Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

# Trabajo de Fin de Grado

Grado en Ingeniería Informática

---

# Mews: music news app

*Mews: music news app*

Pablo Bande Sánchez - Girón

---

La Laguna, *05* de *julio* de 2022

D. **Alejandro Pérez Nava**, con N.I.F. 43.821.179-S profesor Asociado de Universidad asociado al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D. **Francisco Javier Rodriguez González**, con N.I.F. 43.618.712-V profesor Asociado de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

**C E R T I F I C A (N)**

Que la presente memoria titulada:

*“Mews: music news app”*

ha sido realizada bajo su dirección por D. **Pablo Bande Sánchez - Girón**,

con N.I.F. 43.838.242-B.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 5 de julio de 2022.

# Agradecimientos

Me gustaría agradecer profundamente a mi familia por todo el apoyo que me han brindado a lo largo de esta etapa educativa, a mis amigos y compañeros por los momentos compartidos.

Finalmente a mi tutor Alejandro Pérez Nava por su ayuda, paciencia y dedicación.

# Licencia

**Resumen**

*El objetivo de este trabajo ha sido la implementación de una aplicación móvil para Android capaz de reunir noticias relacionadas con el mundo de la música, clasificarlas por artistas y mostrarlas al usuario. Además, contará con un sistema de autenticación mediante correo electrónico y contraseña y permitirá al usuario suscribirse a distintos artistas y acceder a la pestaña de recientes donde se mostrará las últimas novedades de los músicos a los que sigue.*

*Para llevar a cabo este proyecto se ha de realizar un estudio sobre el desarrollo de aplicaciones móviles, el diseño y administración de base de datos y el uso de APIs para obtener los recursos y conectar los distintos componentes del sistema.*

**Palabras clave:** autenticación, bases de datos, APIs

**Abstract**

*The goal of this project is to implement an mobile application for Android capable of retrieving news related to the music, classify them by artists and display them to the user. Moreover, it will have an authentication system by email address and password and will allow the user to subscribe to multiple artists and access to the recent tab where the latest news of the musicians followed will be shown.*

*In order to achieve this objective there must be research into the development of mobile phone applications, the design and management of databases and the uses of APIs to obtain information and link different modules of the system.*

**Keywords:** authentication, databases, APIs

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1 Introducción

Hoy en día, gracias a los avances tecnológicos, la mayor parte de la población hace uso de internet para mantenerse informado sobre temas de la actualidad. La publicación y transmisión de noticias tiene gran importancia en el día a día de las personas ya que constantemente buscamos información sobre temas de interés, ya sea por necesidad o simple curiosidad.

La finalidad de este proyecto es el desarrollo de una aplicación para Android que mediante búsqueda permite obtener noticias recientes de artistas musicales. Además, el usuario será capaz de suscribirse a ellos y acceder a una pestaña con las novedades de los que sigue.

Para ello, se dispondrá de un buzón que contiene un conjunto amplio de información general sobre la música y que se irá actualizando constantemente mediante la suscripción a Google Alerts. Estas noticias a medida que van llegando son analizadas y almacenadas en la base de datos con sus respectivos artistas relacionados. El usuario dentro de la app puede buscar un artista disponible y acceder a todas sus últimas noticias que están guardadas en la base de datos.

## 1.1 Antecedentes y estado actual

Nos encontramos en la era de la información, en un mundo conectado mediante la tecnología donde acceder a fuentes muy variadas y ricas en información digital es de gran importancia. Por ello existen múltiples plataformas que ofrecen este tipo de servicio a los usuarios de forma completa y personalizada. Los principales que dominan el ámbito tecnológico son: Google News, Microsoft News, Apple News y Yahoo News.

En 2014 Google News dejó de estar disponible en España por problemas legislativos, por esta razón otras alternativas que surgieron y ganaron gran popularidad son las nombradas anteriormente, y en especial, Menéame. Actualmente se trata de la plataforma local de noticias más usada en el país.

Además de las plataformas creadas íntegramente para este objetivo, otras redes sociales como Reddit, Twitter o Facebook cuentan con un sistema similar de búsqueda de noticias y suscripción.

## 1.2 Objetivos

Para la realización del proyecto de fin de grado he establecido una serie de objetivos:

**General:**

- Desarrollar una aplicación de noticias relacionadas con la industria de la música para Android.

**Específicos:**

- Obtención y análisis de noticias mediante el correo electrónico con Google Alerts como fuente de información.
- Habilitar un menú de búsqueda donde el usuario pueda introducir el nombre del artista en interés y acceder a sus novedades.
- Registro e inicio de sesión mediante correo electrónico y contraseña.
- Sistema de suscripción a los artistas y acceso a una pantalla con las noticias más recientes de los músicos seguidos.

**Académicos:**

- Aprender acerca del funcionamiento, desarrollo y tipos de aplicaciones móviles.
- Diseñar una aplicación *fullstack* adquiriendo conocimientos tanto de *frontend* como *backend*.
- Diseño y administración de bases de datos.
- Uso de APIs, especialmente REST.

# Capítulo 2 Análisis

## Tipo de aplicación móvil

Lo primero que me planteé a la hora de comenzar con el proyecto fue el tipo de aplicación móvil a crear. Existen tres tipos de aplicaciones: nativas, híbridas y web.

- **Aplicación nativa**: está diseñada para un sistema operativo específico como iOS o Android. Al ser desarrollada para una única plataforma, generalmente están bien optimizadas ya que tienen un mayor control sobre los recursos del dispositivo. Además, al poder conectar directamente con el *hardware* del dispositivo pueden utilizar todo tipo de funcionalidades del teléfono como Bluetooth y NFC. En cuanto al aspecto, emplean elementos propios de la interfaz de usuario del sistema. Algunos ejemplos de lenguajes de programación para su creación son Java, Kotlin, Swift y Objective C.

- **Aplicación web:** funcionan como páginas web que se adaptan al formato móvil.

No pueden ser publicadas en las tiendas de aplicaciones como App Store o Google Play ni emplear algunos elementos del dispositivo como los nombrados previamente en las aplicaciones nativas. Por otro lado, requieren conexión a internet para su funcionamiento, aunque a la hora de decidir este factor no tuvo importancia dada la naturaleza de la aplicación. Para su desarrollo generalmente se emplea HTML5, CSS, JavaScript.

- **Aplicación híbrida o multiplataforma**: A diferencia de las nativas, estas pueden ser ejecutadas en distintas plataformas con una sola implementación. En esencia son aplicaciones web que parecen y se sienten como nativas y a diferencia de las puramente web, no requieren de conexión a internet para su uso. Al poder usarse en distintas plataformas su producción es más económica y consta de un mejor mantenimiento, sin embargo con un rendimiento inferior al de las nativas. Son desarrolladas empleando Ionic, Objective C, Switf y HTML5 entre otros lenguajes.

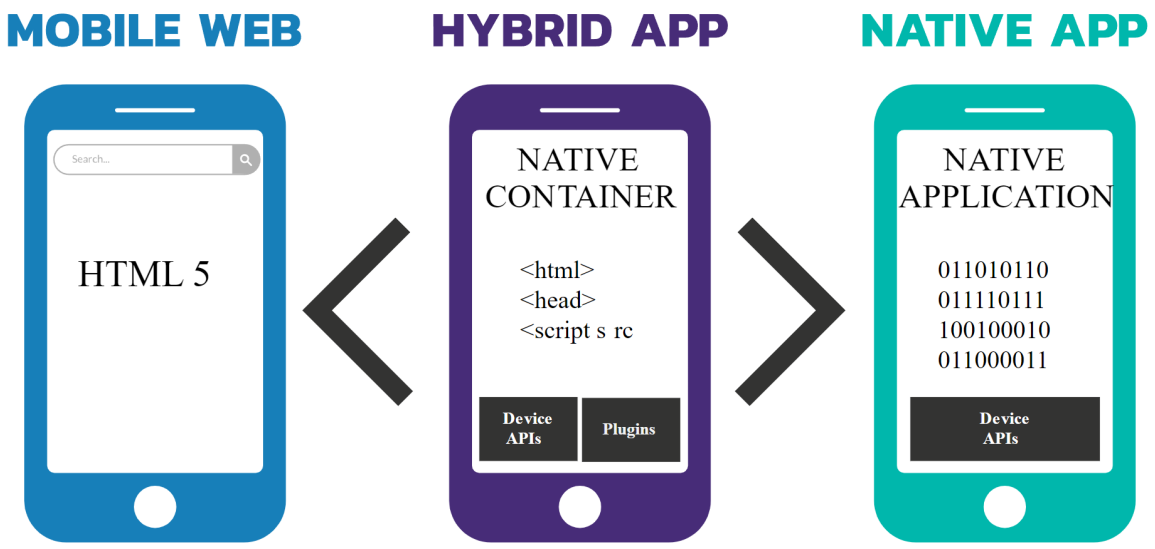


Figura 1: Tipos de aplicaciones móviles

Tras considerar las ventajas y desventajas que ofrecen cada tipo de aplicación, por su mejor rendimiento, estabilidad y grado de personalización he decidido optar por desarrollar una aplicación nativa. Inicialmente comencé con Kotlin, sin embargo no me sentía cómodo con el entorno de trabajo (Android Studio) ni con el lenguaje ya que era completamente nuevo para mí. Posteriormente, un compañero me dio a conocer React Native, una herramienta multiplataforma basada en JavaScript que permite crear aplicaciones nativas. Es una tecnología de desarrollo de apps híbrida ya que permite trabajar en iOS y Android con la misma implementación y para ello traduce el código de JavaScript a componentes nativos específicos, en caso de Android a Java y en iOS a Objective C. A pesar de que el objetivo es desarrollar una aplicación únicamente en Android, al permitirme usar un lenguaje de programación con el que estoy familiarizado para crear una app nativa, considero que React Native es la mejor alternativa para el proyecto.

# Tecnologías usadas

**Visual Studio Code**
Es el editor de código fuente que he empleado durante todo el proyecto. La versatilidad y funcionalidad que ofrece gracias a sus distintas extensiones crean un entorno de trabajo muy cómodo y eficiente.

Figura 2: Logo Visual Studio Code

**npm**
Es un el gestor de paquetes para JavaScript por defecto y lo he empleado tanto en *backend* como en *frontend* para la instalación y gestión de las distintas librerías y herramientas externas que he incorporado en el proyecto.

Figura 3: Logo npm

**Git - Github**
He usado Git como control de versiones local para guardar y hacer un seguimiento de los cambios y funcionalidades nuevas que he ido añadiendo al proyecto. Por otro lado, remotamente llevé a cabo esta tarea con GitHub. Además me fue de mucha utilidad a la hora de desplegar el *back-end* en el servicio de host DigitalOcean para ponerlo en producción

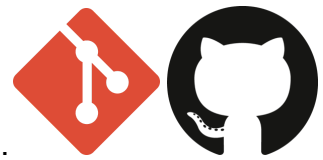

Figura 4: Logos Git y Github

**Figma**
Editor gráfico que he usado para crear los diseños iniciales de la aplicación siguiendo los estándares UI para dispositivos móviles. Sin embargo, dadas las limitaciones a la hora de desarrollar el *frontend* en React Native por la falta de experiencia no fue posible generar una interfaz de usuario como la diseñada inicialmente con esta herramienta.

## Frontend

Es la parte de la aplicación con la que tiene contacto directo el usuario e incluye el diseño de la interfaz de la app.

**React Native [1]**
Framework *open source* para el desarrollo de aplicaciones nativas multiplataforma basado en la librería de JavaScript desarrollada por Facebook: React. Permite a los desarrolladores crear aplicaciones con componentes nativos desde un lenguaje distinto. Permite desarrollar tanto para IOS como para Android (cross platform). Al usar de base un lenguaje interpretado, no necesita compilar y generar el paquete cada vez que se realicen cambios lo que agiliza mucho la programación.

**Axios [2]**
Cliente HTTP basado en promesas para node.js. Lo he usado para realizar las llamadas a la API desde la aplicación. He decidido usarlo sobre Fetch puesto que la información a enviar entre cliente y servidor tiene formato JSON y Axios maneja de forma más simple este tipo de datos. Otra ventaja de Axios es el control de errores, mientras que con el primero detecta las excepciones con el método catch, en Fetch es necesario leer la respuesta para comprobar si la petición ha sido exitosa, ya que lo único que capta el catch es fallo de conexión.

## Backend

Capa interna del programa encargada de gestionar los datos.

**Node.js**
Es un entorno de ejecución de aplicaciones escritas en JavaScript en tiempo real. Está formado por un modelo asíncrono dirigido por eventos que permite controlar múltiples programas al mismo tiempo.

**Google Cloud Platform [3]**
Consta con una serie de servicios que ofrece Google para diseñar aplicaciones y trabajar con datos de forma remota empleando un sistema de gestión de recursos seguro y eficiente. Cuenta con todo tipo de herramientas como máquinas virtuales, bases de datos e inteligencia artificial y aprendizaje automático y administración de APIs. En esta plataforma gestionaré la API de Gmail para acceder a la bandeja de entrada y Cloud Functions para controlar las notificaciones de llegada de correos electrónicos con las noticias a analizar.

**Express [4]**
Es el framework de node.js más popular que ayuda a organizar la aplicación para crear la estructura modelo-vista-controlador que he usado en la parte del servidor. Lo he usado para desarrollar la REST API y simplificar el control de las rutas y peticiones.

**Firebase [5]**
Es una plataforma de desarrollo de aplicaciones administrada por Google. Provee de herramientas que emplearé a lo largo del proyecto como una base de datos NoSQL:

Firestore database y un sistema de autenticación: Firestore Authentication mediante el que el usuario se registrará e iniciará sesión.

**DigitalOcean [6]**
Es un proveedor de infraestructura como servicio para los desarrolladores de *software* y lo he usado para desplegar el *backend* ya en producción. Gracias a las ventajas del pack de estudiantes de Github he obtenido de forma gratuita una máquina virtual denominada droplet que me permite mantener la ejecución del programa de forma indefinida.

**pm2 [7]**
Gestiona la ejecución de procesos en tiempo real de node.js y al igual que DigitalOcean, esta herramienta la usé para controlar la ejecución del *backend* en la máquina virtual y poder acceder a los registros.

**nginx [8]**
Se encarga de redireccionar automáticamente las peticiones al puerto 3000 donde está ubicado el servidor además de ofrecer mayor seguridad y mejor gestión de carga de datos.

# Capítulo 3 Diseño

## 3.1 Estructura de la aplicación: Patrón MVC

Para el diseño del proyecto he decidido llevar a cabo el modelo vista controlador (MVC) ya que es el más comúnmente empleado en el desarrollo de aplicaciones móviles. Consta de las siguientes partes:

- **Modelo**: se encarga de manejar la información contenida en la base de datos y llevar a cabo las distintas tareas y procedimientos para el funcionamiento correcto de la aplicación. En este caso, corresponde con los ficheros encargados de realizar el proceso de obtención, procesamiento, almacenamiento y manejo de noticias. Es parte del *backend* e incluye tanto la base de datos como los módulos que controlan el flujo de noticias y acceso a Firestore.

- **Vista**: corresponde directamente con el *frontend* de la aplicación en React Native y se encarga de la correcta visualización de la información aportada por el controlador. Es la parte de la aplicación a la que tiene acceso el usuario.

- **Controlador**: crea la unión entre la vista y el modelo. En este caso lo he realizado mediante la REST API. Las peticiones que recibe desde la aplicación son

notificadas al modelo que realiza distintas operaciones y modifica los valores de la base de datos a representar en la vista.

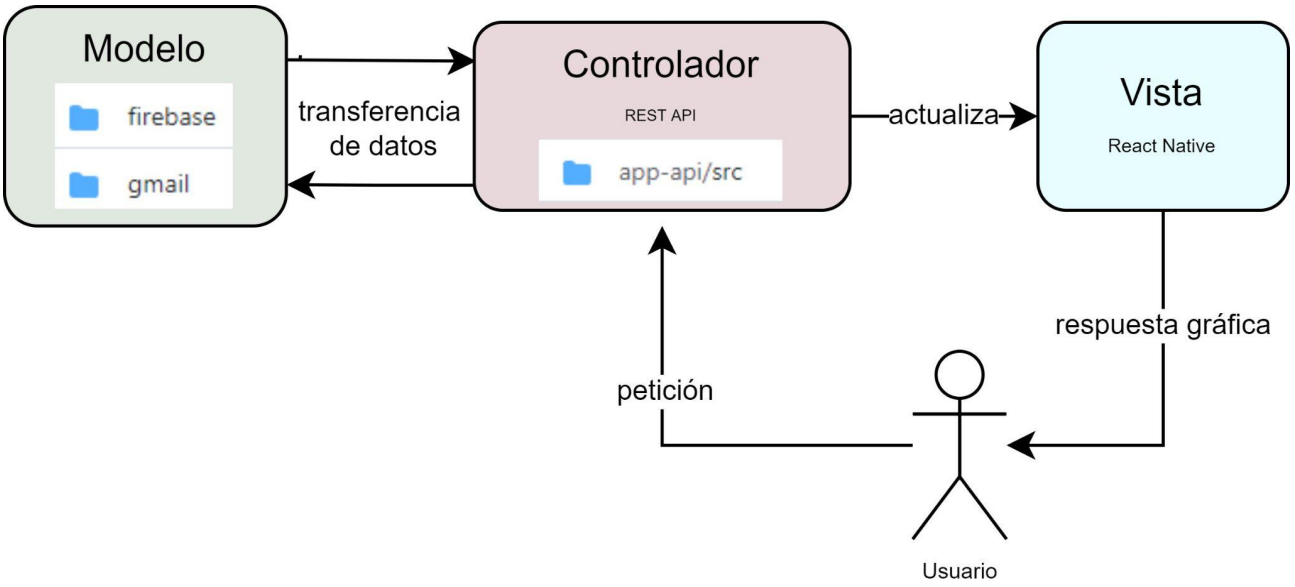


Figura 5: Esquema de funcionamiento del patrón modelo-vista-controlador

Haber empleado el modelo vista controlador ha sido de gran ayuda a la hora de realizar las distintas partes del proyecto ya que gracias a la división de responsabilidades que otorga me ha permitido desarrollar y comprobar el correcto funcionamiento del modelo y de la vista de forma independiente. Una vez finalicé ambas partes fue posible unirlas mediante el controlador con gran facilidad. Además, por esta razón, el mantenimiento de la aplicación es más sencillo y permite una mayor escalabilidad.

## 3.2 Diseño inicial de la interfaz

Una vez definidos los requisitos de usuario para la aplicación, realicé un mockup de la interfaz:

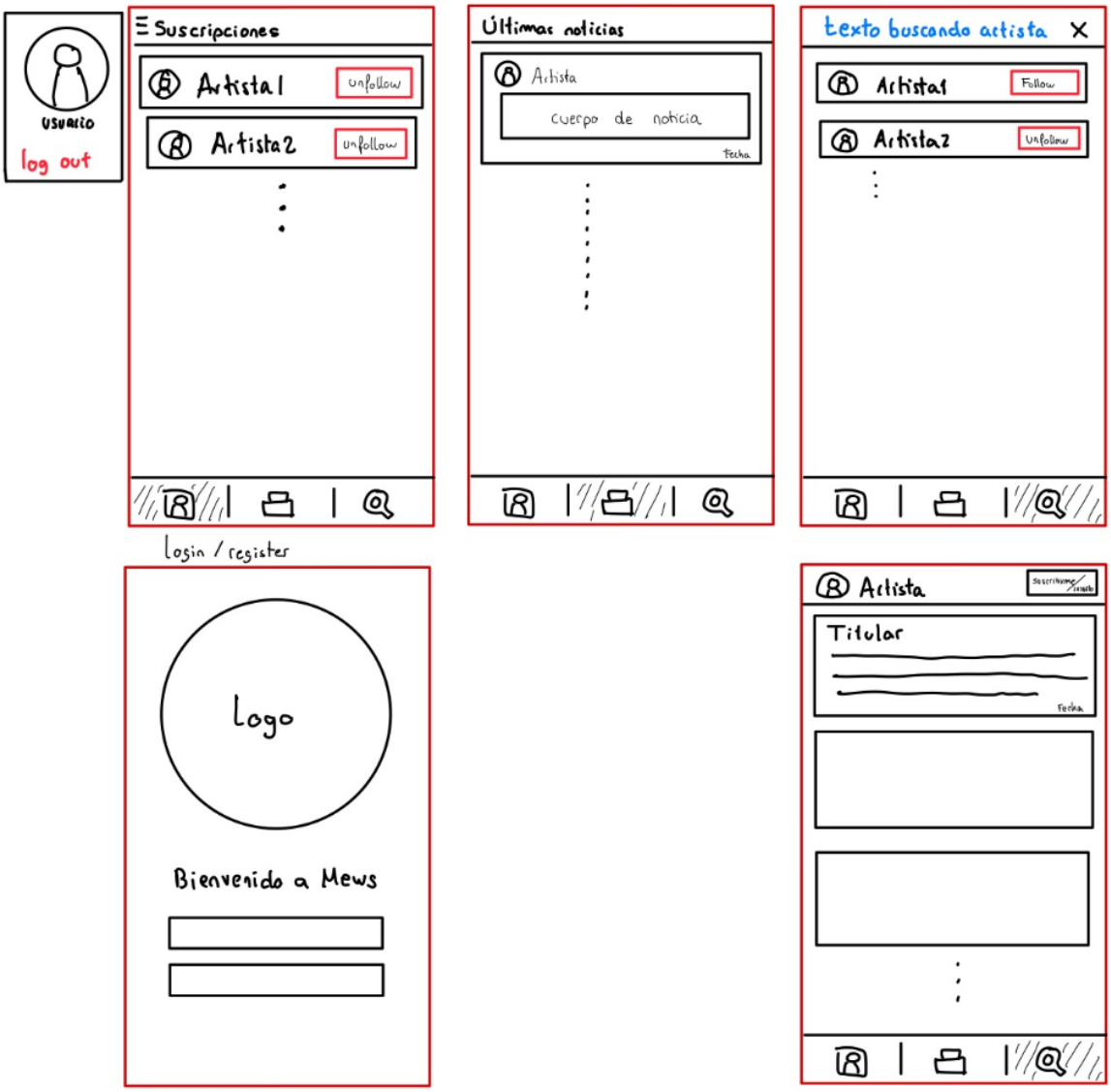


Figura 6: Mockup de la aplicación

## 3.3 Diseño de la base de datos

Decidí usar Firestore, una base de datos NoSQL desarrollada por Google que almacena la información en documentos organizados por colecciones. En contraposición con las SQL, no existen tablas ni registros, los datos son representados mediante documentos, que son árboles JSON con un identificador asociado. Por otro lado, las colecciones son los contenedores de los documentos y es posible crear subcolecciones dentro de estos tal como hice a la hora de diseñar la base de datos. Otra notable diferencia es la escalabilidad, mientras que las SQL escalan verticalmente requiriendo una mejora de los componentes del servidor, las NoSQL escalan horizontalmente aumentando el número de servidores. Cuando mayor sea el volumen de datos, la diferencia de tiempos de carga entre ambos tipos es más notable ya que en las SQL hay que realizar la unión de múltiples tablas para realizar la consulta y esto consume una gran cantidad de recursos, mientras que en NoSQL si está bien estructurada, la información debe estar junta quitando la necesidad de realizar estas operaciones.

La principal razón por la que escogí este tipo de base de datos es por el manejo de distintos tipos de datos y flexibilidad de la estructura de los documentos y la comodidad que supone al estar en formato JSON que es el que uso durante todo el programa.

Además la configuración y uso es sencillo mediante la SDK de node. Tras analizar las necesidades de la aplicación, he diseñado la base de datos con la siguiente estructura:

**Colección de artistas**

Conjunto de documentos que contienen los datos de los artistas a mostrar en las noticias y contiene una subcolección donde son almacenadas las noticias relacionadas con este. He decidido usar una subcolección en vez de un array puesto que Firebase no permite extraer parte de los documentos, por lo que si simplemente necesito los datos básicos del artista tendría que leer todas las noticias. Además, los documentos tienen como límite de tamaño 1 MB por lo que a medida que vaya aumentando el número de noticias podría llegar a sobrepasar. La única desventaja de usar subcolecciones es la dificultad a la hora de eliminarlas aunque en este caso no afecta.

```
Artists [ {
        id: string
        name: string
        image: string
        followers: number
        news [ {
                title: string
                body: string
                date: timestamp
                sourceLink: string
         }]
}]
```

**Colección de usuarios**

Para poder almacenar las suscripciones de los usuarios he tenido que crear una nueva colección ya que el lugar donde Firebase Authentication almacena los datos es inaccesible y no permite extender las propiedades. Por ello he decidido crear documentos para cada usuario con su identificador de usuario como identificador del documento y con los nuevos atributos asociados a ellos. A diferencia del caso anterior, he usado un array de strings para guardar las suscripciones.

```
Users [{
        id: string
        subscription: array
}]
```

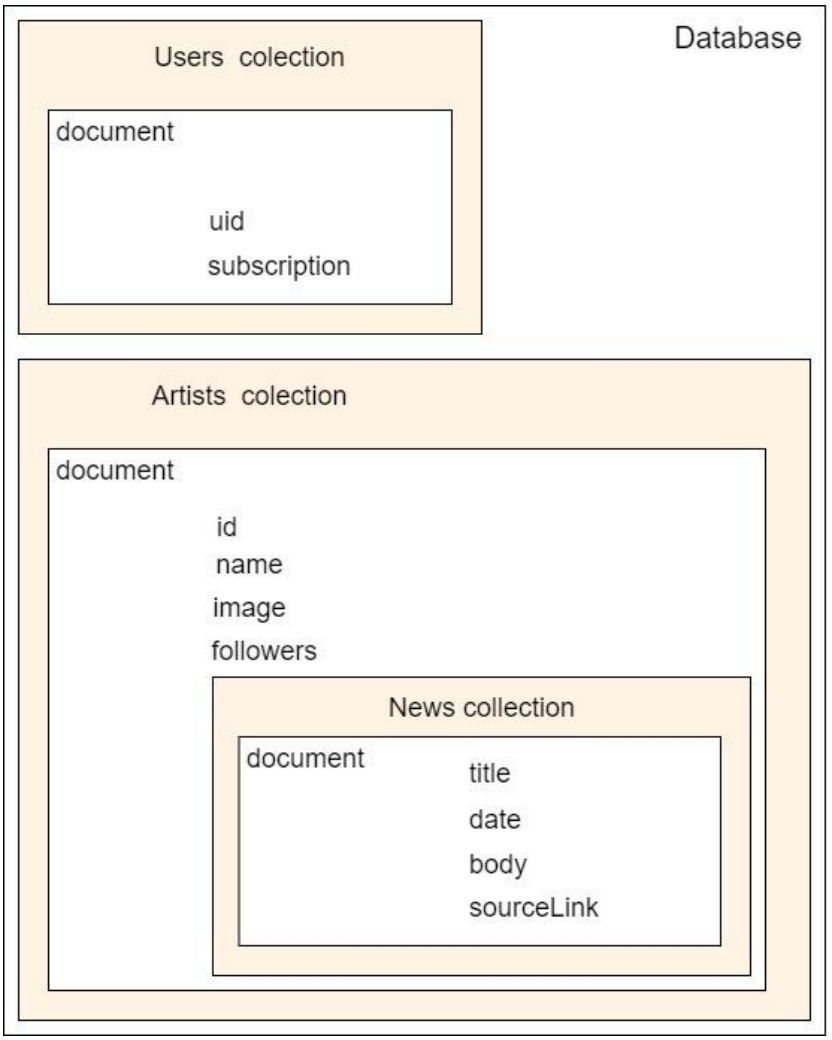


Figura 7: Diseño de la base de datos

# Capítulo 4 Desarrollo

En cuanto al desarrollo de la aplicación, he comenzado con la creación de la interfaz de usuario a partir de las necesidades establecidas inicialmente y para ello he usado *mocking* y datos locales. Una vez que he finalizado el *frontend*, he pasado al desarrollo del *backend* implementando la lógica del programa y generado la base de datos Haber seguido este orden me ha permitido adaptar la app a las necesidades del usuario y no forzarla a seguir las limitaciones del desarrollo del backend ni ser necesario remodelar la base de datos.

# 4.1 Frontend

## 4.1.1 Pantallas

La aplicación consta principalmente de cinco pantallas: autenticación, noticias recientes, artistas seguidos, búsqueda y pantalla de noticias de artista.

**Autenticación:**

Siempre que la aplicación no detecte un usuario registrado, se va a mostrar esta pantalla donde podrá iniciar sesión con una cuenta existente o registrarse con un correo y contraseña.

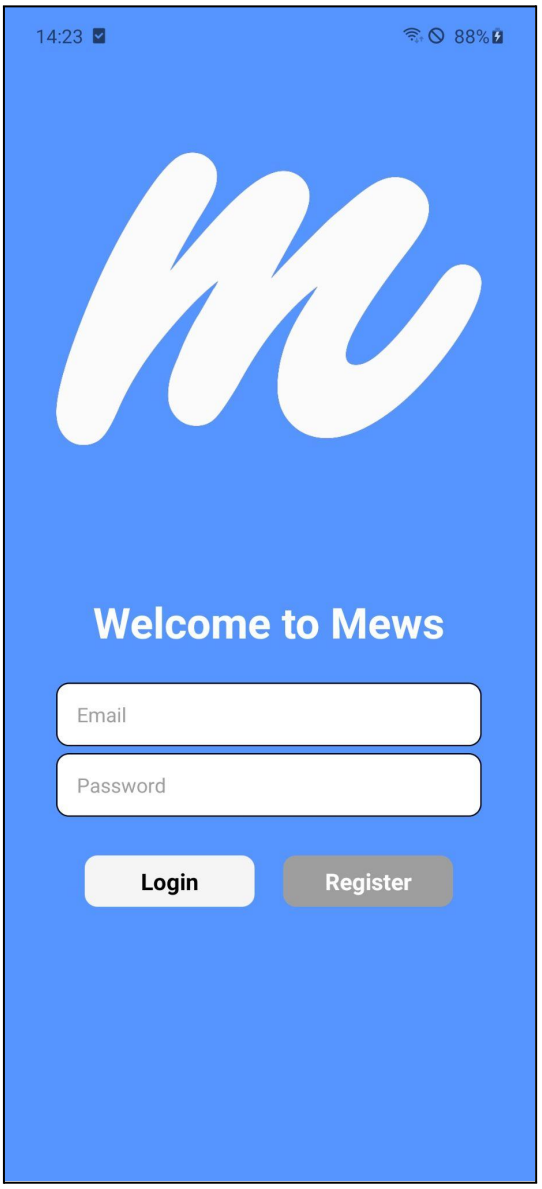


Figura 8: Pantalla de inicio de sesión y registro

**Reciente:**

Aquí se mostrará las noticias relacionadas con los artistas musicales que el usuario sigue actualmente ordenadas por fecha de publicación. Además en el header hay un menú de tres puntos que muestra el correo electrónico del usuario registrado y un pressable para cerrar sesión y volver al menú de autenticación.


Figura 9: Pantalla de noticias recientes

**Siguiendo:**

Aparecerá en forma de lista los distintos artistas a los que el usuario está suscrito junto con un botón de dejar de seguir. He añadido una barra de búsqueda para filtrarlos para facilitar la experiencia de usuario en caso de seguir a muchos artistas.

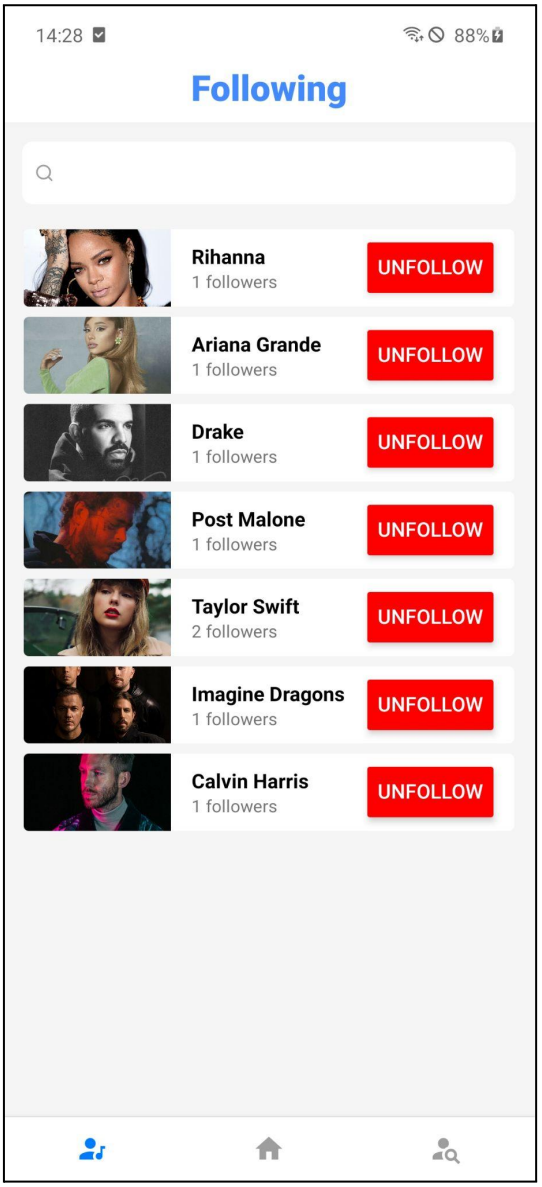


Figura 10: Pantalla de suscripciones

**Búsqueda:**

Al igual que en la anterior pantalla, habrá una barra de búsqueda que mostrará los artistas que cumplen con lo escrito de toda la base de datos. Aquí es donde el usuario podrá suscribirse a nuevos artistas o desuscribirse de los actuales.

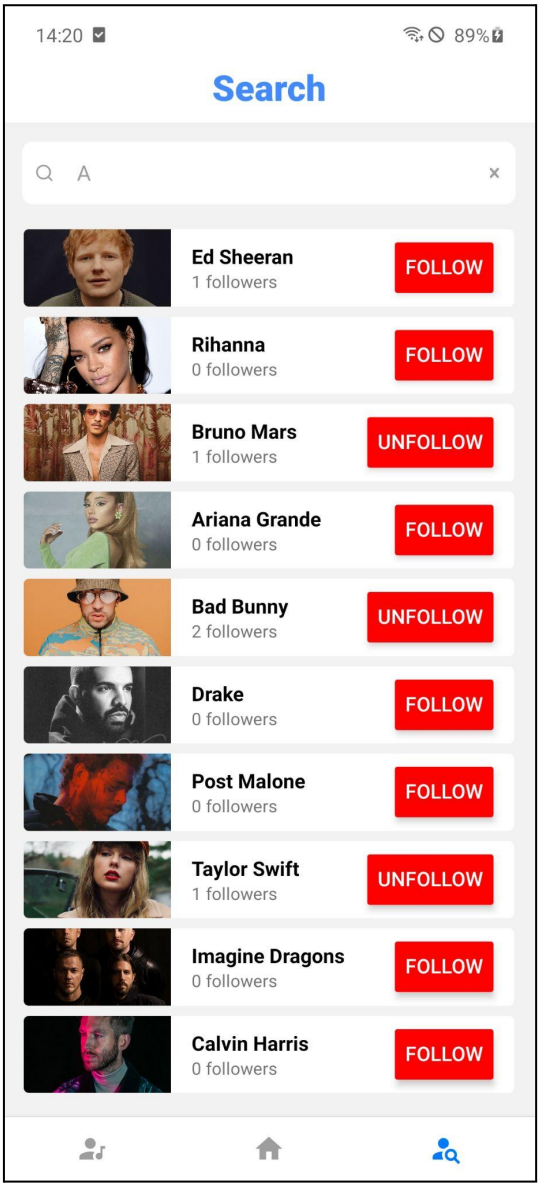


Figura 11: Pantalla de búsqueda

**Artista:**

Muestra una cabecera con la imagen, nombre del artista y suscriptores actuales en Mews. Debajo está la lista de noticias recientes del artista en cuestión.

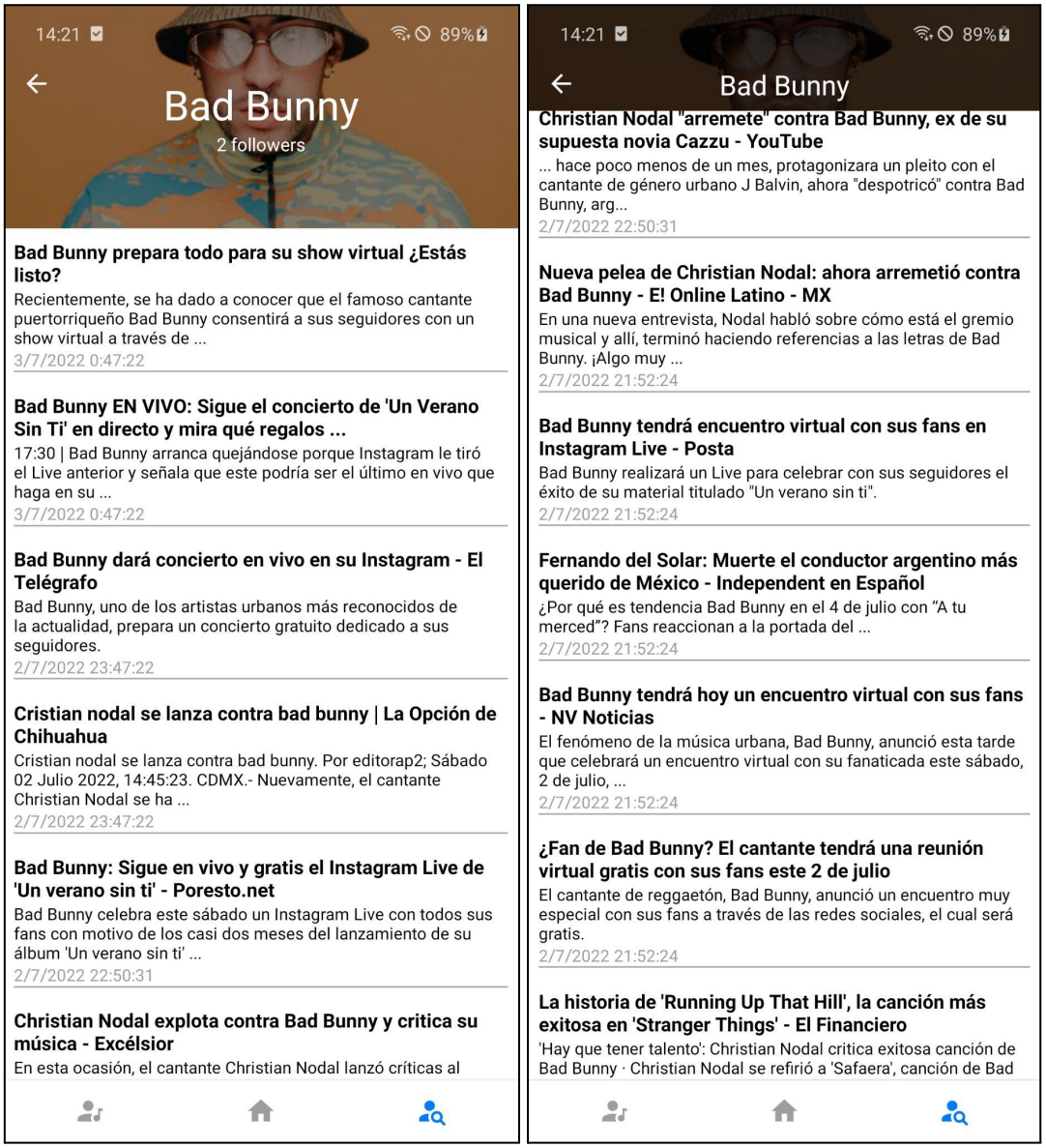


Figura 12: Pantalla de información de artista

### 4.1.2 Navegación

Para conseguir una correcta presentación y transición entre las múltiples pantallas de la aplicación he usado react navigation [9] como navegador y stack navigator y tab navigator como contenedores.

- **Stack navigator**: cada pantalla forma una capa dentro de la pila actual y es colocada encima de la anterior, de tal manera que si queremos retroceder se eliminan las superiores. Permite avanzar y retroceder en la app de forma lineal, además provee gestos y transiciones nativas.

- **Tab navigator**: Otro tipo de navegación muy común en las aplicaciones móviles es mediante pestañas. Esto implica tener una barra en la parte superior o inferior de la pantalla donde muestra las distintas secciones, en este caso siguiendo, reciente y búsqueda.

Se pueden combinar para una mejor experiencia de usuario tal como he hecho en la app. He creado un tabnavigator que contiene los stack navigator de recientes, siguiendo y búsqueda. Estos stacks a su vez contienen pantalla de artista y el webview de la noticia.

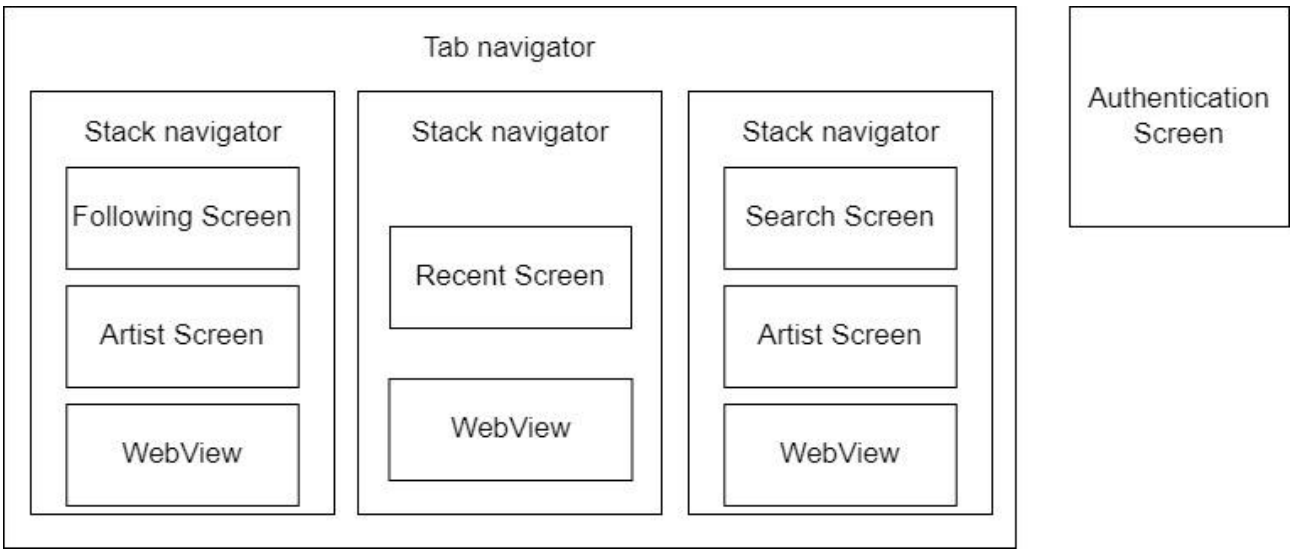


Figura 13: Esquema de navegación

# 4.2 Backend

## 4.2.1 Obtención de noticias

Como comenté previamente, la fuente de información única será Google Alerts. He creado una cuenta de Google dedicada para la app y de forma manual he añadido suscripciones para una variedad de artistas musicales estableciendo como fuente de origen las noticias y de frecuencia cuando se produzcan. A falta de tiempo y desconocimiento, no he implementado la generación de alertas y perfiles de artistas en la base de datos de forma automatizada, sin embargo en caso de crecer la aplicación sería necesaria. Tras añadir las alertas, Google Alerts cada vez que detecte una noticia que incluya al artista en cuestión, enviará un correo electrónico con el contenido a la cuenta asociada.

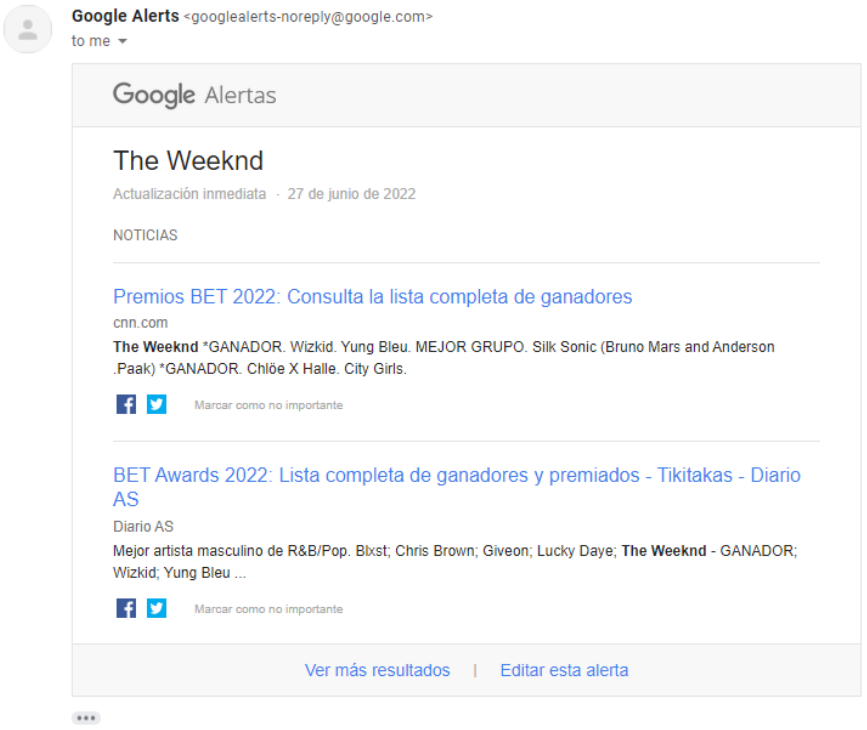


Figura 14: Ejemplo de correo electrónico con noticias

A pesar de haber seleccionado como frecuencia de notificación el momento de aparición de la noticia, he podido observar que Google Alerts tiene cierto retraso para cada suscripción de unas pocas horas provocando que los mensajes enviados contengan múltiples noticias, por lo que será necesario tener esto en cuenta en el proceso de análisis. Tras haber realizado lo anterior, ya tendríamos el canal de acceso a las noticias definido y lo siguiente sería obtener estos mensajes de la bandeja de entrada.

Como la base de datos ha de actualizarse constantemente a medida que el correo electrónico recibe nuevas noticias, fue necesario usar el sistema de notificación en tiempo real mediante el servicio Cloud Pub/Sub de Google. Esta herramienta está formada por dos partes, los temas y los suscriptores. El tema recibirá notificaciones de correos y enviará la información a todos los suscriptores asignados a este. Los pasos a seguir para captar las noticias han sido los siguientes:

1. Lo primero sería en Google Cloud crear un proyecto y habilitar la API de Gmail para poder acceder a su contenido mediante consultas.
2. Crear un tema y asignar permisos de publicación a la cuenta del servicio de Pub/Sub. Para activar las notificaciones del correo electrónico es necesario ejecutar la función *watch* de la API de Gmail [11]. Esto debe realizarse mínimo cada siete días para actualizar la vigencia del servicio.
3. Luego asignamos un suscriptor al topic que será el encargado de captar las notificaciones y enviarlas a nuestro programa. Para ello debe ser de tipo *Push* y requerirá de un *endpoint*. En este punto me surgió un gran problema y es que sólo permite de tipo HTTPS por lo que no pude realizar pruebas de funcionamiento hasta no haberlo desplegado en DigitalOcean.

4. Una vez el programa es capaz de recibir las notificaciones, lo siguiente es obtener los mensajes añadidos a la bandeja de entrada. Otra dificultad que supuso este sistema es la incapacidad de acceder directamente a estos emails ya que la notificación devuelve el identificador del historial, es decir estado actual de la bandeja. Por ello mediante la función *history* de la API de Gmail y pasando como parámetro el identificador del historial previo a la notificación y la opción *historyTypes: 'messageAdded'* para captar sólo los mensajes añadidos podemos obtener sus identificadores.
5. Tras obtener los identificadores de los mensajes ya podemos llamar a la función *getMessages* para obtener el cuerpo de estos. Como dije antes, puede haber varias noticias dentro de un mensaje por lo que debemos realizar el análisis. Primero, lo decodificamos ya que los datos están en base64 y lo convertimos a utf-8 para dar legibilidad. Finalmente siguiendo un patrón definido por observación de la estructuración de los correos de google alerts, extraemos las noticias. Este es otro de los problemas de la obtención de las noticias mediante este servicio y es el hecho de que las noticias no tienen un formato predefinido que posibilite dividirlas en sus distintos componentes (título, cuerpo, enlace) con total fiabilidad. Por ello, tras observar los mensajes identifiqué cierto patrón y con una gran muestra pude comprobar que el formateo fue exitoso.

    - Título: 1 - 2 líneas
    - Fuente: 1 línea
    - Cuerpo: 1 - 3 líneas
    - Enlace: 1 línea

Tras identificar las distintas partes de una noticia, generamos un objeto con el siguiente formato: { date, title, body, sourceLink } y lo clasificamos dentro de la base de datos de artistas existentes. Para ello simplemente se realiza la búsqueda de cada nombre en el contenido y si es exitosa se introduce en su subcolección de noticias del documento correspondiente. Al emplear este método existirá alguna noticia que se duplique ya que es posible que contenga múltiples artistas de la base de datos, sin embargo es la mejor opción planteada.

Finalmente ya habríamos logrado la obtención y clasificación de noticias automáticamente. El esquema de funcionamiento sería el siguiente:

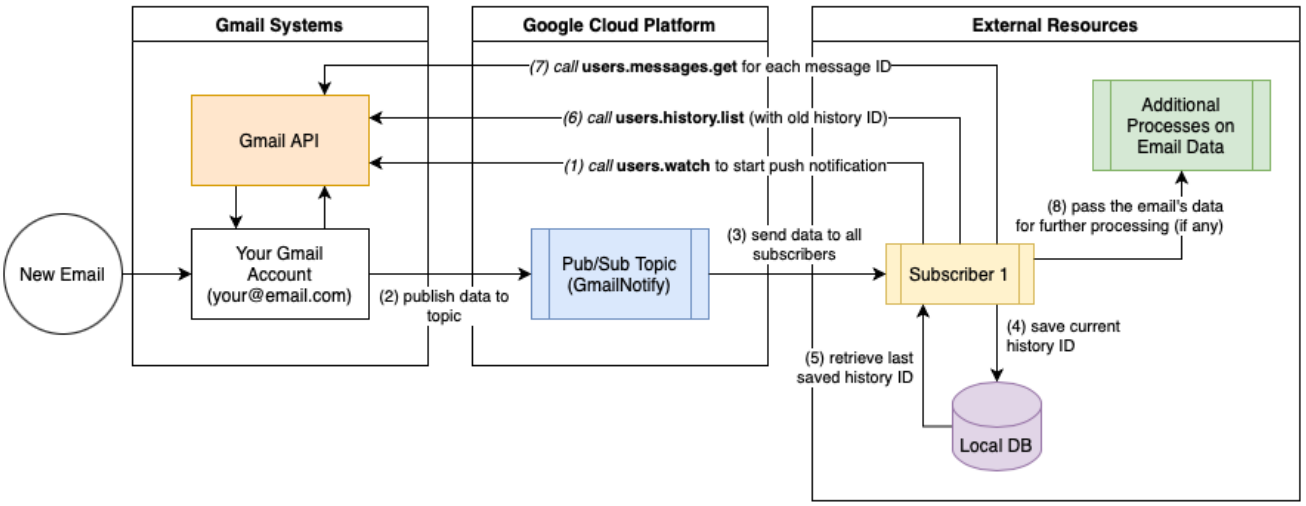


Figura 15: Esquema de funcionamiento del algoritmo de obtención de correos a partir de las notificaciones [10]

### 4.2.2 Autenticación

Para la autenticación de los usuarios en la aplicación he usado Firebase Authentication y en concreto la autenticación mediante correo electrónico y contraseña. En la aplicación introducimos las credenciales que son enviadas al servidor de Firebase donde son validadas y devuelve un *token* de autenticación. Esto permite acceder a la información del usuario desde el *frontend*. Una vez el usuario ha iniciado sesión en la aplicación, el *token* permanece en el almacenamiento local hasta que se llama a la función de cerrar sesión.

La base de datos de usuarios de Firebase Authentication está en una ubicación distinta al resto y no permite acceder directamente y añadir nuevos campos, que en este caso nos interesaría añadir un array de identificadores de artistas para almacenar las suscripciones actuales. Por ello, he generado una nueva colección en la base de datos: 'Users' y en el proceso de registro de usuarios generamos un documento nuevo con el identificador de este y el campo deseado. Además servirá para la futura información del usuario que quiera añadirse.

### 4.2.3 REST API

#### API

Las APIs son mecanismos que conectan dos aplicaciones, generalmente denominadas como cliente, que envía la petición y el servidor que la recibe y devuelve una respuesta. Sirve para simplificar la estructura del sistema ya que permite integrar módulos independientes y realizar cambios en ambos componentes sin afectarse directamente. Según su protocolo existen varios tipos:

- **SOAP**: *Simple Object Access Protocol* es el protocolo más antiguo y carece de flexibilidad por su estricto formato ya que usa XML para los datos de envío. Facilita la conexión entre aplicaciones con distinto lenguaje de programación y entorno.

- **RPC**: *Remote procedure call* realiza llamadas al servidor con múltiples parámetros para que este realice procedimientos y devuelva el resultado. Puede usar tanto XML como JSON para la transmisión de datos.

- **REST:** *Representational State Transfer* es el tipo de API más flexible y por ello ha ganado mucha popularidad ya que permite el envío de datos en una gran variedad de formatos como JSON, HTML y texto plano. El cliente contacta con el servidor enviando información que luego este emplea para llevar a cabo funciones internas y devolver los resultados de vuelta. A diferencia de las RPC, estas están orientadas al uso de recursos y no a las acciones. Además soporta un mayor número de operaciones CRUD.

Tras haber analizado los distintos tipos de APIs nos centraremos en las RESTful, que es la que he implementado en el servidor. Previamente nombré las operaciones CRUD que son los cuatro métodos básicos que permiten al usuario crear y gestionar los datos de la aplicación: crear, leer, actualizar y eliminar (*create, read, update, delete)* . En HTTP existen como los siguientes métodos:

**GET:** obtener un recurso existente en la base de datos.
- **200 ok:** recurso encontrado envía el contenido junto con este código.
- **404 not found:** recurso no encontrado.
- **400 bad request:** formato de la petición no válido.

Ejemplo: obtener la lista de artistas

```
router.get('/api/artist', (req, res) => {
        getAllArtists().then((artists) => {
                res.json(artists);
        });
});
```

**POST:** crear un recurso nuevo en la base de datos.
- **201 created:** éxito en la generación del nuevo recurso.

Ejemplo: registrar un usuario

```
router.post('/api/user/:uid', (req, res) => {
        addUser(req.body).then(() => {
                res.json({ success: true });
        });
});
```

**PUT:** actualiza un recurso y en caso de no existir lo crea. No lo he usado en la REST API para la aplicación ya que consideré que PATCH se adecuaba más a las necesidades puesto que este método debe usarse sólo cuando se reemplace completamente el recurso.

- **200 ok**: modificó el recurso existente.
- **201 created:** si no existía y generó un nuevo recurso.
- **204 no content:** modificó recurso existente.

**PATCH:** actualizar parcialmente un recurso ya existente.
- **200 ok**: modificó el recurso existente.
- **404 not found:** recurso no encontrado.

Ejemplo: Suscribir usuario a un artista

```
router.patch('/api/user/:uid/subs/:artistId', (req, res) => {
        addSubscription(req.params.uid, req.params.artistId).then(() => {
                res.json({ success: true });
        });
});
```

**DELETE:** eliminar un recurso existente.
- **200 ok:** recurso eliminado exitosamente.
- **404 not found:** recurso no encontrado en la base de datos.

Ejemplo: elimina artista de la lista de suscriptores del usuario.

```
router.delete('/api/user/:uid/subs/:artistId', (req, res) => {
        removeSubscription(req.params.uid, req.params.artistId).then(() => {
                res.json({ success: true });
        });
});
```

A continuación muestro en una tabla las equivalencias entre las operaciones CRUD, métodos HTTP y funciones de Firestore.

| CRUD | HTTP | Firestore |
|---|---|---|
| Create | POST, PUT | add, set |
| Retrieve | GET | get |
| Update | PUT, PATCH | update |
| Delete | DELETE | delete |

Tabla 1: Equivalencias entre CRUD, HTTP y Firestore

### 4.3.4 Paginación

Cuando queremos realizar una consulta de datos pero son una cantidad masiva de ellos, como en el caso de las noticias y queremos mostrarlas en la aplicación, envíar la lista

entera de golpe no es buena idea ya que son demasiados datos y procesarlos todos juntos supondría una bajada del rendimiento de la app, por ello usamos la paginación. La idea es mediante los parámetros de la llamada, dividir los datos por partes de tal forma que definimos un límite por petición. En el caso de Mews, cuando queremos obtener noticias de los artistas.
Se dividen los datos por partes y se envían con un tamaño que denominamos límite y a medida que la app requiera más información la consulta al servidor por secciones. De esta forma solo carga la información necesaria y disminuye así los tiempos de envío.
Para ello debemos definir un límite de la cantidad de elementos a enviar y tras la operación en la base de datos, el servidor junto con la información requerida manda el último documento obtenido. De tal forma que la aplicación lo almacena en una variable local y en la siguiente consulta lo añade como parámetro de inicio a la petición de datos. Ahora al acceder a la base de datos comenzará a recolectar la información a partir del comienzo establecido.

```
async function getNewsByArtistUsingLimit(artistId, limit) {
        const snapshot = await db
                .collection('Artists')
                .doc(artistId)
                .collection('News')
                .orderBy('date', 'desc')
                .limit(limit)
                .get();
        const lastNews = snapshot.docs[snapshot.docs.length - 1].id;
        const news = snapshot.docs.map((doc) => doc.data());
        return { news, lastNews };
}
```

Figura 16: Función de obtención de noticias con límite

Como se puede observar en el código, especificamos el número máximo de documentos a extraer mediante *limit* en la consulta a la base de datos. Tras realizar guardamos el último elemento en *lastNews* y lo devolvemos junto al resto de información. Finalmente, la consulta con punto de inicio la realizamos llamando a la función *startAfter*.

```
async function getNewsByArtistUsingLimitAndStartAfter(
        artistId,
        limit,
        startAfter
) {
        const snapshot = await db
                .collection('Artists')
                .doc(artistId)
                .collection('News')
                .orderBy('date', 'desc')
                .startAfter(startAfter)
                .limit(limit)
                .get();
        const lastNews = snapshot.docs[snapshot.docs.length - 1].id;
        const news = snapshot.docs.map((doc) => doc.data());
        return { news, lastNews };
}
```

Figura 17: Función de obtención de noticias con límite y punto de inicio

Esta funcionalidad la implementé como prueba para la aplicación para los envíos de las noticias. Sin embargo, tras realizar una serie de cambios en la base de datos para adaptarlas más al modelo noSQL, me imposibilitó usarla en las noticias recientes ya que no podía aplicar la funcionalidad de firebase de seleccionar el inicio y límite en uniones de múltiples colecciones.

### 4.3.5 Automatización de creación de artistas

Actualmente la adición de artistas a la base de datos se realiza procesando un fichero con sus datos introducidos manualmente al igual que las alertas. Dada la magnitud del proyecto no hay problema en realizarlo de esta forma, sin embargo como diseño de aplicación a mayor escala sería inviable. Por ello intenté automatizar el proceso empleando la API de Spotify [13] para obtener las imágenes de los artistas y mediante el módulo google-alerts-node-npm [12] para la creación de alertas pero me encontré con varios problemas.

En cuanto a Google Alerts, la librería de node.js que añadiría al proyecto está desactualizada y no funciona correctamente, además de la poca documentación que presenta. Por otro lado, la API de Spotify no permite la búsqueda de artistas por nombre ya que no son únicos y la distinción se hace mediante su identificador, por lo tanto implicaría obtener manualmente los identificadores y ya deja de ser un proceso automático como se desea.

### 4.3.6 Despliegue

Una vez terminado el desarrollo del backend, lo desplegué en un *droplet* DigitalOcean que es una máquina virtual alojada en la nube por la empresa. Lo primero que realicé fue la obtención de un dominio privado en Namecheap [15]: mewsapp.me. Con ayuda de los beneficios de Github Student Developer Pack [14] lo adquirí gratuitamente con una duración de un año. Además, me dio acceso a cien euros de saldo para DigitalOcean con los que mantengo el servidor activo.

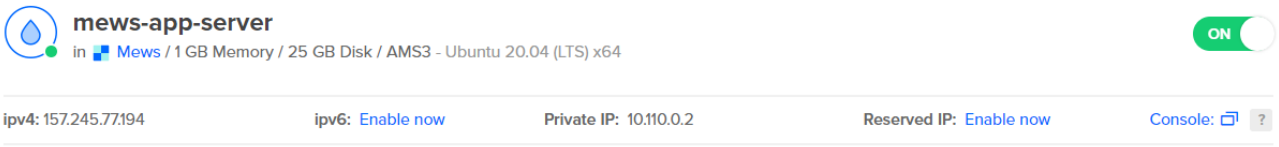


Figura 18: *Droplet* de DigitalOcean

Tras clonar el repositorio del *backend* en la máquina virtual, instalé pm2 y puse en ejecución el fichero que contiene la REST API mediante el siguiente comando:

pm2 start index.js --name mewsAPI

Esto permite que se ejecute en segundo plano mientras la máquina está encendida y da acceso a los registros en tiempo de ejecución, además de guardar en un fichero los errores obtenidos desde que se inició el proceso. Lo ideal sería que el servicio, a pesar de que el servidor se reinicie, se vuelva a lanzar automáticamente ya que por razones de mantenimiento DigitalOcean cada cierto tiempo fuerza el reinicio de las máquinas. Para ello, mediante pm2 startup y pm2 save, añadimos los procesos de pm2 en los scripts del arranque del sistema.

Lo siguiente sería activar un *firewall* para evitar que alguna aplicación no deseada con cualquier tipo de protocolo se pueda conectar al servidor y así sólo permitir HTTP y HTTPS que son los que usamos en el proyecto. En linux viene instalado por defecto el *firewall* ufw y simplemente debemos activarlo y añadir las configuraciones y recargar el servicio:

sudo ufw allow http

sudo ufw allow https

Para evitar la necesidad de escribir el puerto en el que tenemos ejecutándose el programa, instalamos un *proxy* server internamente. Un *proxy* server es básicamente un servidor que se encuentra en medio de otros dos en este caso node.js y el cliente y lo usamos para redireccionar las consultas del puerto 80 que es el por defecto al puerto 3000.

Como el endpoint del suscriptor de Google Pub/Sub requiere el protocolo HTTPS, he de activarlo, ya que actualmente consta como HTTP. Para ello es necesario obtener un certificado SSL y lo haremos de forma gratuita a través de Let 's encrypt [16] y usaremos el bot python-certbot-nginx que ofrece para realizar el proceso automáticamente. Finalmente ejecutamos sudo certbot --nginx -d mewsapp.me -d www.mewsapp.me, y escogemos la opción de redireccionar siempre a HTTPS. Como se puede apreciar en la figura 19, el dominio ya está validado y usa el protocolo deseado.

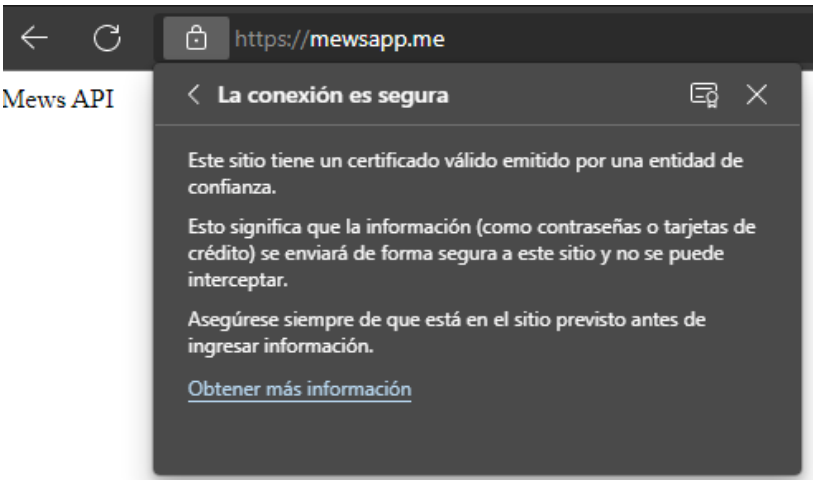


Figura 19: protocolo HTTPS

# Capítulo 4 Conclusiones y líneas futuras

Tras haber realizado con éxito la aplicación y cumplido los objetivos establecidos inicialmente, he adquirido conocimientos técnicos en el desarrollo de aplicaciones móviles con React Native y diseño de interfaces siguiendo una serie de estándares. Además he puesto en práctica los conocimientos previos para diseñar una base de datos que se adapte a las necesidades de la aplicación. Mediante una serie de patrones como el modelo-vista-controlador se ha creado una arquitectura simple, independiente y fácil de mantener. Finalmente, he sido capaz de desplegar el servidor en una máquina alojada en la nube y llevar el proyecto a producción para dar acceso al cliente a la aplicación.

A pesar de todo esto, durante el proceso de desarrollo han surgido problemas, especialmente con la obtención de noticias ya que la documentación relacionada con el sistema de notificaciones de Google Cloud es muy limitada y es un factor clave ya que permite la actualización en tiempo real de las noticias. Por otro lado, no fue posible automatizar la generación de artistas en la base de datos y sería una de las posibles adiciones a realizar en el proyecto para futuras versiones ya que es necesario para el crecimiento de la aplicación. Además, en cuanto a la base de datos, aunque realiza su función correctamente, requeriría de un rediseño ya que actualmente con la estructura definida no hay forma de aplicar la paginación en la obtención de noticias recientes de forma eficiente.

Asimismo, para lograr una mayor vistosidad en la aplicación sería favorable la creación de una nueva pestaña donde figuren los artistas con el mayor número de seguidores. Inicialmente el enfoque que le había dado a la app era centrándose más en un servicio personalizado que permitiera al usuario seguir a los artistas en los que está interesado y recibir notificaciones en relación a ellos y por esa razón he implantado la necesidad de registro para hacer uso de la aplicación. Sin embargo, a efectos prácticos esta imposición provoca la pérdida de interés por parte del usuario y limita en gran medida el público al que está dirigido tratándose por lo tanto de una mala idea de negocio. Es por ello que sería ideal eliminar esta restricción y sólo aplicarla en caso de ser necesario cuando quiera acceder a un servicio que lo requiera como la suscripción a algún artista.

# Capítulo 5 Summary and Conclusions

After successfully making the application and meeting the initially established objectives, I have acquired technical knowledge in the development of mobile applications with React Native and interface design following a series of standards. In addition, I have put into practice the previous knowledge to design a database that adapts to the needs of the application. Through a series of patterns such as the model-view-controller, I have created a simple, independent and easy to maintain architecture. Finally, I was able to deploy the server in a cloud-hosted environment and move the project to production to give the client access to the application.

Despite all this, I have faced several problems during the development process, especially obtaining news since the documentation related to the Google Cloud notification system is very limited and it is a key factor since it allows a real time news update. On the other hand, it was not possible to automate the generation of artists and it would be one of the possible additions to the project for future versions, because it is necessary for the application to expand. In addition, regarding the database, although it works correctly, I would suggest a redesign considering that currently with the defined structure there is no way to apply pagination efficiently when obtaining recent news.

Likewise, to achieve greater visibility, it would be favorable to create a new tab that displays the top artists with the largest number of followers. Initially the approach that I aimed for the app was to create a custom service that would allow the user to follow the artists they are interested in and receive notifications so that is why I made the registration mandatory. However, this imposition causes users to lose interest in the app and limits the target audience, thus making it a bad business idea. For this reason, it would be ideal to remove this restriction and only apply it if necessary when users want to access a service that requires it, such as a subscription to an artist.

# Capítulo 6 Presupuesto

Ahora hablaremos del presupuesto del proyecto hasta el día de hoy. Actualmente se está haciendo uso de varios recursos en su plan gratuito o en su período de prueba. Se han empleado recursos tales como Github Students, los cuales nos dotan de estas ventajas y descuentos en los servicios asociados con los que cuentan. Hasta la fecha, las mayores inversiones que se han realizado, han sido las de la mensualidad del servidor en Digitalocean, y por supuesto, el pago del desarrollador en los meses de desarrollo. Dichos pagos se pueden observar a continuación.

| Tipo | Descripción | Coste | Coste Total |
|---|---|---|---|
| RRHH | Desarrollador | 2000€ | 12000€ |
| Servicio | Namecheap | 0€ | 0€ |
| Servicio | Firebase | 0€ | 0€ |
| Servicio | DigitalOcean | 0€ | 0€ |

Tabla 2: Presupuesto actual

La base de datos de Firebase ofrece suficiente capacidad y potencia en su versión gratuita, por lo cual, no se considera necesario aumentar a planes superiores, no obstante en un futuro se puede reconsiderar, si son necesarios por ejemplos, otros servicios ofrecidos por la plataforma. Namecheap a través de Github Students, nos ha dotado de un dominio temporal, en caso de continuar el proyecto habría que pagar el servicio con su tarifa regular.

DigitalOcean, al igual que el resto de servicios asociados a GitHub Students, nos ha dotado de ventajas a la hora de usar su plataforma, abaratando el coste del servicio un número de meses, y al igual que en los casos anteriores, si se decide continuar con el proyecto, se necesitará pagar de forma regular el servicio, suponiendo un coste adicional. A continuación se pueden observar los costes, después de los pagos adicionales:

| Tipo | Descripción | Coste | Coste Total |
|---|---|---|---|
| RRHH | Desarrollador | 2000€ | 12000€ |
| Servicio | Namecheap | 5€ | 30€ |
| Servicio | Firebase | 0€ | 0€ |
| Servicio | DigitalOcean | 5€ | 30€ |

Tabla 3: Presupuesto sin GitHub Student Developer Pack

# Capítulo 7 Anexo

## 7.1 Obtención de las noticias en el cuerpo del mensaje

```
/**************************************************************************
*
* manage-news.js
*
**************************************************************************
*
* AUTOR: Pablo Bande Sánchez Girón
*
*
* FECHA: 04/7/2022
*
*
* DESCRIPCIÓN: Algoritmo que lleva a cabo el proceso de obtención, filtración y parsing de
* correos electrónicos a partir del identificador del mensaje.
*
**************************************************************************/
async function collectNews(newsId) {
	const message = await getMessage(newsId);
	const date = new Date(message.headers.date);

	let encoded_body = message.data.payload.parts[0].body.data;
	let decoded_body = Buffer.from(encoded_body, 'base64').toString('utf-8');

	let newsRawList = parseEmail(decoded_body);
	let newsList = newsRawList.map((news) => {
		const parsedNews = parseNews(news);
```

```
                return new News(
                        date,
                        parsedNews.title,
                        parsedNews.body,
                        parsedNews.sourceLink
                );
        });
        return newsList;
}
```

## 7.2 Obtención y clasificación de noticias a partir de las notificaciones de Google Pub / Sub

```
/**************************************************************************
 *
 * manage-news.js
 *
 **************************************************************************
 *
 * AUTOR: Pablo Bande Sánchez - Girón
 *
 * FECHA: 04/7/2022
 *
 * DESCRIPCIÓN: Algoritmo encargado de detectar los cambios en la bandeja de entrada y
* extraer los mensajes agregados para luego obtener las noticias y almacenarlas en la base de
* datos en la subcolección de los artistas correspondientes.
 **************************************************************************/
async function notifyAndAddNewsFromMail() {
        let historyId = fs.readFileSync(historyFile, 'utf8');
        let newHistory = await getHistory(historyId);

        if (newHistory.history != undefined) {
                let messages = newHistory.history[
                        newHistory.history.length - 1
                ].messagesAdded.map((msg) => msg.message.id);

                let newsList = await Promise.all(
                        messages.map(async (messageId) => {
                                let news = await collectNews(messageId);
                                console.log(messageId);
                                console.log(news);
                                return news;
                        })
                );
```

```
                newsList = newsList.flat();
                newsList.forEach((news) => {
                        addNews(news.convertToJson());
                });

                historyId = newHistory.historyId;
                fs.writeFileSync(historyFile, historyId);

                console.log('New news added');
        } else {
                console.log('No new news');
        }
}
```

## 7.3 Análisis de correos electrónicos y noticias

```
/*****************************************************************************
*
* parser.js
*
*****************************************************************************
*
* AUTOR: Pablo Bande Sánchez Girón
*
*
* FECHA: 04/7/2022
*
*
* DESCRIPCIÓN: Funciones empleadas para analizar y dar formato a los correos electrónicos y
*noticias.
*****************************************************************************/

function parseEmail(emailBody) {
        let bodyEnd = emailBody.indexOf(
                '\r\n- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -'
        );

        if (bodyEnd == -1) {
                return [];
        }

        emailBody = emailBody.slice(0, bodyEnd).replace(/\=\=\=.+?\=\=\=/, '');
        let newsList = emailBody.split(/\r\n\r\n/).filter(Boolean);
        return newsList;
```

```
}

function parseNews(newsRaw) {
	let news = {};
	let newsList = newsRaw.split(/\r\n/);

	if (newsList.length < 4) {
		throw new Error('Something went wrong parsing the news');
	}
	news.sourceLink = newsList.pop().slice(1, -1);
	let body = newsList.pop();
	while (newsList.at(-1).length > 50) {
		body = newsList.pop() + ' ' + body;
	}
	news.body = body;
	newsList.pop();
	news.title = newsList.join(' ');

	return news;
}
```

## 7.4 Obtención de noticias dado el identificador del artista

```
/*****************************************************************************
*
* firebase-news.js
*
*****************************************************************************
*
* AUTOR: Pablo Bande Sánchez Girón
*
*
* FECHA: 04/7/2022
*
*
* DESCRIPCIÓN: consulta de noticias de un artista determinado. Estas son las tres funciones
*para realizar la extracción: la primera obtiene todas las noticias de su subcolección y las otras
*dos tienen el sistema de paginación aplicado retornando la referencia al último documento
*extraído junto con la información requerida por el usuario.
*****************************************************************************/
async function getNewsByArtist(artistId) {
	const snapshot = await db
		.collection('Artists')
		.doc(artistId)
```

```
                .collection('News')
                .get();
        return snapshot.docs.map((doc) => doc.data());
}

async function getNewsByArtistUsingLimit(artistId, limit) {
        const snapshot = await db
                .collection('Artists')
                .doc(artistId)
                .collection('News')
                .orderBy('date', 'desc')
                .limit(limit)
                .get();
        const lastNews = snapshot.docs[snapshot.docs.length - 1].id;
        const news = snapshot.docs.map((doc) => doc.data());
        return { news, lastNews };
}

async function getNewsByArtistUsingLimitAndStartAfter(
        artistId,
        limit,
        startAfter
) {
        const snapshot = await db
                .collection('Artists')
                .doc(artistId)
                .collection('News')
                .orderBy('date', 'desc')
                .startAfter(startAfter)
                .limit(limit)
                .get();
        const lastNews = snapshot.docs[snapshot.docs.length - 1].id;
        const news = snapshot.docs.map((doc) => doc.data());
        return { news, lastNews };
}
```

## 7.5 Obtención de noticias recientes

```
/*****************************************************************************
*
* firebase-news.js
*
*****************************************************************************
*
* AUTOR: Pablo Bande Sánchez Girón
*
```

```
*
* FECHA: 04/7/2022
*
*
* DESCRIPCIÓN: consulta de noticias recientes de artistas a los que está suscrito el usuario al
*que corresponde el uid.
*****************************************************************************/
async function getRecentNews(uid) {
	const subscribedArtists = await getSubscriptions(uid);
	if (!subscribedArtists || subscribedArtists.length === 0) {
		return [];
	}

	const allNews = await Promise.all(
		subscribedArtists.map(async (artistId) => {
			const artistInfo = await getArtistsById(artistId);
			return getNewsByArtist(artistId).then((news) => {
				news.forEach((n) => {
					n.artist = artistInfo;
				});
				return news;
			});
		})
	);
	return allNews.flat().sort((a, b) => {
		return new Date(b.date) - new Date(a.date);
	});
}
```

# Bibliografía

[1] "React Native · Learn once, write anywhere". React Native · Learn once, write anywhere. https://reactnative.dev/.

[2] "Getting started | axios docs". Axios. https://axios-http.com/docs/intro.

[3] "Cloud computing services | google cloud". Google Cloud. https://cloud.google.com/.

[4] "Express - Node.js web application framework". Express - Node.js web application framework. https://expressjs.com/.

[5] "Firebase". Firebase. https://firebase.google.com/.

[6] "DigitalOcean - The developer cloud". DigitalOcean - The developer cloud. https://www.digitalocean.com/.

[7] "PM2 - home". PM2 - Home. https://pm2.keymetrics.io/.

[8] "Advanced load balancer, web server, & reverse proxy - NGINX". NGINX. https://www.nginx.com/.

[9] "React navigation | react navigation". React Navigation | React Navigation. https://reactnavigation.org/.

[10] N. Shah. "Understanding gmail's push notifications via google cloud pub/sub". Medium.https://medium.com/@eagnir/understanding-gmails-push-notifications-via-google-cloud-pub-sub-3a002f9350ef.

[11] "Gmail API | google developers". Google Developers. https://developers.google.com/gmail/api.

[12] "Google-alerts-api". npm. https://www.npmjs.com/package/google-alerts-api.

[13] "Web API | spotify for developers". Home | Spotify for Developers. https://developer.spotify.com/documentation/web-api/.

[14] "GitHub student developer pack". GitHub Education. https://education.github.com/pack.

[15] "Buy a domain name". Buy a domain name - Namecheap. https://www.namecheap.com/.

[16] "Let's encrypt". Let's Encrypt. https://letsencrypt.org/.

[17] "GitHub - alu0101225296/Mews-Front-End". GitHub. https://github.com/alu0101225296/Mews-Front-End.

[18] "GitHub - alu0101225296/Mews-API". GitHub. https://github.com/alu0101225296/Mews-API.