasos realizados para esta crear el proyecto Angular fueron los siguientes:

1. Instalación previa de Node.js, npm (Node package manager).
2. Instalación de Angular CLI a través de la consola de Windows haciendo uso del comando:

   npm i -g @angular/cli. Esta herramienta es fundamental para poder usar las distintas herramientas que ofrece Angular para la creación de aplicaciones.
3. Creación del proyecto, esta vez desde la terminal de Visual Studio Code. Esta tarea es muy sencilla y simplemente debemos introducir el comando:

   ng new nombre_proyecto

   en nuestro caso hemos definido este proyecto con el nombre de “frontend”. Posteriormente debemos indicar la hoja de estilos que usará el proyecto, en nuestro caso seleccionaremos CSS.

Tras haber realizado esos pasos, tendremos nuestro proyecto Angular creado, y para iniciarlo simplemente introducimos el comando: ng serve -o. Por defecto, los proyectos en Angular arrancan en localhost:4200, esta opción puede modificarse desde el fichero angular.json. Sin embargo, nosotros hemos decidido mantener esta configuración ya que no se solapaba con ningún otro puerto. Si accedemos desde el navegador a esta URL, el proyecto Angular de nueva creación tiene el siguiente aspecto, dándonos la bienvenida de una forma bastante visual. A continuación, se muestra un ejemplo de cómo ser vería un proyecto Angular tras ser creado.

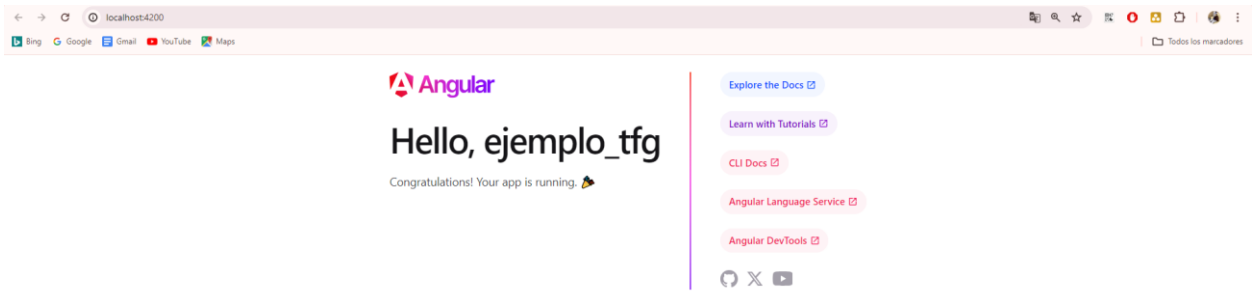

Ilustración 44 : Ejemplo de proyecto Angular de nueva creación

Es en este punto donde podemos dar paso a las tareas realizadas para el desarrollo del frontend. Estas son:

1. Creación, desarrollo e integración de los distintos componentes añadiendo Bootstrap 5 a los estilos de estos.
2. Creación desarrollo e integración de los distintos servicios que permitirán a los componentes ejecutar sus métodos para realizar las peticiones a la parte del servidor
3. Definición de los distintos módulos usados para que se obtenga el resultado esperado, entre ellos el módulo de *routing* que permite definir las rutas que la aplicación debe seguir y decidir que componentes se deben cargar en estas, recordemos que Angular sigue el principio SPA.
4. Elaboración de las distintas solicitudes HTTP al backend teniendo en cuenta que estas operaciones son asíncronas, es decir, realizar un manejo de estados.

Abordaremos a continuación cada uno de estos aspectos detalladamente.

3.3.1 Servicios del frontend

El trabajo de un servicio es el de controlar la información, desde obtenerla, almacenarla, actualizarla y compartirla con los componentes. No hay nada especial acerca de un Servicio en Angular, excepto que estos deben de integrarse con los componentes vía el inyector de Dependencias de Angular tal y como sucedía con los controladores de Nest. En nuestro caso hemos creado tres servicios, cada uno de los cuales estará destinado a manejar las peticiones relacionadas con la obtención de la información de cada uno de los tipos de nodos que reside en la base de datos, es decir: Observation, Devices y Patients. En consecuencia, nuestros servicios son los siguientes: ObservacionService, DeviceService y PatientService. Cada uno de ellos realiza peticiones HTTP GET a los distintos endpoints definidos en nuestro backend, esta tarea ha sido posible principalmente gracias a las siguientes herramientas:

- El servicio HttpClient del módulo de Angular HttpClientModule, que permite hacer solicitudes HTTP a un servidor backend.
- La biblioteca RxJS (Reactive Extensions for JavaScript) concretamente a su clase Observable. La cual permite la creación de objetos Observables siendo el método subscribe() de gran interés para el desarrollo de nuestro proyecto. Este método subscribe() toma como parámetro un observador y nos devuelve una suscripción. El observador se suscribe al observable y cada vez que el observable emite un valor, el observador es notificado. De esta manera es como obtenemos los datos que previamente el backend ha resuelto de la consulta a Neo4j.

A continuación, se exponen los métodos que componen cada uno de los servicios que conforman el proyecto:

Métodos del servicio PatientService

Método	obtenerInformacionHome
Descripción	Realiza una petición GET con el objetivo de obtener a todos los pacientes almacenados en la base de datos.
Parámetros de entrada	Ninguno
URL a la que se hace la petición	http://localhost:3000/pacientes
Respuesta	Un observable preparado para recibir datos con el siguiente formato Observable<{ message: { pacientes: Patient[] } }>

Tabla 17 : Método obtenerInformacionHome

Método	obtenerDetallePaciente
Descripción	Realiza una petición GET con el objetivo de obtener los detalles de un paciente concreto.
Parámetros de entrada	id:string, el cual consiste en el identificador del paciente del cual deseamos obtener su información en detalle. Esta acción puede ser realizada gracias al RoutingModule. Veremos cómo cuando hablemos de este.
URL a la que se hace la petición	http://localhost:3000/pacientes/${id} siendo ${id} el parámetro de entrada del método.
Respuesta	Un objeto Observable preparado para recibir a un Paciente concreto. Observable<Patient>

Tabla 18 : Método obtenerDetallePaciente

Métodos del servicio DeviceService

Método	obtenerInformacionDispositivos
Descripción	Realiza una petición GET con el objetivo de obtener todos los dispositivos almacenados en la base de datos.
Parámetros de entrada	Ninguno
URL a la que se hace la petición	http://localhost:3000/devices
Respuesta	Un objeto Observable preparado para recibir datos con el siguiente formato Observable<{ message: { devices: Device[] } }>

Tabla 19 : Método obtenerInformacionDispositivos

Método	obtenerInformacionDispositivosPaciente
Descripción	Realiza una petición GET con el objetivo de obtener la información de todos los dispositivos asociados a un paciente concreto
Parámetros de entrada	id:string, el cual consiste en el identificador del paciente del cual deseamos obtener la información de los dispositivos que tiene asociados (no las observaciones). Esta acción puede ser realizada gracias al RoutingModule. Veremos cómo cuando hablemos de este.
URL a la que se hace la petición	localhost:3000/pacientes/${id}/dispositivos siendo ${id} el parámetro de entrada del método.
Respuesta	Un objeto Observable preparado para recibir datos con el siguiente formato: Observable<{devices: Device[]}>

Tabla 20 : Método obtenerInformacionDispositivosPaciente

Métodos – Servicio ObservaciónService

Método	obtenerInformacionObservaciones
Descripción	Realizar una petición GET con el objetivo de obtener las observaciones de un dispositivo y paciente concretos.
Parámetros de entrada	id_paciente_string, id_dispositivo:string
URL a la que se hace la petición	localhost:3000/pacientes/${id_paciente}/ dispositivos/${id_dispositivo}/observaciones. Los cuales consisten en el identificador del paciente y dispositivo respectivamente del cual deseamos obtener sus observaciones. Esta acción puede ser realizada gracias al RoutingModule. Veremos cómo cuando hablemos de este.
Salida	Un objeto Observable preparado para recibir datos con el siguiente formato: Observable<{ observaciones: Observation[] }>

Tabla 21 : Método obtenerInformacionObservaciones

3.3.2 Componentes del frontend

En el apartado 2.3.3 de la memoria ya definimos el concepto de Componente, recordemos que estos podemos crearlos o bien manualmente o bien de una manera más sencilla mediante el comando proporcionado por Angular CLI: ng generate component nombre-componente.

Si además buscamos aplicarle a estos componentes los estilos proporcionados por Bootstrap, entonces debemos

añadir a nuestro archivo de config.json las siguientes líneas en las secciones styles y script respectivamente.

```
"node_modules/bootstrap/dist/css/bootstrap.min.css"
"node_modules/bootstrap/dist/js/bootstrap.min.js"
```

En nuestro caso, para el desarrollo de nuestro frontend hemos generado los siguientes componentes

- HomeComponentComponent
- DetallePacienteComponent
- DispositivosComponent
- DispositivosPacienteComponent
- ObservacionDispositivoComponent

Además, si recordamos, Angular por defecto tiene el componente raíz AppComponent. A continuación, realizaremos una breve descripción de cada uno de los componentes por los que se compone nuestra aplicación, como ya sabemos, cada uno de estos componentes está conformado por:

- Plantilla HTML o Template
- Estilos CSS
- Lógica TypeScript

De manera que para entender mejor la explicación de estos componentes nos ayudaremos de las distintas vistas por las que se compone nuestra aplicación, indicándose las funcionalidades que podemos realizar en cada una de ellas.

HomeComponentComponent

En la lógica TypeScript de este componente definimos un objeto Paciente[] que será un array de Pacientes, para ello previamente debe ser importada la interfaz Paciente.ts la cual está definida con los mismos atributos con los que los nodos Patient de Neo4j cuentan. Este componente también importa el servicio PatientService con el objetivo de consumir los métodos que sean necesarios de este. Para realizar esta tarea definimos dentro del constructor del componente HomeComponentComponent una instancia del servicio. Consecuentemente, esta instancia tendrá accesibles todos los métodos de su clase, es decir, del servicio PatientService.

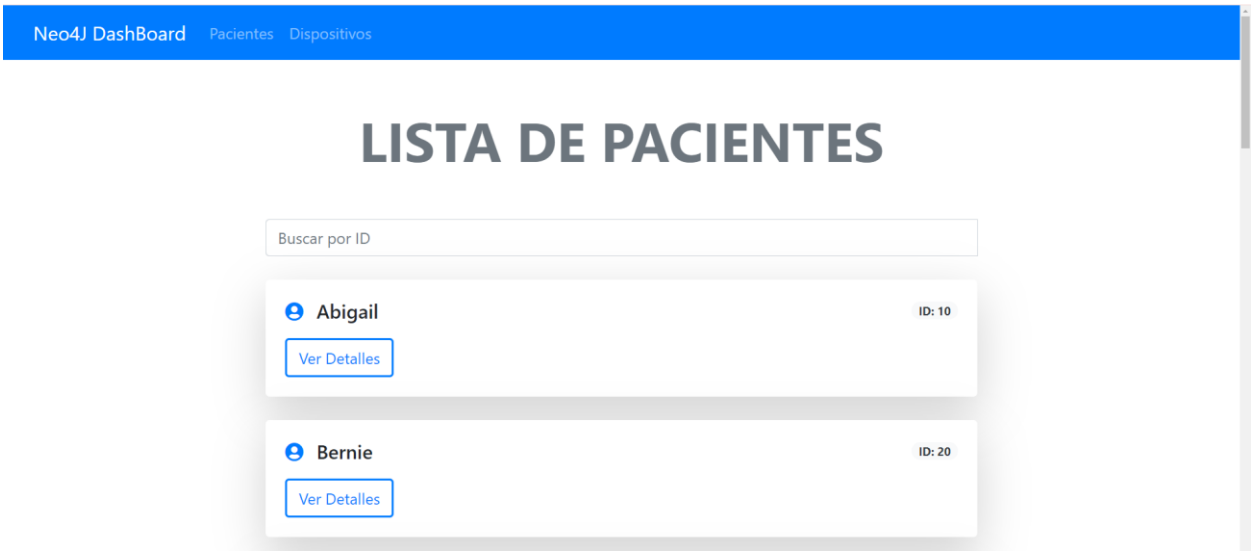

Ilustración 45 : Vista principal de la aplicación – HomeComponentComponent

Tras arrancar nuestro proyecto (base de datos, backend y frontend) la vista principal del mismo será la que se muestra en la ilustración 45, comentamos los siguientes aspectos de esta sección:

- La barra de navegación permite viajar a distintas secciones de la aplicación apoyándose en el RoutingModule así como en las respuestas que reciba del backend
- En esta sección también encontramos un buscador para filtrar a los pacientes por su ID, este será muy útil en caso de que queramos buscar a un paciente específico. Por ejemplo, si queremos filtrar a los pacientes cuyo ID contenga el valor "10" este sería el resultado:

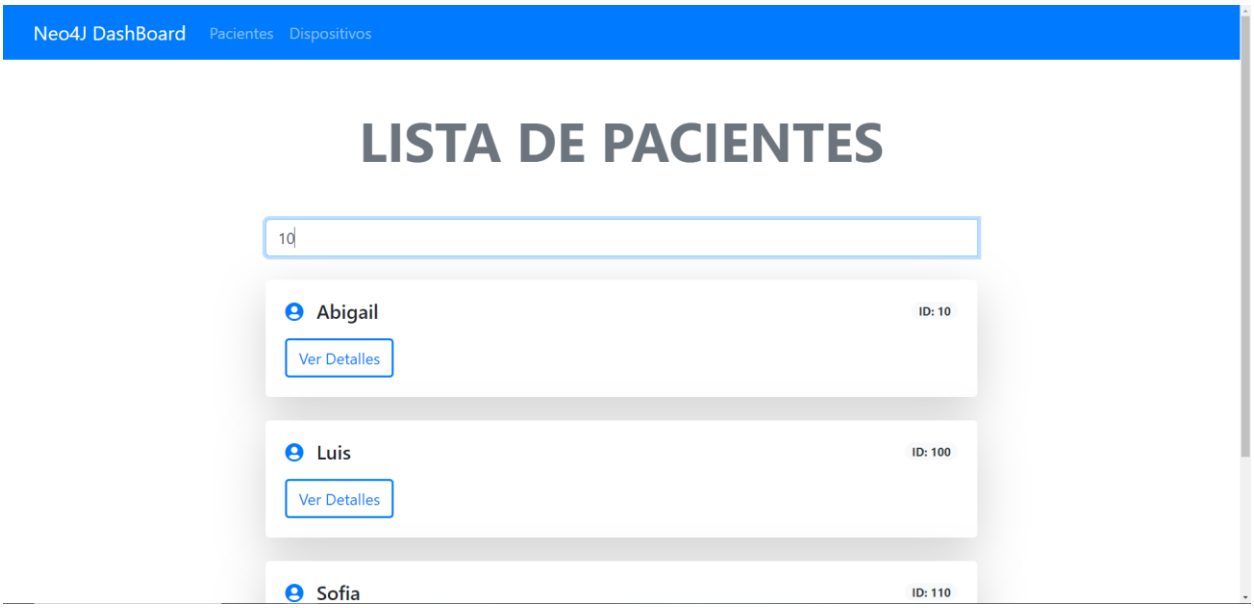

Ilustración 46 : Filtrado de pacientes por ID con secuencia "10"

- Con la ayuda de Bootstrap, hemos desarrollado las tarjetas que representan a cada uno de los pacientes. Cada una cuenta con un botón para mostrar los detalles del paciente en concreto (Ver Detalles). La funcionalidad de este consiste en lanzarnos a otra vista de la aplicación concretamente al componente DetallePacienteComponent. Para poder identificar a cada uno de los pacientes correctamente nos ayudamos tanto del ID como del nombre ya que pueden coexistir en la base de datos dos pacientes con el mismo nombre, pero no con el mismo ID.

Además, haciendo uso de las herramientas de desarrollador podemos comprobar que cada vez que accedemos a este componente, el navegador realiza una petición a la URL http://localhost:3000/pacientes del backend.

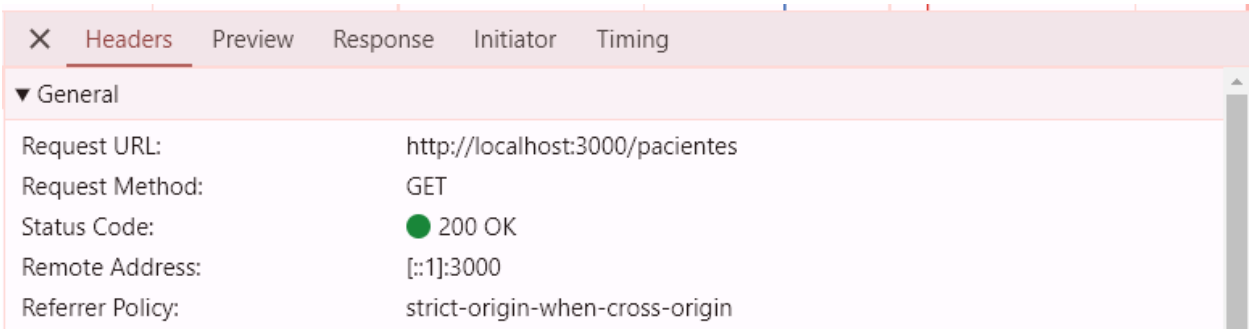

Ilustración 47 : Petición al backend desde HomeComponentComponent

DispositivosComponent

En la lógica TypeScript de este componente definimos un objeto Dispositivo[] que será un array de Dispositivos, para ello previamente debe ser importada la interfaz Dispositivo.ts, la cual está definida con los mismos atributos con los que los nodos Device de Neo4j cuentan. Este componente también importa el servicio DeviceService con el objetivo de consumir los métodos que sean necesarios de este. Para realizar esta tarea definimos dentro

del constructor del componente DispositivosComponent una instancia del servicio. Consecuentemente, esta instancia tendrá accesibles todos los métodos de su clase, es decir, del servicio DeviceService.

Si accedemos a este componente a través de la barra de navegación con la que cuenta la aplicación, aparecerán en tarjetas, al igual que con los pacientes, la lista de dispositivos registrados en la base de datos, esta vez sin la opción de que se muestren los detalles ya que estos aparecen todos en la propia tarjeta como podemos observar en la siguiente ilustración.

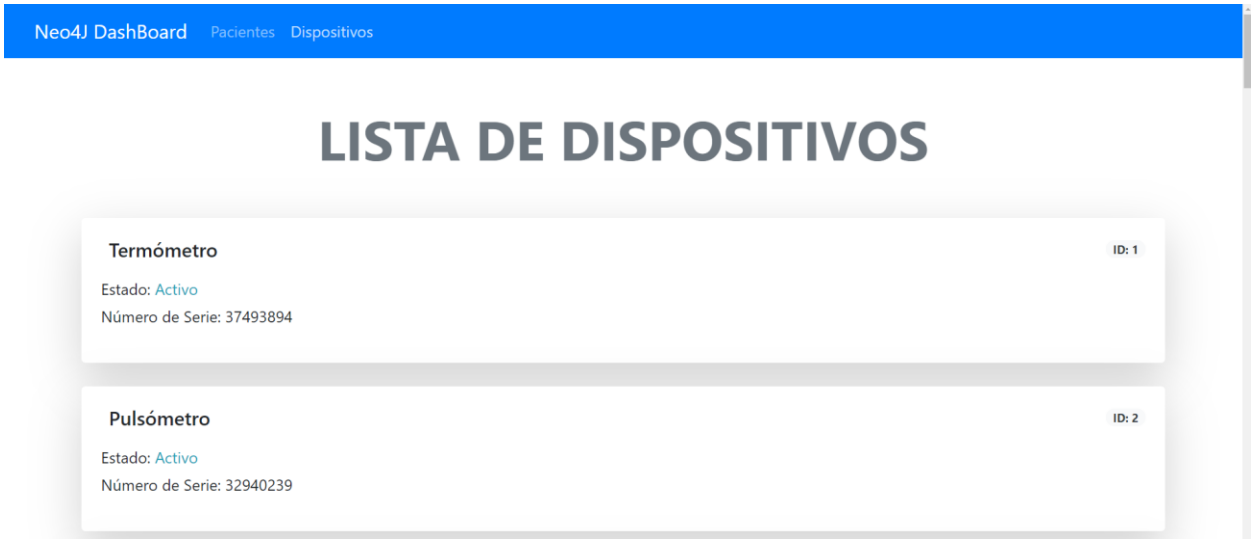

Ilustración 48 : Vista del componente DispositivosComponent

Al igual que ocurría con los pacientes, podemos tener dos dispositivos del mismo tipo en la base de datos sin ningún problema, podremos diferenciarlos gracias a su identificador, el cual puede reflejado en la esquina superior derecha del dispositivo en cuestión. En este caso la URL a la que el navegador hace la petición es http://localhost:3000/devices como puede observarse en la ilustración 49.

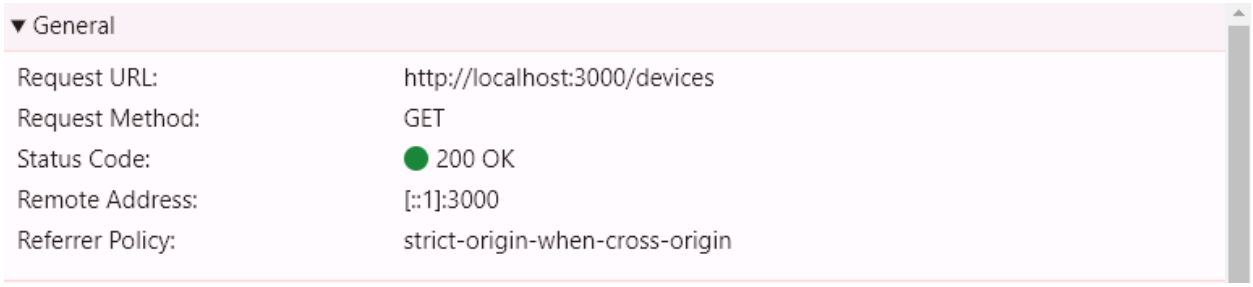

Ilustración 49 : Petición al backend desde DispositivosComponent

DetallePacienteComponent

En la lógica TypeScript de este componente definimos un objeto Paciente, para ello previamente debe ser importada la interfaz Paciente.ts, la cual está definida con los mismos atributos con los que los nodos Patient de Neo4j cuentan. Además, también hemos definido una variable id de tipo string para asignarle valores de interés que viajen en la URL y posteriormente pasarla por parámetro al método que corresponda. Este componente también importa tanto el servicio ActivatedRoute (gracias al cual accederemos a la información de las rutas definidas con RouterModule) como el PatientService, con el objetivo de consumir los métodos que sean necesarios de este. Para realizar esta tarea definimos dentro del constructor del componente DetallePacienteComponent una instancia de sendos servicios. Consecuentemente, podremos trabajar cómodamente con estas instancias haciendo uso de sus métodos y funcionalidades.

Si nos centramos en un paciente en concreto por ejemplo, Abigail con ID = “10” la aplicación nos mostrará el siguiente resultado:

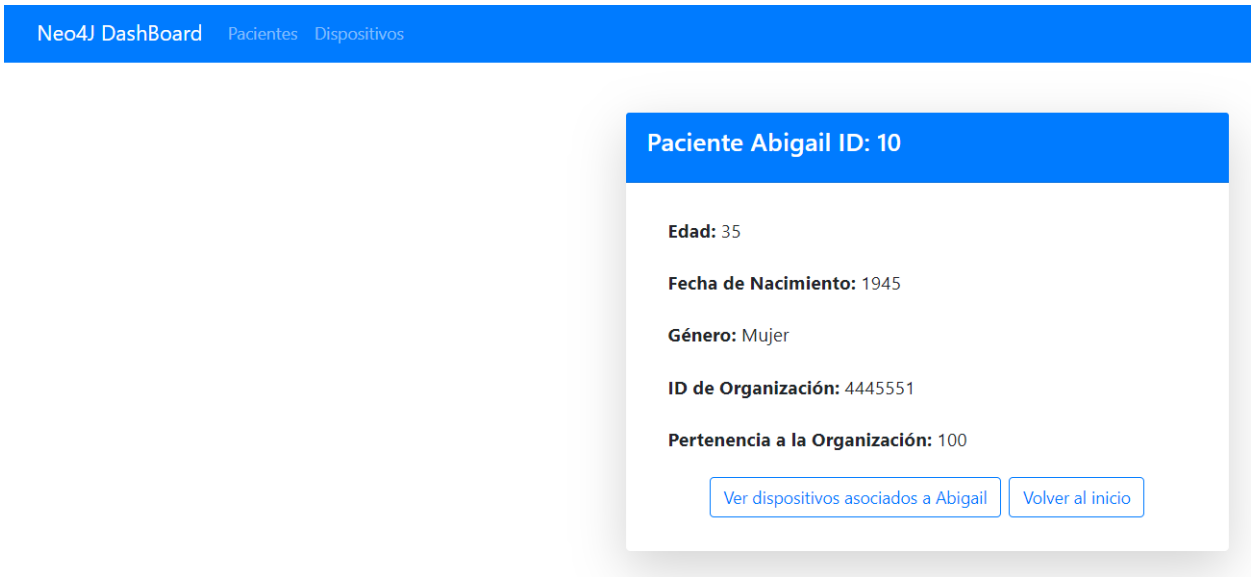

Ilustración 50 : Vista del detalle de un paciente concreto - componente DispositivosComponent

En este caso la petición se ha realizado a la siguiente URL del backend http:localhost:3000/pacientes/10, siendo 10 el identificador de la paciente Abigail

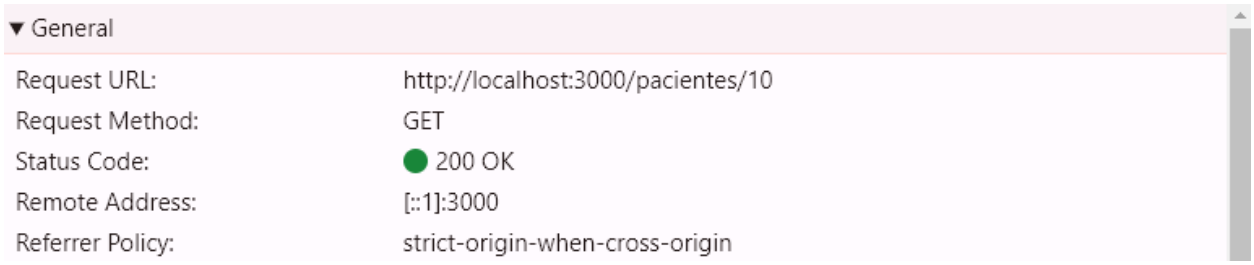

Ilustración 51: Petición al backend desde DetallePacienteComponent

Si en lugar de acceder al detalle de Abigail hubiéramos seleccionado la opción de Ver Detalles del paciente con ID = “110” esta sería la URL a la que se consultaría desde el navegador.

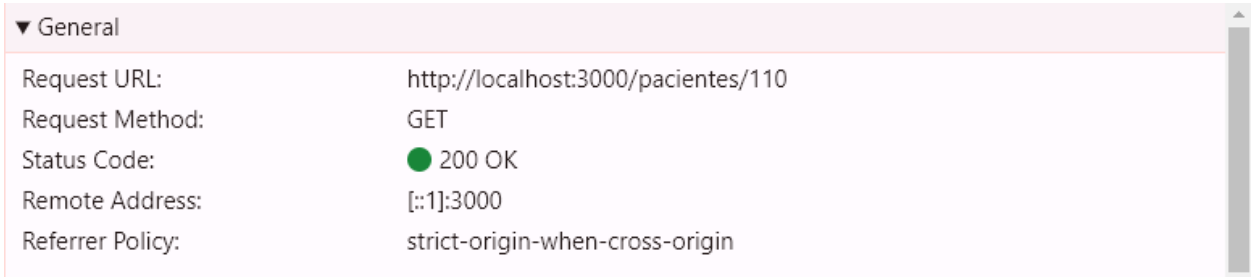

Ilustración 52 : Ejemplo de URL consultada al solicitar los detalles de un paciente con otro ID

Este comportamiento ya fue mencionado en apartados anteriores de la memoria. Por otro lado, en esta vista podemos volver al listado de pacientes o bien seleccionando el botón Volver al Inicio o bien desde la barra superior de navegación. También tenemos la opción de consultar los dispositivos asociados al paciente, en ese caso debemos pulsar el botón Ver dispositivos asociados a nombre_paciente, en este caso Abigail.

DispositivosPacienteComponent

En la lógica TypeScript de este componente definimos un objeto Dispositivo[] que será un array de Dispositivos, para ello previamente debe ser importada la interfaz Dispositivo.ts, la cual está definida con los mismos atributos con los que los nodos Device de Neo4j cuentan. Además, también hemos definido una variable id_paciente de tipo string para asignarle valores de interés que viajen en la URL y posteriormente pasar esta variable por

parámetro al método que corresponda. Este componente también importa tanto el servicio ActivatedRoute (gracias al cual accederemos a la información de las rutas definidas con RouterModule) como el servicio DeviceService con el objetivo de consumir los métodos que sean necesarios de este. Para realizar esta tarea definimos dentro del constructor del componente DispositivosPacienteComponent una instancia de sendos servicios. Consecuentemente, estas instancias tendrán acceso a todos los métodos de sus respectivas clases.

Si seguimos con el ejemplo de Abigail, una vez pulsado el botón Ver dispositivos asociados a Abigail se nos redireccionará al componente DispositivosPacienteComponent. En la siguiente ilustración observamos como se vería este componente.

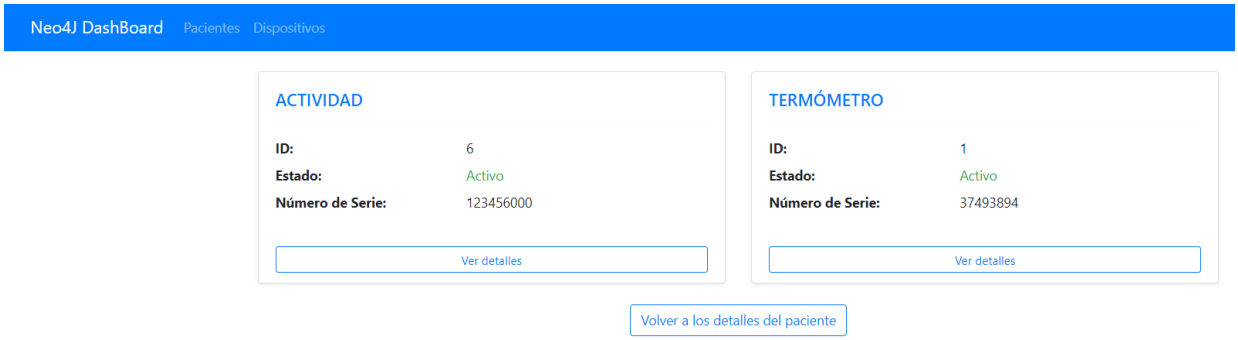

Ilustración 53 : Vista de los dispositivos asociados a un paciente concreto - componente DispositivosPacienteComponent

En este caso la paciente Abigail tiene asociados 2 dispositivos diferentes, un termómetro y un monitor de actividad, comprobándose así que un paciente puede tener asociado más de un dispositivos. Además, cada una de las tarjetas muestra los atributos de cada uno de los dispositivos y nos dan la opción de Ver detalles, es decir, consultar las observaciones registradas en cada uno de ellos. Si quisiéramos volver a los detalles de Abigail, basta con pulsar el botón Volver a los detalles del paciente.

La URL a la que se realiza la petición en este caso es http://localhost/pacientes/10/dispositivos como podemos ver en la siguiente ilustración. De nuevo el valor 10 que viaja en la URL es el valor del ID de Abigail.

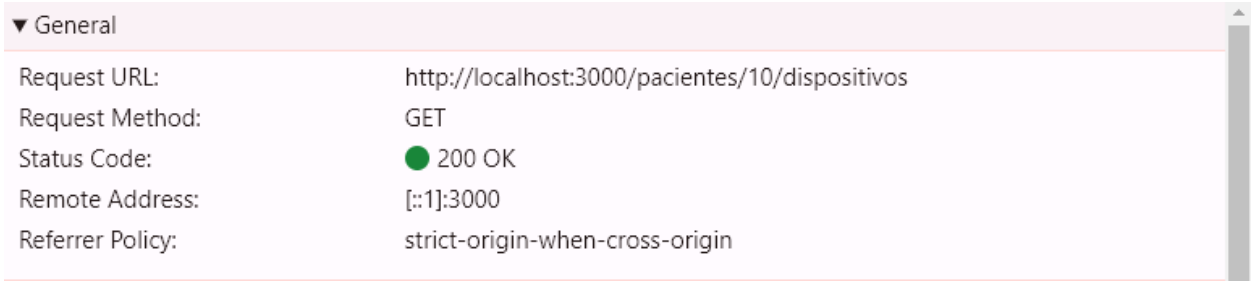

Ilustración 54 : Petición al backend desde DispositivosPacienteComponent

Si hubiéramos accedido a los dispositivos asociados con otro paciente el valor de ese parámetro en la URL sería distinto.

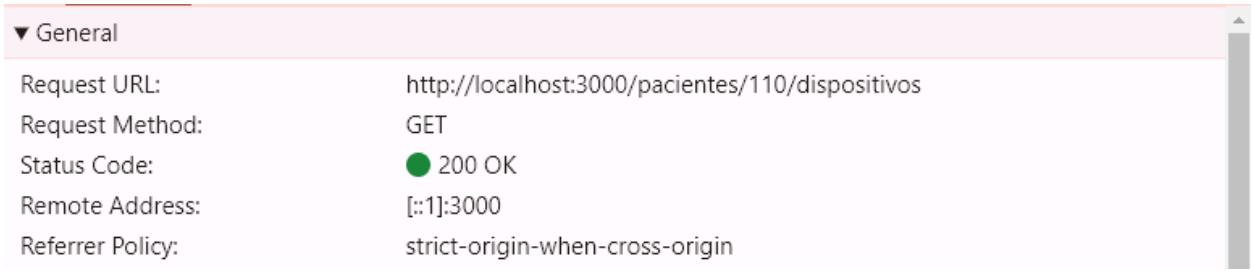

Ilustración 55 : Ejemplo de URL consultada al solicitar los dispositivos de un paciente con otro ID

ObservacionDispositivoComponent

En la lógica TypeScript de este componente, ObservacionDispositivoComponent, definimos varias propiedades y métodos para gestionar y visualizar las observaciones de un dispositivo específico asociado a un paciente. Para ello, previamente importamos las interfaces y servicios necesarios.

Definimos un objeto Observacion[] que será un array de observaciones, para ello previamente debe ser importada la interfaz Observacion.ts, la cual está definida con los mismos atributos con los que los nodos de observación de Neo4j cuentan. Además, hemos definido una variable id_paciente de tipo string para asignarle valores de interés que viajan en la URL y posteriormente pasar esta variable por parámetro al método que corresponda. También definimos la variable idDispositivoConcreto de tipo string para manejar el ID del dispositivo específico.

Este componente importa tanto el servicio ActivatedRoute (gracias al cual accederemos a la información de las rutas definidas con RouterModule) como los servicios ObservacionService y DeviceService con el objetivo de consumir los métodos que sean necesarios de estos. Para realizar esta tarea, definimos dentro del constructor del componente ObservacionDispositivoComponent una instancia de cada uno de estos servicios. Consecuentemente, estas instancias tendrán acceso a todos los métodos de sus respectivas clases.

Además, utilizamos ViewChild para obtener una referencia al elemento del gráfico en el DOM y la biblioteca Chart.js para la visualización de datos.

En resumen, este componente realiza las siguientes funciones:

- Inicializa las propiedades necesarias para manejar las observaciones y la información del dispositivo.
- Obtiene los parámetros de la URL mediante ActivatedRoute para capturar id_paciente e id_dispositivo.
- Llama a los métodos de servicios ObservacionService y DeviceService para obtener las observaciones y la información del dispositivo.
- Crea un gráfico interactivo utilizando Chart.js para visualizar los datos de las observaciones del dispositivo.

Concretamente la implementación de las gráficas se ha llevado a cabo realizando los siguientes pasos:

1. **Importación de Chart.js**. En el componente ObservacionDispositivoComponent usamos la biblioteca Chart.js para crear gráficos interactivos. Primero, importamos los módulos necesarios y registramos los componentes para poder utilizarlos en el componente.
2. **Referencia al elemento del gráfico.** Utilizamos el decorador @ViewChild para obtener una referencia al elemento canvas en el templete o HTML. Esto nos permite acceder directamente al contexto del canvas y manipularlo para crear el gráfico.
3. **Método crearGrafico**. Aquí es donde configuramos y renderizamos el gráfico con los datos de las observaciones del dispositivo. Los pasos clave son:
    - **Obtener el contexto del Canvas**: Conseguimos el contexto 2D del elemento canvas referenciado. Este contexto es necesario para dibujar y manipular el gráfico.
    - **Extracción de datos**: Sacamos las fechas y los valores de las observaciones del dispositivo. Las fechas se utilizan como etiquetas en el eje X, y los valores como puntos de datos en el eje Y. Esto nos permite mostrar cómo evolucionan los valores de observación a lo largo del tiempo.
    - **Destruir el gráfico existente**: Si ya existe un gráfico previamente renderizado, lo destruimos para evitar superposiciones o duplicados. Así, cada vez que hay nuevos datos o actualizaciones, el gráfico se renderiza limpio y sin errores.
    - **Configuración y creación del nuevo gráfico**: Configuramos y creamos el nuevo gráfico utilizando Chart.js. Esto incluye:
        1. **Tipo de gráfico**: Especificamos que el gráfico es de tipo línea.
        2. **Datos del gráfico**: Incluimos las etiquetas y los conjuntos de datos (fechas y valores de observaciones), definiendo qué datos se van a representar y cómo.

3. **Configuración del conjunto de datos**: Ajustamos aspectos visuales como el color de la línea, el color de fondo del área debajo de la línea, si esta área debe rellenarse, la suavidad de la curva de la línea y el ancho de la línea. Estos ajustes mejoran la presentación visual y la legibilidad del gráfico.
4. **Opciones del gráfico**: Ajustamos las opciones del gráfico para hacerlo responsivo y permitir que se ajuste al tamaño del contenedor. También configuramos los ejes X e Y con títulos descriptivos, lo que facilita la interpretación de los datos.

Concretamente las gráficas de las observaciones de los distintos dispositivos de la Paciente Abigail tendrían el siguiente aspecto:

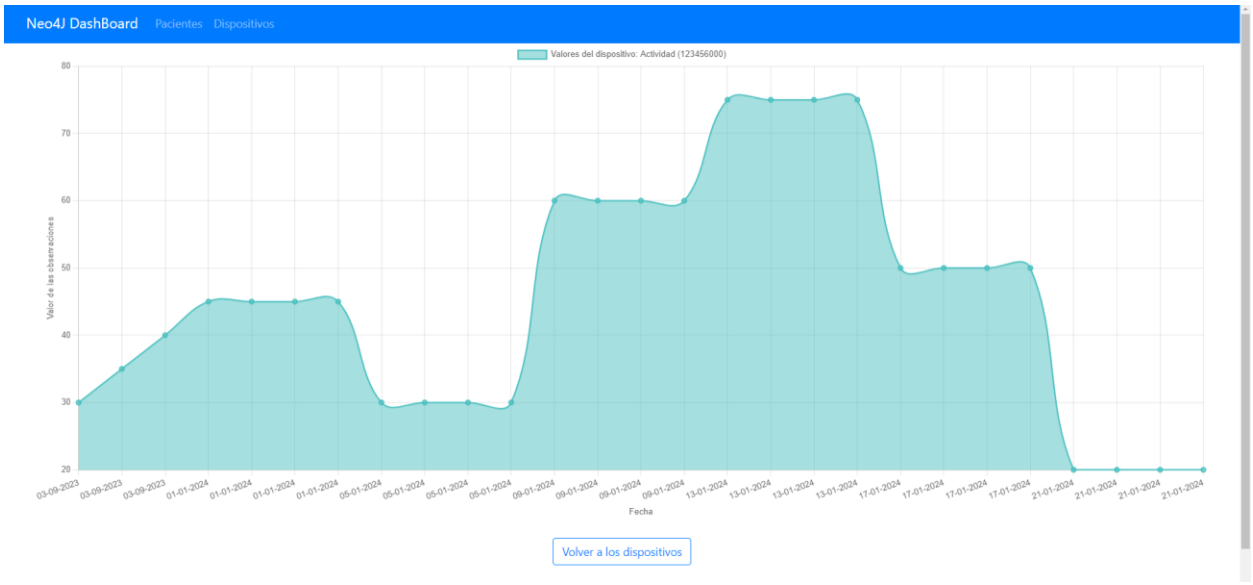

Ilustración 56 : Gráfica de las observaciones del dispositivo de Actividad de Abigail

En ella se pueden ver los distintos valores de las observaciones relacionadas con este dispositivo (eje Y) en el intervalo temporal (eje X).

La URL al backend a la que se realiza la petición para obtener estas observaciones y poder plasmarlas gráficamente es http://localhost:3000/pacientes/10/dispositivos/6/observaciones como podemos ver en la siguiente ilustración.

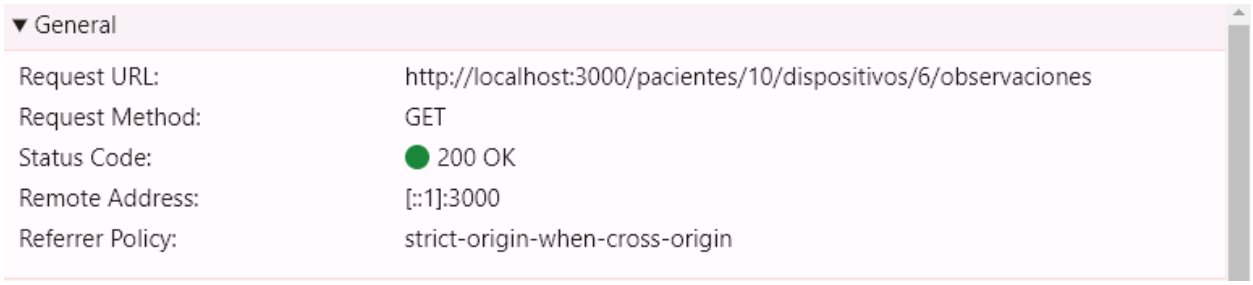

Ilustración 57 : Petición al backend desde ObservacionDispositivoComponent

Los valores 10 y 6 que aparecen en la URL se corresponden con los identificadores del paciente y del dispositivo respectivamente.

La gráfica del dispositivo del termómetro asociado a Abigail sería la siguiente.

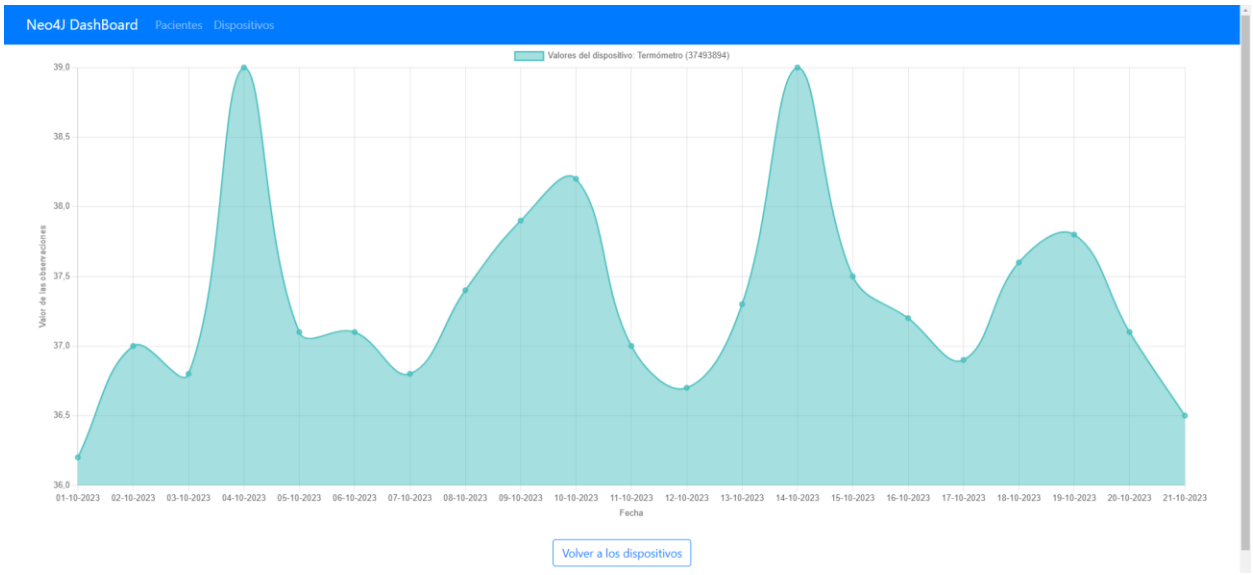

Ilustración 58 : Gráfica de las observaciones del dispositivo de Temperatura de Abigail

De la misma manera se pueden ver los distintos valores de las observaciones relacionadas con este dispositivo (eje Y) en el intervalo temporal (eje X).

La URL a la que se ha hecho la petición es http://localhost:3000/pacientes/10/dispositivos/1/observaiones.

Podemos comprobar que en este caso el identificador del dispositivo es 1.

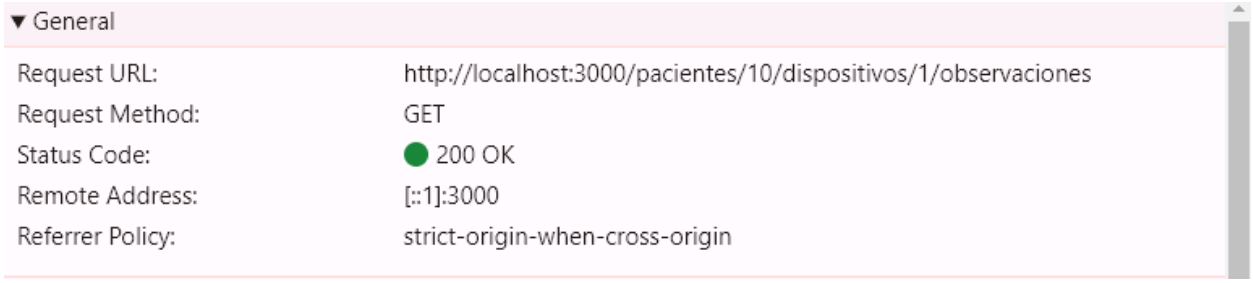

Ilustración 59 : Petición al backend desde ObservacionDispositivoComponent

4 CONCLUSIONES Y LÍNEAS FUTURAS

Cada solución da pie a una nueva pregunta.

- David Hume -

En este proyecto se ha abordado el diseño, desarrollo e implementación de un sistema que simula la monitorización de las observaciones de dispositivos asociados a una serie de pacientes, manteniéndose estos datos almacenados en una base de datos basada en grafos, concretamente utilizando Neo4j. Entre las distintas tecnologías que encontramos disponibles finalmente se han seleccionado aquellas que permitían una integración eficiente no solo entre sí, sino que también con Neo4j.

4.1 Conclusiones

A continuación, destacamos los principales logros alcanzados tras completar el desarrollo de la aplicación.

1. Exploración y compresión de Neo4J: Se ha llevado a cabo un estudio detallado de las bases de datos basadas en grafos y su implementación en Neo4J. Esto incluyó el aprendizaje del lenguaje Cypher para manejar nodos y relaciones, permitiendo una correcta gestión de los datos de monitorización de pacientes.
2. Desarrollo de backend y frontend con tecnologías desconocidas previamente: Gracias a ello no solo hemos conseguido profundizar y reforzar los conocimientos de programación web que teníamos sino que también hemos aprendido a manejar nuevas y populares tecnologías como son Nest y Angular.
3. Creación de una API REST: La cual ha permitido la conexión fluida entre el frontend y el backend, brindando versatilidad y mejorando la capacidad de respuesta del sistema. El desarrollo de esta API ha sido esencial no solo para la visualización y análisis de los datos de monitorización, sino también para definir correctamente los endpoints que conforman la aplicación.
4. Visualización gráfica de los datos: Gracias al uso de la librería Chart.js la cual ha facilitado la creación de los gráficos que representan las observaciones de los distintos dispositivos permitiendo representar estos datos de manera clara y comprensible

Por otro lado, debemos tener en cuenta que el proyecto también cuenta con aspectos negativos, pues las tecnologías utilizadas, especialmente Neo4J y las librerías de frontend, están en constante evolución. Esto implica que las implementaciones actuales podrían requerir actualizaciones frecuentes para mantenerse al día con las nuevas versiones y mejoras. También es importante mencionar que, aunque las bases de datos orientadas a grafos ofrecen ventajas significativas, también presentan una curva de aprendizaje pronunciada y una mayor complejidad en la configuración inicial y en el mantenimiento de las relaciones de datos.

4.2 Futuras líneas de desarrollo

Al finalizar el proyecto hemos logrado alcanzar el punto en el que el objetivo propuesto al principio ha sido alcanzado: diseñar y desarrollar una solución tecnológica que almacene y procese datos de manera efectiva haciendo uso de una base de datos basada en grafos (en la implementación concreta de Neo4J), ofreciendo herramientas para la visualización y análisis de las constantes vitales de los pacientes. Lo cual incluye:

- Exploración y comprensión de las bases de datos basadas en grafos como Neo4J.
- Aplicación de los conocimientos adquiridos durante el grado para el desarrollo de una solución a un problema.
- Diseño, desarrollo, implementación y pruebas de un servicio de backend que se encargue de conectarse y obtener los datos almacenados en la base de datos de manera coherente.
- Diseño, desarrollo, implementación y pruebas de una API alojada en el backend, la cual será consumida por el frontend para así obtener los datos a modo de respuesta del servidor de manera coherente.
- Diseño, desarrollo, implementación y pruebas de una interfaz gráfica para el usuario (GUI) que muestre los datos obtenidos de manera visual.

No obstante, a continuación, se muestran las siguientes líneas futuras a tener en cuenta en caso de que se desee continuar con el desarrollo de este proyecto.

- Ampliación de funcionalidades: Explorar la incorporación de nuevas funcionalidades que mejoren la interacción con el sistema, como la detección de tendencias en los datos de los pacientes y el lanzamiento de notificaciones con alarmas en caso de que las constantes vitales de un paciente estén fuera de los rangos normales.
- Desarrollo de un CRUD completo: Implementar un sistema CRUD (Crear, Leer, Actualizar, Eliminar) que permita a los usuarios no solo obtener datos de la base de datos mediante peticiones GET, sino también crear, eliminar y actualizar registros directamente desde la aplicación.
- Implementación de un roxy: Desarrollar un proxy para que las peticiones al servidor no sean visibles desde el menú de administrador, proporcionando una capa adicional de seguridad y enmascarando las solicitudes.
- Conexión en tiempo real con dispositivos: Conectar dispositivos de monitorización de pacientes directamente con la base de datos para actualizar las gráficas de observaciones en tiempo real, mejorando así la precisión y la inmediatez de los datos mostrados.
- Optimización de rendimiento: Continuar optimizando tanto el backend como el frontend para asegurar un rendimiento superior, especialmente en escenarios con grandes volúmenes de datos.

REFERENCIAS

[1] Ramírez Garrido, Lorena, *Monitorización de constantes vitales en la Unidad de Cuidados Intensivos Revista NPunto*, vol. 6, nº 66, septiembre 2023.

[2] Wikipedia, 2024, *Problema de los puentes Königsberg*. Consultada el 17 de mayo del 2024 en https://es.wikipedia.org/wiki/Problema_de_los_puentes_de_K%C3%B6nigsberg

[3] Amazon Web Services, 2023, *¿Cuál es la diferencia entre una base de datos basada en grafos y una base de datos relacional?*. Consultada el 17 de mayo del 2024 en https://aws.amazon.com/es/compare/the-difference-between-graph-and-relational-database/

[4] Graph Everywere, 2024, *Comparativa Bases de datos de Grafos | ¿Cuál es mejor? ¿Cuál elegir?*. Consultada el 21 de mayo del 2024 en https://www.grapheverywhere.com/comparativa-bases-de-datos-de-grafos-cual-es-mejor-cual-elegir/

[5] BBVA, 2024, *¿Qué es la explicabilidad de la inteligencia artificial? Cómo quitarle misterio a la tecnología*. Consultada el 21 de mayo del 2024 en https://www.bbva.com/es/innovacion/que-es-la-explicabilidad-de-la-ia-como-quitarle-misterio-a-la-tecnologia/

[6] Jhon Stegman, 2024, *Native vs Non-Native Graph Database*. Consultada el 24 de mayo del 2024 en https://neo4j.com/blog/native-vs-non-native-graph-technology/

[7] Neo4J, 2024, *What is a graph database?* .Consultada el 27 de mayo del 2024 en https://neo4j.com/docs/getting-started/get-started-with-neo4j/graph-database/

[8] Neo4J, 2024, *Overview*. Consultada el 27 de mayo del 2024 en https://neo4j.com/docs/cypher-manual/current/introduction/cypher_overview/

[9] Neo4J, 2024, *GraphAademy, Fundamentals*. Consultada el 27 de mayo de 2024 en https://graphacademy.neo4j.com/

[10] Google, 2020, *Introducción a la Documentación de Angular*. Consultada el 3 de junio del 2024 en https://docs.angular.lat/docs

[11] Google, 2024, *TypeScript for JavaScript Programmers*. Consultada el 4 de junio del 2024 en https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html

[12] Google, 2020, *Funcionalidades y Ventajas*. Consultada el 4 de junio del 2024 en https://docs.angular.lat/features

[13] Getboostrap, 2023, *Get started with Boostrap*. Consultada el 7 de junio del 2024 en https://getbootstrap.com/docs/5.3/getting-started/introduction/

[14] Google, 2020. *Introducción a componentes y plantillas*. Consultada el 14 de junio del 2024 en https://docs.angular.lat/guide/architecture-components

[15] Jakub Staron, 2024. *Introduction.* Consultada el 16 de junio del 2024 en https://docs.nestjs.com/

[16] Martin Duran, 2023. *Qué es una Single Page Application, cómo funciona y ejemplo*. Consultada el

17 de junio del 2024 en https://blog.hubspot.es/website/que-es-single-page-application

[17] Microsoft, 2024. *Visual Studio Code*. Consultada el 17 de junio del 2024 en https://code.visualstudio.com/

[18] Neo4J, 2024. *About Neo4J Desktop*. Consultada el 21 de junio del 2024 en https://neo4j.com/docs/desktop-manual/current/about-desktop/

[19] Miguel Angel Alvarez, 2021. *Servicios en NestJS*. Consultada el 21 de junio del 2024 en https://desarrolloweb.com/articulos/servicios-nest

[20] Jakub Staron, 2024. *Controllers*. Consultada el 28 de junio del 2024 en https://docs.nestjs.com/controllers