UNIVERSIDAD TÉCNICA DEL NORTE

Facultad de Ingeniería en Ciencias Aplicadas

Carrera de Software

DESARROLLO DE UN ENVOLTORIO DEL API-REST DE NAPSTER UTILIZANDO EL LENGUAJE DE CONSULTAS GRAPHQL, PARA MEJORAR LA EFICIENCIA DEL CONSUMO DE LOS DATOS BASADO EN LA CARACTERÍSTICA DE EFICIENCIA DE LA NORMA ISO/IEC 25023.

Trabajo de grado previo a la obtención del título de Ingeniería de Software

Autor:

Nelson Daniel Guamán Valencia

Director:

PhD. José Antonio Quiña Mera

Ibarra – Ecuador

2023

UNIVERSIDAD TÉCNICA DEL NORTE
BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

DATOS DEL CONTACTO			
CÉDULA DE IDENTIDAD:	1004117600		
APELLIDOS Y NOMBRES:	GUAMÁN VALENCIA NELSON DANIEL		
DIRECCIÓN:	CARANQUI – SAN FRANCISCO DE CHORLAVISITO – SECTOR NUEVOS HORIZONTES		
E-MAIL:	ndguamanv@utn.edu.ec		
TELEFONO FIJO:	062-550177	TELÉFONO MOVÍL:	0994774491

DATOS DE LA OBRA	
TÍTULO:	DESARROLLO DE UN ENVOLTORIO DEL API-REST DE NAPSTER UTILIZANDO EL LENGUAJE DE CONSULTAS GRAPHQL, PARA MEJORAR LA EFICIENCIA DEL CONSUMO DE LOS DATOS BASADO EN LA CARACTERÍSTICA DE EFICIENCIA DE LA NORMA ISO/IEC 25023.
AUTOR (ES):	GUAMÁN VALENCIA NELSON DANIEL
FECHA:	12/09/2023
PROGRAMA:	☒ PREGRADO ☐ POSTGRADO
TÍTULO POR EL QUE OPTA:	INGENIERÍA EN SOFTWARE
ASESOR/DIRECTOR:	PhD. ANTONIO QUIÑA

2. CONSTANCIAS

2. CONSTANCIAS

El autor (es) manifiesta (n) que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto la obra es original y que es (son) el (los) titular (es) de los derechos patrimoniales, por lo que asume (n) la responsabilidad sobre el contenido de la misma y saldrá (n) en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 12 días del mes de septiembre de 2023

EL AUTOR:

Sr. Guamán Daniel
C.I: 100411760-0

CERTIFICADO DEL DIRECTOR DE TRABAJO DE GRADO

UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA DE CIENCIAS EN APLICADAS

CERTIFICACIÓN DEL DIRECTOR

Certifico que la Tesis previa a la obtención del título de Ingeniera en Software con el tema: "DESARROLLO DE UN ENVOLTORIO DEL API-REST DE NAPSTER UTILIZANDO EL LENGUAJE DE CONSULTAS GRAPHQL, PARA MEJORAR LA EFICIENCIA DEL CONSUMO DE LOS DATOS BASADO EN LA CARACTERÍSTICA DE EFICIENCIA DE LA NORMA ISO/IEC 25023." ha sido desarrollada y terminada en su totalidad por el Sr. Guamán Valencia Nelson Daniel, con cédula de identidad Nro. 100411760-0 bajo mi supervisión para lo cual firmo en constancia.

PhD. Antonio Quiña
DIRECTOR DE TESIS

DEDICATORIA

Este trabajo de titulación se lo dedico principalmente a mi madre quien ha sido la persona que más me ha apoyado a lo largo de mi vida personal y académica, con sus consejos y valores que me ha inculcado hasta llegar a ser la persona que soy hoy en día, el apoyo que me ofreció el cual fue lo que me llevo a finalizar con éxito el presente trabajo de titulación.

A mi hermana y abuelo por el cariño que me brindaron en los buenos y malos momentos, por estar presentes cuando los necesité y sacarme una sonrisa.

AGRADECIMIENTOS

Agradezco a Dios por permitirme llegar tan lejos y dejarme culminar esta etapa de mi vida con éxito, a mi familia por todo el cariño y amor que me dieron en el transcurso de mi vida universitaria y pese a los contratiempos que se presentaron me ayudaron a salir adelante, así convirtiéndose un mi más grande inspiración.

A mi tutor, PhD. Antonio Quiña quien con su conocimiento, me guio y apoyó en la realización y culminación de mi trabajo de titulación, de igual manera a mi opositor MSc. Mauricio Rea quien me dio sus recomendaciones y observaciones para llevar a cabo la finalización de mi trabajo de titulación.

A mis amigos y compañeros quienes estuvieron conmigo en el transcurso de la carrera y me brindaron su amistad y apoyo cuando los necesitaba.

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

En la actualidad existen varios servicios que hacen uso de la arquitectura REST, pero conforme avanza el tiempo se ha visto que esta arquitectura tiene sus inconvenientes al momento de desplegar aplicaciones, por esto se han creado nuevas arquitecturas como GraphQL, que brinda los mismo servicios que REST pero de una manera más ágil y ordenada. Para esto se hará uso de la API-REST de Napster, y se comparará con una API-GraphQL(wrapper), lo que comparará el tiempo de respuesta de ambas arquitecturas, así descubriendo cual es más eficiente.

El presente trabajo hace uso de la API de Napster como caso de desarrollo y muestra la creación de una API que envuelve a la tecnología API-REST transformándola en una tecnología GraphQL(wrapper). Para el desarrollo del envoltorio se trabajó siguiendo la metodología ágil scrum en donde se fue presentando avances del desarrollo en cada sprint. Para validar el software, se realizó un laboratorio experimental el cual ese basó en la guía de Wohlin, en donde se pudo comparar la eficiencia de REST y GraphQL, tanto cuando se hace uso de caché y cuando no se usa caché con respecto a la calidad del producto de software. Para medir la eficiencia se usó la métrica de “tiempo medio de respuesta” definida en la ISO/IEC 25023, lo que dio como resultado que el tiempo medio de respuesta cuando se utiliza caché es mucho menor que cuando no se utiliza caché en cualquiera de las arquitecturas, y de la misma manera se pudo observar que GraphQL es más eficiente que REST al momento de consumir los servicios API. Por lo cual se concluyó que GraphQL es más eficiente que REST cuando se despliegan servicios, sobre todo cuando se utiliza caché.

Palabras clave: API, arquitecturas de software, envoltorios(wrappers), Napster, REST, GraphQL, laboratorio computacional, tiempo medio de respuesta, Wohlin, eficiencia, caché

ABSTRACT

Currently, there are several services that make use of the REST architecture, but as time progresses, it has been observed that this architecture has its drawbacks when deploying applications. Because of this, new architectures like GraphQL have been created, which provide the same services as REST but in a more agile and organized manner. To achieve this, we will make use of the Napster REST API and compare it with a GraphQL API (wrapper), which will compare the response times of both architectures, thereby discovering which one is more efficient.

This work utilizes the Napster API as a development case and demonstrates the creation of an API that wraps the REST technology into a GraphQL technology (wrapper). The wrapper development followed the agile Scrum methodology, where progress was presented in each sprint. To validate the software, an experimental laboratory was conducted based on Wohlin's guidelines, where the efficiency of REST and GraphQL was compared, both when using cache and when not using cache in terms of software product quality. The efficiency was measured using the "average response time" metric defined in ISO/IEC 25023, resulting in the average response time being significantly lower when cache is used compared to when cache is not used in either of the architectures. It was also observed that GraphQL is more efficient than REST when consuming API services. Therefore, it was concluded that GraphQL is more efficient than REST when deploying services, especially when cache is used.

Keywords: API, software architectures, wrappers, Napster, REST, GraphQL, computational laboratory, average response time, Wohlin, efficiency, cache.

INTRODUCCIÓN

Antecedentes

El origen de GraphQL se remonta a los cambios que existieron en la industria móvil. Esto fue cuando Facebook decidió adoptar HTML5 para dispositivos móviles, llegando a tener problemas con el alto uso de la red, como resultado Facebook en el año 2012 decide implementar la tecnología GraphQL como un medio para reducir el uso de la red a la hora de obtener datos (APPTEC, 2020)

El distribuidor de archivos .mp3 Napster fue creado principalmente para realizar un servicio de intercambio de archivos perr-2-peercon el fin de que sus usuarios pudieran intercambiar archivos de música con otros usuarios. A través de un servicio REST, Napster permite a los desarrolladores integrar sus productos nuevos o existentes con el servicio de transmisión de música; para consumir las funciones que ofrece se debe tener conocimientos de la arquitectura REST, y, además el API presenta el inconveniente de no tener flexibilidad para los desarrolladores ya que, al tener muchos EndPoints, se dificultaba el consumo de dichas funciones.

Situación Actual

Una API, o interfaz de programación de aplicaciones, es un conjunto de reglas que determinan cómo las aplicaciones o los dispositivos pueden conectarse y comunicarse entre sí y REST es una interfaz para conectar varios sistemas que se basen en el protocolo HTTP (IBM, 2021). Por el crecimiento de los consumidores y las diferentes necesidades, este estilo arquitectónico reveló debilidades relacionadas con el rendimiento y la flexibilidad de las aplicaciones. Estos son o pueden ser abordados con GraphQL (Landeiro, 2020)

La plataforma de música Napster, tiene una API con arquitectura REST que puede utilizar la comunidad de desarrolladores interesados en estos servicios. Sin embargo, esta plataforma no cuenta con un API GraphQL que use los servicios de una manera más eficiente y personalizada los servicios que ofrece.

Se ha visto que a la hora de consumir el API-REST de Napster, la información que devuelve como respuesta es muy extensa, esta se presenta de una forma muy variada, lo que conlleva a que los desarrolladores les cueste el descubrir, parametrizar y separar los datos haciendo que los tiempos de consumo sean muy altos; además en la actual comunidad de desarrolladores no existe un motor de ejecución que ayude a la organización, presentación y acceso más eficiente de los datos.

Prospectiva

Con el presente trabajo, se procura desarrollar una envoltura del API-REST de Napster aplicando GraphQL, que permita filtrar y presentar de una mejor manera los datos a los desarrolladores; la implementación del envoltorio se realizara con la creación de una API-GraphQL y de un cliente, utilizando un marco de trabajo de calidad de uso basado en el estándar ISO/IEC 25023, permitiendo demostrar la eficacia y flexibilidad que ofrece la nueva tecnología a la hora de ser consumida e implementada.

Planteamiento del Problema

Las limitaciones en la eficiencia y la dificultad de consumo del API-REST de Napster, hacen que usar este servicio sea un poco molesto del lado del cliente por la complejidad de adaptación de dicho servicio con el producto del cliente (aplicaciones web, aplicaciones de escritorio, aplicaciones móviles etc.). La falta de una herramienta (Api GraphQL) más eficiente e intuitiva para el cliente hacen que el proceso de consumo no sea óptimo ni fácil de implementar.

Fig. 1 Árbol de Problemas

Objetivos

Objetivo General

Desarrollar un envoltorio del API-REST de Napster utilizando el lenguaje de consultas GraphQL, para mejorar la eficiencia del consumo de los datos basado en la característica de eficiencia de la norma ISO/IEC 25023.

Objetivos Específicos

- Establecer un marco teórico para el desarrollo del proyecto.

- Implementar un envoltorio API-REST con GraphQL que permita fortalecer el consumo de la plataforma de música Napster.
- Evaluar el envoltorio a través de un experimento para comparar la eficiencia del envoltorio API-GraphQL y el API-REST.
- Analizar los datos obtenidos del proyecto.

Alcance

En este proyecto, para conocer los conceptos importantes a implementar se realizará un marco teórico que realice una investigación que dé a conocer los conceptos más importantes de la arquitectura REST y el lenguaje de GraphQL, así como también los conceptos de las herramientas tecnológicas que se van a utilizar, y además de aclarar el marco de trabajo de calidad de uso, basado en la ISO/IEC 25023.

Se realizará un envoltorio de la API-REST de Napster utilizando GraphQL el cual mejora la comunicación entre las API y los consumidores, evitando la necesidad de peticiones en cascada del estilo REST (Acosta, 2021).

Para la implementación de la arquitectura se utilizará la herramienta tecnológica de Apollo Client, el cual es un cliente JavaScript que permite crear servidores GraphQL y consumir APIs GraphQL (Lázaro, 2020).

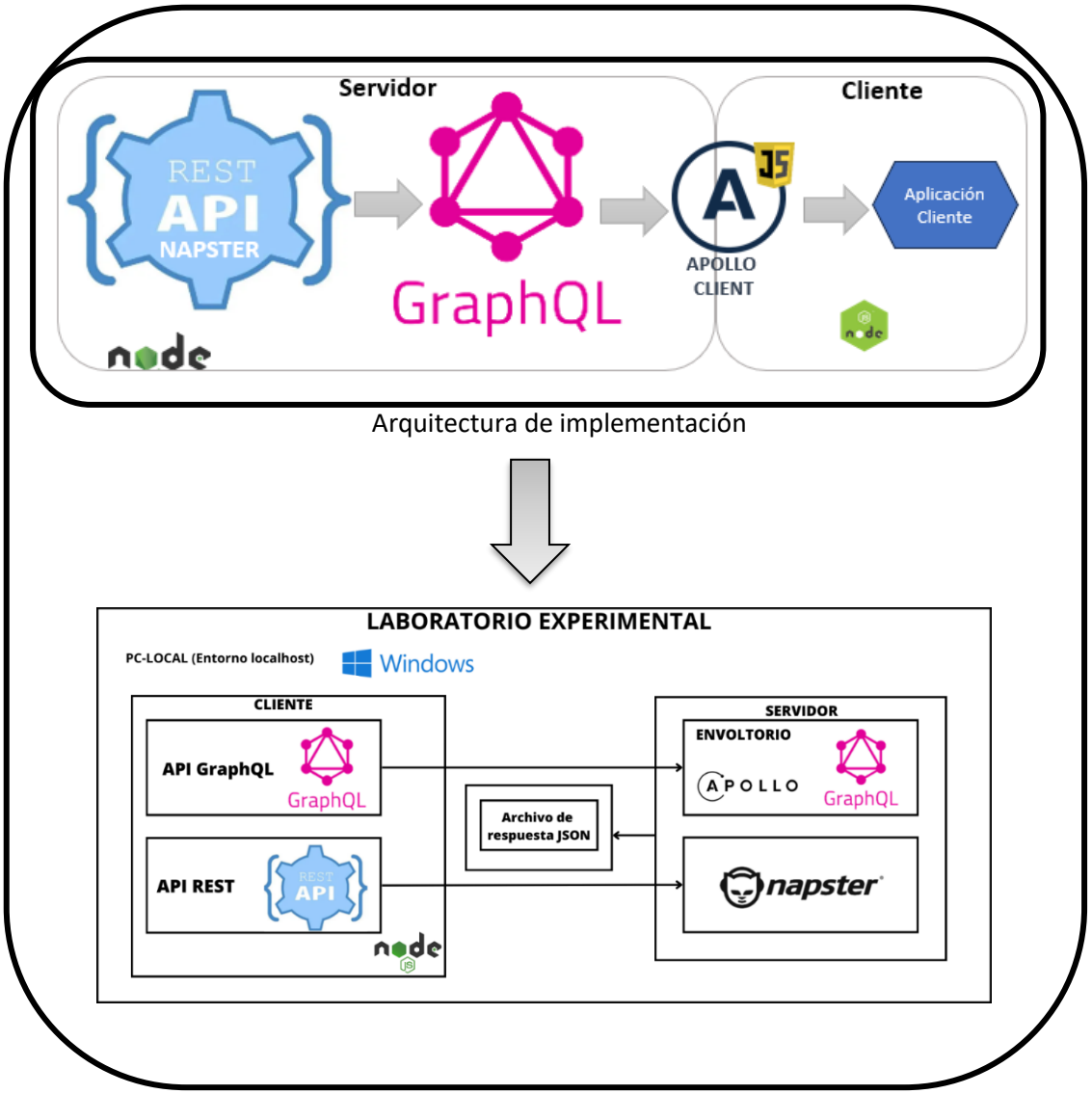
Se utilizará Node.js el cual es un entorno de ejecución para JavaScript, para el desarrollo del envoltorio API-GraphQL, así como para la comparación de los resultados del experimento se lo hará basándose en la guía de Wohlin, la cual se enfoca en realizar experimentos controlados que involucren productos de software (Claes Wohlin, 2012).

Se utilizarán los servicios proporcionados por la plataforma Napster que, debido a que la información que provee es detallada y precisa, pero también extensa a la hora de realizar consultas en el Api-REST, por lo que se espera que el desarrollo del envoltorio ayude a que esta información sea más directa y no tan extensa a la hora de realizar consultas, además de que estos datos cumplen con los requisitos necesarios para realizar el experimento.

Servicios de Napster que se implementarán al proyecto.

- Conexión a la API de Napster
- Listado de Artistas
- Listado de Tracks
- Listado de Álbumes

En la siguiente figura se muestra la arquitectura general del sistema, y la descripción general del proceso del experimento basándose en la guía de Wohlin.



Arquitectura de Implementación y descripción del experimento.
(Fuente: Elaboración Propia)

Justificación

Con el presente trabajo se espera generar un mayor aporte al conocimiento sobre las nuevas herramientas, no solo API-REST, sino que también la efectividad que tiene GraphQL y las distintas oportunidades que dará a los diversos usuarios para que puedan crear soluciones con la información adecuada, es por esto por lo que se dará a implementar una manera (un wrapper) que maneje dicha información más fácilmente.

La contribución de herramientas tecnológicas accesibles a la sociedad se ha vuelto muy habitual, una de las ideas más acopla a esta definición son las herramientas de software libre, que permiten a cierta comunidad con conocimientos XXI específicos, explotar muchos productos o servicios informáticos. La plataforma de música Napster ofrece a la comunidad de desarrolladores un servicio tecnológico con el que se accede a las funcionalidades de gestión de bibliografía mediante un API-REST.

Con la implementación de una manera más flexible (API GraphQL) para obtener los servicios de Napster, el presente trabajo contribuye a la generación de oportunidades de trabajo ya que profesionales pueden usar la herramienta desarrollada y crear sus propias soluciones sin necesidad de tener experticia en áreas específicas.

El enfoque del presente trabajo hacia los objetivos de desarrollo sostenible (ODS) se contempla en 2 de ellos:

Educación de calidad (Objetivo 4): La presente investigación, pretende incentivar y motivar a seguir por un camino para la formación que tenga que ver con la tecnología y proveer información para un mejor conocimiento (Naciones Unidas 2018), así mejorar el ambiente educativo, con el concepto de que un trabajo de calidad genera información de calidad.

Industria, innovación e infraestructura (Objetivo 9): Las tecnologías de la información y comunicación han estado al frente en respuesta de las necesidades, proyectos o innovaciones de las personas y para desarrollar infraestructuras fiables, sostenibles, resilientes y de calidad, para apoyar el desarrollo económico y el bienestar humano (Organización Internacional del Trabajo 2017).

CAPITULO I

Marco Teórico

1.1. Fundamentos de arquitecturas orientas a microservicios.

Según (Azure, 2022) una arquitectura de microservicios cuenta con servicios autónomos y pequeños, donde cada servicio es independiente y tiene una funcionalidad de negocio individual.

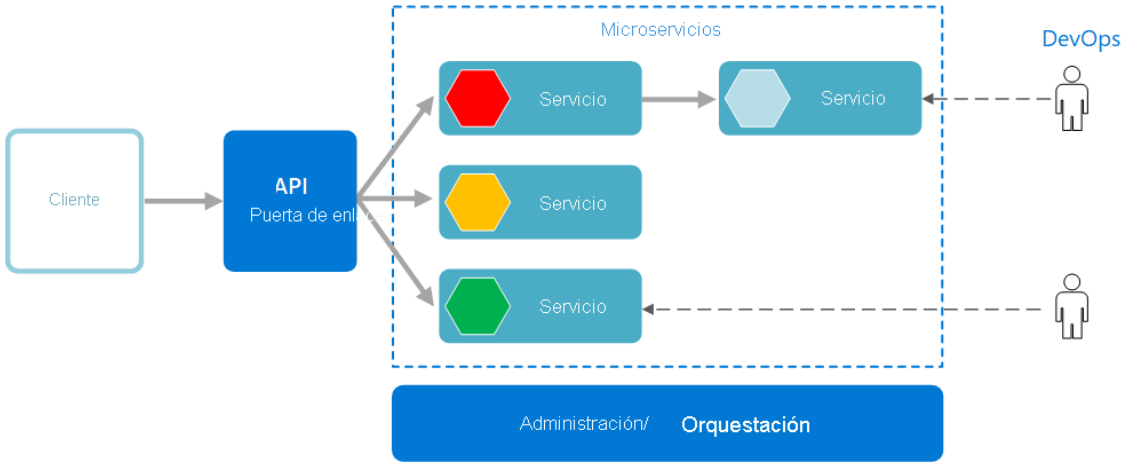


Fig. 2 Arquitectura de Microservicios

Fuente:(Microsoft Azure)

1.1.1. Arquitecturas de software

Las arquitecturas de software son una estructura fundamental en el desarrollo de sistemas de software, definiendo la organización y disposición de los componentes y módulos que conforman el sistema. Según (RedHat, 2022), "la arquitectura de software se refiere a la estructura fundamental de un sistema de software, incluyendo sus componentes, relaciones entre ellos y los principios y directrices que guían su diseño y evolución". Las arquitecturas de software permiten una mejor comprensión y comunicación del sistema entre los miembros del equipo, así como una mayor eficiencia en la gestión del proyecto. Además, también ayudan a mejorar la escalabilidad, mantenibilidad y flexibilidad del sistema de software a largo plazo.

1.1.1.1. Arquitectura Monolítica

La arquitectura monolítica es un estilo de arquitectura de software en el que una aplicación se construye como una sola unidad, con sus diferentes componentes interconectados y dependientes entre sí. Este enfoque se ha utilizado en el desarrollo de aplicaciones empresariales complejas, pero ha sido criticado por su falta de escalabilidad y flexibilidad. En los últimos años, se ha producido un cambio hacia arquitecturas más modulares y distribuidas, como la arquitectura de microservicios, que permite una mayor escalabilidad y flexibilidad (Ramos, 2019).

1.1.1.2. Arquitectura Microservicios

Un microservicio es un tipo de arquitectura de software que se enfoca en dividir una aplicación en servicios más pequeños y autónomos, que se comunican entre sí a través de interfaces bien definidas. Cada servicio es responsable de una tarea específica y está diseñado para ocultar su información interna a los demás servicios. Este enfoque permite una mayor flexibilidad y escalabilidad en el desarrollo y mantenimiento de la aplicación (Velepucha, Flores, & Torres, 2019).

1.1.1.3. Arquitectura Cliente-servidor

La arquitectura cliente-servidor es un modelo de diseño de software que se ha utilizado en los últimos años. Esta arquitectura consiste en separar los procesos de una aplicación en dos tipos de componentes: un cliente y un servidor. El cliente es responsable de solicitar servicios o recursos del servidor, mientras que el servidor proporciona estos servicios o recursos al cliente (Rouse, Client/Server Architecture, 2020).

1.1.2. Interfaz de programación de aplicaciones (API)

Una API permite que dos aplicaciones se comuniquen con un lenguaje que ambas comprenden; una API es un intermediario entre dos aplicaciones permitiendo que estas interactúen eficazmente. Las API también ofrecen una interfaz estandarizada que permite a los desarrolladores interactuar con las funcionalidades de una aplicación de manera programática, sin la necesidad de comprender los detalles internos de cómo funciona la aplicación.

- **API REST**

  Según (Ribas, 2018) una API con un estilo arquitectónico REST debe tomar en cuenta unas características como:

  a) Protocolo cliente/servidor

b) Operaciones como: POST (crear), GET (leer y consultar), PUT (editar) y DELETE (borrar).
c) Identificador URI
d) Una interfaz uniforme y un sistema de capas

- **API GRAPHQL**

Según (RedHat, 2019) la función de una API GraphQL es ofrecer a los clientes exactamente los datos que solicitan ya que permite a los desarrolladores crear consultas para extraer datos de varias fuentes en una sola llamada a la API.

1.1.3. Estilo arquitectónico REST

Representational State Transfer o más comúnmente conocido como REST, es una serie de restricciones que crea un estilo arquitectónico, este estilo se basa en recursos. Un recurso es una entidad, la cual se almacena principalmente en un servidor y el cliente solicita el recurso utilizando servicios Web RestFull (Castro, 2018).

Los principios REST están definidos por cuatro controles de interfaz, que incluyen la identificación de recursos, la gestión de recursos a través de representaciones, comunicaciones autodescriptivas e hipermedia como motor del estado de la aplicación (Naeem, 2020).

- **Recursos**: Los recursos en REST se identifican a través de una URI (Uniform Resource Identifier) conocidos también como EndPoints, la manipulación las URI se lo realiza a mediante un método HTTP, el sistema que soporta el protocolo HTTP es capaz de usar REST (Kobusinska & Hsu, 2017).
- **Representaciones: Los componentes REST se comunican mediante la transferencia de una representación de los datos en un formato coincido con uno de los tipos de datos estándar, seleccionados dinámicamente según las capacidades o deseos del destinatario y la naturaleza de los datos.** (Fielding & Taylor, 2020)
- **Estado**: Es la representación actual de un recurso en un momento dado. Mientras está en el servidor el estado de un recurso puede ser recuperado a través de una petición HTTP GET, PUT, UPDATE y se representa en un formato específico, como JSON o XML. En cambio, el estado se puede representar como la aplicación cuando se encuentra del lado del cliente (Zhang, Ma, Lai, Yang, & Tang, 2018).
- **Operaciones**: Las operaciones más importantes relacionadas con los datos en cualquier sistema REST y la especificación HTTP son cuatro: POST (crear), GET (leer y consultar), PUT (editar) y DELETE (eliminar).

Tabla 1 Operaciones REST sobre los recursos

Acción	Protocolo HTTP	Descripción
CREATE	POST	Obtiene la información de un recurso a través de una petición HTTP.
RETRIEVE	GET	Crea nuevos recursos a través de una petición HTTP.
UPDATE	PUT	Actualiza la información de un recurso a través de una petición HTTP.
DELETE	DELETE	Elimina el recurso a través de una petición HTTP.

Fuente: (Galipienso, 2014)

1.1.4. Lenguaje de consultas GraphQL

El origen de GraphQL se remonta a los cambios que existieron en la industria móvil. Esto fue cuando Facebook decidió adoptar HTML5 para dispositivos móviles, llegando a tener problemas con el alto uso de la red, como resultado Facebook en el año 2012 decide implementar la tecnología GraphQL como un medio para reducir el uso de la red a la hora de obtener datos (APPTEC, 2020). GraphQL permite a los clientes (como aplicaciones móviles o sitios web) enviar solicitudes de datos precisas y personalizadas a los servidores, en lugar de recibir datos excesivos o insuficientes de una API tradicional. Además, permite a los desarrolladores de API especificar la forma y la estructura de los datos que se deben entregar a los clientes (Quiña-Mera, Fernandez, García, & Ruiz-Cortés, 2023).

1.1.4.1. Estructura Consultas y Mutaciones

- **Operaciones**

  GraphQL es capaz de ejecutar las cuatro funciones esenciales de REST (GET, POST, PUT, DELETE) para administrar información; a esta acción se le conoce como CRUD (crear, consultar, actualizar y eliminar). En relación con REST; este posee estas operaciones en forma de verbos las cuales son especificadas en la consulta HTTP, en cambio GraphQL especifica tres tipos de operaciones (Vázquez et al., 2017):

  a) **Consultas:** Obtención de datos (solo lectura).
  b) **Mutaciones:** Realiza una modificación de los datos y la respuesta de la consulta da como resultado una creación, actualización o eliminación.
  c) **Suscripciones:** Solicitudes en respuesta a eventos fuente (tiempo real)

- **Campos**

Son los componentes más básicos del esquema de GraphQL , determinan qué campos ofrece el servicio GraphQL, el lenguaje de esquema GraphQL admite los tipos escalares de cadena, int, flotante, booleano e ID (Ghebremicael, 2017)

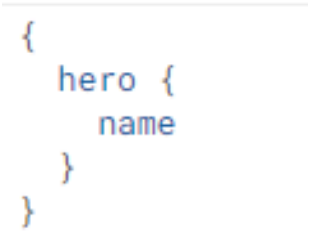


Fig. 3 Campos en formato simple GraphQL

Fuente: graphql.org

- **Argumentos**

  En una consulta GraphQL, los argumentos son información que se relaciona con un

  campo de consulta para realizar una relación tipo clave-valor. También, los argumentos permiten filtrar ciertos datos de una consulta (Porcello & Banks, 2018).

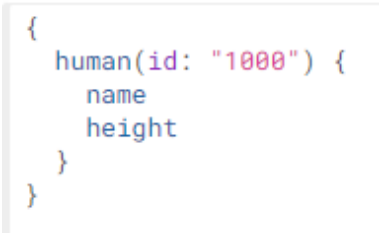


Fig. 4 Argumentos en GraphQL

Fuente: graphql.org

- **Variables**

  Los argumentos de los campos serán dinámicos, no sería una buena idea pasar estos argumentos dinámicos directamente en la cadena de consulta, porque entonces nuestro código del lado del cliente necesitaría manipular dinámicamente la cadena de consulta en tiempo de ejecución y serializarla en un formato específico. Por esto GraphQL tiene una forma de primera clase de factorizar valores dinámicos fuera de la consulta y pasarlos como un diccionario separado. Estos valores se denominan variables (GraphQL-Foundation, 2019).

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

VARIABLES

```
{
  "episode": "JEDI"
}
```

Fig. 5 Variables en GraphQL

Fuente: graphql.org

1.1.4.2. Esquemas y tipos

- **Argumento**

  Los argumentos son un mecanismo que permite a los clientes especificar información adicional en sus consultas para personalizar las respuestas que reciben. Los argumentos se definen en el esquema GraphQL junto con los campos y tipos de datos, y se utilizan para indicar qué datos específicos deben ser devueltos (GraphQL, 2023).

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

Fig. 6 Ejemplo de Argumento

- **Esquema**

  El esquema del servidor GraphQL define en su estructura los tipos de datos y las relaciones que existen en estos, en este esquema también se definen las operaciones de consulta y mutaciones de los datos (Wittern et al., 2018).

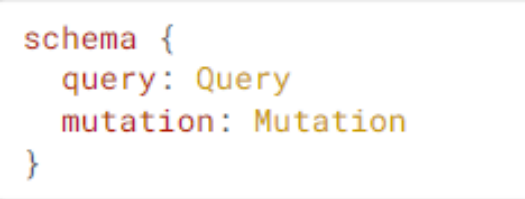


Fig. 7 Ejemplo de esquema

Fuente: graphql.org

- **Campos y tipos de objetos**

  Un tipo representa a un objeto personalizado y cada objeto representa la funcionalidad principal de la aplicación. Un tipo está compuesto por campos que no son más que las características o propiedades del objeto (Porcello & Banks, 2018).

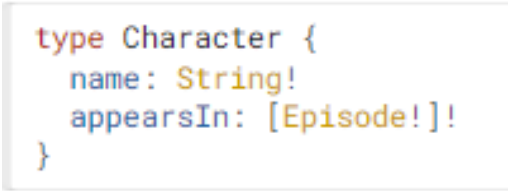


Fig. 8 Ejemplo de campos

Fuente: graphql.org

- **Tipos consulta y mutación (type query, type mutation)**

  En un esquema GraphQL, los tipos de consulta y mutación son dos elementos clave que definen cómo los clientes pueden interactuar con una API. Según la documentación oficial de GraphQL, las consultas son operaciones de solo lectura que permiten a los clientes recuperar datos de la API, mientras que las mutaciones son operaciones de escritura que permiten a los clientes modificar o eliminar datos de la API. Los tipos de consulta y mutación se definen en el esquema GraphQL y se utilizan para especificar qué campos están disponibles en cada tipo de operación. Además, los tipos de consulta y mutación también pueden tener argumentos que se utilizan para personalizar la respuesta de la API. En resumen, los tipos de consulta y mutación en GraphQL son elementos clave del esquema que permiten a los clientes interactuar con la API de manera efectiva y flexible (GraphQL, 2023).

```
type Query {
  hero(episode: Episode): Character
  droid(id: ID!): Droid
}
```

Fig. 9 Ejemplo Query

Fuente: graphql.org

```
mutation CreateReviewForEpisode($ep: Episode
  createReview(episode: $ep, review: $review
    stars
    commentary
  }
}
```

Fig. 10 Ejemplo mutation

Fuente: graphql.org

- **Tipos de entrada(input)**

  Los tipos de entrada son un tipo especial de tipo de objeto que se utiliza para especificar el formato de los datos que se envían a través de las mutaciones, se definen en el esquema GraphQL y se utilizan para especificar los campos y tipos de datos que se deben proporcionar al realizar una mutación. Los tipos de entrada son similares a los tipos de objeto, pero se utilizan de manera diferente: mientras que los tipos de objeto se utilizan para describir los campos y propiedades de los datos que se devuelven en una consulta, los tipos de entrada se utilizan para describir los campos y propiedades de los datos que se envían a través de una mutación (GraphQL, 2023).

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

Fig. 11 Ejemplo de Input

Fuente: Graphql.org

1.1.4.3. Validación

La validación es una parte importante del ciclo de vida de una consulta GraphQL, ya que ayuda a garantizar que la consulta cumpla con los requisitos del esquema GraphQL y que los datos solicitados estén disponibles en la API. Durante el proceso de validación, se comprueba que la consulta contenga solo campos y argumentos válidos y que los tipos de datos y argumentos sean correctos. Si se encuentra un error durante la validación, se devuelve un mensaje de error que indica la causa del problema. En resumen, la validación en GraphQL es una característica importante que ayuda a garantizar que las consultas enviadas a través de la API sean correctas y puedan ser ejecutadas con éxito en el servidor GraphQL (GraphQL, 2023).

1.1.4.4. Ejecución

La ejecución es una parte crucial del ciclo de vida de una consulta GraphQL, ya que es el paso final en la transformación de una consulta en una respuesta. Durante el proceso de ejecución, se procesan los campos solicitados en la consulta y se recuperan los datos correspondientes del origen de datos subyacente. Una vez que se han procesado todos los campos, se crea y se devuelve la respuesta al cliente en formato JSON. En resumen, la ejecución en GraphQL es un proceso importante que convierte una consulta en una respuesta, permitiendo que los clientes obtengan los datos solicitados de la API de manera rápida y eficiente (GraphQL, 2023).

1.1.5. Envoltorios tecnológicos

Un envoltorio o wrapper es una herramienta de programación que proporciona una capa adicional de abstracción sobre una tecnología subyacente, permitiendo a los desarrolladores interactuar con ella de manera más fácil y efectiva. Estos envoltorios pueden proporcionar una interfaz común para interactuar con tecnologías dispares o agregar funcionalidad adicional a una tecnología existente (Margaret, 2018).

1.1.5.1. Envoltorios de Base de Datos

Los envoltorios(wrappers) que se basan de una base de datos son instrumentos que proporcionan una capa de contemplación sobre una base de datos, permitiendo a los desarrolladores interactuar con ella de manera más fácil y eficiente. Estos envoltorios ofrecen una interfaz de programación de aplicaciones (API) común para interactuar con diferentes sistemas de bases de datos, independientemente del lenguaje de programación utilizado. Además, los envoltorios pueden agregar funcionalidad adicional a la base de datos, como la capacidad de almacenar y acceder a datos de manera distribuida (Daniel, 2019).

1.1.5.2. Envoltorios a nivel de lenguaje de programación

Los envoltorios a nivel de lenguaje de programación, también conocidos como wrappers de lenguaje, son herramientas que ofrecen una interfaz de programación de aplicaciones (API) común para interactuar con diferentes bibliotecas o frameworks en un lenguaje de programación específico. Estos envoltorios permiten a los desarrolladores interactuar con diferentes bibliotecas o frameworks utilizando una única API, lo que hace que el proceso de desarrollo sea más fácil y rápido (Lindberg, 2019).

1.1.5.3. Envoltorios basados en API-REST

Los envoltorios basados en API REST son herramientas que ofrecen una capa de abstracción sobre una API REST existente, permitiendo a los desarrolladores interactuar con ella de manera más fácil y eficiente. Estos envoltorios proporcionan una interfaz de programación de aplicaciones (API) común para interactuar con diferentes APIs REST, independientemente del lenguaje de programación utilizado (Murali, 2020).

1.2. Herramientas tecnológicas.

El presente trabajo busca realizar un experimento, el cual hará uso de un envoltorio GraphQL y del servicio REST, este estará compuesto por un servidor BackEnd, y un cliente FrontEnd el cual consumirá ambos servicios. A continuación, se especifica las herramientas a utilizar en el desarrollo del proyecto.

1.2.1. Lenguaje de programación JavaScript

JavaScript es un lenguaje de programación que se puede aplicar a un documento HTML y se pueden usar para establecer interactividad dinámica con los sitios web. JavaScript por sí solo es bastante compacto y flexible, y los desarrolladores han escrito gran cantidad de herramientas encima del núcleo del lenguaje JavaScript (MDN, 2022).

JavaScript se ha convertido en uno de los lenguajes de programación más populares y demandados en la industria del desarrollo de software. A partir de 2018, ha habido una serie de mejoras y actualizaciones importantes en JavaScript, incluyendo la introducción de nuevas características y especificaciones, lo que ha hecho que el lenguaje sea más poderoso y fácil de usar que nunca (StackOverflow, 2021).

1.2.2. Manejador de paquetes node (npm)

El manejador de paquetes de node(npm), es el manejador de paquetes más popular y utilizado para Node.js. Con él, los desarrolladores pueden descargar, instalar y gestionar paquetes de código fuente abiertos para sus proyectos de Node.js. En un artículo publicado en el sitio web de npm en 2020, se destacan algunas de las características más importantes de npm, como la facilidad de uso, la posibilidad de crear paquetes personalizados, el soporte para múltiples versiones de paquetes y la capacidad de publicar paquetes de manera rápida y fácil (npm, 2020).

En un artículo publicado en el sitio web de Medium en 2021, se destaca que npm ha sido fundamental para el crecimiento de la comunidad de desarrolladores de Node.js y ha contribuido significativamente a la evolución de la plataforma. El artículo también menciona que npm ha introducido nuevas características y herramientas para mejorar la calidad y seguridad de los paquetes, como la implementación de la autenticación de dos factores, la validación de los paquetes y la creación de un índice de paquetes vulnerables (Idongesit, 2021).

1.2.3. Framework de desarrollo Node.js

“Ideado como un entorno de ejecución de JavaScript orientado a eventos asíncronos, Node.js está diseñado para crear aplicaciones network escalables.” (Node, 2019). La eficiencia de Node.js se debe a que usa el motor v8 creado por Google para el lenguaje JavaScript; lo cual hace óptima a la administración de recursos y la rapidez del entorno de ejecución (Casciaro, 2014).

Node.js también ha sido adoptado ampliamente por grandes empresas en los últimos años. Empresas como Netflix, LinkedIn, Walmart, Uber y Airbnb han utilizado Node.js para construir aplicaciones de alto rendimiento y escalables. Según un artículo de Forbes de 2020, el uso de Node.js por parte de estas empresas se debe a su capacidad para manejar grandes cantidades de solicitudes y su eficiencia en el consumo de recursos. Además, Node.js también ha sido una opción popular para el desarrollo de aplicaciones en la nube debido a su capacidad para ser desplegado en múltiples plataformas de infraestructura en la nube (Vikram, 2020).

1.2.4. API REST de Napster

Napster es un servicio de intercambio de archivos que se originó en 1999, y fue creado por Shawn Fanning. El servicio permitía a los usuarios compartir archivos de música en

formato MP3 de forma gratuita, lo que llevó a una enorme popularidad, llegando a tener más de 80 millones de usuarios registrados en su apogeo.

La API de Napster fue creada en 2001 para permitir que desarrolladores externos accedieran a su plataforma de música en línea y crearan aplicaciones que pudieran interactuar con ella. Esta API permitió que terceros crearan aplicaciones que permitieran a los usuarios acceder a la biblioteca de música de Napster y crear sus propias listas de reproducción personalizadas.

Varios autores, como Stephen Witt y Douglas Kellner, han discutido la historia de Napster y su impacto en la industria musical y la cultura digital en sus trabajos, sosteniendo que Napster y otros servicios de intercambio de archivos transformaron la forma en que las personas consumían música y que su popularidad llevó a un cambio radical en la industria musical. (Witt, 2015)

- **Recursos Básicos**

Una de las características de REST es el de obtener la información de la API mediante una url, siendo así Napster ofrece una API de metadatos la cual brinda información sobre contenido musical y editorial disponible a través de la plataforma.

Tabla 2 Metadatos de la API de Napster

Recurso	Descripción	URL del recurso
Artistas	Este recurso ofrece una lista paginada opcionalmente de los mejores artistas de todo Napster, impulsada por la actividad de escucha.	https://api.napster.com/v2.2/artists/top
Tracks	Este recurso hace uso de un ID y devuelve una lista de las mejores pistas de Rhapsody, actualizada diariamente.	http://api.napster.com/v2.2/artists/${id}/tracks/top
Albums	Este recurso hace uso de un ID del EndPoint anterior y devuelve una lista de todos los lanzamientos nuevos, actualizada diariamente.	https://api.napster.com/v2.2/artists/${id}/albums
Géneros	Este recurso hace uso de un ID y devuelve un mapa jerárquico de todos los géneros y subgéneros.	https://api.napster.com/v2.2/genres/${id}
Posts	Este recurso hace uso de un ID y devuelve una lista de publicaciones de blog y otras características por género, escritas por el equipo editorial.	https://api.napster.com/v2.2/genres/${id}/posts

(developer.prod.napster.com, s.f.)

1.2.5. ApolloClient

ApolloClient es una biblioteca de JavaScript que se utiliza para construir aplicaciones de cliente que se conectan a una API GraphQL. Lanzado en 2016 por la compañía Apollo, ApolloClient se ha convertido en una de las bibliotecas más populares para trabajar con GraphQL en JavaScript. Apollo Client se integra con frameworks populares como React o Angular, lo que lo convierte en una opción atractiva para desarrolladores de FrontEnd. La biblioteca proporciona una forma sencilla y flexible de trabajar con datos en una aplicación de cliente, y también ofrece características avanzadas como caché en memoria, paginación y gestión de estados. En resumen, Apollo Client es una herramienta poderosa y eficiente para construir aplicaciones de cliente que utilizan GraphQL como fuente de datos. (ApolloClient, 2021)

1.3. Metodologías de desarrollo de software.

Las metodologías de desarrollo de software son enfoques sistemáticos para planificar, diseñar, implementar y mantener software de alta calidad, y según (Rincón-Mejía, López-González, Martínez-Santiago, & Ramírez-Benavides, 2021) existen varias metodologías de desarrollo de software ampliamente utilizadas en la industria, tales como Agile, Scrum, Kanban, Waterfall, entre otras. Estas metodologías se han desarrollado para mejorar la calidad del software, aumentar la eficiencia y reducir los costos de desarrollo. También hay un creciente interés en las metodologías de desarrollo de software ágil, que promueven la colaboración entre los miembros del equipo y la entrega continua de software de alta calidad.

Las metodologías de desarrollo de software son enfoques sistemáticos y estructurados utilizados para gestionar el proceso de creación de software. A lo largo de los últimos años, se han destacado varias metodologías, como Agile, Scrum y DevOps. Según (Rouse, 2019), Agile es una metodología ágil que se enfoca en la colaboración, la adaptabilidad y la entrega incremental de software. Además, un artículo de Paul Krill menciona (Krill, 2020), una metodología dentro del marco Agile que se basa en la colaboración y la flexibilidad para entregar productos de software en ciclos iterativos, además de que se menciona a DevOps como una metodología que combina el desarrollo y las operaciones, promoviendo la colaboración y la entrega continua. Estas metodologías han demostrado ser efectivas para gestionar proyectos de desarrollo de software y adaptarse a los cambios rápidos y las demandas del mercado.

1.3.1. Scrum

Según (Scrum, 2020), Scrum es un marco ligero que ayuda a las personas, los equipos y las organizaciones a generar valor a través de soluciones adaptativas para problemas complejos. o. El marco técnico de Scrum se forma de: roles, artefactos y eventos.

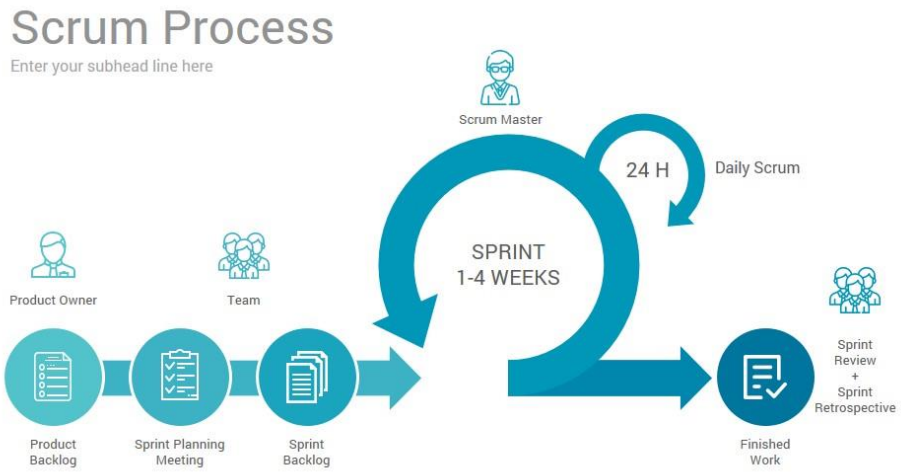


Fig. 12 Desarrollo ágil de software (Scrum)

Fuente: webdesigncusco.com

1.3.1.1. Roles

Cuando se trabaja en proyectos que utiliza la metodología Scrum, se deben considerar los roles principales del equipo de trabajo en el que cada miembro tiene un rol específico. Los roles que componen Scrum son: El Product Owner es el responsable de definir las funcionalidades del producto y priorizarlas en el backlog, mientras que el Scrum Master es el facilitador del proceso y se encarga de asegurar que el equipo de desarrollo tenga todo lo necesario para trabajar. Por último, el Equipo de Desarrollo es el grupo de personas encargado de diseñar, desarrollar y probar el producto (WEST, 2020).

Según (WEST, 2020) los roles de Scrum son:

- **Product Owner:** Este rol es el responsable de definir el producto, establecer los objetivos del proyecto y priorizar las funcionalidades de este. El Product Owner debe asegurarse de que el equipo de desarrollo esté trabajando en las tareas adecuadas y que el valor del producto sea maximizado en todo momento. También es el encargado de mantener el backlog actualizado y de revisar continuamente los requisitos del producto para asegurarse de que se están cumpliendo las expectativas del cliente.

- **Scrum Master:** Actúa como facilitador y defensor de los principios y prácticas de SCRUM dentro del equipo. Se encarga de asegurar que el equipo de desarrollo esté siguiendo el marco de trabajo de SCRUM y de eliminar cualquier obstáculo o impedimento que pueda surgir durante el proyecto. Además, el Scrum Master se enfoca en fomentar la colaboración y la comunicación efectiva dentro del equipo, promoviendo un entorno de trabajo productivo y de alto rendimiento. También brinda apoyo y orientación al Product Owner en la gestión del backlog y en la maximización del valor del producto.
- **Equipo de desarrollo:** Este equipo está compuesto por profesionales multidisciplinarios encargados de diseñar, desarrollar y entregar el producto. El Equipo de Desarrollo es autoorganizado y se organiza internamente para distribuir el trabajo de manera equitativa. Además, se enfoca en mejorar continuamente su capacidad para entregar valor al cliente, y toma decisiones técnicas y de diseño para lograr los resultados esperados.

1.3.1.2. Artefactos

Un artefacto en el contexto de Scrum se refiere a un elemento tangible o documental que proporciona transparencia y visibilidad sobre el trabajo realizado y los resultados obtenidos en un proyecto ágil. En la metodología Scrum, existen tres artefactos principales: el Producto de Backlog, que representa los requisitos y funcionalidades deseadas del producto; el Sprint Backlog, que es una lista de tareas que el equipo se compromete a completar durante un sprint específico; y el Incremento, que es el resultado tangible y potencialmente entregable al final de cada sprint (Scrum.org, 2023).

Según la página de SCRUM (Scrum.org, 2023) los artefactos son:

- **Product Backlog:** La pila del producto es una lista priorizada y dinámica que contiene todas las características, funcionalidades, mejoras y correcciones deseadas para un producto. El Producto de Backlog se mantiene y actualiza continuamente a medida que se obtiene un mayor conocimiento y se reciben comentarios de los usuarios y las partes interesadas. Cada elemento del backlog debe ser claro, conciso y tener un valor medible para el cliente. El equipo de desarrollo, en colaboración con el propietario del producto, es responsable de ordenar y estimar los elementos del backlog para garantizar la entrega de valor máximo en cada incremento del producto.
- **Sprint Backlog:** Es una lista detallada de las tareas específicas que el equipo de desarrollo se compromete a completar durante un sprint. El Sprint Backlog es creado en colaboración con el equipo de desarrollo durante la reunión de planificación del sprint y se actualiza diariamente en la reunión diaria de seguimiento del sprint. Cada tarea en el Sprint Backlog tiene una estimación de esfuerzo y es asignada a un miembro del equipo para su implementación. A medida que se completan las tareas, el equipo realiza un

seguimiento del progreso y se asegura de que se cumplan los objetivos establecidos para el sprint.

- **Incremento:** Un incremento en Scrum se refiere al resultado tangible y potencialmente entregable al final de cada sprint. Es el conjunto de todas las funcionalidades y mejoras completadas durante el sprint, que se han desarrollado, probado y listas para su entrega al cliente o para su integración en el producto en desarrollo. Cada incremento debe ser funcional y cumplir con los criterios de calidad definidos, lo que significa que debe ser utilizable y demostrar un avance real en la creación del producto final.

1.3.1.3. Eventos

Los eventos son momentos estructurados en los que el equipo Scrum colabora para inspeccionar y adaptar el progreso del proyecto.

Según (Schwaber & Sutherland, 2020) los eventos de SCRUM son:

- **Sprint Planning:** Es un evento en la metodología SCRUM donde el equipo Scrum se reúne para definir y acordar el objetivo del sprint y el trabajo a realizar durante el mismo. Durante esta reunión, que tiene una duración máxima de ocho horas para un sprint de un mes, el equipo colabora con el Product Owner para seleccionar los elementos del backlog del producto que serán incluidos en el sprint y los descompone en tareas más pequeñas si es necesario.
- **Daily Scrum:** También conocido como reunión diaria de seguimiento del sprint, es un evento clave que permite al equipo Scrum sincronizarse y planificar el trabajo para el día. Esta reunión, que tiene una duración máxima de 15 minutos, se lleva a cabo todos los días durante el sprint.
- **Sprint Review:** Es un evento clave que se lleva a cabo al final de cada sprint. Durante esta reunión, el equipo Scrum muestra al Product Owner y a otros interesados el trabajo completado durante el sprint. El objetivo principal del Sprint Review es obtener retroalimentación y colaborar en la evaluación del incremento del producto. Durante la revisión, el equipo Scrum presenta las historias de usuario completadas y demuestra las funcionalidades implementadas.
- **Retrospectiva:** Durante esta reunión, el equipo Scrum reflexiona sobre su desempeño durante el sprint y busca oportunidades de mejora continua. El objetivo principal de la retrospectiva es inspeccionar el proceso de trabajo y adaptarlo para aumentar la eficacia y la calidad del equipo. Durante la retrospectiva, se revisan las prácticas, herramientas y colaboración del equipo, y se identifican los aspectos que funcionaron bien y los que se pueden mejorar.

1.4. Experimentación en la ingeniería de software

La Experimentación en la ingeniería de software es un enfoque utilizado para evaluar y validar nuevas técnicas, herramientas y prácticas en el desarrollo de software. En los últimos años, se han publicado varios artículos que destacan la importancia de la experimentación en este campo. Según (Jørgensen & Shepperd, 2022), la experimentación en ingeniería de software permite obtener resultados empíricos sólidos, identificar posibles mejoras en los procesos de desarrollo y reducir los riesgos asociados con la adopción de nuevas tecnologías.

Metodología Wohlin

El fin de la experimentación es identificar las causas por las que se producen determinados resultados. Un experimento modela en el laboratorio, en condiciones controladas, las principales características de una realidad lo que permite estudiarla y comprenderla mejor (Wohlin et al., 2012). Esta metodología consta de las siguientes fases:

- **Selección del objetivo:** En esta fase, se define claramente el objetivo del estudio empírico, estableciendo las preguntas de investigación y los objetivos específicos a alcanzar.
- **Diseño del estudio:** En esta etapa, se planifica la estructura del estudio, incluyendo la selección de los participantes, los métodos de recolección de datos, los criterios de medición y los procedimientos de análisis.
- **Preparación del estudio:** Aquí se establecen los protocolos y procedimientos necesarios para llevar a cabo el estudio, incluyendo la elaboración de cuestionarios, la preparación del entorno de experimentación y la definición de los pasos a seguir.
- **Realización del estudio:** En esta fase, se lleva a cabo el estudio empírico, recolectando los datos según el diseño establecido previamente. Esto puede incluir la ejecución de experimentos, la recolección de datos de campo o la realización de encuestas, entre otros métodos.
- **Análisis de los resultados:** Una vez recopilados los datos, se realiza un análisis detallado para responder a las preguntas de investigación planteadas. Se utilizan técnicas estadísticas y métodos de análisis de datos para obtener conclusiones y extraer conocimientos relevantes.
- **Interpretación de los resultados:** En esta etapa, se interpretan los resultados obtenidos y se vinculan con el objetivo inicial del estudio. Se evalúan las implicaciones prácticas y se proporcionan recomendaciones basadas en los hallazgos del estudio.

- **Informe y presentación:** Finalmente, se elabora un informe detallado que documenta el estudio empírico realizado, incluyendo la descripción de la metodología, los resultados, las conclusiones y las recomendaciones. Además, se pueden realizar presentaciones y discusiones para compartir los hallazgos con la comunidad científica y profesional.

El desarrollo del experimento estará basado en el enfoque de calidad que es:

- **Enfoque de calidad.** El principal efecto estudiado en el experimento es el rendimiento individual. Aquí, se enfatizan dos aspectos específicos. La opción es centrarse en la productividad (KLOC/tiempo de desarrollo) y la densidad de defectos (fallas/KLOC), donde KLOC significa miles de líneas de código

1.5. Estándar ISO/IEC 25000.

La norma ISO/IEC 25000, también conocida como SQuaRE (Software Product Quality Requirements and Evaluation), es un estándar internacional que establece un marco de referencia para la evaluación y gestión de la calidad del software. Esta norma proporciona un conjunto de modelos y técnicas para evaluar distintos aspectos de la calidad del software, como funcionalidad, confiabilidad, eficiencia, usabilidad y mantenibilidad. Además, promueve el uso de métricas y la recolección de datos objetivos para evaluar y mejorar continuamente la calidad del software (Cabrera, Vaca, & Piattini, 2019).

Fig. 13 División de la norma ISO/IEC 2500n

1.5.1. ISO/IEC 25023

La norma ISO/IEC 25023, también conocida como SQuaRE (Software Quality Requirements and Evaluation), es un estándar internacional que proporciona directrices para la evaluación y medición de la calidad del software. Esta norma está diseñada para ayudar a las organizaciones a definir, establecer y evaluar los requisitos de calidad del software. Se basa en un enfoque sistemático y estructurado para evaluar la calidad del software. Proporciona directrices para la definición de métricas de calidad, la selección de técnicas de evaluación apropiadas y la interpretación de los resultados obtenidos (iso.org, 2022). A continuación, se presentan las características de la norma ISO/IEC 25023.

Tabla 3 Características y Sub-características del estándar ISO/IEC 2023

CARÁCTERÍSTICA	SUB-CARÁCTERÍSTICAS
Funcionalidad	Adecuación: capacidad del software para proporcionar las funciones requeridas por el usuario.
	Exactitud: precisión y corrección de los resultados y salidas del software.
	Interoperabilidad: capacidad del software para interactuar y funcionar con otros sistemas
	Seguridad de acceso: protección de la información y los datos del software contra accesos no autorizados.
Confiabilidad	Madurez: capacidad del software para evitar fallos o errores.
	Tolerancia a fallos: capacidad del software para mantener un nivel de rendimiento aceptable incluso en caso de fallos.
	Capacidad de recuperación: capacidad del software para recuperarse rápidamente después de un fallo.
	Cumplimiento: capacidad del software para cumplir con los estándares, regulaciones y requisitos legales aplicables.
Usabilidad	Comprensibilidad: facilidad de comprensión del software para los usuarios.
	Aprendizaje: facilidad de aprendizaje del software para los usuarios.
	Operabilidad: facilidad de operación del software para los usuarios.
	Atractivo visual: diseño atractivo y agradable del software.
Eficiencia	Comportamiento en el tiempo: rendimiento y tiempo de respuesta del software.
	Utilización de recursos: eficiencia en el uso de los recursos del sistema, como CPU, memoria y espacio en disco.
	Capacidad: capacidad del software para manejar grandes volúmenes de datos o transacciones.
Mantenibilidad	Analizabilidad: facilidad de comprensión y análisis del software para el mantenimiento y la modificación.
	Cambiabilidad: facilidad de realizar cambios y modificaciones en el software.
	Estabilidad: capacidad del software para evitar efectos no deseados al realizar cambios.
	Testabilidad: facilidad para realizar pruebas y validar el software.
Portabilidad	Adaptabilidad: capacidad del software para adaptarse a diferentes entornos y plataformas.
	Instalabilidad: facilidad de instalación del software en diferentes entornos.
	Reemplazabilidad: facilidad de reemplazar el software existente por otro software.

1.5.1.1. Característica de Eficiencia de la ISO/IEC 25023

La característica de Eficiencia, dentro de la norma ISO/IEC 25023, se refiere a la capacidad general del software para realizar sus funciones de manera eficiente, minimizando el consumo de recursos y maximizando el rendimiento. Esta característica se enfoca en optimizar la utilización de los recursos disponibles, como tiempo, memoria, CPU, almacenamiento y ancho de banda de red.

Algunos aspectos relevantes de la característica de Eficiencia incluyen:

- Rendimiento: Implica la capacidad de realizar tareas en un tiempo razonable y cumplir con los requisitos de tiempo de respuesta establecidos.
- Utilización de recursos: Un software eficiente debe hacer uso del CPU, la memoria, el almacenamiento y el ancho de banda de red de manera eficaz, minimizando el consumo innecesario y evitando la sobrecarga del sistema.
- Escalabilidad: Un software eficiente debe ser capaz de escalar adecuadamente, manteniendo su rendimiento óptimo incluso en entornos de alta demanda.
- Optimización de algoritmos y estructuras de datos: Implica seleccionar y diseñar algoritmos que realicen las operaciones de manera más rápida y utilizar estructuras de datos que permitan un acceso y manipulación eficientes de la información.
- Optimización de consultas y acceso a bases de datos: En el caso de aplicaciones que interactúan con bases de datos, la eficiencia implica optimizar las consultas y el acceso a los datos.

CAPITULO II

DESARROLLO

La fase de desarrollo del proyecto se divide en dos partes bajo la guía de marco de trabajo SCRUM, la primera parte consta de crear un servidor que envuelva el servicio REST de la API de NAPSTER el cual brindará servicios y recursos a través de una API GraphQL. La segunda parte cuenta con un cliente que consumirá, tanto los recursos y servicios de la API GraphQL y API REST de NAPSTER para probar la funcionalidad de ambas API.

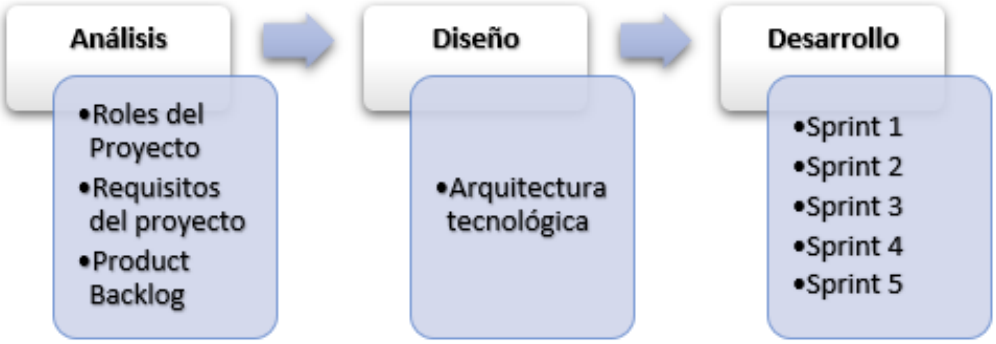


Fig. 14 Estructura del desarrollo del proyecto

2.1. Análisis del Proyecto

2.1.1. Roles del Proyecto

Para el desarrollo del envoltorio API GraphQL de NAPSTER y su prueba de funcionalidad, se ha establecido un grupo de trabajo basándose en el marco de trabajo SCRUM; en la siguiente Tabla 4 se muestra el rol que cumple cada miembro del equipo.

Tabla 4 Roles del grupo de trabajo

Miembros	Descripción	Rol que cumple
PhD. Antonio Quiña	Director del presente Trabajo de Grado y Docente de la Carrera de Software de la Universidad Técnica del Norte	Propietario del Producto (Product Owner)
Sr. Daniel Guamán	Estudiante de la Carrera de Software de la Universidad Técnica del Norte	Jefe del Proyecto (Scrum Master).
Sr. Daniel Guamán	Estudiante de la Carrera de Software de la Universidad Técnica del Norte	Equipo de Desarrollo (Development Team).

2.1.2. Requisitos del Proyecto

2.1.2.1 Historias de Usuario

Para los requisitos del envoltorio y su prueba de concepto se crearon Historias de usuario por parte del propietario del producto, a estas Historias de Usuario se las considera como una herramienta en la metodología SCRUM.

Tabla 5 Primera Historia de Usuario

N.º: HU - 01	**Historia de Usuario 1**
	Nombre de HU: Estructura del Software
	Usuario: ---
	Tiempo estimado: 48 hrs.
	Dependencia: Ninguna

Objetivo:

Realizar pruebas de consumo a la API de NAPSTER, esencial para determinar las capacidades del envoltorio GraphQL y verificar los diversos servicios proporcionados por el gestor. Esto permitirá establecer una sólida base técnica para el desarrollo del software en cuestión.

Aceptaciones

- Para la autentificación de la API REST de Napster se debe establecer un formato entendible para su posterior desarrollo.
- Conocer el resto de los servicios que ofrece la API de NAPSTER que sean necesarios para el funcionamiento del proyecto.

Tabla 6 Segunda Historia de Usuario

N.º: HU - 02	**Historia de Usuario 2**
	Nombre de HU: Gestión de acceso para los recursos de la API NAPSTER.
	Usuario: Desarrollador Backend
	Tiempo estimado: 10 hrs.
	Dependencia: HU-01

Objetivo:

Como desarrollador backend, quiero realizar el acceso a la API de NAPSTER, esto se realiza al obtener un token temporal al autentificar la cuenta de Napster.

Observación:

- Los campos necesarios para la obtención del token de artistas esta detallado en la documentación de Napster: https://developer.prod.napster.com/api/v2.2#auth-implicit

Aceptaciones

- El ingreso de los parámetros para la obtención del token generará una respuesta satisfactoria y un texto con el token necesario para hacer uso de los recursos de la API napster.
- Visualización del token obtenido en la arquitectura REST y GraphQL.

Tabla 7 Tercera Historia de Usuario

N.º: HU - 03	**Historia de Usuario 3**
	Nombre de HU: Consulta del endpoint de 'Artists'
	Usuario: Desarrollador Backend
	Tiempo Estimado: 12 hrs.
	Dependencia: HU-02

Objetivo:

Desarrollar un Backend, el cual realice una consulta al endpoint (API) de artistas en las arquitecturas REST y GraphQL. Esta consulta debe obtener un listado de los artistas registrados en el servidor de NAPSTER.

Observación:

- La estructura necesaria para la obtención del endpoint artists se encuentra detallada en su documentación: https://developer.prod.napster.com/api/v2.2#artists
- El url específico del endpoint para su uso es: https://api.napster.com/v2.2/artists/top

Aceptaciones:

- El ingreso correcto de los parámetros de la API permitirá generar un listado de todos los artistas registrados en el servidor de Napster.
- Mostrar el tiempo de consulta.

Tabla 8 Cuarta Historia de Usuario

N.º: HU - 04	**Historia de Usuario 4**
	Nombre de HU: Consulta del endpoint 'Tracks' por artista.
	Usuario: Desarrollador Backend
	Tiempo estimado: 12 hrs.
	Dependencia: HU-02- HU-03

Objetivo:

Desarrollar un Backend, que realice una consulta al endpoint (API) de tracks en las arquitecturas de REST y GraphQL. Esta consulta se la realiza a través de 2 pasos:

- Acceso al endpoint artists y,
- Acceso al endpoint tracks

Observación:

- La estructura necesaria para la obtención del endpoint tracks se encuentra detallada en su documentación: https://developer.prod.napster.com/api/v2.2#tracks

- El url específico del endpoint para su uso es: https://api.napster.com/v2.2/artists/**ID** /tracks/top
- El campo necesario para la obtención del listado de tracks es el 'id' del artista, de tipo String.

Aceptaciones:

- El ingreso correcto de los parámetros de la API permitirá generar un listado de todas las pistas(tracks) del artista requerido.
- Mostrar el tiempo de consulta.

Tabla 9 Quinta Historia de Usuario

N.º: HU - 05	**Historia de Usuario 5**
	Nombre de HU: Consulta del endpoint 'Albums' por artista
	Usuario: Desarrollador
	Tiempo Estimado: 12 hrs.
	Dependencia: HU-02, HU-03, HU-04

Objetivo:

Desarrollar un Backend, que realice una consulta al endpoint (API) de Albums en las arquitecturas de REST y GraphQL. Esta consulta se la realiza a través de 3 pasos:

- Acceso al endpoint artists,
- Acceso al endpoint tracks y,
- Accesos al endpoint albums.

Observación:

- La estructura necesaria para la obtención del endpoint de albums se encuentra detallada en su documentación: https://developer.prod.napster.com/api/v2.2#albums
- El url especifico de la API para su uso es: https://api.napster.com/v2.2/artists/**ID** /albums
- El campo necesario para la obtención del listado de Albums es el 'id' del artista (obtenido del endpoint tracks), de tipo String.

Pruebas de aceptación:

- El ingreso correcto de los parámetros de la API permitirá generar un listado de todos los albums del artista.
- Mostrar el tiempo de consulta.

Tabla 10 Sexta Historia de Usuario

N.º: HU - 06	**Historia de Usuario 6**
	Nombre de HU: Menú de consulta de servicios REST y GraphQL
	Usuario: Desarrollador Frontend
	Tiempo estimado: 10 hrs.

Dependencia: HU-02, HU-03, HU-04, HU-05
Objetivo: Desarrollar el frontend, el cual constará con un menú que enliste los servicios de REST y GraphQL que se requiere consultar.
Observación: ▪ El menú mostrará una lista con los servicios que requiero consultar: ~ Consumir REST ~ Consumir REST con cache ~ Consumir GraphQL ~ Consumir GraphQL con cache
Aceptación: ▪ Mostrar el tiempo de consulta de cada caso.

2.1.3. Product Backlog

El Product Backlog que se detalla en la Tabla 11 contiene las Historias de usuario ya definidas de acuerdo a la prioridad definida por el Product Owner.

Tabla 11 Historias de Usuario definidas

			PRODUCT BACKLOG	
PRIORIDAD	N.º HU	NOMBRE DE HU	ROL ESTABLECIDO	TIEMPO ESTIMADO
1	HU-01	Estructura del Software	---	48
2	HU-02	Gestión de acceso para los recursos de la API NAPSTER.	Desarrollador Backend	12
3	HU-03	Consulta del endpoint de ‘Artists’	Desarrollador Backend	12
4	HU-04	Consulta de ‘Tracks’ por artista.	Desarrollador Backend	12
5	HU-05	Consulta de ‘Albums’ por artista.	Desarrollador Backend	12
6	HU-06	Menú de consulta de servicios REST y GraphQL	Desarrollador Frontend	10

2.2. Diseño

En el desarrollo del proyecto se cuenta con varios Sprints, contrariamente, para establecer el diseño del proyecto, se realizó un Sprint extra llamado Sprint 0 en donde se definió la arquitectura tecnológica del proyecto.

Tabla 12 Detalle Sprint 0

N.º de Sprint	Fecha de Inicio	Fecha de Fin	Tiempo de duración
Sprint 0	10/abril/2023	24/abril/2023	20 hrs.

Planificación Sprint 0

Reunión para la planificación y definición de la arquitectura tecnológica del proyecto.

Fecha de Inicio: 10 de abril de 2023

Objetivo: Definir la arquitectura tecnológica del proyecto.

Miembros:

- Product Owner
- SCRUM Master y
- Team Developer

Tabla 13 Planificación del primer Sprint

Fase	Objetivo	Tiempo estimado
Diseño	Definir la arquitectura tecnológica del proyecto	5 hrs
Planificación	Detallar las tareas a realizarse para el actual sprint	4 hrs
Revisión	Revisar los resultados que se obtuvieron del sprint.	1 hrs
	Total, de horas	10 hrs

2.2.1. Arquitectura tecnológica

La arquitectura tecnológica consta de dos partes, un servidor (envoltorio), y un cliente (prueba del concepto). Tanto el servidor como el cliente están desarrollados en Node.js con la implementación de tecnologías como GraphQL, ApolloClient y la API de napster.

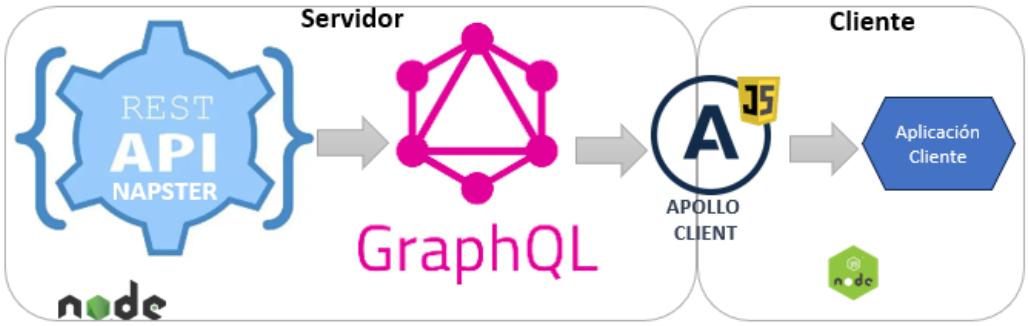


Fig. 15 Arquitectura del proyecto

2.3. Desarrollo

Para la ejecución del proyecto se recurrió a hacerlo mediante Sprints, el tiempo de duración de cada uno de estos Sprints es de unas 40 horas de dos semanas laborables cada uno.

Tabla 14 Detalle General de la duración de los SPRINT

Sprint	Fecha Inicio	Fecha Fin	Duración
Sprint 1	01-may-2023	15-may-2023	40 hrs.
Sprint 2	15-may-2023	29-may-2023	40 hrs.
Sprint 3	29-may-2023	12-jun-2023	40 hrs.
Sprint 4	12-jun-2023	26-jun-2023	40 hrs.

2.3.1. SPRINT 1

- Planificación del Sprint 1

  En el primer sprint se realizan las tareas de exploración de la API de Napster para obtener un insumo con información del uso de los servicios de la API, que servirá como material para el desarrollo del proyecto.

a) Planificación de la Reunión.

   Fecha de Inicio: 01 de mayo de2023

   Asistentes de la Reunión:

   - Product Owner
   - Scrum Master
   - Team Development

   Objetivo de la Reunión: Planificación del Sprint 1

b) Sprint backlog

Tabla 15 Sprint 1

		SPRINT BACKLOG	
N.º HU	**Fase**	**Tarea**	**Tiempo estimado**
HU-01	Analizar	Leer la documentación del API de Napster para poder conocer que requisitos son necesarios, y así definir los recursos que se van a analizar.	6 hrs
	Analizar	Analizar cuál es el proceso con que el que se autentifica la API y así obtener acceso a los EndPoints de la API	3 hrs
	Pruebas	Pruebas de los EndPoints en Postman.	5 hrs

	Documentación	Realizar un documento con la información que se necesita para realizar la autentificación.	4 hrs
	Pruebas	Consumir los recursos de la API de napster con una herramienta que permita peticiones api.	6 hrs
		Pruebas de los EndPoints en Postman	4 hrs
	Documentación	Realizar un documento el cual contenga la información de los EndPoints.	4hrs
Otros	Planificación	Detalla las tareas a realizarse en el sprint actual.	4 hrs
	Revisión	Revisar los resultados del sprint	3 hrs
	Retrospectiva	Analizar los resultados del sprint	1 hrs
		Total, de horas	40 hrs

- **Revisión**

  El cumplimiento del primer sprint estableció cual es la estructura del desarrollo de la API. La información que se muestra a continuación es el resultado de haber hecho una revisión y análisis minucioso a la documentación que ofrece la plataforma 'developer napster', la información conseguida sirve para establecer el funcionamiento de la API GraphQL.

  a) Reunión de revisión del sprint 1.

  Fecha de revisión: 15 de mayo de 2023

  Asistentes de la reunión:

  - Product Owner
  - Scrum Master
  - Team development

  Objetivo de la reunión: Revisar cual es el incremento del proyecto.

  b) Incremento del producto potencialmente entregable

  URL base de la API de Napster: https://api.napster.com/

  Autentificación

  La API de Napster tiene una autentificación la cual se realiza mediante el protocolo de autorización OAuth2, el cual posibilita que haya autorizaciones de las solicitudes a las API mediante tokens de acceso. Actualmente napster admite dos flujos para realizar la autentificación.

Tabla 16 Flujos de autentificación Napster

Flujo	Descripción

Implicit	En este flujo, su aplicación dirige al miembro (en un navegador web) a un formulario de inicio de sesión alojado en Napster, donde se le solicita que autorice su aplicación:
Password Grant	En este flujo, intercambia el nombre de usuario y la contraseña de un miembro (que su aplicación obtiene directamente del miembro) por un token de acceso y un token de actualización.

ENDPOINT ARTISTS

- **Documentación del recurso:** https://developer.prod.napster.com/api/v2.2#artists
- **URI:** /artists/top
- **Atributos principales**

Tabla 17 Principales atributos del endpoint artists

type	String	Tipo de metadata: artists, albums, Tracks, genres, playslist, Stations, editorial posts, tags, search
id	String	Identificador del artista, el cual es establecido en el servidor cuando se crea el recurso, este no se puede modificar.
href	String	Url que dirige directamente al recurso (artists) que está siendo consultado.
name	String	Nombre del artists
shortcut	String	Nombre corto del artista
blurbs	Array	Propaganda realizada por el artista
bios	Array	Información general del artista que detalla el autor y fecha de publicación de la biografía del artista.
albumGroups	Array	Albums de música que tiene el artista y que constan en la plataforma napster.
links	Array	Url que redirige a los albums, imágenes, pistas del recurso consultado.

- **Operación GET**

Tabla 18 Parámetros(opcionales) de la operación GET - artists

GET	
Parámetros:	Descripción
limit	Indica el límite de recursos que puedo consultar: de 1 a 200, default 20
offset	Indica la paginación del EndPoint
range	El rango puede obtener los siguientes valores: ‘day, week, month, year and life’, default month
id	Id para obtener un artista en específico.

images	Devuelve una lista con las imágenes con licencia del artista
albums	Devuelve la discografía completa del artista.
albums/top	Devuelve los álbumes más populares del artistas
Albums/new	Devuelve los álbumes nuevos del artista
tracks	Devuelve la lista de pistas del artista
posts	Devuelve una lista de publicaciones del blog escrita por el equipo editorial.

Ejemplo del consumo del endpoint artists.

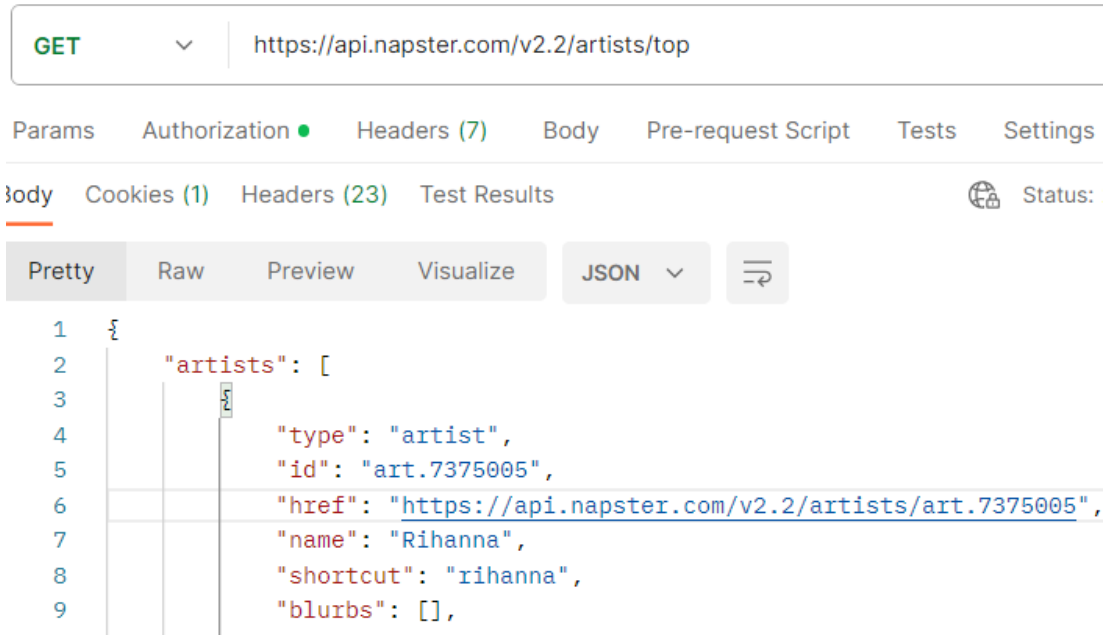


Fig. 16 Ejecución del endpoint artists en Postman

ENDPOINT TRACKS

- **URL del recurso:** https://developer.prod.napster.com/api/v2.2#tracks
- **Atributos principales**

Tabla 19 Principales atributos del endpoint tracks

type	string	Tipo de metadata: artists, albums, Tracks, genres, playslist, Stations, editorial posts, tags, search
id	string	Identificador de la pista, establecido por el servidor al momento de crear el recurso, no es modificable.
disc	int	Indica cuantos discos tiene la pista
href	string	Url que dirige directamente al recurso (artists) que está siendo consultado.
name	string	Nombre de la pista

shortcut	string	Nombre corto del artista y la pista
isrc	string	Indica el código ISRC de identificación de la pista.
blurbs	array	Propaganda realizada por el artista
artistid	string	Indica el id del artista al que pertenece la pista consultada.
artistname	string	Indica el nombre del artista a quien pertenece la pista consultada.
albumname	string	Indica el nombre del álbum al que pertenece la pista
formats	array	Indica el tipo de formato que tiene la pista
albumid	string	Indica el id del álbum al que pertenece la pista
contributors	array	Indica los contribuidores para la creación de la pista.
links	array	Url que redirige a los albums, imágenes, pistas del recurso consultado.

- Operación GET

Tabla 20 Parámetros(opcionales) de la operación GET - tracks

GET	
Parámetros:	Descripción
limit	Indica el límite de recursos que puedo consultar: de 1 a 200, default 20
offset	Indica la paginación del EndPoint
range	El rango puede obtener los siguientes valores: ‘day, week, month, year and life’, default month
id	Id para obtener una pista en específico del artista.

Ejemplo del consumo del endpoint tracks.

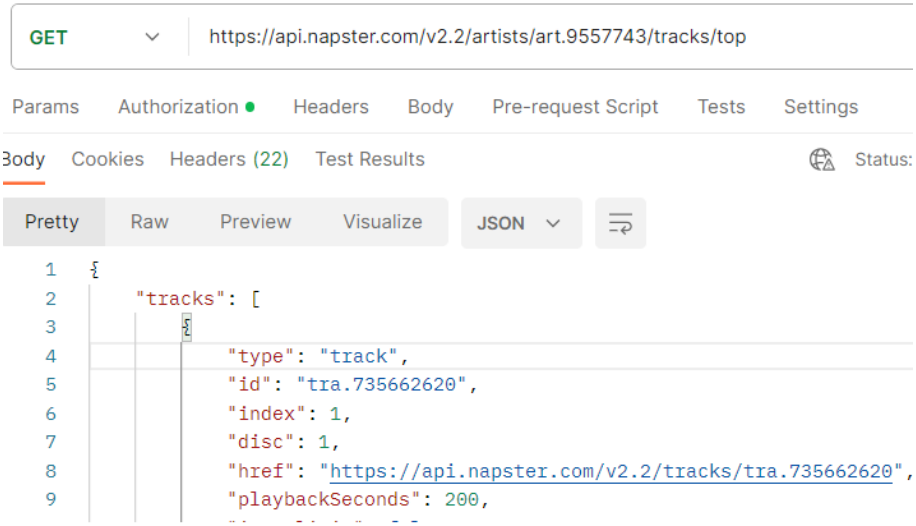


Fig. 17 Ejecución del endpoint tracks en Postman

ENDPOINT ALBUMS

- **URL del recurso:** https://developer.prod.napster.com/api/v2.2#albums
- **Atributos principales**

Tabla 21 Principales atributos del endpoint albums

type	string	Tipo de metadata: artists, albums, tracks, genres, playslist, Stations, editorial posts, tags, search
Id	string	Identificador del álbum, el cual es establecido en el servidor cuando se crea el recurso, este no se puede modificar.
shortcut	string	Nombre corto del del álbum
upc	int	Indica el código del álbum como producto.
href	string	Url que dirige directamente al recurso (artists) que está siendo consultado.
name	string	Nombre del álbum
released	string	Indica la fecha en que fue liberado el álbum.
blurbs	array	Propaganda realizada por el artista
label	string	Muestra una etiqueta con información del álbum.
copyright	string	Indica el propietario de los derechos del álbum.
artistName	string	Nombre del artista
links	array	Url que redirige a los albums, imágenes, pistas del recurso consultado.

- Operación GET

Tabla 22 Parámetros(opcionales) de la operación GET - albums

GET	
Parámetros:	Descripción
limit	Indica el límite de recursos que puedo consultar, default 20
offset	Indica la paginación del EndPoint
range	El rango puede obtener los siguientes valores: 'day, week, month, year and life', default month
id	Id para obtener un álbum en específico del artista.

Ejemplo del consumo del endpoint albums.

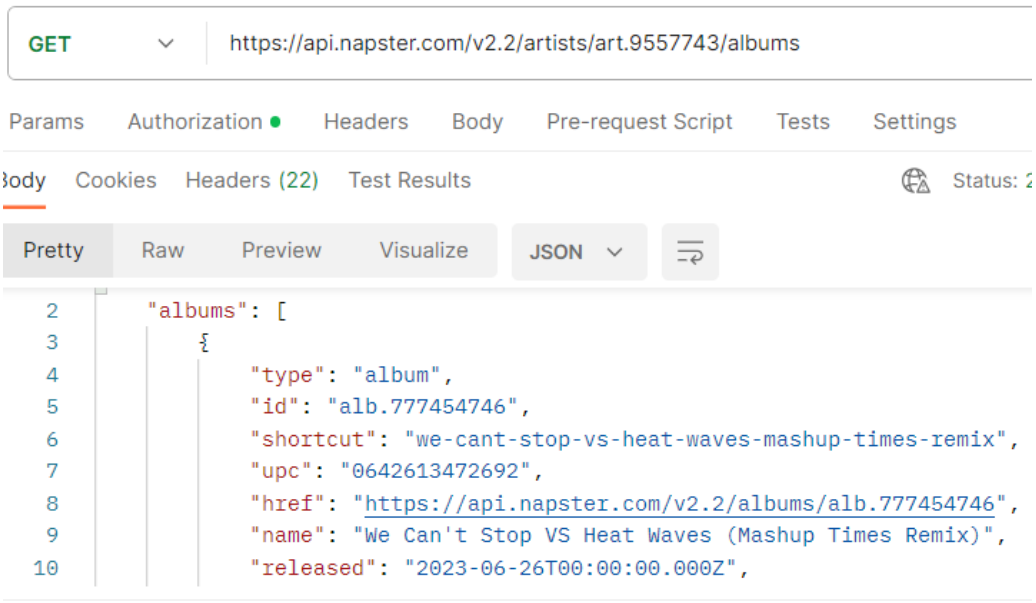


Fig. 18 Ejecución del endpoint albums en Postman

- Retrospectivas Sprint 1

a) Reunión Retrospectiva

Fecha de Reunión: 15 de mayo de 2023

Asistentes de la reunión:

- Product Owner
- Scrum Master
- Team development

Objetivo: Analizar los aciertos, que errores se obtuvieron y que mejoras se puede realizar.

Tabla 23 Retrospectiva Sprint 1

Retrospectiva

Aciertos: ¿Qué salió bien del Sprint?	Los recursos(endpoints) proporcionados por la API son claros y fáciles de entender para el desarrollo del envoltorio.
Errores: ¿Qué no salió bien del Sprint?	Existieron recursos que tardaban más de lo necesario a la hora de realizar una consulta.
Mejoras: ¿Qué mejoras se pueden implementar?	Se puede mejorar la estructura del proceso de investigación con el objetivo de obtener recursos de mayor eficiencia.

2.3.2. SPRINT 2

- Planificación Sprint-2

a) Planificación de Reunión

Fecha de Inicio: 15 de mayo de 2023

Asistentes de la reunión:

- Product owner
- Scrum Master
- Team development

Objetivo de la reunión: Planificación del Sprint 2

b) Sprint Backlog

Tabla 24 Sprint 2

		SPRINT BACKLOG	
N.º. HU	**Fase**	**Tarea**	**Tiempo estimado**
OTROS	Desarrollar	Crear un proyecto Node, e instalar todas las dependencias necesarias para el desarrollo de servidor.	3 hrs
HU-02	Desarrollar	Crear un query obtenertoken para tener acceso a los recursos de la API.	4 hrs
		Implementar un resolver para el consumo del token	4 hrs
	Pruebas	Prueba de obtención de token	2 hrs
HU-03	Desarrollar	Crear un type ResponseArtists en el schema en GraphQL con los campos del endpoint correspondiente.	6 hrs
		Implementar los resolvers para la operación listar del recurso artists	4 hrs
		Crear un tipo query con la operación Get para acceder al tipo ResponseArtists definido en el esquema graphql y al resolver del recurso artista.	6 hrs

	Pruebas	Pruebas de concepto de implementación.	3 hrs
Otros	Planificación	Detalla las tareas que deben realizarse.	4 hrs
	Revisión	Revisar los resultados	3 hrs
	Retrospectiva	Analizar los resultados	1 hrs
		Total Horas	40 hrs

- Revisión Sprint-2

a) Reunión revisión sprint 2

Fecha de reunión: 29 de mayo de 2023

Asistentes de la reunión:

- Product owner
- Scrum Master
- Team development

Resultado: Revisión del cumplimiento de las tareas del sprint.

b) Desarrollo del proyecto potencialmente entregable

Obtener token

Para obtener un token que permita el acceso a los recursos se realiza un query que acepta la consulta ‘obtenertoken’ se obtiene como respuesta un token tipo String.

Fig. 19 Estructura de Obtener token - GraphQL

ENDPOINT ARTISTS

Listar artists

Para listar los artistas se especifica los parámetros de entrada que necesitemos, todo esto para obtener un listado personalizado, además de que en las variables podemos ingresar el límite de datos que deseemos; llamando a la consulta ‘obtenerArtist’ se obtiene como resultado un tipo ‘artists’.

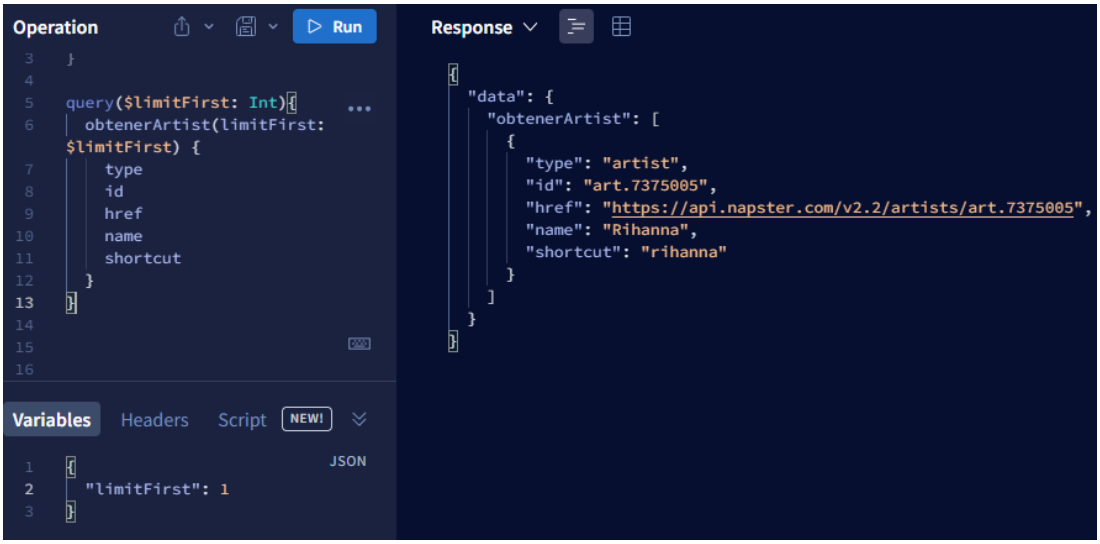


Fig. 20 Estructura de Listar el endpoint artists en GraphQL

- Retrospectiva Sprint 2

a) Reunión de retrospectiva

Fecha de reunión: 29 de mayo de 2023

Asistentes de la reunión:

- Product owner
- Scrum Master
- Team development

Objetivo de la reunión: Analizar cuáles son los aciertos, que errores se encontraron, y que mejoras se pueden realizar.

Tabla 25 Retrospectiva Sprint 2

Retrospectiva	
Aciertos: ¿Qué salió bien del Sprint?	Las consultas fueron fáciles de realizar
Errores: ¿Qué no salió bien del Sprint?	Se subestimo el tiempo de desarrollo, hubo partes del esquema graphql que tuvieron que ser corregidas.
Mejoras: ¿Qué mejoras se pueden implementar?	Se puede implementar ApolloClient y restlink al envoltorio para que, tanto rest y graphql tengan el mismo diseño arquitectónico.

2.3.3. SPRINT 3

- Planificación del Sprint 3

a) Reunión Planificación Sprint 3

Fecha de inicio: 29 de mayo de 2023

Asistentes de la reunión:

- Product owner
- Scrum Master
- Team development

Objetivo de la reunión: Planificación del Sprint 3

b) Sprint backlog

Tabla 26 Sprint-3

			SPRINT BACKLOG
N.º de HU	**Fase**	**Tarea**	**Tiempo estimado**
HU-04	Desarrollar	Crear un type ResponseTracks en el schema en GraphQL con los campos correspondientes del endpoint	5 hrs
		Implementar los resolvers para la operación listar del recurso tracks	5 hrs
		Crear un tipo query con la operación Get para acceder tanto al tipo ResponseTracks definido en el esquema graphql y al resolver del recurso tracks.	5 hrs
	Prueba	Pruebas de funcionamiento del proyecto.	1 hrs
HU-05	Desarrollar	Crear un type ResponseAlbums en el schema en GraphQL con los campos del endpoint correspondiente.	5 hrs
		Implementar los resolvers para la operación listar del recurso albums	5 hrs
		Crear un tipo query con la operación Get para acceder tanto al tipo ResponseAlbums definido en el esquema graphql y al resolver del recurso albums.	5 hrs
	Pruebas	Pruebas de funcionamiento del proyecto.	1 hrs
Otros	Planificación	Detallas que tareas se realizaran en el sprint	4 hrs
	Revisión	Revisar los resultados	3 hrs
	Retrospectiva	Analizar los resultados	1 hrs
		Total, de horas	40 hrs

- Revisión sprint 3

a) Reunión revisión sprint 3

Fecha de reunión: 12 de junio de 2023

Asistentes de la reunión:

- Product owner
- Scrum Master
- Team development

Resultado de la reunión: Revisión del avance del desarrollo del proyecto..

b) Desarrollo del proyecto potencialmente entregable

ENDPOINT TRACKS

Listar tracks

Para listar los tracks se debe especificar los parámetros de entrada que necesitemos ocupar, en este caso se necesitan los parámetros tanto del recurso ‘artists’ y del recurso ‘tracks’ todo esto para obtener un listado personalizado que contenga la información de ambos recursos, además de que en las variables podemos ingresar el límite de datos que deseemos, en este caso ingresamos el límite para el primer endpoint como para el segundo; llamando a la consulta ‘obtenerArtist’ se obtiene como resultado un tipo ‘artists’ con datos del recurso tracks.

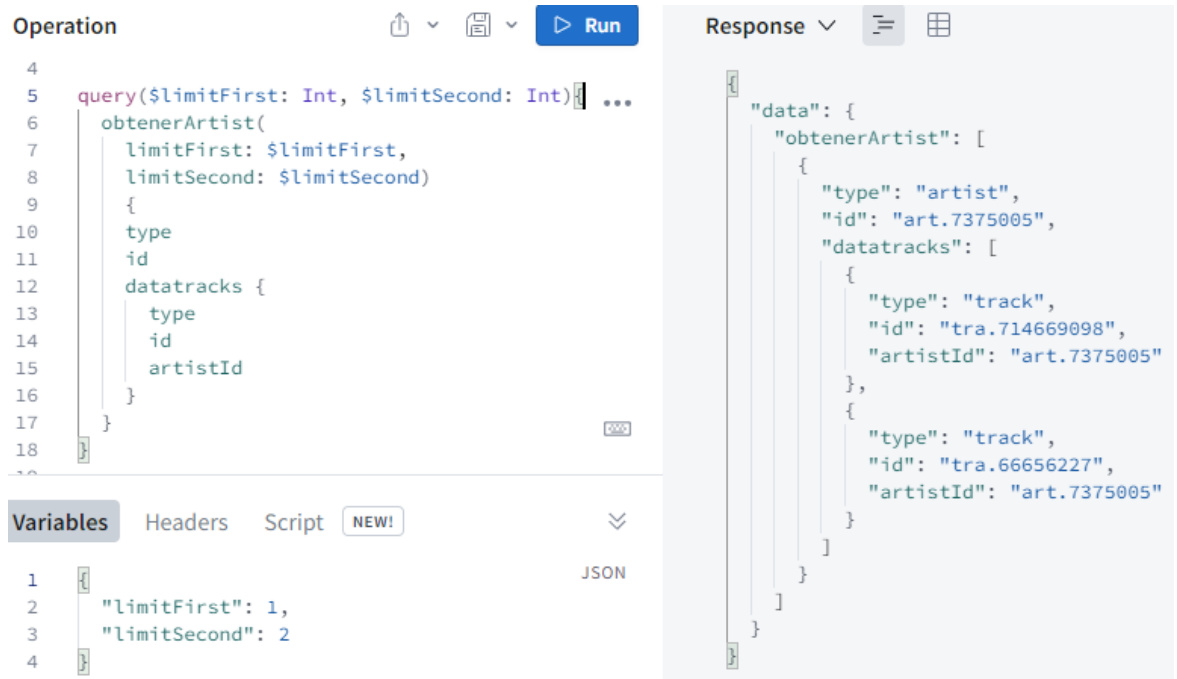


Fig. 21 Estructura de Listar el endpoint tracks en GraphQL

RECURSO ALBUMS

Listar albums

Para listar los albums se debe especificar los parámetros de entrada que necesitemos ocupar, en este caso se necesitan los parámetros de tres recursos, el recurso ‘artists’, ‘tracks’ y ‘albums’ todo esto para obtener un listado personalizado con la información de los tres endpoint, además de que en las variables podemos ingresar el límite de datos

que deseemos, en este caso ingresamos el límite para el primer, segundo y tercer; llamando a la consulta 'obtenerArtist' se obtiene como resultado un tipo 'artists' con datos de los recursos tracks y albums.

Fig. 22 Estructura de Listar el endpoint albums en GraphQL

- Retrospectiva Sprint-3

a) Reunión de retrospectiva

Fecha de reunión: 12 de junio de 2023

Asistentes de la reunión:

- Product Owner
- Scrum Master
- Team Development

Objetivo de la reunión: Analizar los aciertos, los errores que hubo, y que mejoras se pueden realizar.

Tabla 27 Retrospectiva Sprint 3

Retrospectiva	
Aciertos: ¿Qué salió bien del Sprint?	Las consultas por cada recurso resultaron fáciles de realizar.
Errores: ¿Qué no salió bien del Sprint?	Se subestimo el tiempo de desarrollo, hubo partes del esquema graphql que tuvieron que ser corregidas.

Mejoras: ¿Qué mejoras se pueden implementar?	Mayor comprobación y comprensión de tecnologías y herramientas.

2.3.4. SPRINT 4

- Planificación Sprint 4

a) Reunión de planificación.

Fecha de inicio: 12 de junio de 2023

Asistentes de la reunión:

- Product Owner
- Scrum Master
- Team Development

Objetivo de la reunión: Planificar el desarrollo del sprint 4

b) Sprint backlog

Tabla 28 Sprint 4

N.º: de HU	Fase	Tarea	Tiempo estimado
OTROS	Desarrollar	Crear un proyecto en Node, e instalar las dependencias que se requieran para poder consumir el servidor API-REST y API-GraphQL	5 hrs
HU-06	Desarrollar	Crear un menú en el que se visualice las opciones de consulta: • Consumir Rest • Consumir Rest con Cache • Consumir GraphQL • Consumir GraphQL con Cache	5 hrs
		Crear un segundo menú donde se visualice el caso de prueba o recurso que voy a consultar. • Caso de Prueba 1 (Artists) • Caso de Prueba 2 (Tracks) • Caso de Prueba 3 (Albums)	5 hrs
		Implementar los resolvers para la operación listar de los recursos Rest e implementar la conexión al envoltorio Graphql	5 hrs
	Pruebas	Pruebas de funcionamiento del proyecto.	1 hrs
Eventos	Planificación	Detallar las tareas que se realizarán en el sprint	4 hrs
	Revisión	Revisar los resultados	3 hrs
	Retrospectiva	Análisis de los resultados	1 hrs
		Total, de horas	40 hrs

- Revisión del sprint-4

a) Reunión de revisión

Fecha de reunión: 26 de junio de 2023

Asistentes de la reunión:

- Product Owner
- Scrum Master
- Team Development

Resultado: Revisión del cumplimiento de las tareas del sprint.

b) Desarrollo del proyecto potencialmente entregable

MENU PRINCIPAL

En este menú de acceso, se pude consumir los servicios tanto de Rest como de Graphql, estos servicios pueden ser con cache o sin cache para ver cómo funciona cada consulta.

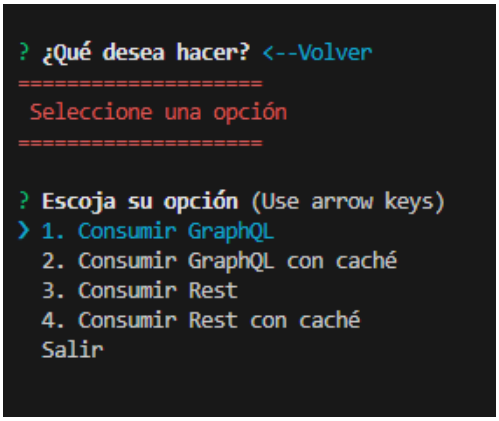


Fig. 23 Menú Principal del Proyecto

MENU SECUNDARIO

En este menú podemos seleccionar el recurso que deseamos consumir, aquí podemos llamar a los resolver y al envoltorio, dando como resultado el tiempo de consulta que tarda en responder el endpoint.

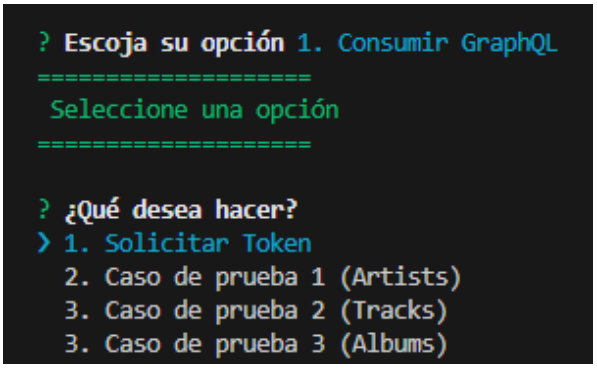


Fig. 24 Menú Secundario del proyecto - Consumo Casos de Uso

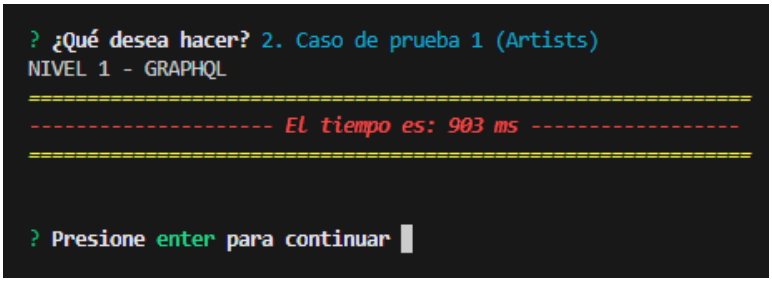


Fig. 25 Ejemplo de tiempo de Consumo

- Retrospectiva sprint-4

b) Reunión retrospectiva

Fecha de reunión: 26 de junio de 2023

Asistentes de la reunión:

- Product Owner
- Scrum Master
- Team Development

Objetivo: Analizar los aciertos, que errores hubo, y que mejoras se pueden realizar.

Tabla 29 Retrospectiva Sprint 4

Retrospectiva	
Aciertos: ¿Qué salió bien del Sprint?	Se conectó correctamente al envoltorio API Graphql y a los resolver de la API Rest
Errores: ¿Qué no salió bien del Sprint?	La duración del 'code' para la obtención del token, es de solamente 24 horas, como consecuencia se debe actualizar el parámetro code para el funcionamiento del proyecto.
Mejoras: ¿Qué mejoras se pueden implementar?	Se puede implementar un parámetro extra que convierta los milisegundos en minutos si así es necesario.

2.4. Pruebas de Aceptación

Una vez finalizado con el desarrollo del proyecto a través de diversas pruebas de aceptación, se debe verificar el funcionamiento del proyecto.

Tabla 30 Pruebas de Aceptación.

N.º de HU	Nombre de Historia de usuario	Tarea	ACEPTACIÓN	
			SI	NO

HU-01	Estructura del software	Proceso de autentificación de la API Napster.	X	
		Documentación de los recursos principales y secundarios de Napster.	X	
HU-02	Gestión de acceso para los recursos de la API NAPSTER.	Acceso al token de autentificación para la API	X	
HU-03	Consulta del endpoint de ‘Artists’	Listar Artists	X	
HU-04	Consulta de ‘Tracks’ por artista.	Listar Tracks	X	
HU-05	Consulta de ‘Albums’ por artista.	Listar Albums	X	
HU-06	Menú de consulta de servicios REST y GraphQL	Mostrar consumo de token	X	
		Mostrar servicios y recursos	X	
		Mostrar tiempos de consulta	X	

CAPITULO III

VALIDACION E INTERPRETACIÓN DE RESULTADOS

Para este capítulo se la validación e interpretación de resultados se lo realiza mediante un experimento controlado, todo esto basándose en la guía de Wohlin del año 2012 “Experimentación en Ingeniería de Software”. El experimento consiste en comparar las arquitecturas REST y GRAPHQL, con casos de uso que se presentaran más a continuación. Para esto se planteó la siguiente pregunta de investigación ***PI***: ¿En qué situaciones es más recomendable utilizar arquitecturas híbridas que integren tanto REST como GraphQL al desarrollar aplicaciones web o servicios API?, para esto se utilizó la norma ISO/IEC 25023 con respecto a la característica de eficiencia de calidad de productos de software.

3.1. Entorno experimental

3.1.1. Objetivo

Realizar una comparación del desempeño de la API-GraphQL y de la API-REST de la plataforma de música napster en una ambiente local(localhost), con relación a la calidad de productos de software.

3.1.2. Factores y tratamientos

El factor que se investiga es la arquitectura de software, más concretamente el despliegue de las APIs de los servicios GraphQL y REST. Los tratamientos que se usaron son los siguientes:

- Arquitectura REST para el despliegue de servicios API.
- Arquitectura GraphQL para envoltorios para el despliegue de servicios API.

En la arquitectura se desplego su respectiva API con y sin cache.

3.1.3. Variable

La variable independiente que se definió es a la arquitectura de software para el despliegue de servicios API, la arquitectura REST y arquitectura GraphQL para envoltorios. Del mismo modo se definió la variable dependiente la cual es la calidad de productos de software en base a la característica de eficiencia de rendimiento.

Tabla 31 Variables del experimento

Variable Independiente	Arquitectura REST para el despliegue de servicios API.
	Arquitectura GraphQL para envoltorios para el despliegue de servicios API.
Variable Dependiente	Calidad de productos de software en base a la característica de eficiencia de rendimiento.

Seguidamente, se muestra cual es la métrica referida a la característica de eficiencia de acuerdo a la norma ISO/IEC 25023.

Tiempo de respuesta

El tiempo de respuesta se refiere al tiempo que tarda en culminar una petición, es decir el tiempo medio empleado en el proceso. Una vez teniendo en claro que es el tiempo de respuesta utilizamos la siguiente función:

$$\boldsymbol{X = \sum_{i=1\ an} (B_i = A_i)/n}$$

En donde, $\boldsymbol{A_i}$ representa el tiempo de inicio; $\boldsymbol{B_i}$ representa el tiempo final; y $\boldsymbol{n}$ representa la cantidad de peticiones hechas.

3.1.4. Hipótesis

En esta sección se plantearon las preguntas de investigación derivadas de la ***PI.***

- $\boldsymbol{PI_1}$**:** ¿Cuáles son los efectos que se produce en la calidad de productos de software cuando se despliegan los servicios API con una implementación hibrida que combinen REST y GraphQL?

Una vez se haya planteado la pregunta de investigación, se han establecido unas hipótesis que proponen una respuesta al experimento que se propone en PI_1

- $\boldsymbol{Hipotesis_0}$**:** No se presenta una diferencia que pueda ser significativa en la calidad de productos de software, cuando se despliegan las arquitecturas REST con una implementación hibrida.
- $\boldsymbol{Hipotesis_1}$**:** Se puede presenciar una clara diferencia en la calidad de productos de software cuando se aplican servicios GraphQL/REST con implementación hibrida. En donde la calidad del servicio GraphQL es superior a la calidad que ofrece el servicio REST.
- $\boldsymbol{Hipotesis_2}$**:** Se puede presenciar una clara diferencia en la calidad de productos de software cuando se aplican servicios REST/GraphQL con implementación hibrida. En donde la calidad del servicio REST es superior a la calidad que ofrece el servicio GraphQL.

3.1.5. Diseño

El objetivo es el de establecer condiciones o recursos que sirvan para realizar una comparación de la eficacia de las APIs de la arquitectura REST como de la arquitectura GraphQL. Con este fin, se estableció cuatro tareas experimentales las cuales tienen el mismo

escenario. La primera tarea consiste en implementar servicios API REST sin utilizar caché, mientras que la segunda tarea implementa servicios API REST utilizando caché. La tercera tarea implicó la implementación de servicios del API GraphQL sin utilizar caché, y la cuarta tarea involucró la implementación de servicios del API GraphQL utilizando caché.

En la siguiente tabla se muestra el diseño del experimento, donde se definen tres casos de uso.

Tabla 32 Distribución General del experimento

Caso de Uso	Endpoint	Repeticiones	REST (caché/sin caché)	GraphQL (caché/sin caché)
CU-01	Artists	3	Registros: 1, 50, 250, 3150, 32800, 100000	Registros: 1, 50, 250, 3150, 32800, 100000
CU-02	Tracks	3	Registros: 1, 36, 256, 3136, 32761, 99856	Registros: 1, 36, 256, 3136, 32761, 99856
CU-03	Albums	3	Registros; 1, 27, 216, 3375, 32768, 97336	Registros; 1, 27, 216, 3375, 32768, 97336

Utilizando la métrica definida en la sección 3.1.3 se pudo evaluar la eficiencia de las API, para encontrar la eficiencia se obtuvo el tiempo de respuesta cada caso de uso, los cuales fueron expuestos a tres repeticiones cada uno.

3.1.6. Tareas experimentales

En esta sección se definen las tareas experimentales definidas en el anterior capitulo como ENDPOINTS en las historias de usuario. Mediante tareas experimentales se pudo establecer la construcción del laboratorio computacional tal y como se muestra en la Fig. 26

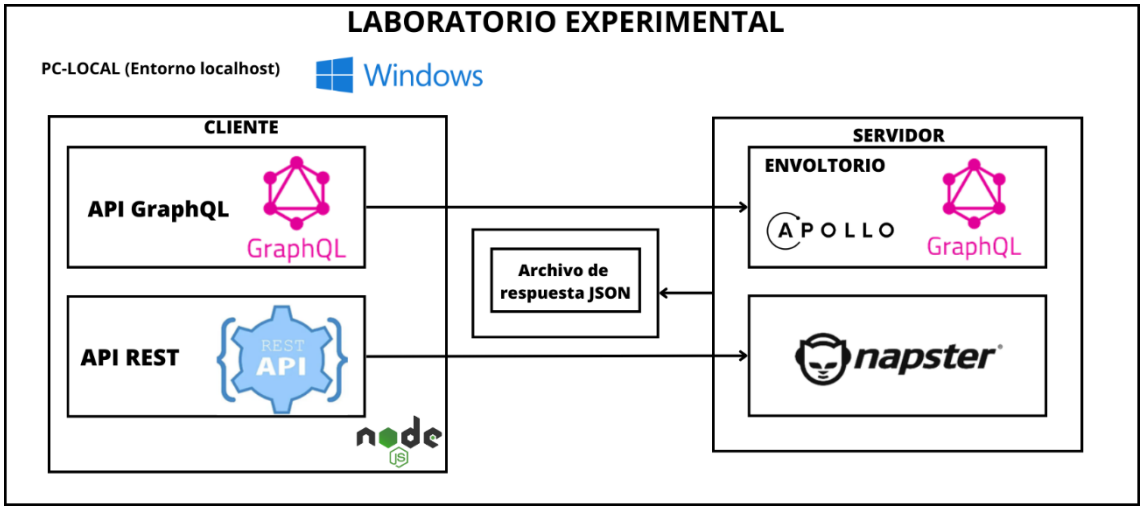


Fig. 26 Arquitectura Laboratorio Experimental

3.1.7. Instrumentación

En este apartado se especifica cual es la infraestructura, librerías y tecnologías que contine el laboratorio experimental.

Características del PC-Local:

- SO: Windows 10 Pro, 22H2 x64 bits
- Procesador: Intel(R) Core (TM) i7-4510U CPU @ 2.00GHz 2.60GHz
- Memoria: RAM 6,00 GB

Ambiente de Desarrollo: Se encuentra conformado por las siguientes tecnologías.

- IDE: Visual Studio Code v1.79.2
- Lenguaje de programación: JavaScript, CommonJS
- Entorno de tiempo de ejecución de JavaScript: NodeJS v19.2.0
- Librería npm para el API GRAPHQL: @apollo/client v3.7.14, apollo-link-rest v0.9.0, apollo-server v3.11.1, apollo-server-plugin-http-headers v0.1.4, graphql v15.8.0, graphql-tag v2.12.6, merge-graphql-schemas v1.7.8, nodemon v2.0.20, qs v6.11.2
- Mapeo de estructura de API: node-fetch v3.3.0
- Aplicación cliente para el consumo del API REST: Postman web version
- Aplicación para consumir la API GRAPHQL: Apollo sandbox

Recolección y análisis de datos: Consta de las siguientes aplicaciones

- IBM SPSS v29.0.1.0
- Microsoft Excel 365

3.1.8. Recolección de datos

En la Tabla 33 se muestra la estructura de datos que se obtuvo del registro de los resultados en el archivo Excel, obtenida a través del experimento descrito en la Tabla 32.

Tabla 33 Estructura para la recolección de datos

Variable	Descripción
Nro.	Numero de datos de la tabla
Caso de Uso	El caso de uso que se ejecutó
Nivel	El nivel de Complejidad
Nro. De Registros	Cantidad de datos consultados o insertados.
Repeticiones	Numero de repeticiones del caso de uso (3 veces)
Arquitecturas	REST o GraphQL con diferente implementación (Con caché/Si caché)

Tiempo	Tiempo de respuesta medido en milisegundos

3.1.9. Análisis

Para el análisis estadístico y normalización de los datos de la experimentación, se lo realizo en la herramienta IBM SPSS Statics.

3.2. Ejecución del experimento

En base al entorno experimental establecido en la sección 3.1.6 se realizaron los siguientes puntos:

- Ejecución del Proyecto (Cliente)
- Consumo Casos de Uso
- Repeticiones (3 veces) por cada número de registros en cada caso de uso.

3.2.1. Muestra

Para este apartado, el experimento se ejecutó de acuerdo al diseño de los tres casos de uso establecidos en la sección **¡Error! No se encuentra el origen de la referencia.**, con el fin de o btener los datos necesarios para la experimentación.

3.2.2. Preparación

El funcionamiento de las tareas especificados en la sección 3.1.6, se verifican con la ejecución del experimento de manera iterativa por cada caso de uso; es decir se realizaron tres repeticiones a cada caso de uso por cada registro y arquitectura. Por ejemplo para realizar las consultas del CU-03 se aplicó la distribución 1, 27, 216, 3375, 32768 y 97336 registros. En la Fig. 27 se detalla el proceso de experimentación.

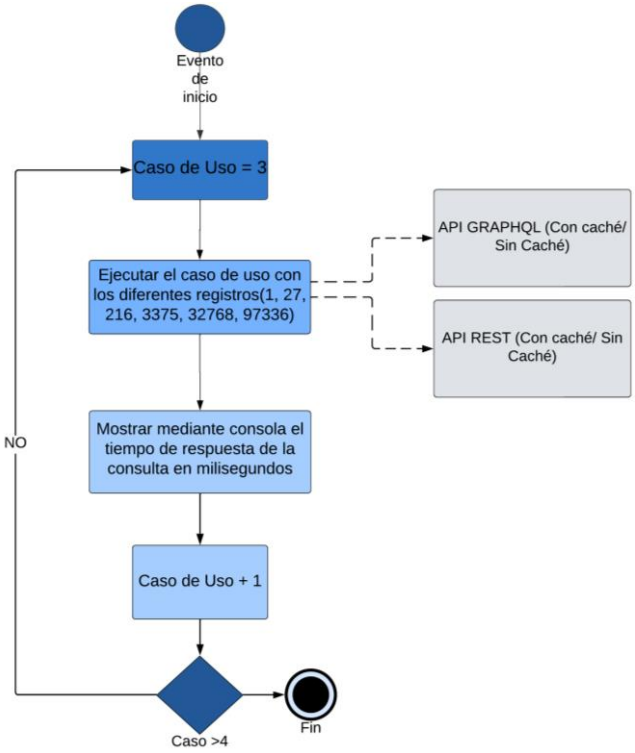


Fig. 27 Diagrama de flujo General para los 3 casos de uso

3.2.3. Recolección de datos

Una vez haya finalizado la ejecución de los casos de uso de acuerdo al proceso establecido en la Fig. 27. Se realizo la recolección y registro de los tiempos de respuesta en un archivo de Excel para su respectiva tabulación. En la Tabla 34, Tabla 35, Tabla 36 y Tabla 37 se muestra un ejemplo de los registros de datos y sus respectivos tiempos de respuesta(milisegundos) obtenidos a través de la consola de Visual Studio Code.

Tabla 34 Ejemplo de recolección de datos Rest sin caché

Arquitectura REST					
Nro.	Caso Uso	Nivel	Nro. Registros	Repeticiones	Tiempo(ms)
1	CU-01	1	100.000	1	1.909.874
2				2	903.533
3				3	698.682

Tabla 35 Ejemplo de recolección de datos Rest con caché

Arquitectura REST caché					
Nro.	Caso Uso	Nivel	Nro. Registros	Repeticiones	Tiempo(ms)
1	CU-01	1	100.000	1	106.262

2				2	690
3				3	651

Tabla 36 Ejemplo de recolección de datos GraphQL sin caché

Arquitectura GraphQL					
Nro.	Caso Uso	Nivel	Nro. Registros	Repeticiones	Tiempo(ms)
1				1	1.645.981
2	CU-01	1	100.000	2	767.732
3				3	683.046

Tabla 37 Ejemplo de recolección de datos GraphQL con caché

Arquitectura GraphQL caché					
Nro.	Caso Uso	Nivel	Nro. Registros	Repeticiones	Tiempo(ms)
1				1	515.637
2	CU-01	1	100.000	2	0
3				3	0

3.3. Análisis de resultados

En esta sección se analizó el impacto que se da en las arquitecturas cuando se despliegan tanto servicios REST y servicios GraphQL. Para lo cual, basándose en la métrica 'tiempo medio de respuesta' de la característica de eficiencia de rendimiento de la norma ISO/IEC 25023, los resultados se obtuvieron a través de la ejecución del experimento especificado en la sección 3.2.

3.3.1. Análisis estadísticos

Caso de Uso 1

En el primer caso de uso (Listar Artists) en la Fig. 28 se visualiza la media obtenida por cada arquitectura. El tiempo medio de respuesta observado es de 263671,3333ms utilizando la arquitectura REST sin caché y utilizando caché nos da un tiempo medio de 10191,0556ms, lo cual el tiempo es mucho mayor al tiempo medio respuesta de GraphQL de 235146,2778ms sin usar caché y 16215,7222 con caché. En la Tabla 38 se muestra en mayor detalle el tiempo medio de respuesta por cada arquitectura, además de mostrar el porcentaje de eficiencia comparando la misma arquitectura en ambas APIs cuando utilizan cache y cuando no lo usan, en donde que REST es un 96,1% más eficiente cuando usa cache y GraphQL es un 84,5% más eficiente cuando se usa caché. Así mismo se comparan las APIs utilizando las

arquitecturas REST y GraphQL sin cache y REST y GraphQL con cache por separado, en donde podemos ver que GraphQL es un 8,80% más eficiente que REST cuando no utilizan caché y cuando se usa cache GraphQL es un -59,05% menos eficiente que REST. Si hablamos de la misma arquitectura el uso de caché permite aumentar la eficiencia del producto, en cambio si comparamos ambas arquitecturas, podemos decir que cuando no se utiliza caché GraphQL es más rápido que REST, pero cuando se utilizó caché Rest resulto mucho más rápido.

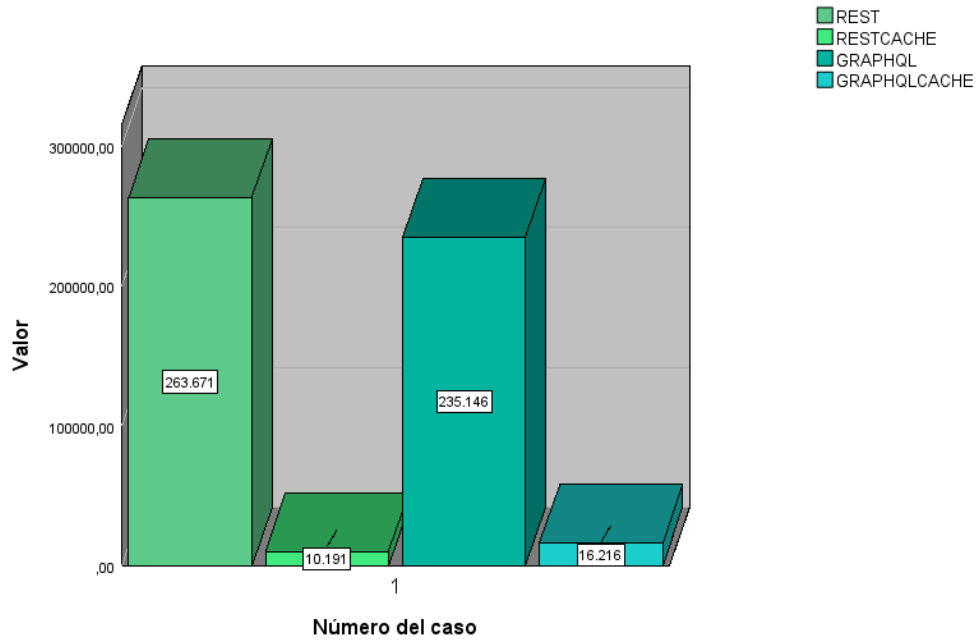


Fig. 28 Tiempo medio de respuesta por cada Arquitectura - CU-01

Tabla 38 Porcentaje de eficiencia - CU-01

Arquitectura	Media	Arquitectura	Media	**Eficiencia Arquitectura REST - GraphQL**
Rest sin caché	263671,3333	GraphQL sin caché	235146,2778	8,80 %
Rest con caché	10191,0556	GraphQL con caché	16215,7222	- 59,05 %
Eficiencia REST con/sin cache	96,1 %	**Eficiencia GraphQL con/sin cache**	84,5 %	

Caso de Uso 2

Para el segundo caso de uso (Listar Tracks) en la Fig. 29 se muestra la media que se obtuvo por cada arquitectura. Se puede observar que el tiempo medio de respuesta de la arquitectura REST es de 294678,6111ms sin utilizar caché y 2658,2222ms con cache, lo cual el tiempo es mucho mayor al tiempo medio respuesta de GraphQL de 7671,3333ms sin usar caché y 1247,4000ms con caché. En la Tabla 39 se muestra en mayor detalle el tiempo medio de respuesta por cada arquitectura, además de mostrar el porcentaje de eficiencia comparando la misma arquitectura en ambas APIs cuando utilizan cache y cuando no lo usan, en donde que REST es un 98,76% más eficiente cuando usa cache y GraphQL es un 84,5% más eficiente cuando se usa caché. Así mismo se comparan las APIs utilizando las arquitecturas REST y GraphQL sin cache y REST y GraphQL con cache por separado, en donde podemos ver que GraphQL es un 97,11% más eficiente que REST cuando no se utiliza caché y cuando se usa cache podemos observar que GraphQL es un 53,06% más eficiente que REST. De estos resultados podemos decir que GraphQL ya sea utilizando cache o no es más eficiente que REST, pero si la caché es una opción viable, "GraphQL con caché" es una buena alternativa para mejorar aún más los tiempos de respuesta.

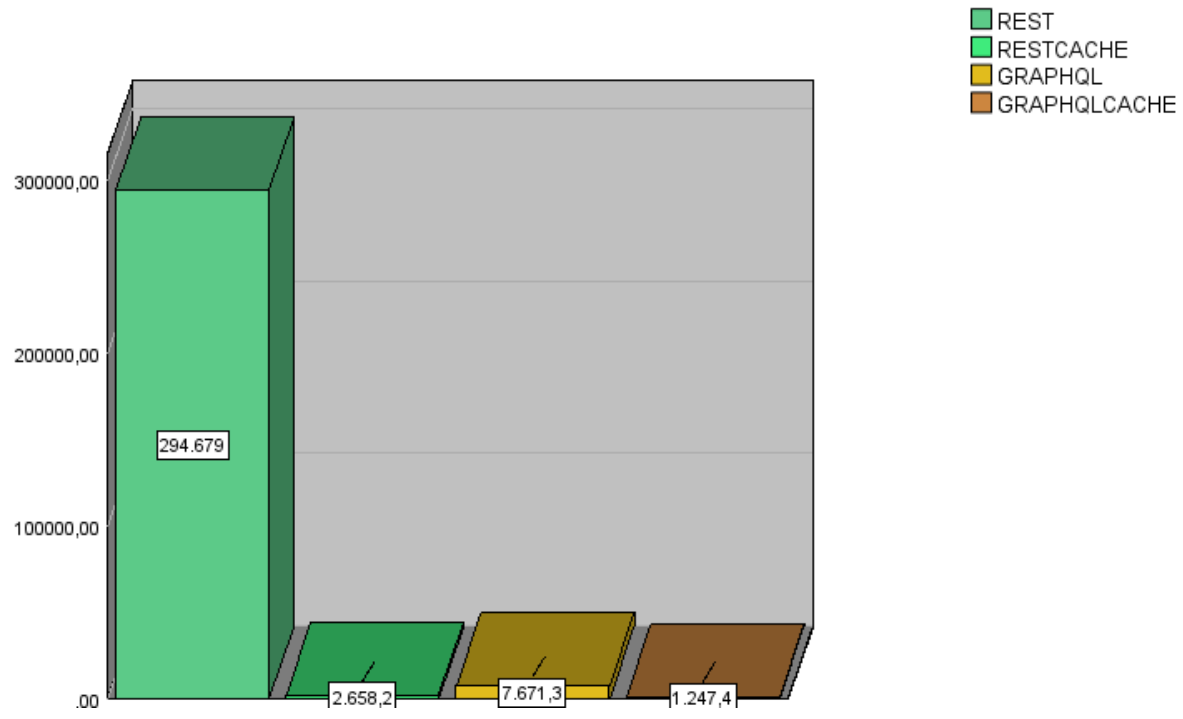


Fig. 29 Tiempo medio de respuesta por cada Arquitectura - CU-02

Tabla 39 Porcentaje de eficiencia - CU-02

Arquitectura	Media	Arquitectura	Media	**Eficiencia Arquitectura REST - GraphQL**
Rest sin caché	294678,6111	GraphQL sin caché	7671,3333	97,11%
Rest con caché	2658,2222	GraphQL con caché	1247,4000	53,06 %
Eficiencia REST con/sin cache	98,76 %	**Eficiencia GraphQL con/sin cache**	83,76 %	

Caso de Uso 3

Para el tercer caso de uso (Listar Albums) en la Fig. 30 se muestra la media que se obtuvo por cada arquitectura. Se puede observar que el tiempo medio de respuesta de la arquitectura REST es de 239482,6111 sin utilizar caché y 2177,6667 con cache, lo cual el tiempo es mucho mayor al tiempo medio respuesta de GraphQL de 5049,83sin usar caché y 1501,72 con caché respectivamente. En la Tabla 40 se muestra en mayor detalle el tiempo medio de respuesta por cada arquitectura, además de mostrar el porcentaje de eficiencia comparando la misma arquitectura en ambas APIs cuando utilizan cache y cuando no lo usan, en donde que REST es un 91,11% más eficiente cuando usa cache y GraphQL es un 70,27% más eficiente cuando se usa caché. Así mismo se comparan las APIs utilizando las arquitecturas REST y GraphQL sin cache y REST y GraphQL con cache por separado, en donde podemos ver que GraphQL es un 97.93% más eficiente que REST cuando no utilizan caché y cuando se usa cache GraphQL es un 31,05% más eficiente que REST. De estos resultados podemos decir que GraphQL ya sea utilizando cache o no es más eficiente que REST, si la caché es una opción viable, "GraphQL con caché" es una buena alternativa para mejorar aún más los tiempos de respuesta.

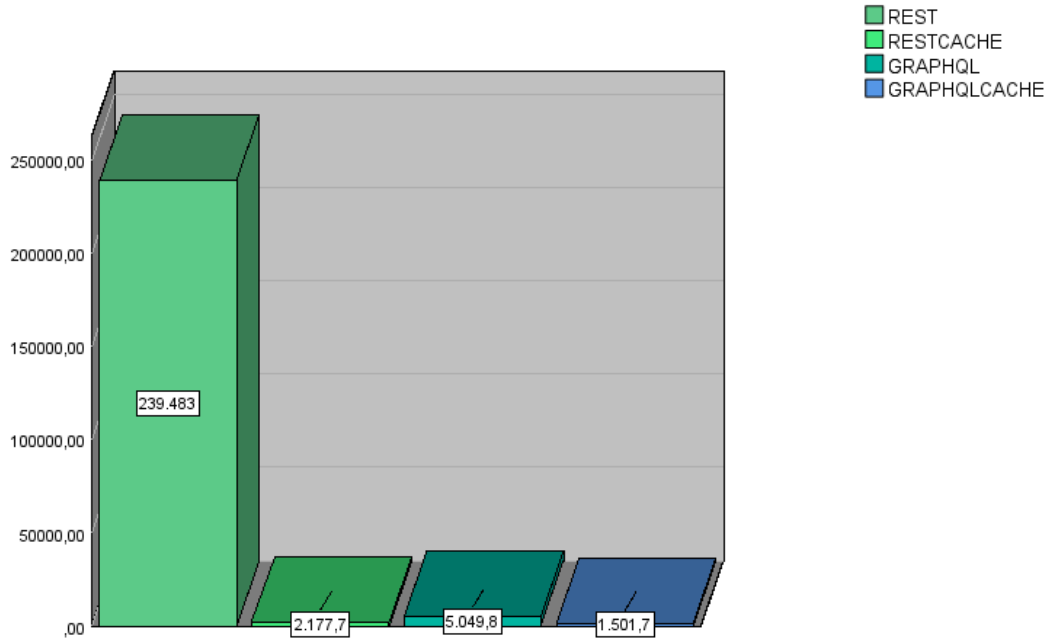


Fig. 30 Tiempo medio de respuesta por cada Arquitectura - CU-03

Tabla 40 Porcentaje de eficiencia - CU-03

Arquitectura	Media	Arquitectura	Media	**Eficiencia Arquitectura REST - GraphQL**
Rest sin caché	239482,6111	GraphQL sin caché	5049,83	97,93%
Rest con caché	2177,6667	GraphQL con caché	1501,72	31,05%
Eficiencia REST con/sin cache	91,11 %	**Eficiencia GraphQL con/sin cache**	70,27%	

En la Fig. 31 se visualiza el comportamiento de los cuatro casos de uso con las diferentes arquitecturas. En el caso de uso 1 se realizó la consulta a un endpoint, en el caso de uso 2 se consultó dos endpoint, y en el caso de uso 3 se hizo la consulta a tres endpoint. Por lo cual se puede asumir que mientras más aumenta el nivel, más aumenta el tiempo medio de respuesta, también se puede observar que GraphQL ya sea con o sin caché su tiempo de respuesta es más rápido que REST.

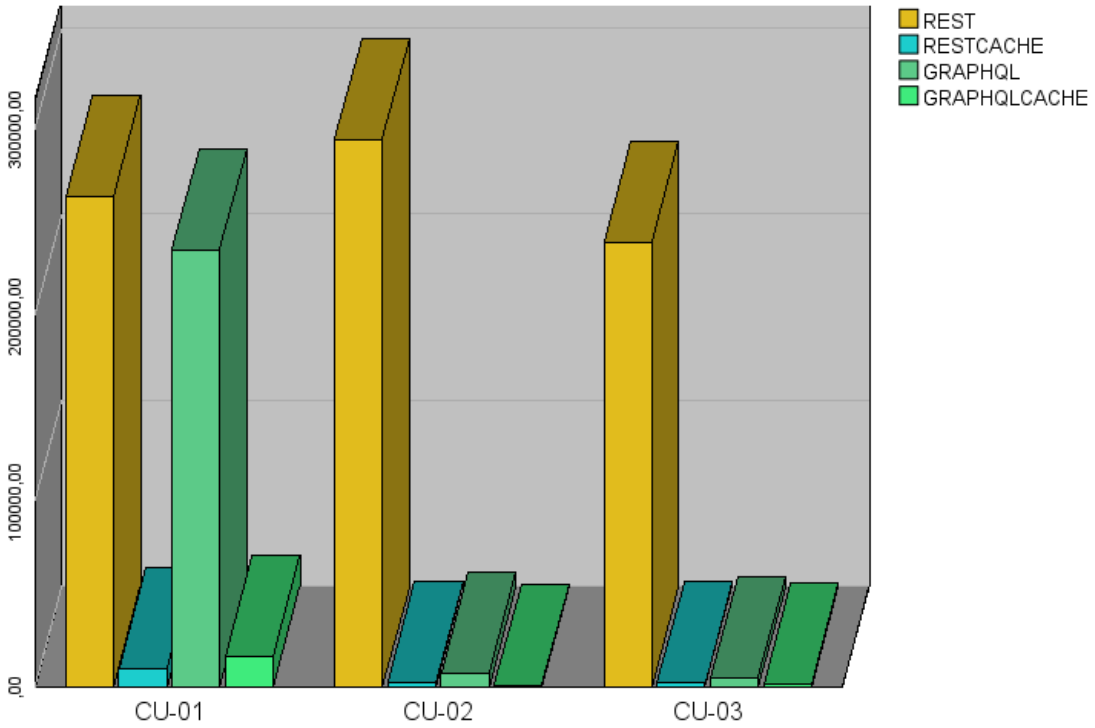


Fig. 31 Valor medio de eficiencia entre GraphQL y REST

En las Fig. 32 y Fig. 33 se puede visualizar los datos estadísticos globales que se obtuvo de los tres casos de uso, en donde se puede apreciar que el tiempo medio general de respuesta de la arquitectura REST sin caché es de 265944,1852ms; 27668,10430 (media; desviación estándar) con un intervalo de confianza del 95% de [197212,8039 - 334675,5665] y de 6321,6148ms; 4494,23546 con cache y un intervalo de confianza del 95% de [-6155,3183 - 16173,2813], dando como resultado que Rest con cache es mucho más eficiente que cuando no se utiliza caché.

De la misma manera podemos observar que el tiempo medio de respuesta de la arquitectura GraphQL sin cache es de 82622,4815ms; 132095,98556 con un intervalo de confianza del 95% de [-245522,1378 - 410767,1008], y con cache el tiempo medio es de 5008,9815ms; 8569,49188 con un intervalo de confianza del 95% de [-14966,1831 - 27609,4128], dando como resultado que GraphQL con cache es más eficiente a la hora de utilizar caché.

Arquitectura				Estadístico	Error estándar
Rest sin cache	Tiempo	Media		265944,1852	15974,18747
		95% de intervalo de confianza para la media	Límite inferior	197212,8039	
			Límite superior	334675,5665	
		Media recortada al 5%		.	
		Mediana		263671,3333	
		Varianza		765523995,71	
		Desv. estándar		27668,10430	
		Mínimo		239482,61	
		Máximo		294678,61	
		Rango		55196,00	
		Rango intercuartil		.	
		Asimetría		,367	1,225
		Curtosis		.	.
Rest con cache	Tiempo	Media		5008,9815	2594,74805
		95% de intervalo de confianza para la media	Límite inferior	-6155,3183	
			Límite superior	16173,2813	
		Media recortada al 5%		.	
		Mediana		2658,2222	
		Varianza		20198152,381	
		Desv. estándar		4494,23546	
		Mínimo		2177,67	
		Máximo		10191,06	
		Rango		8013,39	
		Rango intercuartil		.	
		Asimetría		1,710	1,225
		Curtosis		.	.
GraphQL sin cache	Tiempo	Media		82622,4815	76265,65282
		95% de intervalo de confianza para la media	Límite inferior	-245522,1378	
			Límite superior	410767,1008	
		Media recortada al 5%		.	
		Mediana		7671,3333	
		Varianza		17449349402	
		Desv. estándar		132095,98556	
		Mínimo		5049,83	
		Máximo		235146,28	
		Rango		230096,44	
		Rango intercuartil		.	
		Asimetría		1,731	1,225
		Curtosis		.	.
GraphQL con cache	Tiempo	Media		6321,6148	4947,59844
		95% de intervalo de confianza para la media	Límite inferior	-14966,1831	
			Límite superior	27609,4128	
		Media recortada al 5%		.	
		Mediana		1501,7222	
		Varianza		73436191,075	
		Desv. estándar		8569,49188	
		Mínimo		1247,40	
		Máximo		16215,72	
		Rango		14968,32	
		Rango intercuartil		.	
		Asimetría		1,730	1,225
		Curtosis		.	.

Fig. 32 Media General de los tres casos de uso

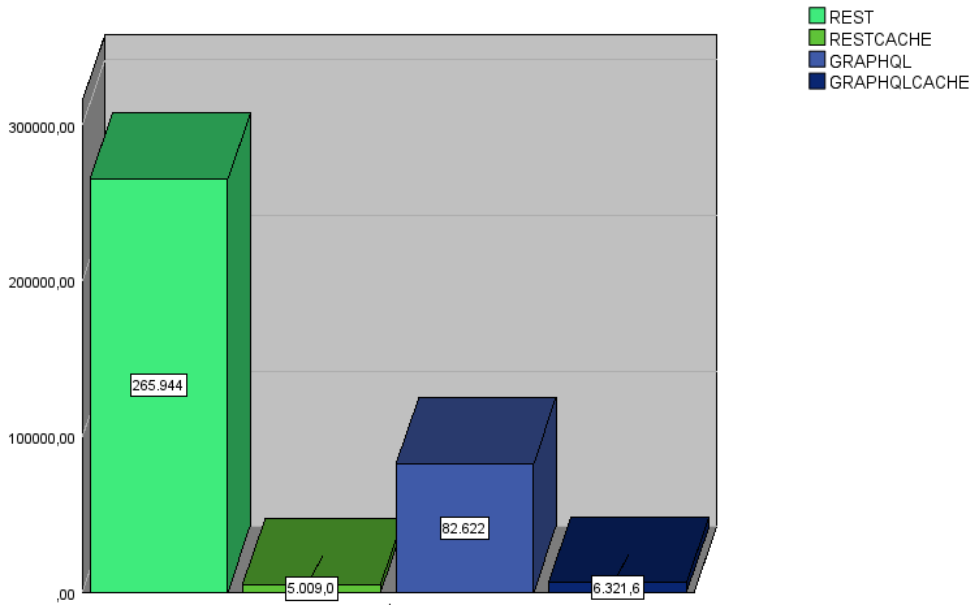


Fig. 33 Estadística descriptiva de eficiencia entre las arquitecturas

3.3.2. Análisis de impactos

Una vez culminado los análisis estadísticos de eficiencia, los resultados de la Tabla 41, determinaron que al usar cache en Rest es mucho más eficiente que cuando no se utiliza, del mismo modo en la arquitectura GraphQL es más eficiente cuando se utiliza cache que cuando no se hace uso de esta, cuando se despliegan servicios API.

En cambio si comparamos las arquitecturas Rest y GraphQL por separado podemos decir que la eficiencia de GraphQL sin cache es un 68.93% mucho más eficiente que REST sin cache; y del mismo modo al comparar las arquitecturas con el uso de cache, podemos decir que GraphQL con caché es 26.20% más eficiente que Rest con caché.

Tabla 41 Eficiencia de la arquitectura GraphQL

Tiempo medio de los tres casos de uso				
Arquitectura	Media(ms)	Arquitectura	Media(ms)	**Eficiencia Arquitectura REST - GraphQL**
Rest sin caché	265944,19	GraphQL sin caché	82622,48	68,93%
Rest con caché	6321,61	GraphQL con caché	5008,98	26,20%
Eficiencia REST con/sin cache	98,11%	**Eficiencia GraphQL con/sin cache**	92,34%	

CONCLUSIONES

- Mediante una investigación teórica, se estableció el marco teórico logrando crear los fundamentos conceptuales necesarios para definir tanto la estructura tecnológica y las tecnologías utilizadas en la creación de la API-GraphQL, el cual actúa como una capa de conexión para las APIs de Napster. Además, se estableció la base conceptual de la familia de normas ISO/IEC 25000, más precisamente la ISO/IEC 25023, utilizada para medir y mejorar la calidad de uso del software.
- Con la base conceptual establecida, se implementó el proyecto conforme al concepto de creación de un envoltorio GraphQL (servidor) y un ambiente que pueda consumir los servicios REST (cliente), estableciendo a Node.js como framework base para todo el proyecto, donde se realizó la integración de los servicios, para el posterior despliegue de la aplicación backend.
- Una vez creados el servidor como el cliente, para poder medir ambos enfoques, se evaluó siguiendo la guía Wohlin mediante un experimento; en donde se planteó la siguiente Pregunta de investigación, ***PI:*** *¿En qué situaciones es más recomendable utilizar arquitecturas híbridas que integren tanto REST como GraphQL al desarrollar aplicaciones web o servicios API?*. Para responder a esta pregunta se plantearon tres casos de uso en el laboratorio experimental, estos casos permitieron comparar la eficiencia Rest-GraphQL mediante la métrica de 'tiempo medio de respuesta' establecida en la ISO/IEC 25023.
- Se analizó los resultados del experimento computacional mediante análisis estadísticos que lograron responder la Pregunta de investigación planteada, y de acuerdo con la ejecución de las tareas experimentales y su respectivo análisis estadístico, se demostró que el tiempo medio de respuesta de las arquitecturas que utilizan cache es más eficiente que las arquitecturas que no utilizan cache. De la misma manera en cada caso de uso se pudo demostrar que la arquitectura GraphQL ya sea utilizando caché o no es más eficiente que REST, a excepción del caso de uso uno donde Rest con cache resulto más eficiente que GraphQL con caché.

RECOMENDACIONES

- Los resultados obtenidos a través de la investigación teórica brindan una sólida base conceptual para futuros desarrollos de software. Se sugiere investigar varios artículos científicos, revistas, libros que abarquen el consumo de arquitecturas GraphQL y REST.
- Para realizar consultas de los diferentes servicios se puede crear un ambiente controlado (de Api rest y Api graphql), donde podamos desplegar nuestro propio servidor y evitar problemas con servidores externos.
- Se recomienda utilizar ES Modules, que es framework basado en Node, el cual permite la carga síncrona y asíncrona, permitiendo fácilmente a implementación de varios servicios.
- Profundizar el tema de la estadística para una mejor resolución y validación de resultados de la investigación, para así adecuarse a las métricas que se utilicen en el trabajo realizado.

BIBLIOGRAFÍA

APPTEC. (2020). *GraphQL: Historia, origen y funcionamiento*. https://www.apptec.cl/blog/graphql-historia-origen-y-funcionamiento#:~:text=La historia de origen de,alto uso de la red.

ApolloClient. (2021). Documentation. Retrieved from Apollo Docs: https://www.apollographql.com/docs/react/

Azure. (2022). *Estilo de arquitectura de microservicios*. https://docs.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices

Cabrera, D., Vaca, C., & Piattini, y. M. (2019). ISO/IEC 25000: A systematic mapping study. Journal of Systems and Software.

Casciaro, M. (2014). *Node.js Design Patterns*. https://www.bookdepository.com/es/Node-js-Design-Patterns-Mario-Casciaro/9781785885587

Castro, A. (2018). *Servicios Web: RESTful*. https://blog.bi-geek.com/servicios-web-restful/

Cho, H., & Ryu, S. (2014). *REST to JavaScript for better client-side development*. https://dl.acm.org/doi/10.1145/2567948.2579219

Daniel, J. (2019). What are Database Wrappers and When Should You Use Them? Retrieved from ScaleGrid: https://scalegrid.io/blog/what-are-database-wrappers-and-when-should-you-use-them/

developer.prod.napster.com. (n.d.). Retrieved from https://developer.prod.napster.com/api/v2.2#genres

Feng, X., Shen, J., & Fan, Y. (2009). REST : An Alternative to RPC for Web Services Architecture. *First International Conference on Future Information Networks*. https://doi.org/10.1109/ICFIN.2009.5339611

Fielding, R. T., & Taylor, R. N. (2020). *Principled design of the modern Web architecture*. https://dl.acm.org/doi/10.1145/514183.514185

Galipienso, M. I. (2014, Diciembre). Servicios Rest. Retrieved from Experto JAVA: http://expertojava.ua.es/experto/restringido/2014-15/rest/rest.html

Ghebremicael, E. S. (2017). *Transformation of REST API to GraphQL for OpenTOSCA*. http://dx.doi.org/10.18419/opus-9352

GraphQL. (2023). Schemas and Types. Retrieved from GraphQL.org: https://graphql.org/learn/schema/#arguments

GraphQL-Foundation. (2019). *Introducción a GraphQL*. https://graphql.org/learn/

IBM. (2021). *¿Qué es una API REST?* https://www.ibm.com/mx-es/cloud/learn/rest-apis

Idongesit, V. (2021). What is npm? A beginner's guide to Node Package Manager. The Startup. Retrieved from https://medium.com/swlh/what-is-npm-a-beginners-guide-to-node-package-manager-9ae4deae87f0

iso.org. (2022). ISO/IEC 25023:2016 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality. Retrieved from ISO: https://www.iso.org/standard/35747.html

ISO-25023. (2016). *ISO/CEI 25023:2016*. https://www.iso.org/standard/35747.html

Jørgensen, M., & Shepperd, M. (2022). Empirical software engineering: The evolving field. Journal of Systems and Software. 183. doi:https://doi.org/10.1016/j.jss.2021.111101

Kobusinska, A., & Hsu, C.-H. (2017). Towards increasing reliability of clouds environments with RESTful web services. *Future Generation Computer Systems*. https://doi.org/https://doi.org/10.1016/j.future.2017.10.050

Krill, P. (2020). Scrum: The Agile project management framework. InfoWorld. Retrieved from https://www.infoworld.com/article/3537561/scrum-the-agile-project-management-framework.html

Landeiro, M. I. (2020). *Software Engineering for Agile Application Development*. Analyzing GraphQL Performance: A Case Study. https://www.igi-global.com/chapter/analyzing-graphql-performance/250439

Lindberg, O. (2019). What are Wrappers and How Do You Use Them in Python?" por Oliver Lindberg en Python Software Foundation. Retrieved from Python Software Foundation: https://peps.python.org/pep-0602/

Margaret, R. (2018, Febrero). Wrapper. Retrieved from techopedia: https://www.techopedia.com/definition/4389/wrapper-software-engineering

MDN. (2022). *Fundamentos de JavaScript*. https://developer.mozilla.org/es/docs/Learn/Getting_started_with_the_web/JavaScript_basics

Murali, A. (2020). REST API Wrappers. Retrieved from Medium: https://medium.com/@anirudh.murali/rest-api-wrappers-209a16f862de

Naeem, T. (2020). *Definición de API REST: comprensión de los conceptos básicos de las API REST*. https://www.astera.com/es/tipo/blog/definición-de-la-API-de-descanso/

Node. (2019). *Node.js*. https://nodejs.org/es/

npm. (2020). Getting started with npm. Retrieved from npm Docs: https://docs.npmjs.com/getting-started

Núñez, L. (2014). *Arquitectura de Software*. https://revista.jovenclub.cu/arquitectura-de-software/

Oracle. (2019). *Java Documentation*. https://docs.oracle.com/en/java/javase/

Palacio, J. (2015). *Scrum Manager I Las reglas de scrum*. https://www.scrummanager.net/files/scrum_I.pdf

Porcello, E., & Banks, A. (2018). *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. https://www.amazon.es/Learning-GraphQL-Eve-Porcello/dp/1492030716

Quiña-Mera, A., Fernandez, P., García, J. M., & Ruiz-Cortés, A. (2023, Febrero 02). GraphQL: A Systematic Mapping Study. ACM Computing Surveys. Retrieved from https://dl.acm.org/doi/full/10.1145/3561818

Ramos, A. I. (2019). Transformación de la construcción y la arquitectura en los últimos veinte años: prospectivas y perspectivas. Análisis Bibliométrico de los tópicos más desarrollados en revistas internacionales de alto impacto. ARQUITECTURAS DEL SUR.

RedHat. (2019). *¿Qué es GraphQL?* https://www.redhat.com/es/topics/api/what-is-graphql#graphql

RedHat. (2022). ¿Qué es una arquitectura de aplicaciones? Retrieved from Red Hat: https://www.redhat.com/es

Ribas, E. (2018). *Qué es Api Rest y por qué debes de integrarla en tu negocio*. https://www.iebschool.com/blog/que-es-api-rest-integrar-negocio-business-tech/#api

Rincón-Mejía, L. F., López-González, L. G., Martínez-Santiago, J. G., & Ramírez-Benavides, C. (2021). Metodologías de desarrollo de software: una revisión sistemática. Investigación e Innovación en Ingenierías. Retrieved from https://revistas.unilibre.edu.co/index.php/iiei/article/view/9864/9492

Rouse, M. (2019). Agile methodology. TechTarget. Retrieved from https://searchsoftwarequality.techtarget.com/definition/Agile-methodology

Rouse, M. (2020). Client/Server Architecture. Techopedia. Retrieved from https://www.techopedia.com/definition/438/clientserver-architecture

Schwaber, K., & Sutherland, J. (2020). The 2020 Scrum GuideTM. Retrieved from Scrum

Guides: https://scrumguides.org/scrum-guide.html

Scrum. (2020). *WHAT IS SCRUM?* https://www.scrum.org/resources/what-is-scrum

Scrum.org. (2023). What is Scrum? Retrieved from Scrum.org: https://www.scrum.org/learning-series/what-is-scrum/the-scrum-artifacts

StackOverflow. (2021). The State of Developer Survey 2021. Retrieved from Stack Overflow: https://insights.stackoverflow.com/survey/2021#technology-programming-languages

Stellman, A., & Greene, J. (2014). *Learning Agile*. O'Reilly Media, Inc. https://learning.oreilly.com/library/view/learning-agile/9781449363819/

Thiran, P., Hainaut, J.-L., & Houben, G.-J. (2005). *Database wrappers development: towards automatic generation*. https://doi.org/10.1109/CSMR.2005.22

Vargas, A., Sánchez Rivero, D., Valdéz, Á., Bernechea, M., Castillo, N., & Colqui, R. (2013). *Evaluación de la calidad de la Información extraída por wrappers, de un sitio web*. http://sedici.unlp.edu.ar/handle/10915/27136

Vázquez, A., Cruz, J., & García, F. (2017). *Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL*. https://doi.org/https://doi.org/10.1145/3144826.3145437

Velepucha, V., Flores, P., & Torres, J. (2019). MOMMIV: Model for the decomposition of a monolithic architecture towards a microservices architecture under the principle of information hiding. RISTI - Revista Iberica de Sistemas e Tecnologias de Informacao. Retrieved from https://www.researchgate.net/publication/331178205_MOMMIV_Model_for_the_decomposition_of_a_monolithic_architecture_towards_a_microservices_architecture_under_the_principle_of_information_hiding

Vikram, J. (2020). "Why Big Companies Use Node.js" de Forbes (2020). Retrieved from Forbes: https://www.forbes.com/sites/forbestechcouncil/2020/03/12/why-big-companies-use-nodejs/?sh=59e8f867282a

WEST, D. (2020). Agile scrum roles and responsibilities. Atlassian. Retrieved from https://www.atlassian.com/agile/scrum/roles

Witt, S. (2015). How Music Got Free: A Story of Obsession and Invention. Penguin.

Wittern, E., Cha, A., & Laredo, J. A. (2018). *Generating GraphQL-Wrappers for REST(-like) APIs*. https://link.springer.com/chapter/10.1007/978-3-319-91662-0_5

Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C., Regnell, B., & Wesslen, A. (2012). Experimentation in Software Engineering. *SRPINGER*. https://doi.org/10.1007/978-3-

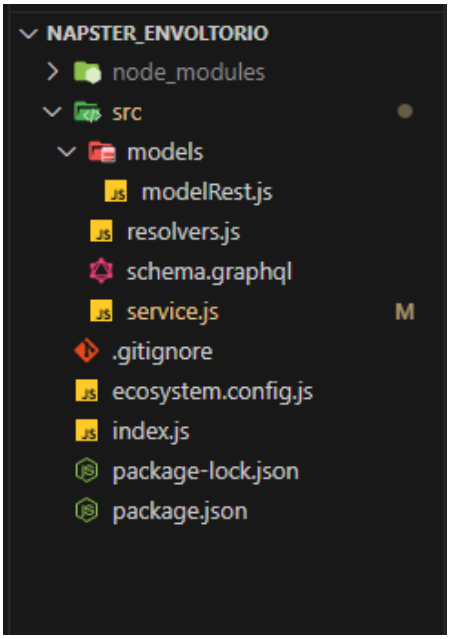
642-29044-2

Zhang, H., Ma, D., Lai, X., Yang, X., & Tang, H. (2018). The Optimization of Auxiliary Detection Coil for Metal Object Detection in Wireless Power Transfer. *IEEE PELS Workshop on Emerging Technologies: Wireless Power Transfer (Wow)*. doi:10.1109/WoW.2018.8450924

ANEXOS

Anexo A: Estructura del proyecto

- Desarrollada en Node.js
  - ~ Creación del componente ‘models’ para adaptarle a ApolloClient de REST.



  - ~ Componente model, para crear el modelo y parametrización de consulta de la API

```
const queryArtists= gql`
query artists($offsetParam:int!,$limitParam:int){
    Execute Query
    artists(offsetParam:$offsetParam,limitParam:$limitParam)
    @rest(
        type:"artists"
        path:"artists/top?limit={args.limitParam}&offset={args.offsetParam}",)
        {
            artists
        }
}`;
```

  - ~ Componente service para consumir el API de napster

```
//Artists
const getArtists = async (token, limit) => {
    let data = [];
    let limitTemp = limit > 50? 50: limit;
    let offset = 0;
    let diference = limit;
    client.setLink(
        new RestLink({
          bodySerializer: (body) => JSON.stringify(body),
          uri: "https://api.napster.com/v2.2/",
          headers: { Authorization: "Bearer " + token },
        })
      );
      do {
        const dataResponse = await client.query(
            { query:queryArtists,
              variables:{
                offsetParam:offset,
                limitParam:limitTemp}});
        offset += 50;
        diference -= limitTemp;
        limitTemp = diference <= 50 ? diference : 50;
        data = [...data, ...dataResponse.data.artists.artists];
      } while (diference !== 0);
    return data
};
```

~ Componente resolver para consultar en la arquitectura GraphQL

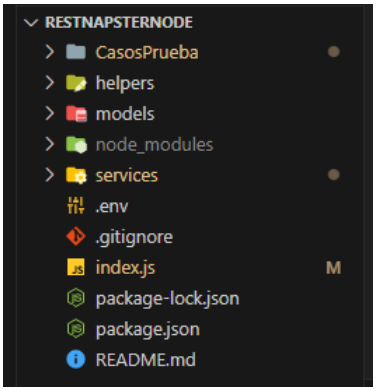
```
const resolvers = {
  Query: {
    obtenerToken:async () => {
      tokenData=await getToken();
      return tokenData.access_token;
    },

    obtenerArtist: async (_,{offset,limitFirst,limitSecond,limitThird}) => {
      const data = await getArtists(tokenData.access_token,limitFirst);
      return data.map(artists=>{
        return{
          ...artists,
          limitSecond,
          limitThird
        }
      });
    },
```

~ Verificación del estado del servidor GraphQL

npm run dev

```
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node .`
Run server in the URL: http://localhost:4000/
```

~ Creación y configuración del cliente



- **Anexo B:** Código fuente del proyecto

Proyecto	Repositorio
Envoltorio GraphQL del API REST de Napster	https://github.com/dans1410/Napster_envoltorio
Cliente que consume tanto el API REST y el API GraphQL	https://github.com/dans1410/RestNapsterNode