GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

POPCO, UNA APLICACIÓN PARA CINÉFILOS

Autor: Lorenzo Colino Hernández

Director: Israel Alonso Martínez

Co-Director: David Contreras Bárcena

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Popco, una aplicación para cinéfilos

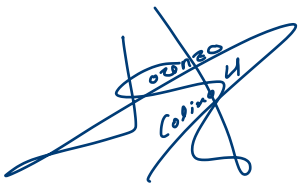
en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2023/24 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.



Fdo.: Lorenzo Colino Hernández Fecha: 03/ 07/ 2024

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo: Israel Alonso Martínez Fdo: David Contreras Bárcena

Fecha: 08/ 07/ 2024

Agradecimientos

A mis padres.

POPCO, UNA APLICACIÓN PARA CINÉFILOS

Autor: Colino Hernández, Lorenzo.
Director: Alonso Martínez, Israel.
Co-director: Contreras Bárcena, David.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

El proyecto Popco ha desarrollado una aplicación móvil para cinéfilos que combina funcionalidades sociales y de gestión de contenido cinematográfico, utilizando tecnologías avanzadas. Los resultados incluyen una interfaz de usuario atractiva y funcional, integración eficiente de datos y herramientas robustas de *backend*, logrando así una plataforma completa y lista para el disfrute del usuario.

Palabras clave: aplicación móvil, cine, arquitectura, desarrollo *sofware*.

1. Introducción

En la era digital, el consumo de contenido cinematográfico ha crecido exponencialmente, impulsado por plataformas de *video on demand*. Sin embargo, la necesidad de una plataforma que permita a los cinéfilos no solo consumir, sino también interactuar y compartir experiencias de manera integral, se ha hecho evidente. Este proyecto aborda esta necesidad mediante el desarrollo de una aplicación móvil que integra funcionalidades sociales y herramientas de gestión de contenido cinematográfico.

2. Definición del proyecto

Popco consiste en una aplicación móvil destinada a los cinéfilos, que permite seguir la actividad cinematográfica de amigos, compartir reseñas de películas, acceder a estadísticas personalizadas de la vida y experiencias cinematográficas de otros usuarios. Cabe destacar sobre todo la posibilidad de participar en comunidades de cada película con sistemas de reconocimiento que reconozcan a los usuarios más entusiastas su pasión y cultura de cine a través de rankings induciendo así un sentimiento de pertenencia y un espíritu competitivo que consiga establecer un vínculo entre la aplicación y el usuario. Popco provee a cualquier cinéfilo de todas las herramientas para una sustancial mejora de su experiencia consumiendo cine.

3. Descripción de la aplicación

El proyecto se compone de dos piezas fundamentales: el *frontend* y el *backend*. El reto principal reside en encajar estas dos piezas de manera eficiente para garantizar una experiencia de usuario óptima. El frontend es la interfaz de usuario, cuya calidad influye en gran medida en la satisfacción en la experiencia del usuario. Este componente actúa como un lienzo dinámico que necesita ser alimentado con información relevante. Esta información proviene del *backend* desarrollado con Firebase y de diversas API, como TheMovieDatabase (TMDB), que proporciona datos detallados sobre contenido cinematográfico. Utilizando tecnologías como React Native y Expo, se ha logrado crear una interfaz intuitiva, rápida y visualmente atractiva, que facilita la interacción del usuario con la aplicación. Firebase

provee un arsenal de herramientas completo y adaptable a cualquier proyecto. No obstante, una aplicación tan completa y técnicamente tan profunda implica la implementación de muchas otras tecnologías. El *caching* evita la redundancia de peticiones al *backend*. Las peticiones implican un coste económico y un deterioro de la experiencia de uso debido al retardo temporal que entrañan. Una tecnología como el *caching*, que registra la información en memoria evitando así futuras peticiones innecesarias, hacen de Popco una aplicación mucho más profesional. La implementación de *caching* ha posible a través de React Query. React Native ha dotado al desarrollo de compartimentalización y reutilización de componentes además de facilitar el estado de la aplicación, una cuestión vital para hacer una aplicación de esta envergadura un proyecto escalable y perfectamente gestionable. El almacenamiento en la memoria local del dispositivo ha sido posible gracias a AsyncStorage. Finalmente, la materialización de todo el código en una aplicación descargable desde cualquier tienda de aplicaciones ha pasado por el uso de Expo. Difícilmente se puede llegar a resumir el uso de tan variado y extenso conjunto de tecnologías que hacen de Expo una herramienta robusta, eficiente, segura y muy útil para cualquier cinéfilo.

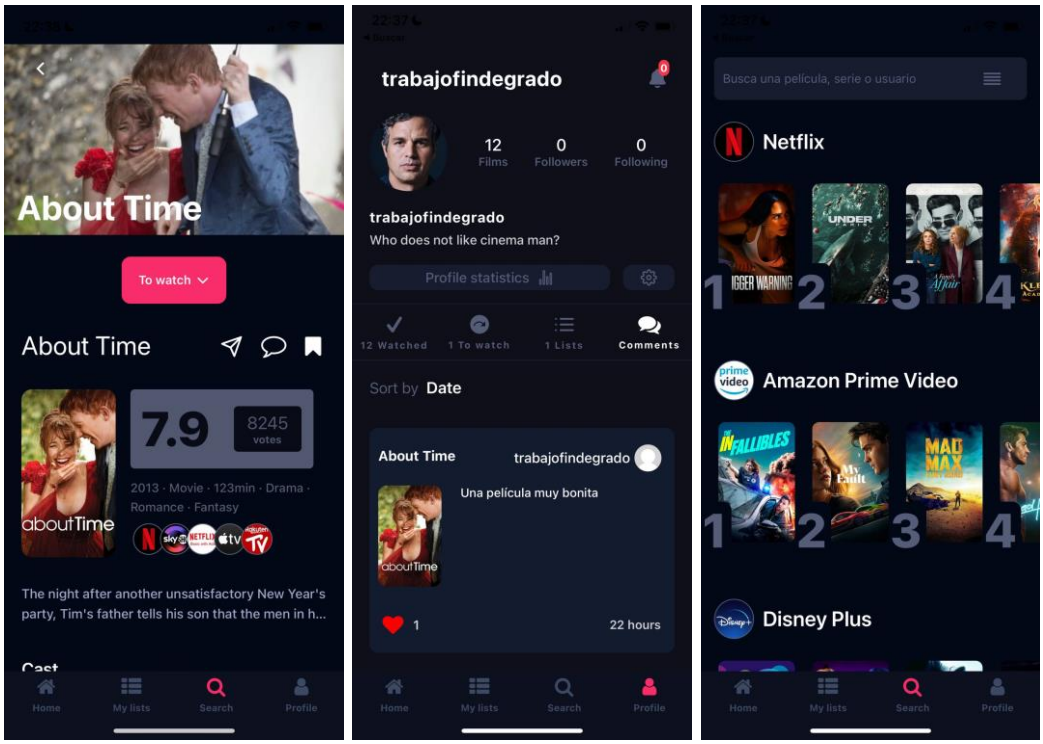


Ilustración 1 – Pestañas más importantes de la aplicación

4. Resultados

El propósito final de cualquier proyecto es materializar la idea inicial en un producto útil y funcional. En este caso, no solo se han cumplido con creces las expectativas, sino que la aplicación ya es una realidad tangible y disponible para su descarga en cualquier dispositivo. La capacidad de utilizarla personalmente y observar su comportamiento en un entorno real representa un completo éxito, validando el esfuerzo y la dedicación invertidos en su desarrollo. Este logro es un testimonio del trabajo meticuloso y la planificación estratégica llevada a cabo a lo largo del proyecto.

La interfaz de usuario, inicialmente esbozada en un boceto, se ha materializado de manera correcta, logrando una apariencia visual seria y profesional. El éxito en el *frontend* se debe al estudio exhaustivo de tecnologías modernas y componentes nativos, creando una interfaz atractiva, funcional y accesible. En cuanto al *backend*, optar por Firebase como BaaS ha sido una decisión acertada, proporcionando herramientas robustas y adaptables, permitiendo un desarrollo rápido y eficiente. Aunque la dependencia de Firebase puede requerir reevaluación futura a medida que crezca la base de usuarios, esta elección ha asegurado una fácil escalabilidad y manejo eficiente del crecimiento.

Otro resultado exitoso ha sido la integración de una amplia gama de tecnologías en un solo proyecto, un logro que ha sido posible gracias a un profundo entendimiento de cada una de ellas. Este enfoque holístico ha permitido crear una aplicación que ofrece una diversidad de posibilidades a los usuarios, equiparándose a las mejores aplicaciones del mercado. La conjunción de React Native, Expo, y React Query, entre otras, ha enriquecido la plataforma, dotándola de funcionalidades avanzadas y una experiencia de usuario de alta calidad. Este éxito no solo resalta la capacidad técnica detrás del proyecto, sino también la visión de ofrecer a los usuarios una herramienta completa y sofisticada que satisface múltiples necesidades.

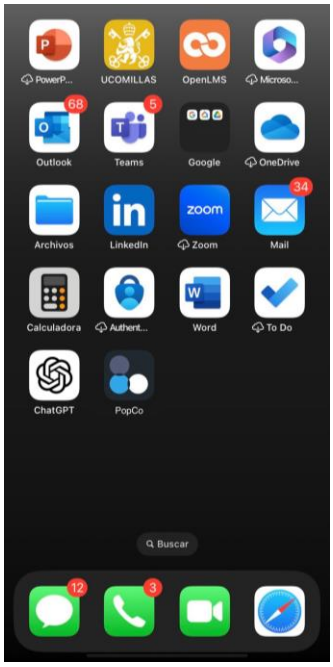


Ilustración 2 – Aplicación beta descargada en un móvil de Apple

5. Conclusiones

El mayor logro de este proyecto ha sido la creación de una red social dedicada a los cinéfilos, donde los usuarios pueden buscar y obtener información detallada sobre películas, interactuar, compartir opiniones y conectar con otros entusiastas del cine. Popco ofrece perfiles personalizados, seguimiento de películas, listas temáticas y un sistema de recomendaciones personalizadas, entre otras funcionalidades. A lo largo del desarrollo, he adquirido una comprensión integral del proceso de conceptualización y materialización de

una aplicación móvil, abarcando desde la idea inicial hasta la implementación final, con un enfoque constante en la excelencia y atención meticulosa en cada fase.

Este proyecto ha nutrido en mí un espíritu perfeccionista, impulsándome a profundizar en el aprendizaje de todas las tecnologías disponibles y asegurando que cada decisión y funcionalidad implementada estuvieran alineadas con los más altos estándares de calidad. La planificación y conceptualización detalladas antes de comenzar con la programación se han revelado como métodos productivos, permitiendo la identificación de posibles obstáculos y la formulación de soluciones eficientes. El diseño meticuloso y la iteración continua han sido cruciales para ajustar y mejorar la aplicación, asegurando que cumpla con los estándares técnicos y satisfaga las expectativas de los usuarios.

Además, el desarrollo de Popco me ha permitido mejorar mi capacidad para aprender y adaptarme a la documentación técnica de diversas tecnologías. He aprendido a navegar eficientemente por guías de usuario, manuales técnicos y recursos en línea, extrayendo información relevante y aplicándola de manera práctica. Esta habilidad ha sido fundamental para resolver problemas técnicos, implementar nuevas funcionalidades y optimizar el rendimiento de la aplicación. En resumen, este proyecto me ha proporcionado una visión integral del desarrollo de aplicaciones móviles, destacando la importancia de una planificación meticulosa, el aprendizaje continuo y la adaptación a nuevas tecnologías, conocimientos que serán valiosos en futuros proyectos.

6. Referencias

[1] Firebase y Firestore. Recuperado de https://firebase.google.com/docs/firestore?hl=es-419

[2] API The Movie Data Base (TMDB). Recuperado de https://www.themoviedb.org/

[3] React. Recuperado de https://es.react.dev/

[4] React Native. Recuperado de https://reactnative.dev/

POPCO, UNA APLICACIÓN PARA CINÉFILOS

Author: Colino Hernández, Lorenzo.
Supervisor: Alonso Martínez, Israel.
Co-Supervisor: Contreras Bárcena, David.
Collaborating Entity: ICAI – Universidad Pontificia Comillas

ABSTRACT

The Popco project has developed a mobile application for movie enthusiasts that combines social and cinematic content management functionalities using advanced technologies. The results include an attractive and functional user interface, efficient data integration, and robust backend tools, resulting in a complete platform ready for user enjoyment.

Keywords: mobile application, cinema, architecture, software development.

1. Introduction

In the digital era, the consumption of cinematic content has grown exponentially, driven by video-on-demand platforms. However, the need for a platform that allows movie enthusiasts not only to consume but also to interact and share experiences comprehensively has become evident. This project addresses this need by developing a mobile application that integrates social functionalities and tools for managing cinematic content.

2. Project definition

Popco is a mobile application designed for movie enthusiasts, allowing users to follow the cinematic activity of friends, share movie reviews, and access personalized statistics about the cinematic lives and experiences of other users. Notably, it includes the possibility to participate in communities for each movie with recognition systems that acknowledge the most passionate and knowledgeable users through rankings, fostering a sense of belonging and a competitive spirit that establishes a bond between the application and the user. Popco provides any movie enthusiast with all the tools for a substantial improvement in their movie-watching experience.

3. Application description

The project consists of two fundamental components: the frontend and the backend. The main challenge lies in fitting these two components together efficiently to ensure an optimal user experience. The frontend is the user interface, whose quality greatly influences user satisfaction. This component acts as a dynamic canvas that needs to be fed with relevant information. This information comes from the backend developed with Firebase and various APIs, such as TheMovieDatabase (TMDB), which provides detailed data on cinematic content. Using technologies like React Native and Expo, an intuitive, fast, and visually appealing interface has been created, facilitating user interaction with the application. Firebase provides a comprehensive and adaptable toolset for any project. However, such a complete and technically deep application implies the implementation of many other technologies. Caching avoids redundancy in backend requests, which entail economic costs and deterioration of the user experience due to the associated temporal delay. A technology

like caching, which records information in memory thus avoiding unnecessary future requests, makes Popco a much more professional application. Caching implementation has been made possible through React Query. React Native has endowed the development with component compartmentalization and reuse, as well as facilitating application state management, a vital issue for making a project of this scale scalable and perfectly manageable. Local device storage has been made possible through AsyncStorage. Finally, the materialization of all the code into a downloadable application from any app store has been achieved using Expo. It is difficult to summarize the use of such a varied and extensive set of technologies that make Expo a robust, efficient, secure, and very useful tool for any movie enthusiast.

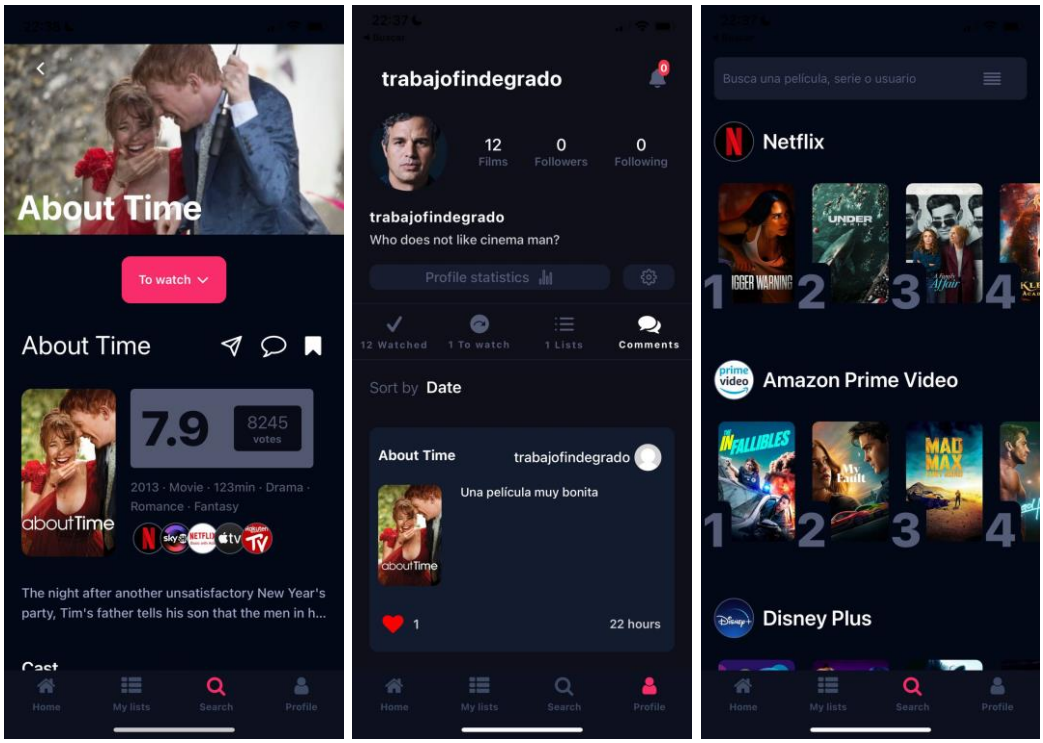


Illustration 1 – Most important tabs of the application

4. Results

The goal of any project is to materialize the initial idea into a useful and functional product. In this case, not only have expectations been far exceeded, but the application is already a tangible reality and available for download on any device. The ability to use it personally and observe its behavior in a real environment represents a complete success, validating the effort and dedication invested in its development. This achievement is a testament to the meticulous work and strategic planning carried out throughout the project. The user interface, initially sketched in a draft, has materialized impeccably, achieving a serious and professional visual appearance. The success in the frontend is due to the thorough study of modern technologies and native components, creating an attractive, functional, and accessible interface. Regarding the backend, opting for Firebase as BaaS was a wise decision, providing robust and adaptable tools, allowing for quick and efficient development. Although reliance on Firebase may need reevaluation in the future as the user base grows,

this choice has ensured easy scalability and efficient growth management. Another successful outcome has been the integration of a wide range of technologies into a single project, an achievement made possible by a deep understanding of each. This holistic approach has allowed the creation of an application that offers users a diversity of possibilities, equaling the best applications on the market. The conjunction of React Native, Expo, and React Query, among others, has enriched the platform, endowing it with advanced functionalities and a high-quality user experience. This success not only highlights the technical capability behind the project but also the vision of offering users a complete and sophisticated tool that meets multiple needs.

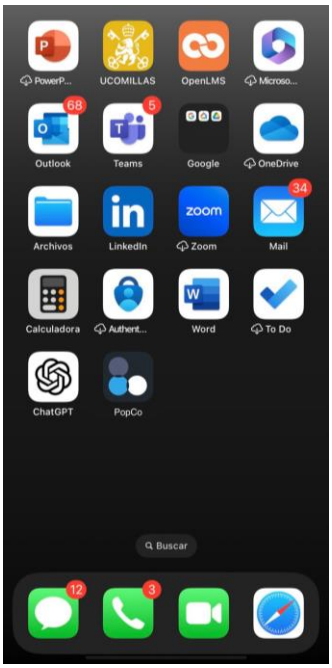


Illustration 2 – Beta application downloaded on an Apple mobile

5. Conclusions

The greatest achievement of this project has been the creation of a social network dedicated to movie enthusiasts, where users can search and obtain detailed information about movies, interact, share opinions, and connect with other movie lovers. Popco offers personalized profiles, movie tracking, thematic lists, and a personalized recommendation system, among other functionalities. Throughout the development, I have gained a comprehensive understanding of the process of conceptualizing and materializing a mobile application, from the initial idea to the final implementation, with a constant focus on excellence and meticulous attention at every stage. This project has nurtured a perfectionist spirit in me, driving me to deepen my learning of all available technologies and ensuring that every decision and implemented functionality aligned with the highest quality standards. Detailed planning and conceptualization before starting programming have proven to be productive methods, allowing for the identification of potential obstacles and the formulation of efficient solutions. Meticulous design and continuous iteration have been crucial for adjusting and improving the application, ensuring it meets technical standards and user expectations. Furthermore, developing Popco has enhanced my ability to learn and adapt to

the technical documentation of various technologies. I have learned to navigate user guides, technical manuals, and online resources efficiently, extracting relevant information and applying it practically. This skill has been fundamental in solving technical problems, implementing new functionalities, and optimizing application performance. In summary, this project has provided me with a comprehensive view of mobile application development, highlighting the importance of meticulous planning, continuous learning, and adaptation to new technologies, knowledge that will be valuable in future projects.

6. References

[1] Firebase y Firestore. Retrieved from https://firebase.google.com/docs/firestore?hl=es-419

[2] API The Movie Data Base (TMDB). Retrieved from https://www.themoviedb.org/

[3] React. Retrieved from https://es.react.dev/

[4] React Native. Retrieved from https://reactnative.dev/

Índice de la memoria

Índice de figuras

Capítulo 1. INTRODUCCIÓN

Motivación del proyecto

En la era digital, la sociedad está invirtiendo cada vez más tiempo en actividades de entretenimiento, particularmente en el consumo de películas y series. Este cambio en los hábitos de consumo se ha visto acelerado por el surgimiento de plataformas de *video on demand* (VOD). Un informe de Nielsen de 2021 reveló que los estadounidenses pasaban un promedio de 4 horas y 49 minutos diarios viendo contenido en estas plataformas, un aumento significativo comparado con años anteriores. Este fenómeno es global, con un aumento considerable del tiempo dedicado al entretenimiento digital en todo el mundo, evidenciando un cambio en la manera en que las personas disfrutan de su tiempo libre.

En este contexto, una aplicación que conecte a los cinéfilos y les permita compartir sus experiencias y recomendaciones se presenta como una propuesta muy interesante y relevante.

Mi motivación para emprender este proyecto se originó de una necesidad personal como cinéfilo de contar con una plataforma donde pudiera seguir de cerca la actividad cinematográfica de mis amigos. Siempre he anhelado un espacio donde pudiera ver qué películas están viendo mis amigos, cuáles desean ver, y sus opiniones sobre cada una. La idea de poder compartir y discutir nuestras experiencias cinematográficas en un entorno dedicado exclusivamente al cine me resultaba muy atractiva. Quería crear una red social que no solo facilitara estas interacciones, sino que también ofreciera un ambiente donde los amantes del cine pudieran conectarse y compartir su pasión de manera más profunda y significativa.

Además de la función social, era crucial para mí que la plataforma también funcionara como una herramienta integral para los cinéfilos. Imaginé una aplicación que permitiera a los

usuarios guardar y organizar las películas que desean ver, buscar información detallada sobre ellas y acceder a estadísticas personalizadas. Saber cuántas películas ha visto un usuario, tener una lista ordenada de deseos y ver sinopsis, repartos y críticas en un solo lugar eran características esenciales que deseaba implementar. La idea era simplificar y enriquecer la experiencia de descubrir y gestionar el contenido cinematográfico, eliminando la necesidad de utilizar múltiples aplicaciones y métodos desorganizados.

Otro aspecto fundamental de mi visión era la creación de comunidades dentro de la plataforma que reconocieran a los usuarios más apasionados. Implementar un sistema de *likes* en las reseñas permitiría identificar y destacar a los mayores fans de una serie o película, fomentando un sentido de pertenencia y competencia amistosa. Estas comunidades no solo incentivarán la participación activa, sino que también crearán un ambiente donde los usuarios se sientan valorados y reconocidos por sus contribuciones. Al permitir que los cinéfilos se conecten y compartan sus opiniones en un entorno que celebre su pasión, la aplicación se convertiría en un lugar de encuentro indispensable para los amantes del cine, proporcionando tanto valor social como funcional.

Finalmente cabe resaltar que soy un entusiasta de la programación, especialmente de la programación de aplicaciones móviles, y siempre he tenido el anhelo de crear un proyecto de gran envergadura que involucre tanto el *backend* como el *frontend*. La idea de desarrollar una aplicación completa, desde la concepción de la idea hasta su implementación final, ha sido una considerable fuente de motivación.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

Este capítulo pretende explicar todas las tecnologías de las que se ha hecho uso a lo largo del proyecto. Cabe resaltar la distinción entre aquellas tecnologías jerárquicamente más importantes, los marcos de trabajo y aquellas tecnologías utilizadas en pos de la consecución de ciertas funcionalidades.

El proyecto ha fundado sus bases sobre cuatro herramientas esencialmente.

2.1 NPM

El Node Package Manager [11] es el gestor de paquetes predeterminado de Node.js. Es una herramienta que permite instalar, actualizar, borrar e incluso compartir dependencias desarrolladas en Javascript.

2.2 NODE.JS

Node.js es un entorno de ejecución de Javascript. Permite la ejecución de Javascript en servidores, a pesar de que originalmente era un lenguaje pensado única y exclusivamente para navegadores.

2.3 REACT Y REACT NATIVE

React [3] es una biblioteca de JavaScript para construir interfaces de usuario, desarrollada y mantenida por Facebook y una comunidad de desarrolladores.

Entre las ventajas que tiene React se han de destacar:

- Compartimentalización

React facilita la creación de componentes convirtiendo la aplicación en un ensamblaje de diferentes módulos, que además son reutilizables.

- Virtual DOM

React hace uso de un DOM virtual para minimizar las manipulaciones e interacciones con el DOM real. El DOM es conceptualmente el árbol jerárquico de componentes a renderizar. El DOM gestionado por Vanilla Javascript cuenta con enormes ineficiencias en la medida que renderiza todo el proyecto sin importar qué componentes han cambiado. El DOM virtual de React solo cambia aquellos componentes dependientes de variables o estado que haya cambiado, reduciendo el coste que supone renderizar a estrictamente lo necesario.

- Comunidad

React es mundialmente popular. Ha desplazado al resto de tecnologías especialmente a vanilla Javascript, el Javascript original.

React Native [4] es un entorno de desarrollo de aplicaciones móviles nativas, basado en Javascript y React. De forma que con un código íntegramente escrito en Javascript y JSX a través de React Native un desarrollador es capaz de diseñar aplicaciones perfectamente funcionales en IOs y Android.

2.4 EXPO

Expo [6] es una plataforma y un conjunto de herramientas que simplifican la creación de aplicaciones React Native. Ofrece una variedad de bibliotecas y servicios que agilizan el desarrollo, la prueba y el lanzamiento de aplicaciones.

Capítulo 3. ESTADO DE LA CUESTIÓN

Al desarrollo de cualquier proyecto debe preceder un estudio de investigación en el cual se busquen minuciosamente alternativas ya existentes a lo que se pretende diseñar. El propósito de esta misión no es identificar si el proyecto es viable o si ya hay alternativas en el mercado, sino absorber las mejores ideas de proyectos parecidos para poder nutrir y fortalecer el propio.

¿Hay algo similar en el mercado?

La respuesta a esta pregunta es parcialmente afirmativa. Actualmente existe una aplicación de referencia en la comunidad cinéfila llamada Letterboxd. Esta es una aplicación muy similar a Popco y ofrece, en términos generales, propuestas parecidas a la versión actual de Popco. Letterboxd es una red social enfocada al cine y además ofrece información acerca de cualquier película.

Es importante recalcar la apostilla "versión actual" que se ha utilizado para denominar la aplicación tal y como es actualmente, ya que Popco aspira a ser una aplicación muy diferente en un futuro. La versión actual es tan solo una versión beta.

Letterboxd es una aplicación muy acogida por la comunidad esencialmente gracias a su longevidad. Letterboxd se fundó en 2011 y desde entonces no ha habido grandes cambios estructurales. Letterboxd, como la gran mayoría de aplicaciones de cine carece de diversidad de funcionalidades y se limita a la posibilidad de reseñar películas además de proveer información sobre las mismas.

Popco es solo un embrión de una aplicación que pretende ser muy diferente a lo que es hoy.

Capítulo 4. DEFINICIÓN DEL TRABAJO

4.1 JUSTIFICACIÓN

En el contexto actual, el consumo de cine y series ha alcanzado niveles sin precedentes gracias al auge de las plataformas de video on demand (VOD) como Netflix, Amazon Prime, HBO y Disney+.

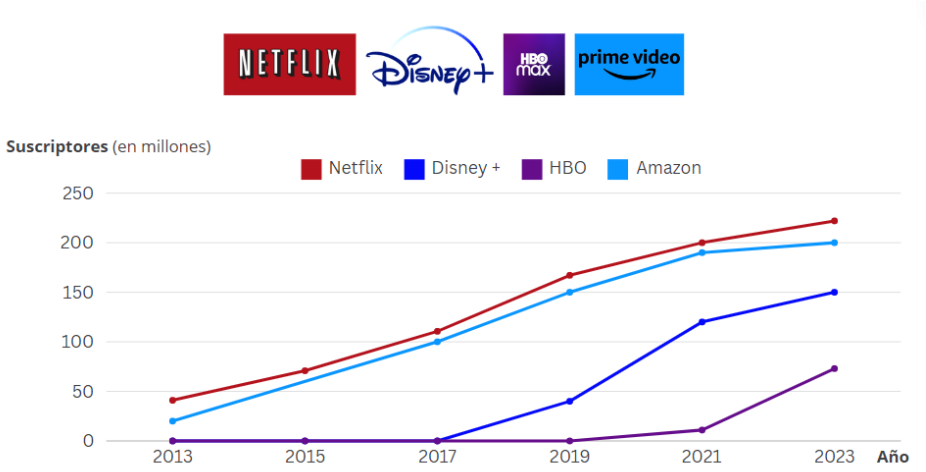


Ilustración 3. Crecimiento de las plataformas VOD a lo largo de los años

La facilidad de acceso a una vasta biblioteca de contenido en cualquier momento y lugar ha transformado los hábitos de entretenimiento de la sociedad, impulsando a millones de usuarios a dedicar cada vez más tiempo a ver películas y series. En este escenario, surge la necesidad de una aplicación especializada que no solo permita gestionar y descubrir contenido, sino también conectar a los cinéfilos en una comunidad dedicada exclusivamente a su pasión por el cine. Esta aplicación no solo responde a la creciente demanda de herramientas que faciliten el consumo y la interacción alrededor del cine, sino que también aprovecha una oportunidad de mercado significativa en un sector en plena expansión.

Popco, diseñada como una red social exclusivamente para cinéfilos, permite a los usuarios actuar entre sí mediante reseñas, *likes* y comentarios, creando una comunidad dinámica y comprometida. Este enfoque especializado no solo atrae a una audiencia apasionada, sino que también incrementa el tiempo de permanencia en la plataforma, lo que se traduce en mayores oportunidades de monetización.

Además, Popco ofrece una serie de herramientas únicas que satisfacen todas las necesidades de un cinéfilo. Desde herramientas para decidir qué película ver cuando la indecisión hace imposible elegir una, la capacidad de rastrear estadísticas personales, como cuántas horas de cine han visto en un año o cuál es el actor cuyas películas más han visto, hasta funcionalidades de gamificación como rankings en cada comunidad basados en los usuarios con más *likes* en sus reseñas. Estas características no solo enriquecen la experiencia del usuario, sino que también fomentan la interacción y la competencia amistosa, incentivando a los usuarios a ser más activos y participativos. Este nivel de personalización y compromiso es altamente valorado en el mercado actual, donde las experiencias personalizadas y las comunidades de nicho son claves para el éxito

Popco no es simplemente una aplicación, es un negocio. Popco es una fuente natural y orgánica de información. Los usuarios comparten voluntariamente todas sus preferencias y experiencias cinematográficas. Desde qué han visto, qué desean ver hasta qué opinan de cada película.

Con dichos datos se podría potencialmente conocer con verdadera exactitud qué películas gustarán a cada usuario y, por tanto, a qué usuarios promocionar una película en concreto según las características de esta.

La industria del cine genera mucho dinero, y parte del éxito de las mejores películas y series es indudablemente el marketing. Las productoras, distribuidoras y plataformas de *video on demand* serían las primeras interesadas en disponer de un espacio de promoción dentro de Popco.

4.2 OBJETIVOS

A medida que el proyecto avanza y se consolida, se plantean una serie de objetivos a futuro que permitirán llevar la aplicación a un nivel comercial y masivo. Estos objetivos son fundamentales para transformar la versión beta actual en una plataforma ampliamente utilizada y reconocida en el mercado.

4.2.1 COMERCIALIZACIÓN DIGITAL

Uno de los objetivos clave es la comercialización digital de la aplicación. Esto implica preparar la plataforma para su lanzamiento oficial al mercado, asegurando que todos los aspectos técnicos, legales y de usuario estén completamente optimizados. La comercialización digital abarca desde la creación de perfiles profesionales en las principales tiendas de aplicaciones hasta la elaboración de material promocional y la implementación de estrategias de marketing, que se verán en apartados siguientes.

4.2.2 LANZAMIENTO EN LA APP STORE Y PLAY STORE

Un paso crucial en la comercialización es sacar la aplicación a la App Store y Play Store. Este proceso requiere cumplir con todas las directrices y requisitos establecidos por Apple y Google, respectivamente. La preparación para este lanzamiento incluye la realización de pruebas exhaustivas para garantizar que la aplicación funcione perfectamente en diferentes dispositivos y sistemas operativos, la creación de descripciones atractivas y precisas, la generación de capturas de pantalla y videos demostrativos, y la obtención de todas las certificaciones y aprobaciones necesarias.

4.2.3 ESTRATEGIAS DE MARKETING

El éxito comercial de la aplicación depende en gran medida de una estrategia de marketing efectiva. Se planea a futuro desarrollar campañas de marketing digital que incluyan publicidad en redes sociales, colaboraciones con *influencers* del cine, y la creación de contenido atractivo para blogs y canales de video. Estas estrategias están diseñadas para

atraer y retener usuarios, incrementando la visibilidad y el reconocimiento de la aplicación. Además, se considerarán tácticas de marketing y eventos especiales para captar la atención del público objetivo de manera creativa y efectiva.

4.2.4 MONETIZACIÓN

Finalmente, para asegurar la viabilidad económica a largo plazo, es fundamental implementar un modelo de monetización sostenible. Esto puede incluir opciones como suscripciones premium que ofrezcan beneficios adicionales o anuncios integrados dentro de la aplicación. La monetización no solo ayudará a cubrir los costos operativos y de desarrollo continuos, sino que también permitirá reinvertir en mejoras y expansiones futuras de la aplicación. Evaluar y adaptar continuamente las estrategias de monetización garantizará que sean atractivas tanto para los usuarios como para los inversores.

Estos objetivos futuros son ambiciosos pero alcanzables con la dedicación y constancia que han caracterizado el desarrollo del proyecto hasta ahora. La combinación de una comercialización digital efectiva, un lanzamiento estratégico en las principales tiendas de aplicaciones, robustas estrategias de marketing y un modelo de monetización sostenible permitirá transformar la versión beta actual en una aplicación exitosa y rentable. Con el esfuerzo continuo y una planificación cuidadosa, es muy viable alcanzar estos objetivos y asegurar el éxito a largo plazo de la plataforma.

4.3 METODOLOGÍA

Para alcanzar los objetivos del proyecto, se sigue una metodología rigurosa y estructurada. El primer paso es conceptualizar todas las ideas. En esta fase, se identifican y definen las funcionalidades clave que la aplicación debe ofrecer. Este proceso de conceptualización no solo es mental, sino que pasa por la elaboración de bocetos y diagramas que exponen sobre todo de forma visual cómo se vería y funcionaría cada componente de la aplicación. La claridad en esta etapa es crucial para las subsiguientes etapas.

Un caso práctico para ejemplificar las etapas de la metodología es la implementación de la API en la pantalla MovieScreen.

El propósito de esta funcionalidad es tener una pantalla que reúna todos los datos relevantes sobre una película en concreto. La API TheMovieDataBase provee los datos de las películas. Para poblar esta pantalla de datos será necesario conectar el componente con la API de TheMovieDataBase. El esquema mental primitivo es precisamente eso, una conexión entre la aplicación y la API que devuelve la información. El esquema visual que se diseñó a modo de boceto representa las partes de la pantalla que requieren de información, cómo se dispondrá la información al usuario y una indicación de la conexión con TheMovieDataBase.

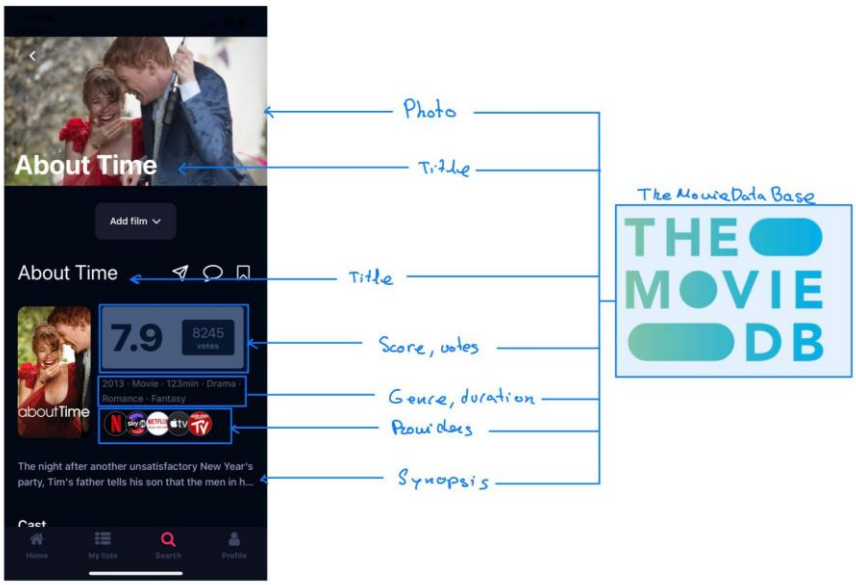


Ilustración 4. Esquema de la conexión con TheMovieDataBase

Una vez conceptualizada la idea, se procede a evaluar su plausibilidad según las tecnologías existentes en el mercado. Esto implica una investigación detallada para identificar las herramientas y plataformas más adecuadas para implementar las funcionalidades deseadas.

Es precisamente la razón por la que cobra vital importancia la primera fase, se ha de tener una visión muy clara de qué funcionalidad se plantea desarrollar para saber qué tecnologías buscar o por lo menos las palabras clave en dicha búsqueda. Normalmente, para verificar que realmente se puede llevar a cabo la funcionalidad con una tecnología en concreto, se buscan precedentes donde desarrolladores hayan diseñado funcionalidades parecidas a la que se planea desarrollar.

La viabilidad de la pantalla MovieScreen pasa por encontrar una base de datos remota que disponga del contenido cinematográfico que se requirió en la etapa de conceptualización. La conexión en sí no entraña demasiada dificultad ya que se trata de una conexión a una API, en principio una práctica básica para cualquier desarrollador. Por tanto, no habrá que preocuparse por la conexión en sí, sino por encontrar una base de datos que cumpla con los requerimientos.

Después de un proceso de investigación, contraste entre alternativas, y escuchar muchos testimonios se llegó a la conclusión de que TheMovieDataBase sería un buen candidato para dicha tarea. Este estudio consiste en analizar cada *endpoint* que ofrece la API y por supuesto si es de acceso público.

Con las tecnologías seleccionadas, el siguiente paso es aprender a utilizarlas en profundidad. En lugar de limitarse a aprender únicamente lo necesario para implementar la funcionalidad en sí, se adopta un enfoque más exhaustivo. Esto implica un estudio profundo de la documentación oficial, la realización de tutoriales o incluso cursos, además de la experimentación práctica para comprender plenamente el potencial y las limitaciones de cada tecnología. Este enfoque no solo garantiza una implementación más robusta y eficiente, sino que también permite anticipar y resolver problemas potenciales antes de que surjan durante el desarrollo. Asimismo, el aprendizaje en profundidad ha resultado ser muy rentable a largo plazo puesto que normalmente surgen nuevas ideas que también requieren del uso de las tecnologías perfeccionadas. El aprendizaje con el propósito de entender las entrañas de la tecnología puede aparentar un esfuerzo innecesario, no obstante, para poder escalar la

aplicación, mejorarla y ser solvente frente a imprevistos y errores sobrevenidos ha resultado ser vital conocer las tecnologías con los que se trabaja. Aprender en profundidad proporciona una base sólida que es esencial para la calidad y el éxito del proyecto.

Aprender a utilizar una API consiste en la familiarización con todos sus recursos. En esta etapa es clave practicar de antemano en documentos de prueba. Habituarse a los formatos de las respuestas y saber dónde encontrar rápidamente los recursos en la documentación.

Después de adquirir una comprensión sólida de las tecnologías, se procede a diseñar conceptualmente la implementación de cada funcionalidad. Cada característica se planifica detalladamente, asegurando que se integre perfectamente con el resto del sistema y cumpla con los requisitos establecidos en la fase de conceptualización. Este diseño incluye la creación de esquemas de base de datos, flujos de usuario y prototipos de interfaz. La atención al detalle en esta etapa asegura que la implementación técnica sea coherente con la visión general del proyecto y que todas las piezas funcionen en armonía. Esta conceptualización es totalmente diferente a la del primer paso puesto que se conoce con qué tecnología se va a hacer y qué posibilidades tiene la misma. Normalmente esta conceptualización es ligeramente más técnica y enfocada a la tecnología en cuestión. Ya no se piensa únicamente en términos abstractos sino en cómo materializar la funcionalidad con la tecnología seleccionada.

En esta etapa la conceptualización ya parte de un conocimiento técnico y tiene un propósito más concluyente, diseñar las partes tal y como van a ser. En esta etapa se diseña un esquema parecido al de la primera conceptualización, la única diferencia reside en el conocimiento sobre cuáles son los *endpoints* específicos de cada pieza de información.

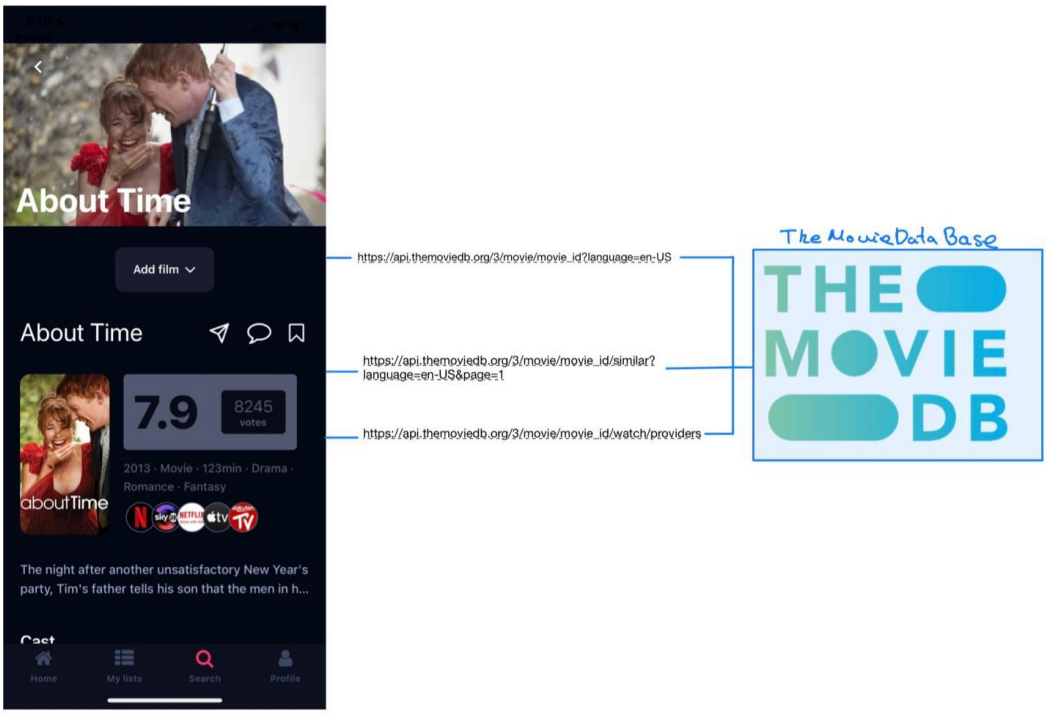


Ilustración 5. Esquema conceptual con endpoints

Finalmente, con el diseño conceptual en mano, se lleva a cabo la implementación del proyecto. Cada funcionalidad se desarrolla y luego se prueba exhaustivamente, utilizándola como si se fuese un usuario final. Idealmente el *feedback* debería provenir de un usuario real no sesgado, pero en su defecto las pruebo personalmente e intento ser profundamente imparcial juzgándola como cinéfilo. En esta etapa se codifica la implementación de la API en la pantalla MovieScreen, se utilizaron inicialmente useEffects para ejecutar las peticiones pertinentes a la base de datos, que a su vez utilizaron *async-await* y *fetch*.

Este enfoque permite identificar áreas de mejora y realizar iteraciones continuamente. Al probar y refinar cada componente en un entorno realista, se garantiza que no solo funcione correctamente, sino que también ofrezca una experiencia de usuario óptima. Este ciclo de prueba y mejora es clave para perfeccionar la aplicación y asegurar que cumpla con las expectativas y necesidades de los cinéfilos. A través de esta metodología iterativa y centrada en el usuario, se logra un producto final que es tanto funcional como agradable de usar. Las

iteraciones en el ejemplo de la implementación de la API supusieron la inclusión de nuevos datos que el usuario podría disfrutar y que inicialmente no se concibieron como tal. Además, el uso de React Query no se dio desde un principio, por tanto, se tuvo que rediseñar la lógica de todo el componente para poder gozar de las ventajas como el *caching* que provee React Query.

4.4 PLANIFICACIÓN Y ESTIMACIÓN ECONÓMICA

4.4.1 PLANIFICACIÓN

Aunque en el sentido técnico el proyecto se divide en dos partes muy diferenciadas *frontend* y *backend* en la práctica se implementarán conjuntamente ya que el funcionamiento de cualquier componente es consustancial al correcto funcionamiento de ambos a la vez, por tanto, la presente sección de planificación se dividirá por bloques generales que engloban muchas funcionalidades.

Antes de empezar conviene destacar que todas las funcionalidades del proyecto son interdependientes y por tanto no cabe el desarrollo de una funcionalidad o parte de la aplicación sin la parcial existencia de otros. Sin embargo, sí que habido una planificación temporal en la que en cada etapa o periodo se ha desarrollado de forma relativamente independiente al resto de bloques de la aplicación.

La división fundamental ha sido las tres páginas principales de navegación: Perfil, Página principal y Comentarios populares.

De octubre a diciembre de 2023

Inicialmente se desarrolló y diseñó la página de perfil. La página de perfil está estrechamente relacionada con la información del usuario, almacenada en la base de datos. Después de esbozar un boceto en código del *frontend* de la página, seguidamente se empezó la configuración de Firebase, Firestore y la autenticación.

Posteriormente poco a poco se fue mejorando la calidad de experiencia de usuario, la interfaz visual y paralelamente el modelo y estructura de los datos relacionados sobre todo con la colección de usuario.

De enero a marzo de 2024

En este periodo se implementó la página principal, esta incluye el buscador de películas la página informativa de una película, la página de actor, el historial, la página de lista entre otras. En este lapso de tiempo también se diseñaron todas las conexiones entre la página de perfil y la página principal y subsecuentes. Cada usuario puede interactuar con las películas de múltiples modos. Puede guardar una película en una lista, puede indicar si ha visto o si planea ver la película. Estas interacciones tienen influencia en ambas partes a la vez. Mientras que la película se puede guardar en una lista desde la página principal, las listas solo son accesibles desde el perfil.

Adicionalmente en este periodo se incluye la implementación de la API a la lógica de la aplicación, puesto que el diseño de la página de película bebe en términos de contenido de la base de datos de TheMovieDataBase.

De marzo a mayo de 2024

Finalmente, en los últimos compases del proyecto es decir desde marzo hasta mayo se ha desarrollado la funcionalidad de los comentarios. Los comentarios son transversales en tanto en cuanto se puede acceder a alguna sección con comentarios desde cualquier parte de la aplicación. Esto implica que la construcción de esta sección ha estado estrechamente ligada a las dos anteriores. Este periodo ha sido fundamental para encajar todas las piezas y solidificar la aplicación entera.

4.4.2 ESTIMACIÓN ECONÓMICA

El coste del proyecto ha sido prácticamente mínimo en términos directos. Sin embargo, un análisis estricto debería incluir la amortización de todas las herramientas utilizadas a lo largo

del desarrollo del proyecto. Esto abarca el ordenador y el suministro de internet, que son esenciales para cualquier trabajo de desarrollo de software. Además, se debe considerar la licencia de desarrollador de Apple Connect, que asciende a 100 euros anuales. Estos costes representan una inversión inicial mínima, pero son cruciales para el desarrollo y mantenimiento de la aplicación.

Potencialmente, en un futuro cercano, si la aplicación ganase popularidad, los costes podrían aumentar debido a la factura de Firebase, el BaaS elegido para el proyecto. Los costes de los BaaS siguen una curva exponencial, diseñados para atraer proyectos en desarrollo y cobrar cuando dichos proyectos empiezan a funcionar a mayor escala. Esto significa que, aunque los costes iniciales pueden ser bajos o incluso gratuitos, a medida que la base de usuarios crezca y el uso de recursos aumente, los gastos se incrementarán significativamente. Es importante prever estos costes para evitar sorpresas financieras en el futuro.

En un escenario hipotético donde se continúe desarrollando y expandiendo la aplicación, habría que hacer frente a gastos adicionales en publicidad para dar a conocer y popularizar la aplicación. La inversión en marketing digital, campañas en redes sociales y colaboraciones con influencers del cine podría ser necesaria para atraer una base de usuarios más amplia. Estos esfuerzos de promoción son esenciales para aumentar la visibilidad y atraer nuevos usuarios, lo que a su vez incrementará el valor de la aplicación.

Además de estos costes, otros gastos realistas que podrían surgir incluyen el mantenimiento y actualización de la aplicación, así como posibles mejoras de infraestructura. A medida que la aplicación crezca, podría ser necesario personal adicional, como desarrolladores y especialistas en marketing, para gestionar el crecimiento y mejorar las funcionalidades. También se podría considerar la inversión en análisis de datos y herramientas de optimización para mejorar la experiencia del usuario y retener a los usuarios actuales.

El valor de la aplicación depende sustancialmente de la base de usuarios. Aunque la aplicación cuente con funcionalidades intrínsecamente valiosas, parte de la propuesta de valor nace bajo el supuesto de que exista una comunidad activa y comprometida. Sin una

base de usuarios sólida, las funcionalidades sociales y de interacción pierden su atractivo. Por lo tanto, es crucial no solo desarrollar una aplicación técnicamente sólida, sino también invertir en estrategias que fomenten la creación y el crecimiento de una comunidad activa alrededor de la aplicación.

En un análisis financiero algo más estricto teniendo en cuenta que un desarrollador de software de media en España por hora cobra 17,95 € y presuponiendo que se ha trabajado todos los días un total de 3 horas desde la fecha en la que oficialicé "Popco, una aplicación para cinéfilos" como objeto del TFG, es decir, el 1 de octubre, amontonaría un total de 14539,5 euros de salario.

Al coste del salarial habría que agregar el coste de la amortización de la herramienta preponderante en el desarrollo de este proyecto, el ordenador. El modelo que se ha usado tiene un precio de mercado de aproximadamente 2000 € y una vida útil de cinco años el uso de dicho ordenador ascendería a 333,33€.

Los costes de Firebase es decir los costes del *backend* son variables según el uso de la base de datos, además cada tipo de petición tiene un coste diferente. Firebase para incentivar el uso de Firestore, una de sus herramientas, asigna un umbral mínimo de peticiones por debajo del cual los propietarios del proyecto no pagarán nada. Al ser un proyecto en desarrollo y a pesar de haber hecho peticiones no se ha llegado en ningún mes al umbral mínimo por lo tanto los costes del *backend* son inexistentes.

La licencia de desarrollador de Apple Connect se paga anualmente y asciende a los 100 € en España. La licencia se compró en marzo de 2024 y por tanto solo se ha amortizado tres meses de la licencia, lo que equivale a 25 euros.

El total de los costes del proyecto en una estimación relativamente realista atendiendo a los precios de mercado asciende a un total de 14897,83.

Capítulo 5. MANUAL DE USO

En el presente capítulo se explicarán paso por paso todas las funcionalidades de la aplicación, sin apreciaciones ni explicaciones técnicas. Será ya en el próximo capítulo donde se profundizará en los fundamentos técnicos de la lógica, el *frontend* y el *backend* involucrados en el proyecto. Por tanto, este apartado se limitará a una mera explicación dirigida a un usuario no técnico, acompañada de imágenes.

5.1 AUTENTICACIÓN

La primera pantalla que aparecerá nada más abrir la aplicación, por primera vez, es la pantalla de registro. La apostilla por primera vez es sustancialmente importante, una vez el usuario se registra se guardará localmente de forma automática un token. Este token representa de forma exclusiva al usuario, de modo que en subsiguientes usos dicho token se mandará, nuevamente de forma automática, al *backend* que verificará su validez, de manera que el usuario no tendrá que volver a introducir sus credenciales.

La pantalla de inicio de sesión se usará muy infrecuentemente, la razón es que el usuario será redirigido únicamente a esta pantalla si previamente ha cerrado sesión.

El funcionamiento de ambas es simple, rellenar el formulario y clicar en el botón.

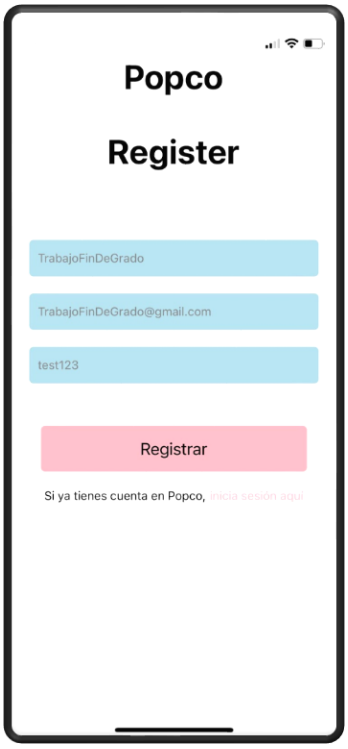


Ilustración 6. Pantalla de registro

Ilustración 7. Pantalla de inicio de sesión

5.2 COMENTARIOS POPULARES

La primera pantalla que sale por defecto al abrir la aplicación con el inicio de sesión ya completado es la página de comentarios populares. Únicamente contiene el título de la página seguido de una lista de comentarios. Esta página se llama "*Popular this week*", que literalmente se traduce a "lo más popular de la semana".

Los comentarios son reseñas que los usuarios emiten acerca de películas en concreto, más adelante se explicará el cómo. Un comentario está compuesto del título de la película, portada de la película, nombre de usuario emisor del comentario, su foto de perfil, cuanto tiempo ha transcurrido desde la emisión y por supuesto el contenido del comentario en la parte central del contenedor.

Los comentarios pueden recibir likes, un like es una muestra de apreciación que cualquier usuario puede emitir a través de un click sobre el corazón colocado en la parte inferior izquierda de cada comentario.

La popularidad se mide precisamente en número de likes, siendo los comentarios más populares aquellos que más likes reciben.

La lista de "*Popular this week"* está ordenada por número de likes, además solo aparecen comentarios emitidos durante los últimos 7 días, convirtiéndose así en una lista actualizada de comentarios democráticamente recomendados. La lista crece a medida que el usuario desliza verticalmente hacia abajo, de forma que nunca carece de contenido.

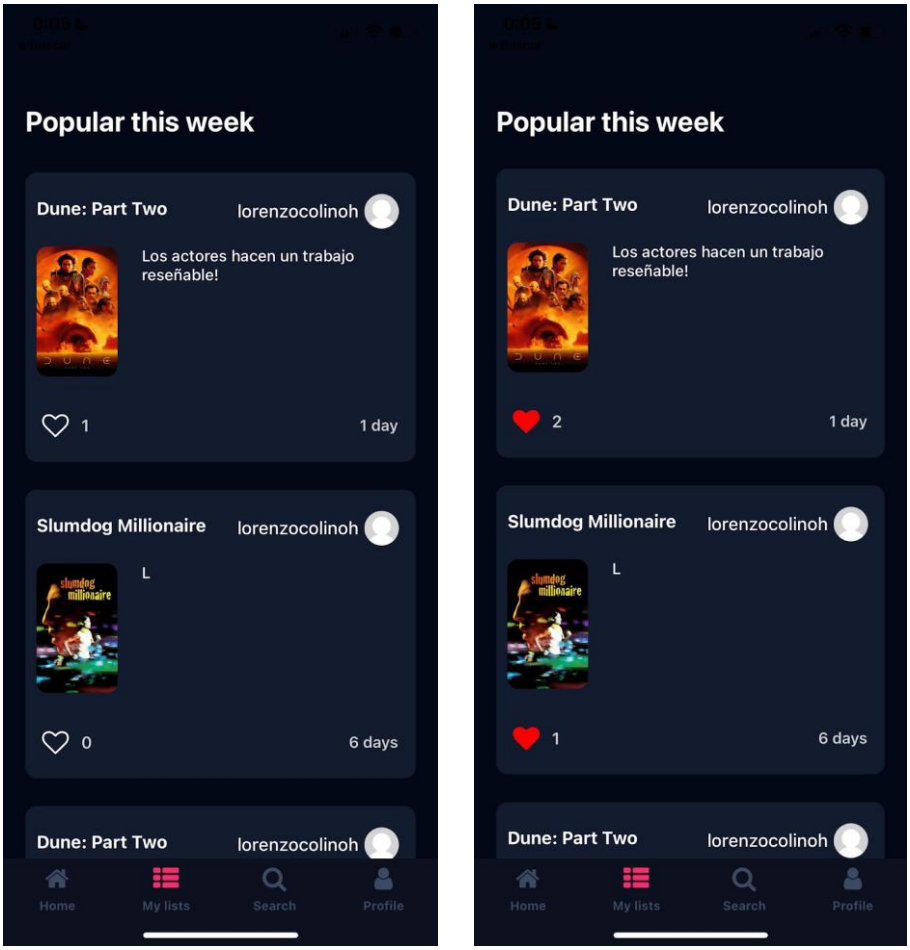


Ilustración 8. Popular this week

En la diapositiva dispuesta a la derecha, se puede apreciar que el usuario ha dado dos likes a dos comentarios.

5.3 HOMEPAGE

La página de *Home* dispone de un buscador en la zona superior de la pantalla. Debajo del buscador hay una zona deslizable (verticalmente) con listas. Dichas listas son las películas más populares disponibles en el catálogo de ciertos servicios de video on demand. Las películas disponibles en, por ejemplo, Amazon Prime España, pueden diferir de las disponibles en Amazon Prime Alemania. Esto sucede así porque los derechos de las películas se compran por regiones geográficas.

Aunque no hay de qué preocuparse, ya que Popco automáticamente detecta la ubicación del usuario.

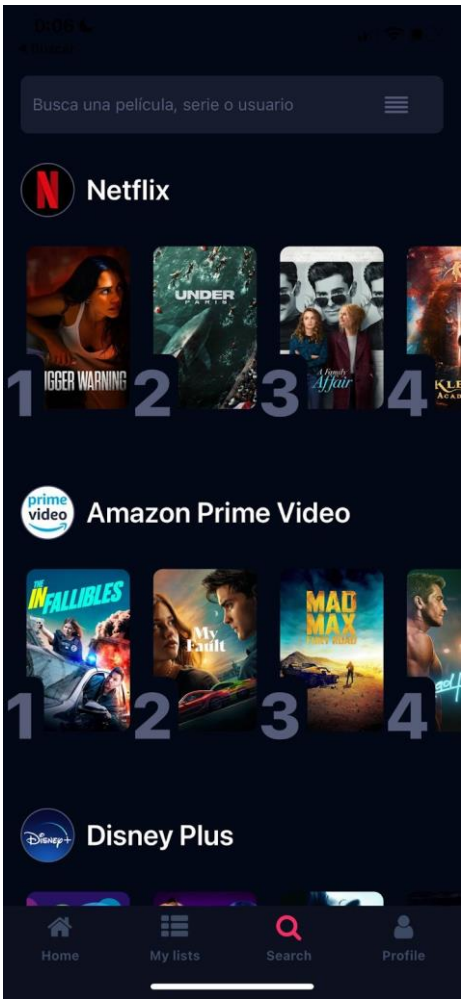


Ilustración 9. Homepage

Hay que tener en cuenta la variedad de suscripciones de los usuarios, dar por hecho que la totalidad de los usuarios de Popco son suscriptores de Netflix, sería injusto para los que no lo fuesen. Los usuarios podrán configurar en la página de ajuste, de qué proveedores de VOD tienen suscripción, para que solo aparezcan las listas de películas populares de los mismos.

5.4 BUSCADOR

Una vez se clica en el buscador, automáticamente se abre una pestaña aparentemente diferente. Aunque pueda parecer una radicalmente distinta, en términos programáticos no lo es. Es un componente que se monta superponiéndose al anterior.

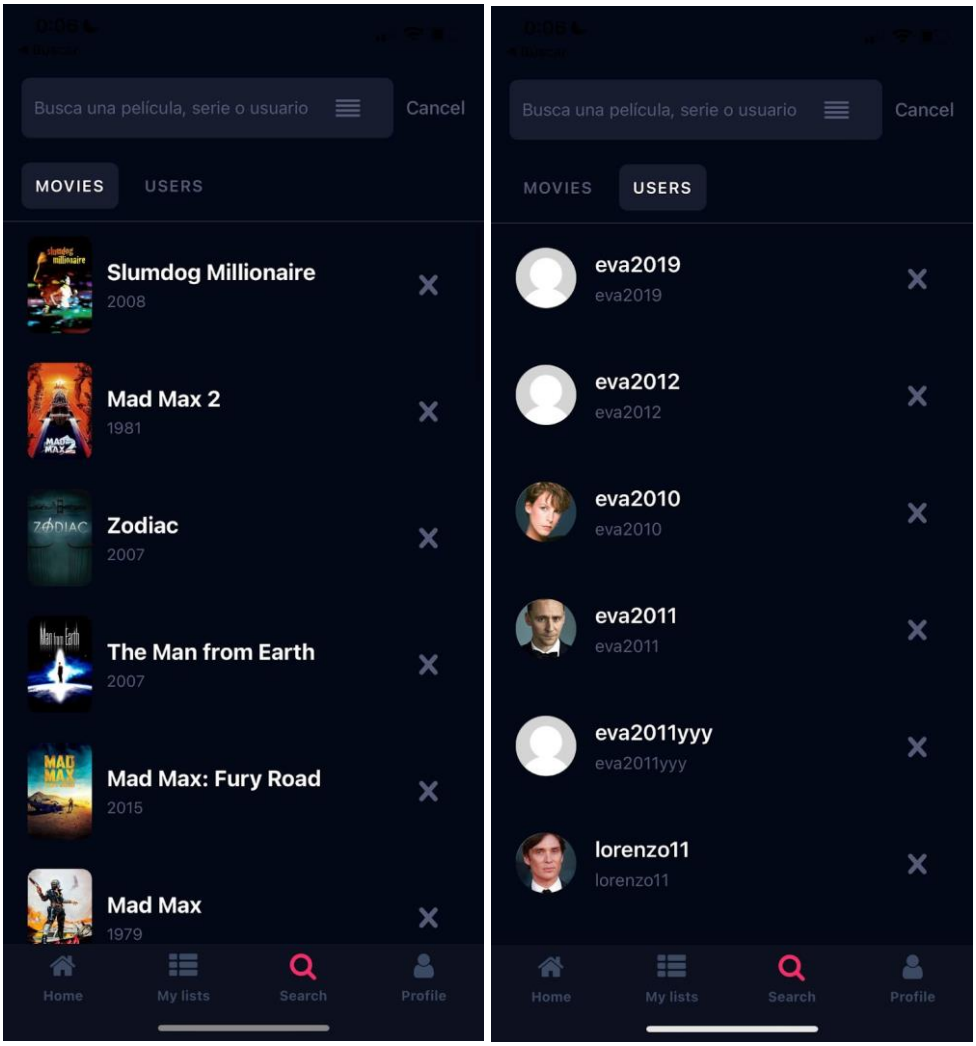


Ilustración 10. Buscador de películas, serie o usuario

Para retroceder bastará con clicar el botón cancelar dispuesto en la esquina superior derecha.

Debajo del buscador hay dos opciones seleccionables, dependiendo del tipo de búsqueda que desee el usuario. Al tratarse de una red social, el buscador tiene incorporada la opción de buscar opciones.

Si no se escribe ningún carácter en el buscador, por defecto se renderiza un historial de búsqueda. Es una lista cuyas entradas corresponden a las búsquedas más recientes. En el lateral derecho de cada una de estas entradas hay una cruz, que al ser clicada borrará del historial la entrada en cuestión.

La sección de búsqueda de usuarios también cuenta con su propio historial de búsquedas, este replica el funcionamiento del previamente explicado.

En caso de escribir algún carácter automáticamente se empezarán a buscar coincidencias de títulos con los caracteres introducidos, o en su defecto usuarios cuyo nombre de usuario coincida con los caracteres en cuestión.

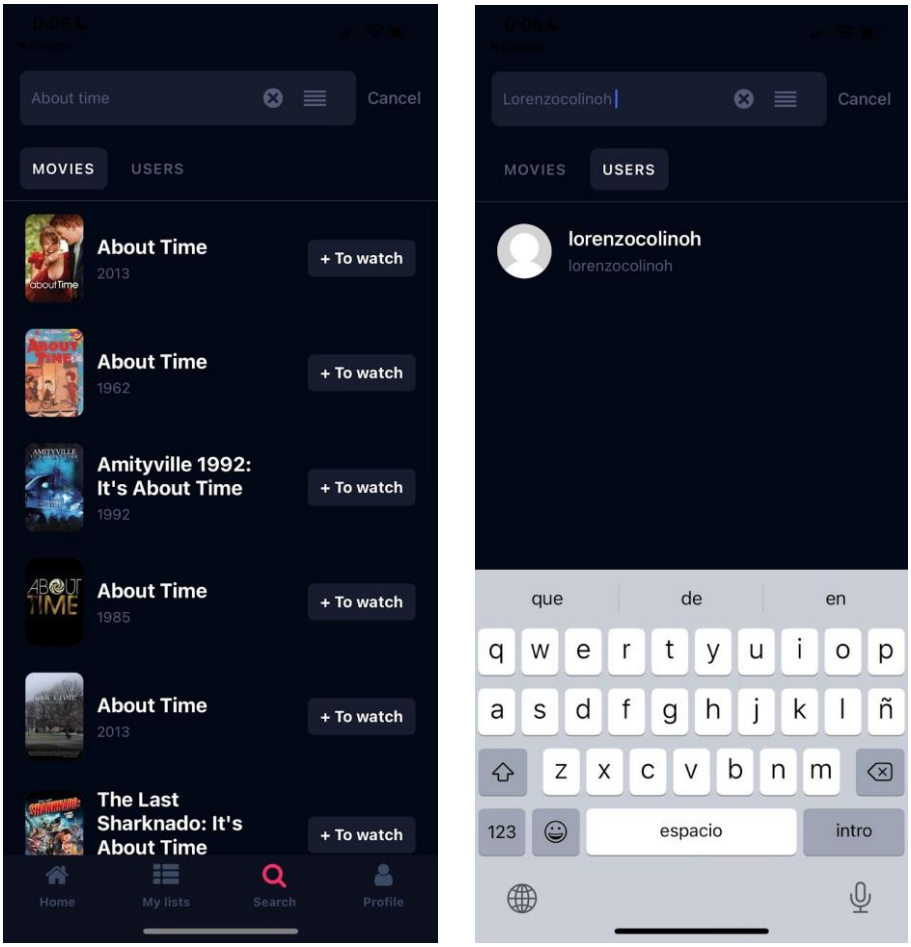


Ilustración 11. Ejemplo de una búsqueda de película y usuario

En la sección de películas se incluye la opción de incluir filtros a la búsqueda.

Muchas ocasiones el usuario no tiene en mente un título de película en concreto y prefiere buscar en base a ciertos filtros que circunscriban el tipo de película que quiere ver.

Dentro del propio contenedor, en el extremo derecho hay un botón que abre el panel de filtros, desde el cual se podrá elegir opcionalmente el género de la película, el tipo de contenido e incluso ordenar los resultados.

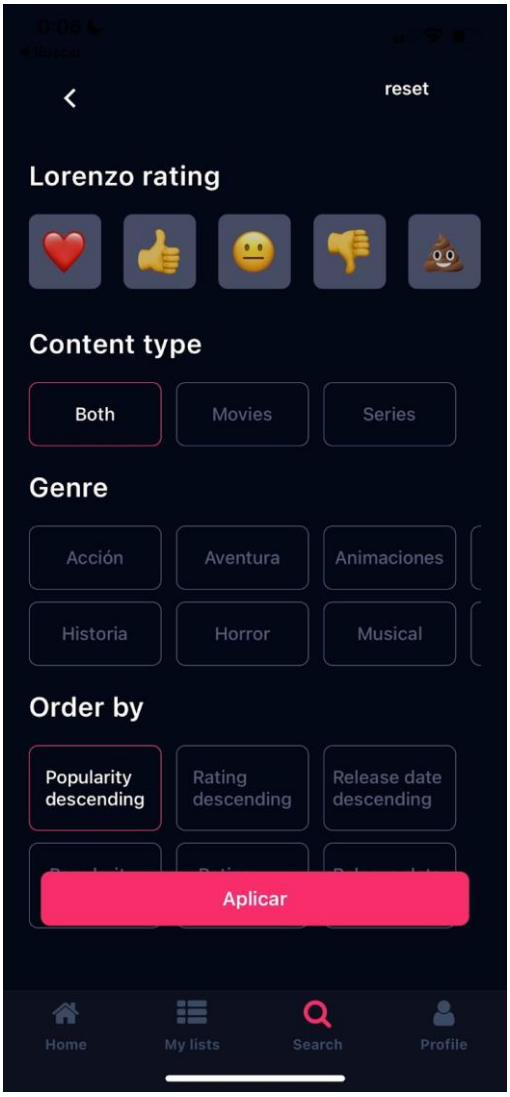


Ilustración 12. Filtros para elegir la película

5.5 PÁGINA DE PELÍCULA

En esta página se puede ver toda la información referente a una película en concreto. Toda la información que un cinéfilo pueda necesitar acerca de una película estará disponible.

En la parte superior, además del botón para retroceder a la pantalla anterior, se dispone de una imagen, superpuesta con el título de la película.

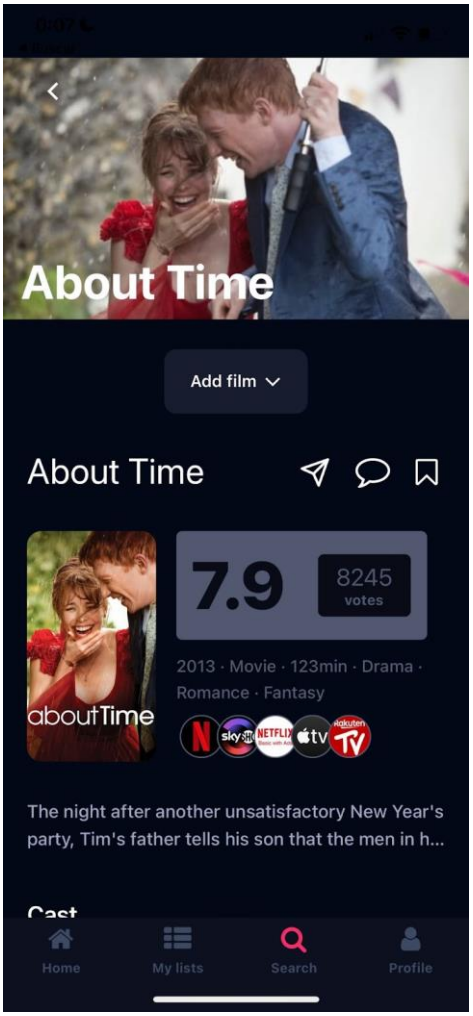


Ilustración 13. Página de la película

Debajo de la imagen hay una sección horizontal dedicada a un botón a través del cual el usuario podrá cambiar el estado de la película.

Solo habrá tres opciones que se desplegarán nada más el usuario pulse el botón.

- *Watched*: es la opción para indicar que se ha visto la película. Si se pulsa simultáneamente cambiará el estado con una animación y en segundo plano se agregará la película a la lista de películas vistas por el usuario. Esto tendrá especial

relevancia en la página de perfil de usuario, donde se puede ver qué películas ha visualizado el usuario

- *To watch*: es la opción para indicar que, a pesar de no haber visto la película, al usuario le gustaría verla. Nuevamente, en segundo plano, la aplicación agrega la película a la lista de películas que el usuario querría ver. Esta información es útil, en tanto en cuanto, permite a los usuarios de un vistazo saber qué películas interesan a sus amigos.
- *Add film*: es el estado por defecto del botón, indica que el usuario no ha interactuado todavía con la película o bien que no quiere visualizarla.

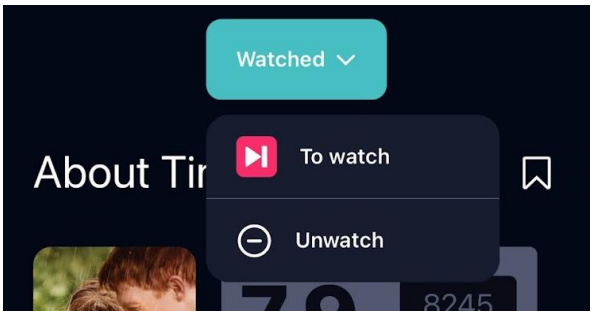


Ilustración 14. Tres estados del botón en la página de la película

La parte central es una sección dedicada íntegramente a la metainformación de la película, desde la portada, la valoración media de los usuarios, el año del estreno hasta los géneros de la película entre otros.

En la parte inferior de la sección se encuentra la sinopsis de la película. Los puntos suspensivos son una discreta e intuitiva forma de indicar al usuario que se trata de un componente desplegable, simplemente clicando sobre el componente.

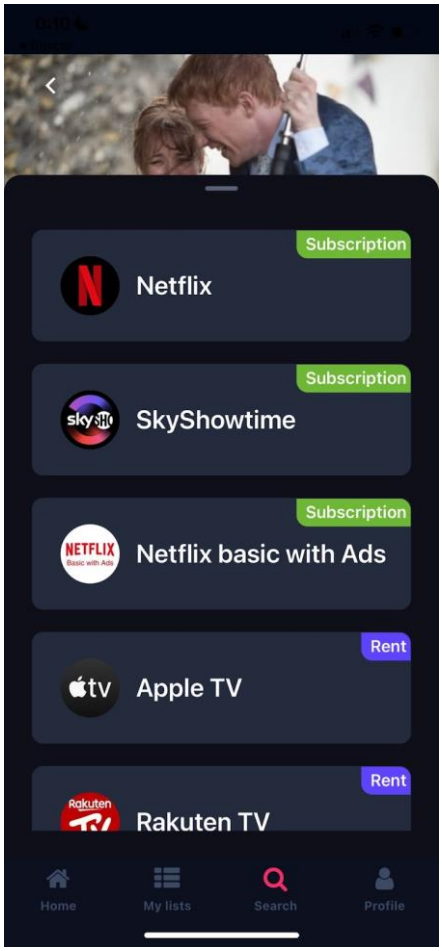


Ilustración 15. Proveedores del VOD

Debajo del género de la película hay logos de proveedores de *video on demand* (VOD). Este componente es clicable, en caso de ser pulsado emergerá un componente deslizable con información más detallada sobre las plataformas.

No solo interesa si una plataforma en concreto cuenta con la película en su catálogo, sino además cuál es la disponibilidad de la misma.

Una película puede estar disponible de forma gratuita para un suscriptor, sin embargo, hay películas que se deben alquilar o incluso comprar para poder visualizarlas. Toda la información estará disponible en dicho componente, que cuenta con una lista deslizable verticalmente.

En la parte inferior de la pantalla a la que se accede simplemente deslizando para abajo, hay tres secciones más.

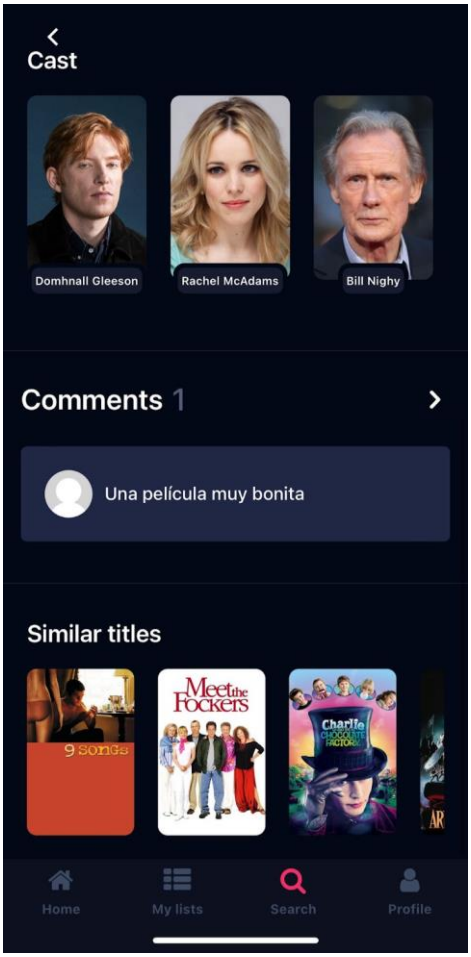


Ilustración 16. Ampliación de la página de la película

El reparto de la película es una lista horizontal con imágenes de los actores y su respectivo nombre, son componentes clicables, que navegan al usuario hasta la página del actor.

Seguidamente se encuentra la sección de comentarios más populares. Aparecerán aquellos comentarios más populares, siempre y cuando no haya ningún comentario de un amigo, en cuyo caso aparecería este.

La tercera sección es nuevamente una lista horizontal de películas con tramas parecidas a la película buscada.

Ilustración 17. Iconos de la pantalla principal

Estos tres iconos nos llevan a pantallas emergentes.

La primera pantalla consiste en un sistema de mensajería a través del cual se puede mandar la película a un amigo acompañada de un mensaje. En la parte superior hay un buscador, para poder buscar el nombre de usuario de un amigo en especial al que le quieras enviar la película. Seguidamente una lista con todos los amigos del usuario. Aparece su foto de perfil en formato circular con su respectivo nombre de usuario, y en caso de ser clicados indicaremos a la aplicación que el mensaje irá destinado a dicho usuario (se podrán elegir múltiples usuarios).

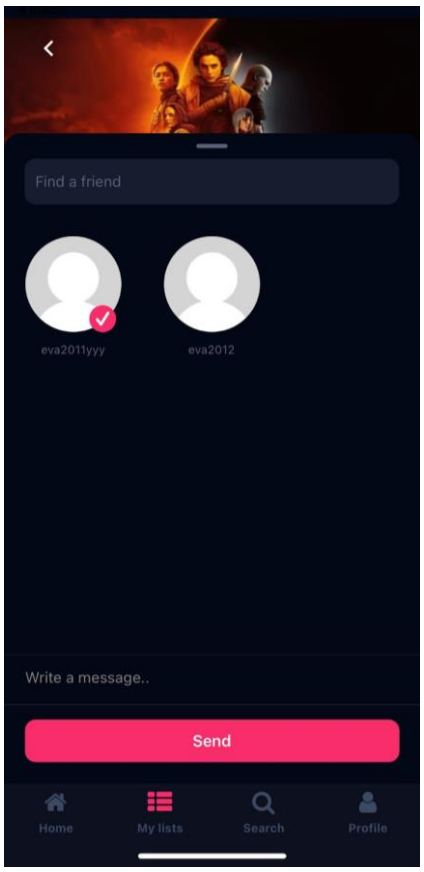


Ilustración 18. Usuarios a los que enviar películas.

Finalmente, y de forma opcional, se puede mandar un mensaje mediante el cuadro de texto inferior. El mensaje se mandará pulsando el botón de *send* e inmediatamente después se cerrará la pantalla.

El segundo botón desencadena que emerja la pantalla de comentarios, donde el usuario podrá visualizar y emitir comentarios. Los comentarios están ordenados por orden de popularidad únicamente.

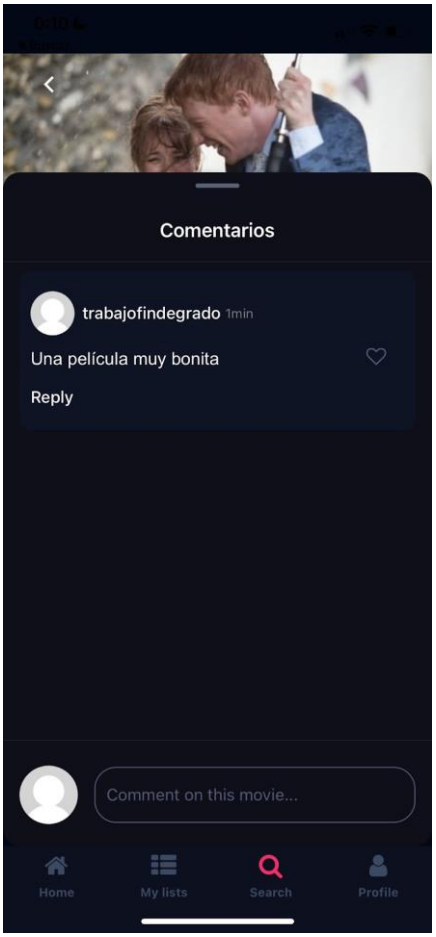


Ilustración 19. Comentarios en una película

Lógicamente toda interacción que se pudiese hacer con un comentario en la pantalla de comentarios populares se podrá replicar en esta. Desde poner un *like* hasta responder a los comentarios.

La única diferencia es que los comentarios, al estar en un espacio más escueto en términos de dimensión, son ligeramente más pequeños. Como es lógico, los comentarios que aparecerán en la lista serán comentarios únicamente relacionados con la película.

El último icono abre otro componente deslizable. En Popco los usuarios pueden crear listas donde almacenar películas, y por defecto, en cada perfil de usuario, viene incorporada una lista inamovible, "*watched films*".

En esta pantalla se renderizan todas las listas a las que tenga acceso y capacidad de modificación el usuario. En caso de clicar una lista, la película se almacenará dentro. En la siguiente visual se puede apreciar que dicha película ya está en la lista.

Ilustración 20. Pantalla del icono de guardar o añadir a lista

5.6 PÁGINA DE ACTOR

La página de actor está dedicada a la filmografía del actor en cuestión. Esta trae una pequeña descripción acompaña a una foto del actor en la parte superior de la pantalla.

La sección intermedia cuenta con botones para configurar visualmente las películas o filtrarlas.

En las siguientes visuales se pueden apreciar los distintos formatos de visualización de películas a disposición del usuario, configurables con los iconos de la sección intermedia.

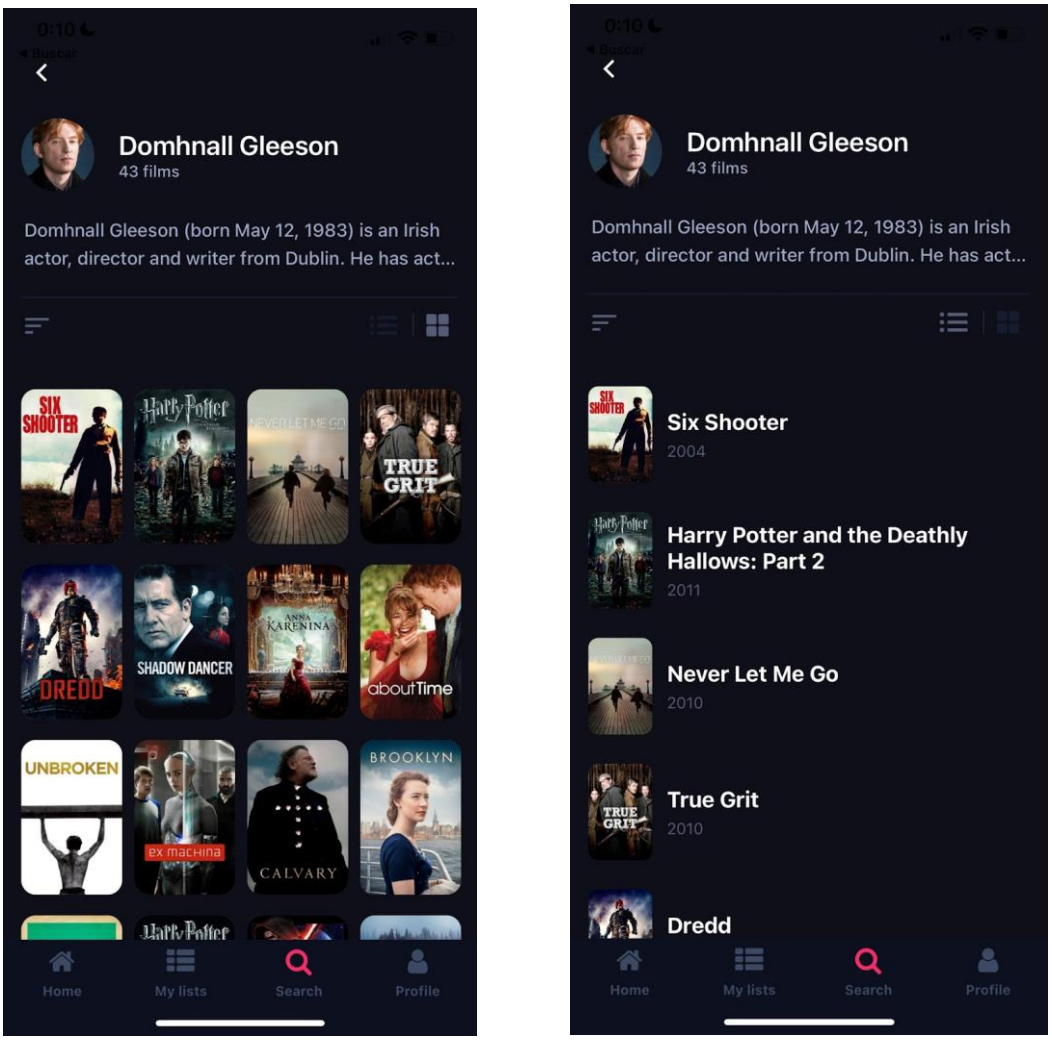


Ilustración 21. Página de perfil del actor

5.7 PERFIL DE USUARIO

La página de perfil de usuario contiene toda la información referente al usuario. Esta es una página frecuente en la práctica totalidad de las redes sociales.

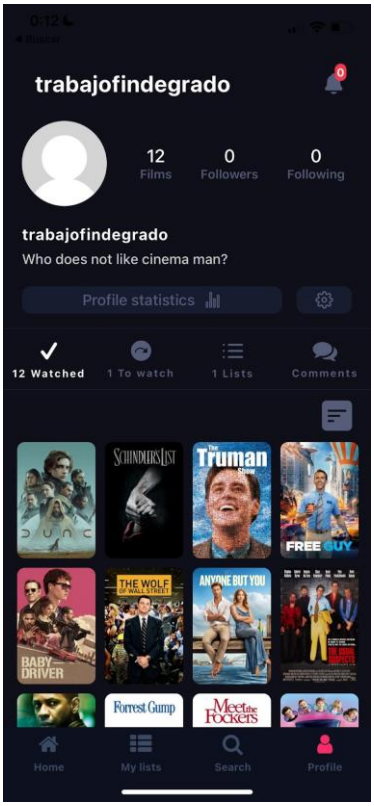


Ilustración 22. Página del perfil del usuario

En una aplicación de cinéfilos debe preponderar la información estrictamente relacionada con el cine, y es precisamente lo que representa la página de perfil de usuario, un conjunto de datos sobre las experiencias cinematográficas del usuario.

La parte superior cuenta con tres contadores, uno de películas vistas, otro de seguidores y el último de seguidos. Además, en la parte superior derecha hay un icono pulsable que navega al usuario a la bandeja de entrada de sus notificaciones.

En la sección central hay un menú con cuatro opciones. Cada opción dará lugar a información distinta: las películas que ha visto el usuario, las que desea ver, las listas públicas del usuario y finalmente los comentarios del usuario.

- Películas vistas por el usuario: es un *grid* con todas las películas vistas por el usuario y ordenadas temporalmente. Las películas visualizadas recientemente aparecerán encima. [*Ilustración 22*]
- Películas que el usuario desea ver: es visualmente igual a las películas vistas, razón por la que se omitirá dicha sección.
- Listas: las listas tienen un aspecto parecido, se agrupan en forma de *grid*. El usuario podrá ordenarlas según el criterio elegido a través de los filtros, y se podrán disponer de forma horizontal si así se desea con los iconos localizados en la sección intermedia. Las listas cuentan en la zona superior con una sección informativa, con datos como los miembros de la misma. [*Ilustración 23*]
- Comentarios: esta sección es una lista con todas las reseñas emitidas por el usuario cuyo perfil se visita. Los comentarios están ordenados por defecto por orden de popularidad, sin embargo, el orden es configurable a través del botón "*Sort by*". Este desplegará una pantalla donde modificar el orden. (La foto dispuesta a la derecha) [*Ilustración 24*]

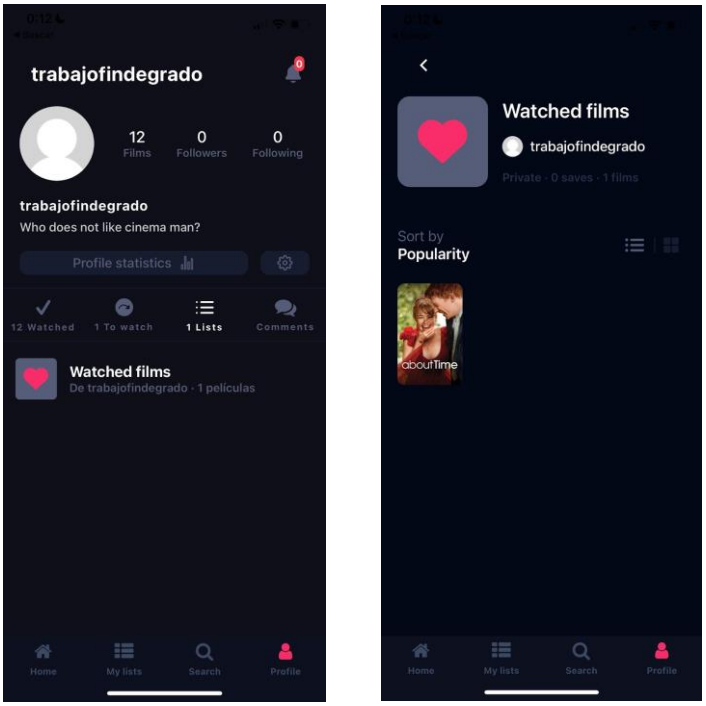


Ilustración 23. Listas en el perfil del usuario

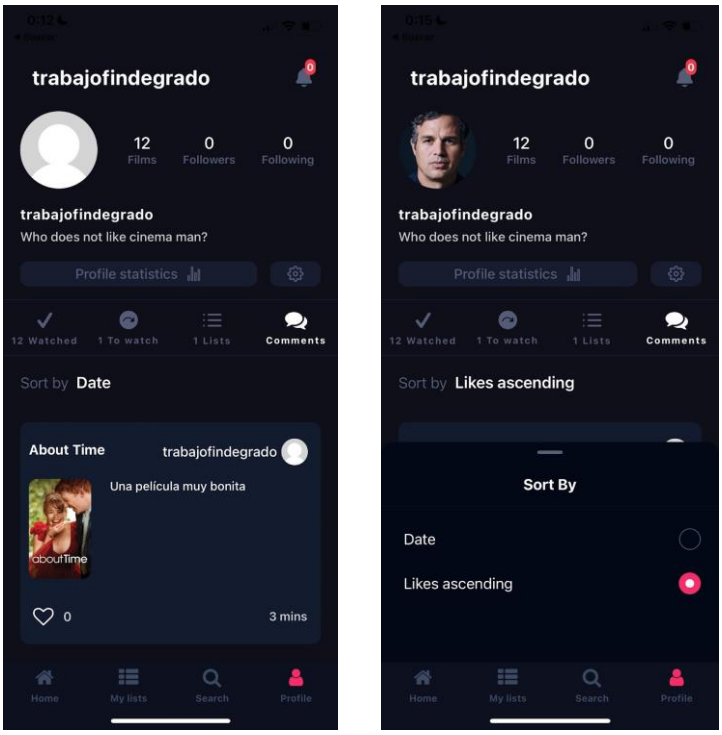


Ilustración 24. Comentarios en el perfil del usuario

5.8 CONFIGURACIÓN

En la página de configuración el usuario podrá cambiar cualquier de los siguientes atributos: nombre, email, *username*, foto de perfil, contraseña, descripción y suscripción a plataformas. La página consiste en un formulario, donde cualquier cambio requiere de una confirmación a través del botón "*Save*".

Además, se podrá cerrar sesión.

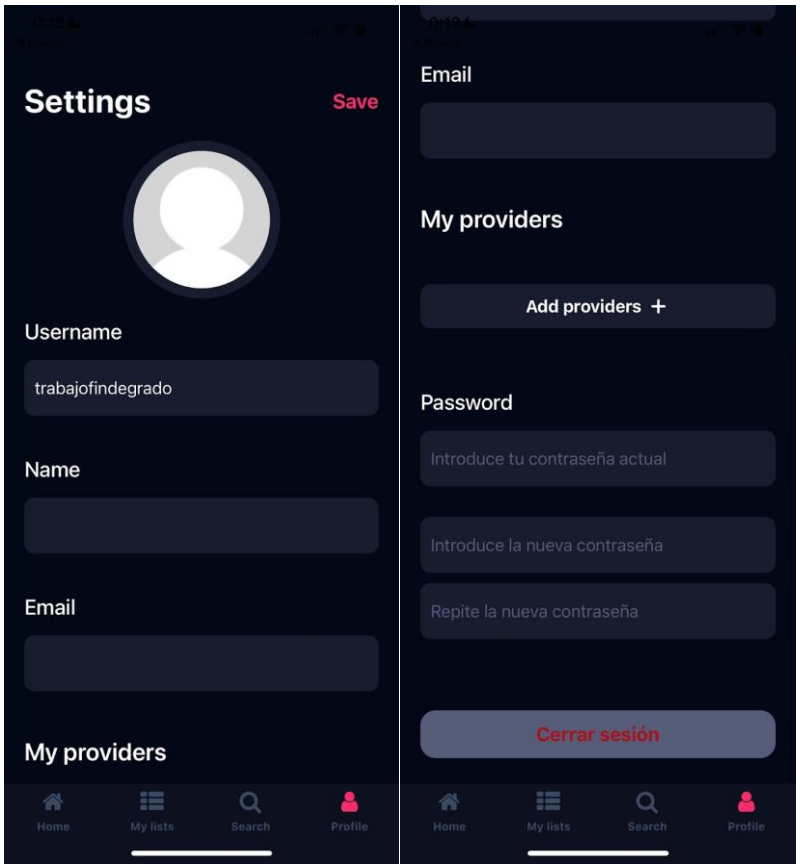


Ilustración 25. Formulario de configuración del perfil

5.9 FOTO DE PERFIL

En Popco los usuarios solo podrán elegir de foto de perfil a un actor. En el panel de cambio de foto de perfil el usuario podrá seleccionar cualquiera de los actores que aleatoriamente aparezcan en cada *refresh*. La opción de sacar una foto con la cámara del dispositivo no está descartada en futuras versiones, pero de momento no está disponible.

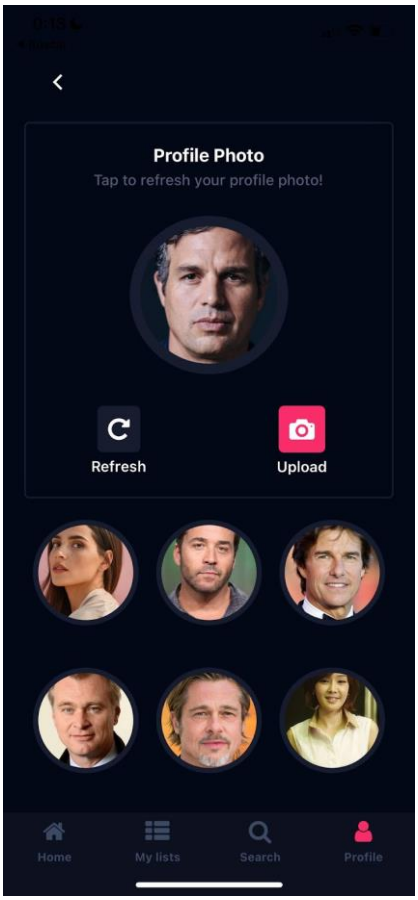


Ilustración 26. Elegir foto de perfil

5.10 PLATAFORMAS

En esta página el usuario podrá registrar a qué plataformas de *VOD* está suscrito. Esta información sirve fundamentalmente para la *Homepage*. A modo de recordatorio en la *Homepage* el usuario tiene a su disposición las listas de películas más populares actualmente de cada proveedor.

Desde esta pantalla el usuario podrá limitar las listas de películas populares en cada plataforma a únicamente las plataformas a las que esté suscrito, que es lo que realmente interesa.

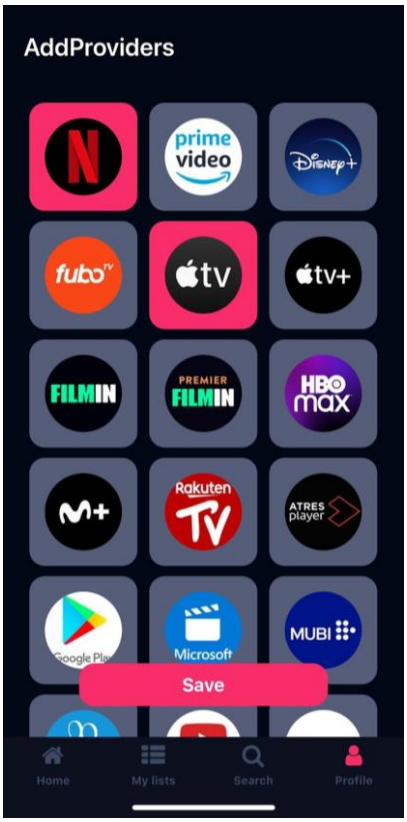


Ilustración 27. Plataformas de VOD

5.11 LISTAS DE USUARIOS

Cada usuario tendrá seguidores y seguidos. Es decir, terceros que han decidido seguirle o terceros a los que sigue respectivamente.

Desde el perfil se puede acceder a cualquiera de las dos listas, la lista de seguidores y la lista de seguidos. A continuación, se expondrán dos visuales, cada una correspondiente a un tipo de lista.

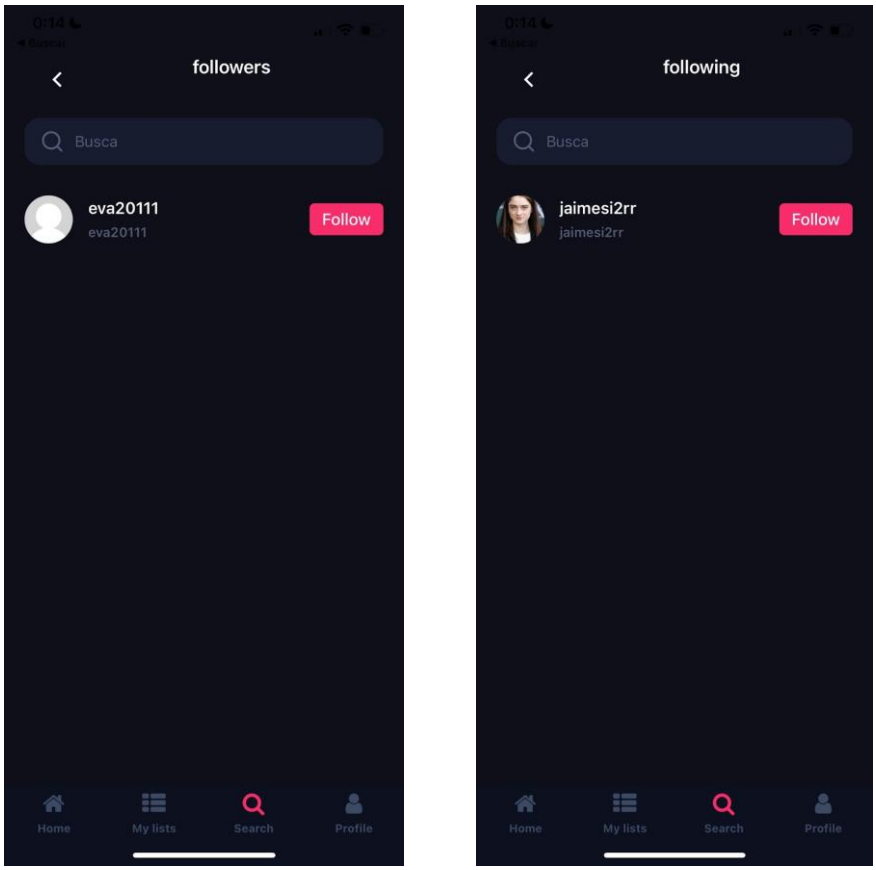


Ilustración 28. Lista de seguidores y seguidos

5.12 RESPUESTAS

Los comentarios son un componente clicable. De hecho, reaccionan de manera diferente según la frecuencia de *click*:

- Si se clica dos veces seguidas (rápidamente) se da un *like* a un comentario.
- Si únicamente se pulsa una vez, el usuario será llevado hacia la página de respuestas.

Cada comentario tiene un hilo de respuestas, las respuestas son comentarios de un cáliz diferente que versan sobre una reseña en cuestión.

A continuación, un ejemplo del funcionamiento de las respuestas. En la parte superior se encuentra el comentario objeto del hijo, posteriormente la lista de respuestas.

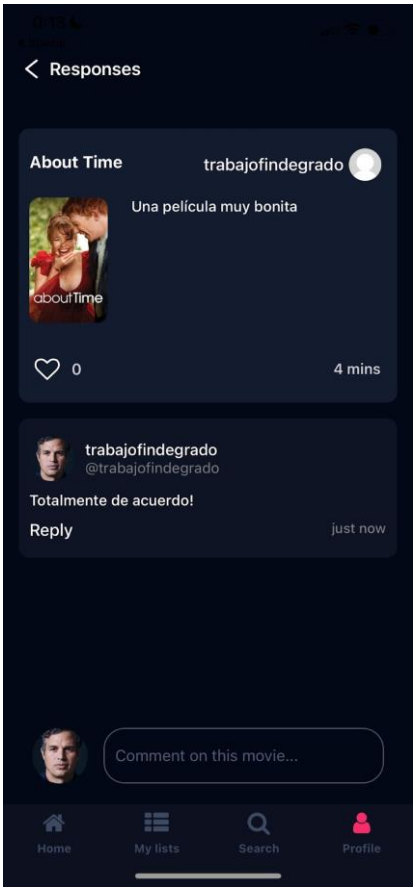


Ilustración 29. Respuestas a un comentario

Capítulo 6. APLICACIÓN DESARROLLADA

En este capítulo se describirá el proyecto realizado. Se ha creído pertinente recorrer la aplicación a través de las librerías utilizadas, las cuales corresponden a los subapartados de este capítulo.

6.1 ASYNCSTORAGE

AsyncStorage [5] es una librería de React Native que almacena asíncrona y localmente información en el dispositivo del usuario en un formato clave-valor.

La gran mayoría de la información en aplicaciones móviles se almacena en los servidores, es decir, en el *backend*. El motivo principal es la importancia de poder acceder a dicha información siempre que sea necesario. Un ejemplo para clarificar dudas contextualizado en Popco, son los comentarios. Si cada comentario que emita un usuario se almacenará localmente, dentro de su dispositivo, el acceso sería complicado, dificultando poder acceder a los comentarios que ha emitido dicho usuario.

Pero hay ciertas píldoras de información que única y exclusivamente resultan de utilidad a un usuario y, por tanto, es información que no tiene necesariamente que viajar a los servidores.

Esta práctica resulta también ventajosa en términos económicos y de experiencia de usuario. Cada petición que se hace al servidor, demandando información de vuelta implica un coste, y además es una transacción que se demora en el tiempo, es decir, no es inmediata. Perjudicando a veces gravemente la experiencia de usuario. El grado de perjuicio depende de la saturación que padezca el servidor o incluso de la calidad de la red del usuario.

El almacenamiento local tarda solamente milisegundos en consultar la información en memoria, resultando a ojos del usuario una transacción inmediata.

En la aplicación se hace uso de esta tecnología en dos ocasiones. El historial de búsqueda y el inicio de sesión automático sin necesidad de que el usuario introduzca sus credenciales cada vez que quiera acceder a la aplicación.

6.1.1 HISTORIAL DE BÚSQUEDA

En la barra de búsqueda que se encuentra en la página principal se pueden buscar usuarios o películas. El usuario introduce una entrada y se produce una búsqueda con las coincidencias.

En caso de que el usuario no haya introducido ningún carácter, se renderiza una lista con las búsquedas más recientes. Dicha lista es el historial de búsqueda.

La información de qué películas o qué usuarios ha buscado recientemente un usuario no se necesita en ninguna otra parte de la aplicación y es encima información confidencial.

Cumple por tanto los requisitos para ser clasificada como información propicia a ser almacenada en la memoria local del dispositivo. Ofreciendo así dicha información de manera inmediata sin recurrir a peticiones al servidor.

El historial de búsqueda recurre constantemente a AsyncStorage, pues se pueden añadir películas al historial e incluso borrarlas si el usuario quiere eliminar cierta película de su historial de búsqueda.

El código expuesto a continuación cuenta con toda la lógica implementada.

Cabe resaltar que el manejo de errores es una práctica recomendada durante el uso de AsyncStorage, a pesar de ser una tecnología potente es susceptible de fallos.

Los métodos *getItem* y *setItem* son de la librería AsyncStorage y sirven para recabar la información de memoria y almacenar la información en memoria respectivamente.

El formato de la información son pares clave-valor, y el tipo de contenido sufre manipulaciones por razones de eficiencia, por eso se hace uso de JSON.parse y JSON.stringify.

- SearchScreen.js (I)

```
const borrarObjetoPorId = async (historialName, id) => {
    try {
      const presentHistorial = await AsyncStorage.getItem(historialName);

      if (presentHistorial) {
        const historialParsed = JSON.parse(presentHistorial);

        // Filtrar la lista para excluir el objeto con el id proporcionado
        const nuevaLista = historialParsed.filter((objeto) => objeto.id !== id);

        // Almacenar la nueva lista en AsyncStorage
        await AsyncStorage.setItem(historialName, JSON.stringify(nuevaLista));
      } else {
        console.log("No hay datos en AsyncStorage.");
      }
    } catch (error) {
      console.error("Error al borrar el objeto desde AsyncStorage", error);
    }
  };
```

En el método borrarObjetoPorID, que recibe el tipo de lista y el id del objeto, consiste en borrar de la lista almacenada en memoria aquellos objetos que desee eliminar el usuario.

Se desencadena cuando el usuario clica sobre una cruz dispuesta en el lateral derecho de cada entrada.

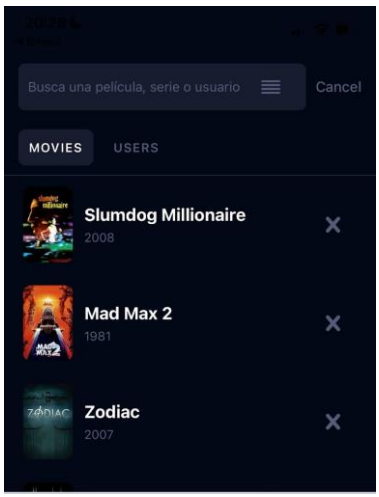


Ilustración 30. Historial de búsqueda

Se habla de objetos y no directamente de películas, ya que existen dos historiales. Uno para usuarios y otro para películas o series. Se puede apreciar en la imagen anterior ambas secciones.

- SearchScreen.js (II)

```
const getHistorial = async (historialName) => {
    try {
      const listaAlmacenada = await AsyncStorage.getItem(historialName);
      if (listaAlmacenada) {
        setHistorial(JSON.parse(listaAlmacenada));
      }
    } catch (error) {
      console.error("Error al recuperar la lista desde AsyncStorage", error);
    }
  };
```

getHistorial es un método para obtener el historial especificado en el argumento.

El historial que se recabe de memoria se guardará en una variable de estado que posteriormente se renderizará en el JSX dentro de una FlatList

- SearchScreen.js (III)

```
const saveInHistorial = async (historialName, nuevoObjeto) => {
    try {
```

```
      const listaActual = await AsyncStorage.getItem(historialName);

      if (listaActual) {
        const listaObjetos = JSON.parse(listaActual);
        let hasItemAlready = false;
        listaObjetos?.forEach((item) => {
          if (item.id === nuevoObjeto.id) {
            hasItemAlready = true;
          }
        });
        if (!hasItemAlready) {
          listaObjetos.unshift(nuevoObjeto);
          await AsyncStorage.setItem(
            historialName,
            JSON.stringify(listaObjetos)
          );
        } else {
          const nuevaLista = listaObjetos.filter(
            (item) => item.id !== nuevoObjeto.id
          );
          nuevaLista.unshift(nuevoObjeto);
          setHistorial(nuevaLista);
          await AsyncStorage.setItem(historialName, JSON.stringify(nuevaLista));
        }
      } else {
        const nuevaLista = [nuevoObjeto];
        await AsyncStorage.setItem(historialName, JSON.stringify(nuevaLista));

        console.log("Primer objeto agregado con éxito.");
      }
    } catch (error) {
      console.error("Error al agregar el objeto a AsyncStorage", error);
    }
  };
```

La función saveInHistorial sirve para guardar la entrada recién accedida en el historial.

Básicamente guarda el id del nuevo objeto en la lista del historial, no sin antes proceder a comprobar si el objeto en cuestión ya existía, en cuyo caso se deberá reordenar la lista eliminando el objeto y seguidamente colocándolo en primera posición.

Hay otros formatos de historial, pero este es el preponderante en la gran mayoría de redes sociales.

6.1.2 INICIO DE SESIÓN AUTOMÁTICO

El inicio de sesión es una tarea generalmente tediosa y poco deseable. Es por tanto vital que solo se requiera una vez, en aras de una mejor experiencia de usuario.

Una práctica que comparten la gran mayoría de las aplicaciones es almacenar localmente en el dispositivo del usuario un token que identifique al mismo. Dicho token viaja automáticamente al servidor cada vez que el usuario abre la aplicación, y es el servidor el encargado de comparar dicho token y verificar la autenticidad del usuario.

Este proceso de autenticación automática pasa por tanto por el almacenamiento del token, que se guardará en el registro del usuario. Y solo se borrará en caso de que el usuario cierre la sesión (una rareza en el comportamiento del usuario, es decir, poco habitual). En este segundo supuesto cuando el usuario vuelva a iniciar sesión se guardará nuevamente el token en la memoria local para posteriores inicios de sesión.

Se procede a exponer el código que ha sido necesario para configurar el inicio de sesión automático.

- firebase-config.js

```
export const auth = initializeAuth(app, {
  persistence: getReactNativePersistence(ReactNativeAsyncStorage)
});
```

6.2 THEMOVIEDATABASE

TheMovieDataBase (TMDB) [1] es una base de datos cinematográficos pública, en otras palabras, es una API.

Una API, es un servidor que contiene información accesible a través de ciertas direcciones (en inglés y en términos ligeramente más técnicos, estas direcciones son conocidas como

endpoints). En cada dirección que hayan diseñado se halla información que se provee en la respuesta en un formato JSON en este caso.

TMDB es una reputada base de datos debido a su rápida capacidad para actualizar los datos e introducir nuevas entradas, además de la relativa rapidez con la que responde a las peticiones. Esta mezcla de virtudes fue vital a la hora de elegir el proveedor de la información más sustancial de Popco, el meta contenido cinematográfico.

TMDB cuenta con numerosos *endpoints* que cubren casi la práctica totalidad de la información referente a películas y series. Para poder hacer peticiones el único requerimiento será un API key, que es esencialmente una clave proveída por el propio TMDB y que será de obligado uso en cada petición. En cada petición además de especificar la url según el recurso que se quiera obtener, habrá otras partes como los *headers* donde se incluye información relevante para la comunicación. Uno de los *headers* es el Authorization header donde se incluye el API Key provisto por TheMovieDataBase con anterioridad.

Las peticiones se pueden discernir según la intencionalidad de dicha petición. La información almacenada en la base de datos tiene usos diferentes, se puede leer, se puede modificar, se puede crear y se puede borrar. La totalidad de las peticiones que se hacen en el presente proyecto son peticiones GET, cuya finalidad es recabar datos.

La conexión con la API se hace a través de una librería nativa de Javascript llamada *fetch*. *Fetch* es una función que recibe un primer argumento de tipo *String* que corresponde a la url del *endpoint* al que se desea hacer la petición. El segundo argumento es un objeto compuesto por el método de la petición, su clave es "*method*" y su correspondiente valor es el método de la petición en un *string*. Adicional y opcionalmente se pueden especificar otras claves, en el caso de las peticiones realizadas en el presente proyecto se utilizó en la práctica totalidad de los casos la clave *headers* que tiene como valor un objeto donde se especifica el formato del contenido de la respuesta y el token de autorización.

Fetch devuelve una promesa, una promesa es un objeto que representa eventualmente el resultado de una operación asíncrona o en su defecto un fallo en caso de que dicha operación no se haya resuelto exitosamente. Entre el envío de la petición y la recepción de la respuesta hay un espacio temporal, ergo lógicamente la variable en la que se almacene la respuesta no estará inmediatamente a disposición para su uso, sino que habrá de esperar a que llegue la información. Hasta entonces la variable será un objeto promesa.

La pregunta que le podría surgir al lector es, ¿cómo comprobamos pues si la operación ha sido realizada? Pues nuevamente Javascript acude al auxilio a través de una funcionalidad introducida en su versión ES6, el *async await.*

Antes de continuar con la explicación del *async await*, es importante apostillar que alternativamente se podría utilizar un método asíncrono de las promesas, el *then*. *Then* espera a la resolución de la promesa para ejecutar el *resolve*.

El *async await* es una estructura fácilmente legible. El *async* es una forma de especificar en código que la función definida a continuación contiene promesas y que por tanto deberá ser categorizada como asíncrona. El *await* se coloca delante de aquello que vaya a devolver una promesa y obliga a esperar a que se resuelva la promesa antes de continuar con el resto del bloque de código afectado por el *async*.

Después de haber explicado en profundidad el mecanismo, se expone a continuación un ejemplo práctico y real de la aplicación.

```
const getActorInfo = async () => {
    const options = {
      method: "GET",
      headers: {
        accept: "application/json",
        Authorization:
          "Bearer
eyJhbGciOiJIUzI1NiJ9.eyJhdWQiOiIyZWIwNTY0NzcxYjAwYTFiMTE2NjQ5NWQzZWZmYzI3MSIsInN1
YiI6IjY1MTNmMDE2Y2FkYjZiMDJiZGViYWMwZSIsInNjb3BlcyI6WyJhcGlfcmVhZCJdLCJ2ZXJzaW9uI
joxfQ.jqZeYyXUptvRW386HRXK0ih7cWHAIF52D90xJ8fb0nY",
      },
    };
```

```
  const request = await fetch(
    `https://api.themoviedb.org/3/person/${actorID}?language=en-US`,
    options
  );
  if (request.status === 200) {
    const data = await request.json();
    setActorInfo(data);
  }
};
```

Esta función cambia el estado del componente ActorScreen. Como se puede intuir, gracias al explicativo nombre del método, práctica por cierto muy recomendable, el método configura el estado de una variable llamada actorInfo, rellenándola con la información de un actor específico. El método es asíncrono, gracias al async, se recuerda que esto implica que dicho método no se ejecutará secuencialmente, sino que se apilará en la cola de funciones asíncronas de Javascript que automáticamente verifica qué promesas han sido resueltas y cuáles no.

Inicialmente se define el objeto *options*, que tiene la clave *method*, cuyo valor corresponde al método de la petición y la clave *headers*, cuyo valor corresponde a un objeto que a su vez contiene el formato en el que se desea recibir la información y las credenciales requeridas para poder lanzar peticiones contra la API de TMDB.

A continuación, se define la variable *request*, que será la petición per se. Se utiliza el *await* antes del *fetch*, ya que el *fetch* devuelve la promesa. El primer argumento del *fetch* es la url correspondiente al *endpoint* que devuelve la información de un actor, y actorID es la ID del actor en cuestión. El segundo argumento es el objeto *options* previamente definido.

Finalmente, se confecciona un condicional para asegurar que el código de la respuesta es un 200, es decir, que todo ha ido correctamente para después extraer la información pertinente y a través de un set guardarla en una variable de estado.

Se puede acceder a la propiedad *status* de la promesa sin temor alguno a que todavía no se haya resuelto ya que por definición todo lo que suceda al *await* se ejecutará siempre y cuando el *await* ya se haya ejecutado.

Este flujo es sustancialmente importante a lo largo de toda la aplicación debido a la cuantiosa cantidad de información que se demanda a TMDB. Bajo el criterio del autor era por tanto profundizar técnicamente e incluso ejemplificar el uso de dicho flujo.

6.3 NAVIGATION

Popco es una aplicación con numerosas funcionalidades y consecuentemente requiere de numerosas interfaces. Se entiende interfaz como parte visualmente independiente de la aplicación donde se dispone una funcionalidad. De modo que habrá aproximadamente el mismo número de interfaces que de funcionalidades.

Una incógnita que surge naturalmente a la hora de diseñar e implementar todas las interfaces es cómo conectarlas entre sí, para que el usuario pueda navegar a su merced por la aplicación.

Para esta labor se ha hecho uso de una librería de React Native llamada `@react-navigation/native`.

Esta librería provee todos los componentes para crear una estructura de navegación por toda la aplicación. Hay dos tradicionales formas de navegar por una aplicación: Tab Navigation y Stack Navigation [7].

Tab navigation consiste en un menú posicionado por defecto en la parte inferior de la pantalla y ocupa horizontalmente toda la pantalla. Desde el menú se puede acceder a cualquiera de las pantallas definidas en el mismo.

Entre las pantallas a las cuales se puede acceder desde el menú se encuentran la siguientes:

- *Home*
- *My lists*
- *Search*
- *Profile*

El menú de tab navigation es jerárquicamente superior al stack navigation, y es por tanto necesario definirlo en la jerarquía de componentes por encima del stack navigator.

El Stack Navigaton consiste en una pila de pantallas que se agrupan encima unas de otras a medida que se navega por ellas. De forma que, si se dispone de las pantallas A, B y C e inicialmente el usuario se encuentra en A y viaja en B, la pila estará compuesta de B en la parte superior por encima de A.

El usuario se desplazará por las diferentes pantallas apilando cada una por las que pasa encima de la pila.

Para volver atrás, el usuario tendrá únicamente que deslizar horizontalmente de derecha a izquierda, esta acción desapila la última pantalla de la lista y vuelve a la pantalla anterior.

Este flujo es habitual en la mayoría de las aplicaciones, un flujo interiorizado por tanto por cualquier usuario de dispositivo móvil moderno, y de prácticamente de obligada inclusión en cualquier aplicación.

6.4 REACT QUERY

Cuando se trabaja con peticiones al *backend* o incluso a APIs de terceros normalmente se desea minimizar, siempre que sea necesario, el número de llamadas a dichas bases de datos. Como se ha explicado con anterioridad, una petición implica una espera preferiblemente

evitable y costes de mantenimiento del *backend*. Por tanto, se deberá reutilizar siempre la información anteriormente obtenida. Esta práctica se conoce por el anglicismo *cashing*.

El *cashing* consiste en almacenar en memoria información obtenida a través de peticiones asignando a dicha información una clave. Si en ulteriores peticiones se requiere información asociada a dicha clave, en vez de ejecutar la petición se comprobará si existe dicha clave en memoria, en cuyo caso la información almacenada prevalecerá y será utilizada.

De forma que gracias a esta práctica la información aparecerá inmediatamente eludiendo las poco deseables pantallas o animaciones de carga.

6.4.1 LIBRERÍA TAN STACK QUERY

Tan stack query [8] es una librería transversal a cualquier marco de trabajo de javascript e incluso a otros lenguajes, que sirve precisamente para manejar el estado de los datos remotos. Esta no se limita a hacer *cashing* sino que además sincroniza todos los datos, pagina, gestiona los errores y estado de la información, entre otras cosas.

El *cashing* se utiliza a lo largo de toda la aplicación. A continuación, se expondrá un ejemplo representativo de la utilización del *cashing* con React Query.

Cada comentario puede recibir *likes*. En el extremo derecho de cada comentario hay un corazón que en caso de ser clicado lanza el evento de dar «me gusta» a dicho comentario. Visualmente se aprecia gracias a que el corazón se vuelve rojo y además el número de *likes* aumenta en una unidad.

En el siguiente ejemplo, inicialmente el comentario no contaba con ningún *like*, y gracias al recibido por el usuario, el contador indica un *like*.

Ilustración 31. Comentario de una película al que se le ha dado like

Cuando se cargan los comentarios, además del contenido de cada comentario, se ha de cargar si el usuario que visualiza dicho comentario ha dado anteriormente *like* a dicho comentario. Por ejemplo, en la siguiente visual (*Ilustración 32*) el usuario ha dado *like* al primer y tercer comentario de la lista.

Ilustración 32. Varios comentarios en una película, algunos con likes

Este es un excelente ejemplo para explicar cómo el *cashing* puede potencialmente ahorrar peticiones innecesarias que buscan información ya almacenada en memoria. La primera vez que se abra la sección de comentarios se ejecutará automáticamente el useQuery, haciendo

un *fetching* al *backend* definido en la función checkIfLiked. Seguidamente se guardará en la memoria la clave "isCommentLiked + commentID" y su correspondiente valor, de forma temporal. TanStack Query es muy configurable, se puede incluso configurar cuánto tiempo se ha de guardar la información en la memoria caché.

Las posteriores ocasiones en las que dicho comentario vuelva a aparecer requiriendo nuevamente toda la información de si el usuario le ha dado *like* o no, en vez de hacer la petición, el sistema de *cashing* se ocupará de verificar si hay alguna clave en memoria que coincida con "isCommentLiked+ commentID"

```
const checkIfLiked = async (commentID) => {
    const likesCollectionRef = collection(db, "likes");
    const q = query(
      likesCollectionRef,
      where("userID", "==", auth.currentUser.uid),
      where("commentID", "==", commentID),
      limit(1)
    );
    const querySnapshot = await getDocs(q);

    if (querySnapshot.docs.length !== 0) {
      return true;
    } else {
      return false;
    }
  };

  const { data: isCommentLikedByMe, isLoading } = useQuery({
    queryKey: ["isCommentLiked", commentID],
    queryFn: async () => checkIfLiked(commentID),
    //staleTime: 5 * 60 * 1000
  });
```

TanStack Query también proporciona otra funcionalidad interesante, la sincronización de datos. En una aplicación que cuente con múltiples funcionalidades y pantallas, posiblemente surjan algunas dependencias entre sí. De forma que ciertos cambios en una pantalla vendrán propulsados por eventos ocasionados en otras pantallas.

Para la resolución de esta cuestión será pues necesario un buen manejo y gestión del estado global. El estado global es toda aquella información cuya accesibilidad y disposición es demandada por diferentes partes de la aplicación.

En una aplicación con alta demanda de peticiones es probable que la información de alguna de ellas se vea comprometida o modificada por otras, por ende, surge la necesidad de mantener consistencia en todo rincón de la aplicación.

TanStack Query ayuda notablemente en esta labor, proveyendo de un arsenal de instrumentos para manejar el estado de la información remota.

No se hará una indagación exhaustiva de todas las posibilidades que ofrece TanStack Query a este respecto, sin embargo, sí se explicarán las utilizadas en el proyecto.

Para incluir una película o serie en una lista se ha de navegar hasta la página de dicha película y pulsar en el botón de guardar en lista. Una vez completado el primer paso se abrirá un desplegable que contiene todas las listas del usuario, si el usuario pulsa cualquiera de esas listas la película o serie en cuestión se guardará en dicha lista.

Evidentemente si se abre la página de la lista donde se ha introducido la película, cabría esperar que estuviese allí. Pero nada más lejos de la realidad, de alguna forma hay que avisar al componente List que una nueva película se ha añadido a la lista, y que por tanto la información disponible estaba anticuada.

Dentro del método que se ejecuta cuando se selecciona una lista entre todas ellas se encuentra la invalidación de una *query* o clave. (“playlist + playlistID”)

Dentro del método ejecutado cuando se selecciona una lista:

```
queryClient.invalidateQueries(['playlist', item?.id]);
```

Cuando se invalida una *query*, automáticamente allá donde exista un useQuery con dicha clave, se volverá a ejecutar, de forma que el useQuery en el componente *List* vuelve a pedir las películas y series que contiene una lista, pero esta vez, modificada.

En el componente *List*:

```
const { data, isLoading } = useQuery({
    queryKey: ["playlist", playlistID],
    queryFn: async () => {
      const playlistData = (
        await getPlaylist(playlistCollectionRef, playlistID)
      ).data();

      return {
        id: playlistID,
        ...playlistData,
      };
    },
  });
```

A propósito de enseñar más ejemplos de la sincronización de datos, precisamente se utilizará un ejemplo anterior, los *likes* en los comentarios.

Cada vez que un usuario da un *like* a un comentario, en toda pantalla donde aparezca dicho comentario, el *like* deberá aparecer. Es decir, el mismo comentario abierto en diferentes pestañas deberá tener el mismo número de *likes* en todas ellas. Para el propósito anterior se ha utilizado el método setQueryData del ya mencionado objeto queryClient.

```
          queryClient.setQueryData(["isCommentLiked", commentID], false);
```

Esta línea de código se ejecutará en el método de dar like. Si un usuario da like, se ejecutará la línea anterior y automática e inmediatamente todas las partes del código que tengan alguna relación con la query “isCommentLiked, commentID” se modificarán y gozarán del nuevo estado de la variable.

6.4.2 PAGINACIÓN INFINITA O INFINITE SCROLLING

Una función presente en cualquier aplicación donde haya muchas entradas es la paginación. La paginación ayuda a segmentar en bloques potencialmente cualquier cantidad de entradas. Esta función es configurable en la medida que se puede elegir cuántas entradas hay disponibles por página. Si hay 100 entradas y se desean 10 entradas por página, habrá pues 10 páginas. Normalmente en el pie de la interfaz existe un menú desde el cual el usuario puede navegar entre páginas.

Esta práctica se hace para evitar descargar del *backend* todas las entradas simultáneamente, esto es altamente ineficiente en tanto en cuanto el usuario podría requerir muchas menos. De esta forma, solo se descargarán aquellas entradas que el usuario elija.

La paginación convencional es tal y como se ha explicado, sin embargo, con el tiempo, se ha experimentado con formas alternativas que mejoren la experiencia de usuario. Una práctica común son las llamadas *Infinite Scrolling List*. Esto son listas que son aparentemente interminables. Estas cargan el contenido a medida que el usuario desliza hacia abajo. Es tal el éxito que este formato de paginación ha tenido, que difícilmente se encuentran aplicaciones nativas que cuenten con el formato convencional.

En Popco hay más de una parte donde estas listas son requeridas. Donde la cantidad de información podría tornarse cuantiosa. En estas ocasiones como ya se ha comentado, es poco deseable en términos de rendimiento y eficiencia descargar toda la información. Se recomienda paginar la información de forma que solo se descargue si el usuario se queda sin entradas que visualizar.

Para que el lector pueda solidificar este abstracto concepto, se expondrá un ejemplo para aterrizar la idea.

Cada película tiene una sección de comentarios, tantos como la comunidad quiera poner, de forma que no parece impensable que la sección de comentarios de una película relativamente

conocida pueda poblarse de comentarios con facilidad. Este es el escenario ideal para implementar el *Infinite Scrolling*.

A continuación, dos imágenes de Popco que ilustran el funcionamiento del *Infinite Scrolling*.

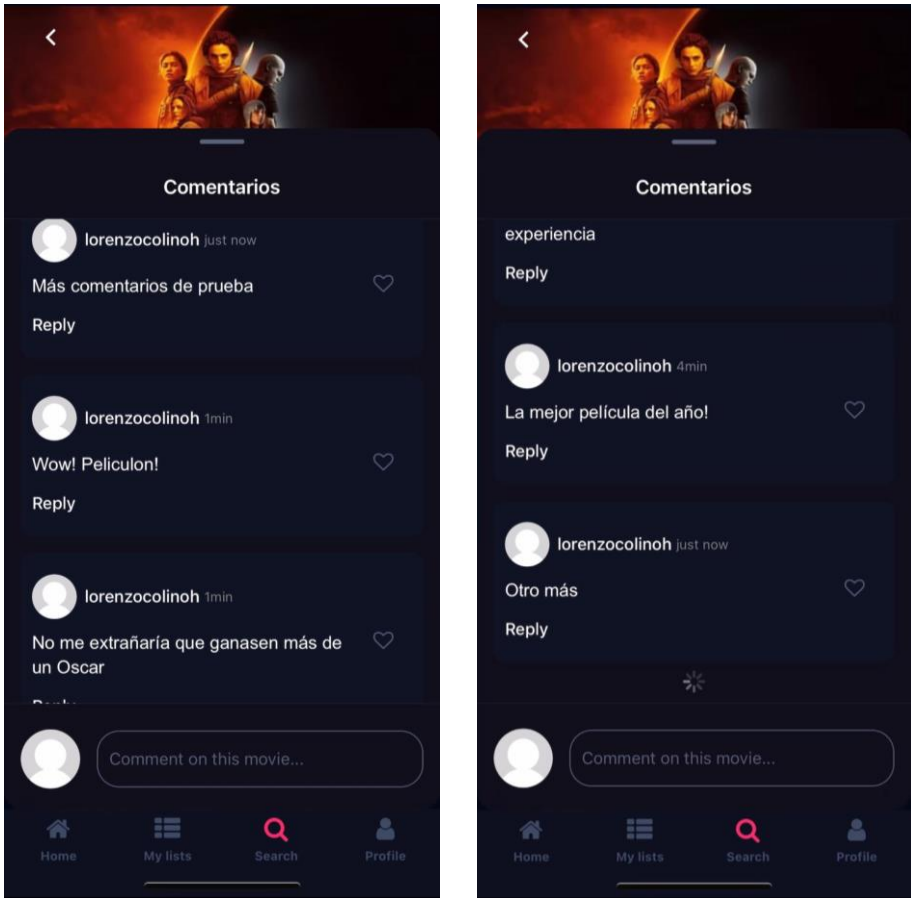


Ilustración 33. Funcionamiento del Infinite Scrolling en comentarios

Inicialmente se cargarán 5 comentarios, dos más de los que caben en la pantalla. La primera imagen corresponde a los primeros 3 comentarios (los que caben en la pantalla), y a medida que el usuario deslice verticalmente hacia abajo se cargarán discretamente nuevos comentarios (segunda imagen). Los comentarios están ordenados por popularidad, de forma que los primeros de la lista serán los que más *likes* hayan recibido. En la segunda imagen se

puede comprobar que, llegado a un umbral vertical de la pantalla, se renderiza una animación de carga simbolizando que se están descargando nuevos comentarios.

El *Infinite Scrolling* se ha implementado en numerosas ocasiones para aquellas las listas que potencialmente puedan contar con demasiadas entradas, como para cargarlas todas a la vez.

- Lista de seguidores

Cada usuario cuenta con una lista de seguidores y seguidos. La magnitud de estas listas depende de los usuarios que hayan solicitado seguir al usuario en cuestión y los usuarios a los que siga el usuario en cuestión, respectivamente.

Evidentemente, lo más habitual será encontrar usuarios que no tengan más seguidores o seguidos que su círculo cercano de amigos. Sin embargo, existe la posibilidad de otras casuísticas, por ejemplo, un usuario famoso cuyas reseñas cinematográficas cuenten con cierta fama y que por tanto tenga muchos seguidores.

No se puede dar por descontada la posibilidad de listas grandes de seguidores o seguidos, sería un craso error. Por tanto, se ha implementado el *Infinite Scrolling*. Así es como ha resultado el diseño visualmente. Inicialmente se cargan 9 usuarios.

- Lista de comentarios populares

Esta es la lista de comentarios más populares emitidos en los últimos 7 días. En este caso se hace patente la necesidad de cargar paulatinamente sólo aquellos comentarios que pueda visualizar el usuario, en caso contrario habría que descargarse simultáneamente todos los comentarios que se hayan emitido los últimos 7 días, que podría ascender a un número enorme.

El código de *Infinite Scrolling* en listas de usuarios es el siguiente:

```
const {
    data,
    isLoading,
```

```
    isError,
    error,
    fetchNextPage,
    hasNextPage,
    refetch,
  } = useInfiniteQuery({
    queryKey: [`${userUID}-${listType}`, inputText],
    queryFn: fetchUsers,
    getNextPageParam: (lastPage) => lastPage?.nextPage ?? undefined,
  });
```

useInfiniteQuery es un hook de TanStack Query, como cualquier useQuery cuenta con una queryKey y una queryFn, que por cierto se explicará en detalle a continuación.

La particularidad es la propiedad getNextPageParam, una función que devuelve si existe siguiente página. Esta propiedad es importante para evitar hacer *fetching* a elementos inexistentes.

Además, useInfiniteQuery pasa como argumento automáticamente el pageParam. Si existe siguiente página, la variable equivaldrá a la siguiente página, en caso contrario será programáticamente equivalente a un false (*null/undefined*)

```
const fetchUsers = async ({ pageParam = null, queryKey }) => {
    const usersCollectionRef = collection(db, "users");
    let q;
    if (queryKey[1].trim().length > 0) {
      q = query(
        usersCollectionRef,
        where(types[listType], "array-contains", userUID),
        where("username", ">=", queryKey[1].trim().toLowerCase()),
        where("username", "<=", queryKey[1].trim().toLowerCase() + "\uf8ff"),
        orderBy("username"),
        startAfter(pageParam),
        limit(7)
      );
    } else {
      q = query(
        usersCollectionRef,
        where(types[listType], "array-contains", userUID),
        orderBy("username"),
        startAfter(pageParam),
        limit(7)
      );
    }
```

```
    const snapshot = await getDocs(q);
    const lastVisible = snapshot.docs[snapshot.docs.length - 1];

    const data = snapshot.docs.map((doc) => {
      return { id: doc?.id, ...doc.data() };
    });
    return { data, nextPage: lastVisible };
  };
```

fetchUser es un método que devuelve los siguientes 7 usuarios de la lista, el cuándo se ejecuta se explorará más adelante, de momento el análisis se limitará a estudiar el método per se.

```
<FlatList
        style={{ marginTop: 10 }}
        keyExtractor={(user) => user?.id}
        data={useros}
        renderItem={(user) => {
          return <UserItem userData={user?.item} listType={listType} />;
        }}
        //onEndReached={loadMoreData}
        onEndReached={() => {
          if (hasNextPage) fetchNextPage();
        }}
        onEndReachedThreshold={0.5}
        //pagingEnabled={true}
        initialNumToRender={3}
        refreshControl={
          <RefreshControl
            refreshing={isLoading}
            onRefresh={refetch}
            tintColor="white" // Set the color to white
          />
        }
        showsVerticalScrollIndicator={false}
      />
```

6.5 ESTADO GLOBAL LOCAL

El estado global es toda información que necesariamente ha de estar disponible y accesible a lo largo y ancho de toda la aplicación. Cada componente en React tiene sus variables de estado, y su uso está constreñido o circunscrito al propio componente.

Sin embargo, hay cierta información que han de utilizar muchos componentes, y en aras de un código más legible, menos redundante y una consistencia mucho mayor, dicha información ha de convertirse en estado global.

En secciones anteriores se ha explicado cómo es el manejo y gestión del estado global con datos remotos, es decir, con la información sustraída del *backend* o APIs.

El estado global local permite que cualquier componente pueda acceder y cambiar una variable que se ha definido centralizadamente.

React Native cuenta con un hook totalmente nativo, useContext, que facilita la creación y compartición de variables globalmente.

Un contexto es en términos simplistas, una zona del código que comparte toda la aplicación. Todo lo definido en el contexto será en términos de programación visible y usable por otros componentes.

Para poder llevar a cabo dicho propósito no solo basta con crear el contexto, sino que además hace falta llevarlo a lo más alto de la jerarquía de componentes.

6.5.1 CONFIGURACIÓN DEL CONTEXTO

Se ha de crear el contexto utilizando el método createContext, que devuelve un objeto *Context*. Después se ha de crear un componente que devuelva *Context.Provider*, que es una propiedad del objeto *context* y que fundamentalmente da accesibilidad a cualquiera de los componentes hijo a todas las variables que se encuentren en el *value*.

De forma que, para utilizar dichas variables compartidas, falta importar el useContext allá donde se requieran. useContext es un método que acepta como argumento el contexto y devuelve un objeto que contiene todas las variables del *value*.

Con importar el useContext (denominado useAuth en el proyecto porque sirve entre otras cosas para la autenticación del usuario) sería suficiente para hacer uso de todas las variables globales.

```
import { useContext, createContext } from "react";

const Context = createContext();

export const useAuth = () => {
  return useContext(Context);
};

export const Contexto = ({ children }) => {
  const [contador, setContador] = useState(0);

  return (
    <Context.Provider value={{ contador, setContador }}>
      {children}
    </Context.Provider>
  );
};
```

6.5.2 JERARQUÍA EN APP.JS

En cualquier proyecto con React hay un componente raíz. Dicho componente será el único ejecutado, de forma que todo lo que se desee incluir en la aplicación deberá estar implícito en App.js.

El App será, por tanto, el punto de entrada sobre el que React monta la aplicación. En términos técnicos, el componente App.js será el primer nodo en el DOM (Document Object Model).

Por ende, para que el Contexto no sea estéril deberemos incluirlo en los eslabones más altos de la jerarquía, y eso es precisamente lo que se ha hecho. Este es el código del App.js:

```
<QueryClientProvider client={queryClient}>
      <AuthProvider>
        <NavigationContainer>
          <PreApp token = {token}/>
        </NavigationContainer>
      </AuthProvider>
    </QueryClientProvider>
```

Se puede comprobar que el AuthProvider, es decir el proveedor del contexto, está incluido en el JSX del componente App, y envuelve al resto de la aplicación que se encuentra implícito en el componente PreApp.

A modo de curiosidad, hay otras funcionalidades que necesitan estar en lo más alto de la jerarquía para funcionar correctamente, como TanStack Query o Navigation.

6.5.3 PUESTA EN ACCIÓN

Entre las numerosas variables que se comparten globalmente se encuentra la variable de estado country y su correspondiente setter, setCountry. Se instrumentaliza esta variable para explicar el funcionamiento del estado global, no obstante, cualquier otra habría valido.

A modo de recordatorio, cabe resaltar que las variables globales no son estrictamente necesarias. Alternativamente podría definirse una variable estado en la raíz de la aplicación y secuencialmente ir compartiéndola entre componentes. Sin embargo, esta tarea es realmente tediosa, convierte el código en poco legible y redundante y además resultaría en un deterioro notable del rendimiento de la aplicación. Este empeoramiento del rendimiento viene inducido por los numerosos re-renderizados que habría que hacer.

Al utilizar useContext, en caso de cambio en el valor de la variable global, solo se renderizan de nuevo aquellos componentes que hagan uso de dicha variable. Sin embargo, al omitir

useContext en el diseño obligamos a que muchos componentes de paso que ni siquiera utilizan la variable, se rendericen forzosa e innecesariamente.

La variable country hace referencia al país donde se encuentre el usuario. Es importante conocer el país donde se encuentre el usuario ya que la disponibilidad de películas en los diferentes proveedores de *video on demand* depende del país. Por ejemplo, Netflix puede tener en su catálogo de reproducción de suscripción España una película que no tenga en Francia. Esto pasa porque en la industria cinematográfica las productoras venden los derechos a distribuidoras que ulteriormente venden a los diferentes proveedores por países.

En Popco hay diferentes interfaces donde aparecen los proveedores. En la *HomePage* aparecen las películas más populares que tienen en catálogo los proveedores a los que el usuario está suscrito, pero además aparecen en cada página de película.

Este es el escenario ideal para el uso de variables globales, se necesita accesibilidad al país de residencia del usuario en dos interfaces distintas.

Por tanto, en el Context se crea la variable country, y a través del useContext se utilizará la variable en las interfaces que dependen de ella.

- Movie Screen:

```
const { country } = useAuth();

const { data: providersData, isLoading } = useQuery({
    queryKey: ["providers", id.toString()],
    queryFn: async () => {
      const data = await getProviders2();
      const myProviders = settingProviders(data.results[country]);
      const allProviders = data.results[country];
      return { myProviders, allProviders };
    },
  });
```

Se utiliza el country en la página de información de película, para solo seleccionar y enseñar al usuario aquellos proveedores que tengan en catálogo dicha película en su país.

6.6 *PUSH NOTIFICATIONS*

Las *push notifications* [9] son mensajes que las aplicaciones pueden enviar al usuario, si el usuario está utilizando el dispositivo, la notificación emergerá normalmente en la parte superior de la pantalla, en su defecto la notificación se apilará en el centro de notificaciones con el resto de las notificaciones provenientes de otras aplicaciones.

Las notificaciones están normalmente asociadas a un evento. El evento puede desencadenarlo el propio usuario receptor de la notificación o incluso otro usuario. Sin embargo, se pueden configurar las notificaciones *push* para que surjan con cierta frecuencia.

Para poder mandar notificaciones *push* hace falta un proveedor de servicios de notificaciones *Push*, en este proyecto se ha utilizado Firebase Cloud Messaging de Google. Este proveedor es el intermediario entre el dispositivo final y el servidor de la aplicación.

La aplicación necesita el permiso explícito del usuario para poder mandarle notificaciones, la convención es mandar un mensaje en forma de Alert, donde se incluye la posibilidad de aceptar o declinar que la aplicación en cuestión te mande notificaciones.

Si el usuario ha dado su permiso automáticamente se consigue un token identificativo del dispositivo del usuario no cambiante. Por cuestiones logísticas, en Popco una vez el usuario da el permiso y se obtiene el token, inmediatamente se guarda en el objeto de dicho usuario en el *backend*. De forma que siempre se disponga de dicho token para poder mandar notificaciones.

De momento solo existe un tipo de notificación. Cuando un usuario emite una reseña de una película o serie, se enviará una notificación a cada uno de sus seguidores.

Traducido a lenguaje técnico, el evento que desencadena el envío de la notificación es la creación de un documento en la colección *comments*.

La lógica dentro del método se limita a primero recabar la información de la reseña emitida. Datos como el nombre de usuario, el contenido de la reseña, y el nombre de la película serán importantes más adelante.

```
exports.commentNotif = onDocumentCreated('comments/{comment}', async (event) => {
    const snapshot = event.data
    const data = snapshot.data()

    const userId = data.userID
    const nombreDeUsuario = data.userName
    const comentario = data.commentText
    const pelicula = data.movieName
```

Como se aprecia en las líneas anteriores el comentario se recibe como argumento, y a través del método data de la propiedad data, se accede a toda la información necesaria.

Después, se define la url del servidor de expo, el primer intermediario de la transacción. Esta url es a la que se tendrá que hacer la petición con toda la información pertinente.

En los *headers*, que son parte de la petición, se especifica el formato, la codificación, el tipo de información y el host, como se puede apreciar en el siguiente extracto del código.

```
const url = "https://exp.host/--/api/v2/push/send";
    const headers = {
     "Host": "exp.host",
     "Accept": "application/json",
     "Accept-Encoding": "gzip, deflate",
     "Content-Type": "application/json",
    };
```

Cabe recordar que el propósito es enviar la notificación a todos los seguidores del usuario emisor de la reseña. Para enviar una notificación a un usuario se requiere su token identificativo, privativo para cada uno de ellos.

Se han de recabar todos los seguidores del usuario.

Posteriormente se hará una petición contra la base de datos, concretamente preguntando por los tokens de cada uno de los seguidores del usuario en cuestión.

Este es un proceso iterativo, razón por la que usaremos un bucle. Por cada iteración, meteremos dentro de la lista expoTokens, el token del usuario iterado.

```
expoTokens = []
    const followers = (await db.doc(`users/${userId}`).get()).data().followers
for (const followerID of followers){
      const followerData = (await db.doc(`users/${followerID}`).get()).data()
      const userToken = followerData?.expoPushToken
      console.log(userToken)
      expoTokens.push(userToken)
    }
```

Una vez hechos los preparativos, solo resta hacer las peticiones pertinentes a los intermediarios que ulteriormente enviarán la notificación.

El método de la petición será un post, el formato del *body* se expone a continuación.

La petición se hace con el método *fetch*, que devuelve una promesa

```
const responses = expoTokens.map((expoToken) => {
      const data = {
       "to": `${expoToken}`,
       "title": "BoxOffice",
       "body": `${nombreDeUsuario} ha comentado en ${pelicula}: ${comentario}`,
      };
      const response = fetch(url, {
        method: 'POST',
        headers: headers,
        body: JSON.stringify(data)
      });
      return response
    });
```

El método de Arrays map devuelve otro array, esta vez un array de promesas. Para resolverlas, aunque hay alternativas, se ha decidido usar el Promise.all, que resuelve promesas almacenadas en un array.

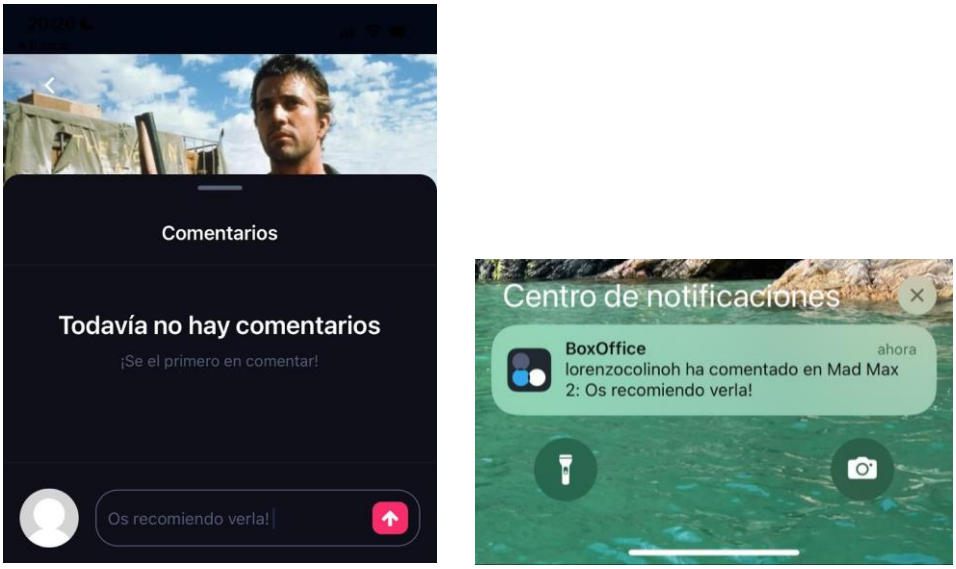


Ilustración 34. Notificación de comentarios

La ilustración de la izquierda es el usuario emisor de la reseña, que posteriormente llegará a sus seguidores. La ilustración de la derecha es el centro de notificaciones de uno de los usuarios que seguían al emisor y por tanto receptor de la notificación.

6.7 TESTFLIGHT

La consecución de cualquier aplicación móvil pasa por la publicación del mismo. Publicar una aplicación es una etapa precedida por el *testing*.

Ninguna aplicación, por bueno que sea su diseño, está exenta de fallos, bugs o margen de mejora. La experiencia de usuario es una cuestión sujeta a constantes cambios en cualquier aplicación.

Suena lógico introducir dentro de las etapas de desarrollo una dedicada al *testing*. Entre los muchos tipos de *testing* se encuentra el más intuitivo, el *feedback* proporcionado por usuarios reales.

Hay diferentes formas de compartir con otros usuarios la aplicación. Una forma algo más rudimentaria y poco escalable, es la instalación de una de las herramientas de Expo, ExpoGo. Esta es una plataforma en forma de aplicación móvil, que permite la ejecución de proyectos realizados con expo. De forma que a través de ExpoGo cualquier proyecto, desarrollado con Expo, se puede ejecutar en un dispositivo móvil, siempre que este ya cuente con la instalación de ExpoGo.

Solo hace falta iniciar el servidor local y conectar a dicho servidor el dispositivo en el que se quiera descargar la aplicación. Este paso es el que se muestra en la *Ilustración 35.*.

Ilustración 35. Iniciación del servidor local

Aunque este método resulte efectivo, desde luego no es eficiente. Para poder probar fidedignamente una aplicación existen diferentes alternativas, entre ellas una que gusta especialmente en la comunidad de desarrolladores, Testflight [10].

TestFlight es una herramienta proporcionada por Apple que permite a los desarrolladores distribuir versiones beta de sus aplicaciones iOS a *testers* internos y externos antes de su lanzamiento oficial en la App Store. Esta herramienta facilita la detección de errores y la obtención de *feedback* valioso sobre las aplicaciones en desarrollo.

Para hacer uso de TestFlight, será primero necesario empaquetar en un formato funcional en el ecosistema IOs, dicho formato es .ipa. Este proceso es llevado a cabo por Expo automáticamente.

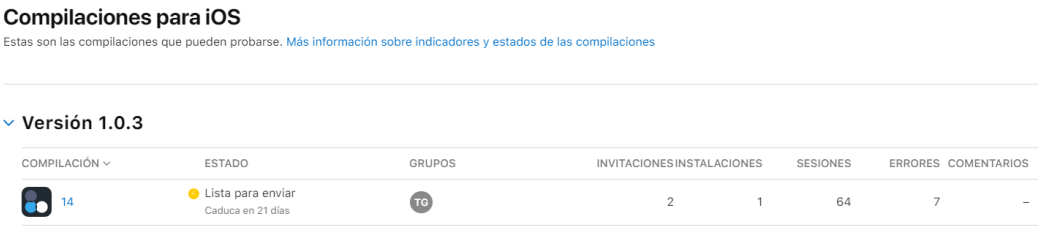


Ilustración 36. Panel de Apple Store Connect

6.8 COMPONENTES DE TERCEROS

Las librerías de nativas de React Native y de Expo son generalmente suficientes en tanto en cuanto cuentan con numerosa cantidad de componentes. Sin embargo, hay ciertas funciones que no son proveídas nativamente por React Native ni por Expo.

Esto no supone ningún impedimento para los desarrolladores de React o Expo, de hecho, una de las grandes ventajas y posiblemente una de las razones de su extendida popularidad, es la gran comunidad que tiene.

Muchas de las librerías más utilizadas en React son desarrolladas íntegramente por terceros.

Con NPM (node packet manager) es relativamente sencillo crear, distribuir e instalar paquetes no nativos. Desde el prisma del autor del presente solo será necesario instalar paquetes desarrollados por terceros.

Para la instalación de paquetes, descontando que ya se ha encontrado el paquete que se desea instalar, únicamente se ha de introducir el comando *"npm install"* seguido del nombre del

paquete en cuestión. Siempre que dicho paquete no sea producto del equipo de expo, la única excepción a la regla.

La gran mayoría de paquetes instalados ya se han mencionado con anterioridad (TanStack Query, Navigation, Firebase etc.), y motivo por el cual, este apartado se limitará a exponer el único componente no nativo que se renderiza visualmente, el BottomSheet. El nombre específico del paquete es:

```
@gorhom/bottom-sheet": "^4.5.1
```

BottomSheet es un componente que emerge verticalmente desde abajo hacia arriba, ocupando horizontalmente toda la pantalla y verticalmente un porcentaje de pantalla previamente definido.

Este componente es un lienzo en blanco sobre el cual se puede renderizar cualquier otro componente. Es especialmente útil en aquellas pantallas donde se requiera de una pantalla complementaria sin necesariamente abandonar la actual.

Para visualizar mejor el componente se ilustrará su uso en diferentes partes de la aplicación.

6.8.1 MOVIE SCREEN

Es la pantalla donde más prepondera el uso del BottomSheet. Es la pantalla informativa de las películas o series. Entre las distintas interacciones que el usuario puede tener con una película se encuentran compartir una película, guardar la película en una lista o comentar en una película. Pese a que estas interacciones no sean estrictamente información de la película, no sería deseable que el usuario tuviese que abandonar la pantalla principal para hacer cualquiera de ellas. En términos de experiencia de usuario no es cómodo.

Además, a pesar de ser una pantalla sobrecargada con datos cinematográficos referentes a la película en cuestión, todas estas opciones están disponibles en MovieScreen gracias al uso de BottomSheet.

Es decir, BottomSheet ayuda proporcionando espacio en pantallas potencialmente sobrecargadas y es una cómoda alternativa para obligar al usuario a abandonar una página en términos de experiencia de usuario.

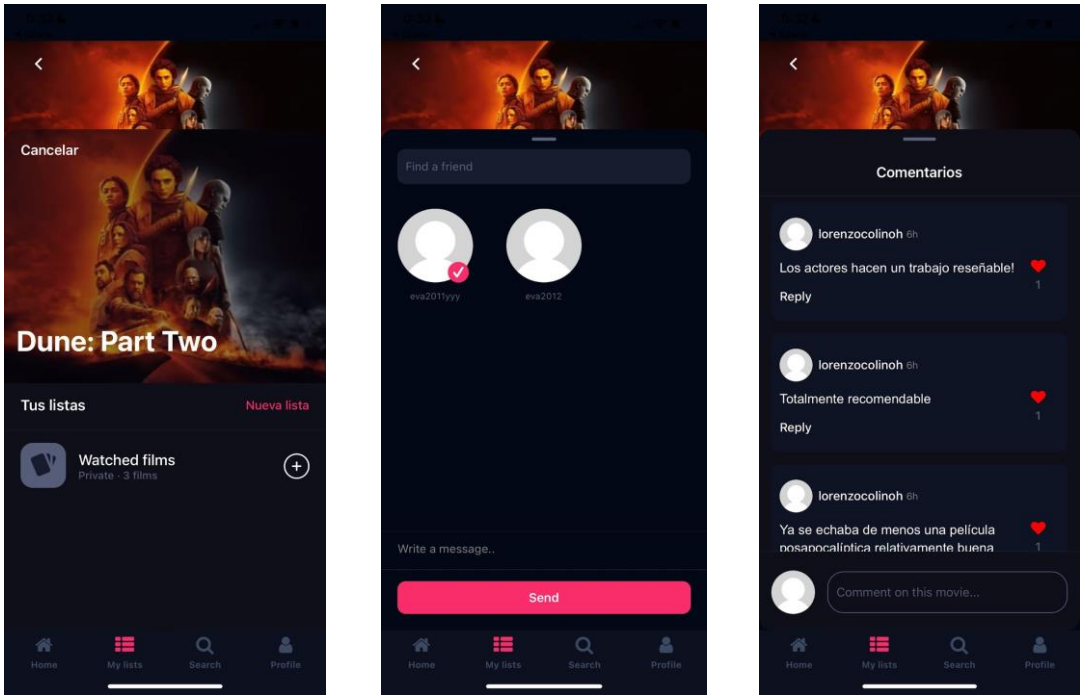
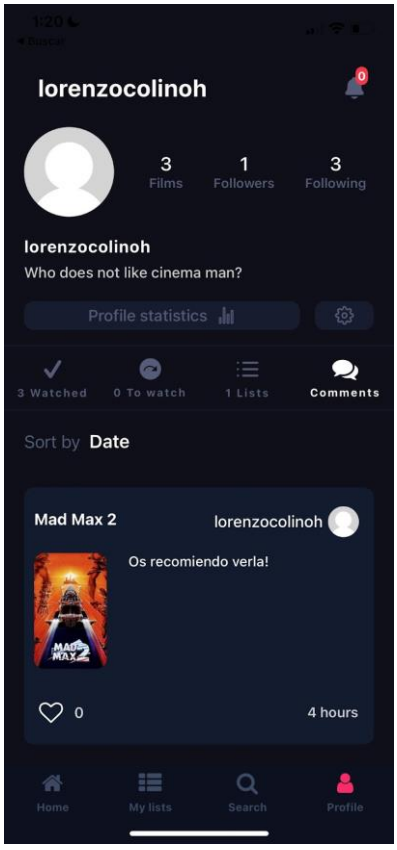
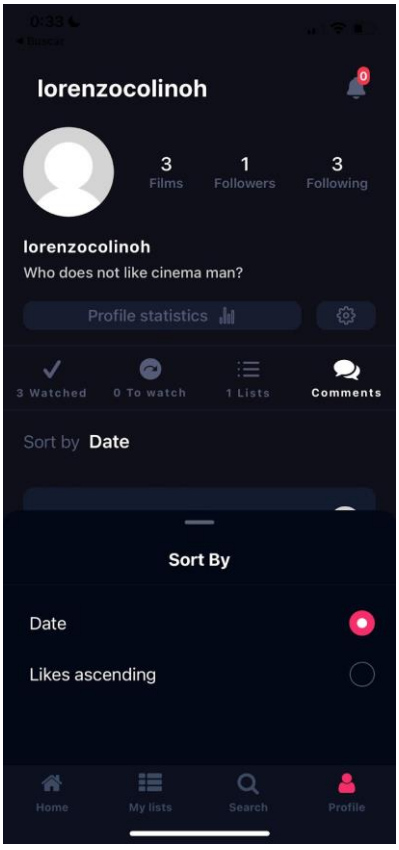


Ilustración 37. Los 3 tipos de BottomSheet que se despliegan según que elija el usuario.

6.8.2 PERFIL

En la página de perfil presenta numerosos datos sobre un usuario, entre ellos una sección dedicada íntegramente a los comentarios emitidos por el usuario. Por defecto aparecerán los comentarios más recientes, pero el orden es configurable y es precisamente la opción de cambiar el orden la que utiliza el BottomSheet, a continuación, una ilustración del funcionamiento.





En la primera visual en el centro de la pantalla hay una frase clicable que versa “Sort by Date”. En caso de ser clicada desencadena un método que desplegará el BottomSheet (segunda visual) desde el cual se podrá seleccionar el orden de los comentarios.

6.9 BACKEND

El *backend* es la parte del sistema que gestiona la base de datos, la autenticación de usuarios, la lógica de servicio y la integración de servicios entre otras. El *backend* en una aplicación no tiene necesariamente por qué incluir todas estas funcionalidades, sin embargo, Popco integra todas ellas.

Existen incontables tecnologías para desarrollar el *backend*. Hay una polarizada distinción entre tipos de *backend*, los proveídos por terceros denominados, *backend as a service* y los *backend* tradicionales en los cuales el desarrollador parte desde cero.

Ambos tienen ventajas e inconvenientes, no es objeto del presente documento explicarlas en profundidad. Se hará un análisis simple, pero concluyente.

Los *backend as a service* normalmente suelen acarrear más costes. El proveedor es un negocio que obtiene beneficio por el ensamblaje del sistema. Despreocuparse del montaje de un sistema tan complejo es, no obstante, una ventaja suficientemente grande como para obviar las desventajas en el presente proyecto.

Normalmente, cuantas más tecnologías del proveedor se aprovechen, más amortizado estará el servicio. En el caso de Popco, se utilizan suficientes como para preferir el *backend as a service*. Asimismo, los costes de las tecnologías de *backend as a service* son exponenciales, y se pueden instrumentalizar como base de *testing* prácticamente a coste cero.

Una ventaja importante del *backend* tradicional es que es más configurable en la medida que se diseña el sistema entero al antojo del desarrollador, sin embargo, Popco no cuenta con dificultades que no se puedan afrontar con el proveedor de *backend.*

Dicho esto, la elección depende totalmente del proyecto.

El *backend as a service* que yo he elegido es Firebase [2], una plataforma desarrollada por Google que es mundialmente conocida y trabaja con importantes proyectos entre ellos aplicaciones móviles como AliExpress o The Economist.

Entre los servicios proveídos por Firebase se encuentran;

- Firebase Authentication: Manejo de la autenticación de usuarios con diversos proveedores como Google, Facebook, Twitter, y más.
- Cloud Firestore y Realtime Database: Soluciones de bases de datos NoSQL.
- Firebase Storage: Almacenamiento de archivos.
- Firebase Hosting: Hosting web para contenido estático.
- Firebase Cloud Messaging: Envío de notificaciones push.
- Firebase Analytics: Análisis de datos de uso y eventos en las aplicaciones.

De este listado se hará uso de Firebase Authentication, Cloud Firestore y Realtime Database y Firebase Cloud Messaging hasta la fecha.

Para la configuración de Firebase será necesario crear una cuenta de google y posteriormente de Firebase. La configuración del proyecto y la asociación con el proyecto React Native está pautada en la documentación de Firebase. Y pasa por el siguiente documento que ha de colocarse en la raíz del proyecto.

```
import { initializeApp } from "firebase/app";
import { initializeAuth, getReactNativePersistence } from 'firebase/auth';
import { getFirestore } from "firebase/firestore";
import { getStorage } from "firebase/storage"
import ReactNativeAsyncStorage from '@react-native-async-storage/async-storage';

const firebaseConfig = {
  apiKey: "AIzaSyAXFjdLqcWaeX5euALDuBcSPZl2yjTUuvE",
  authDomain: "popcodev-9f501.firebaseapp.com",
  projectId: "popcodev-9f501",
  storageBucket: "popcodev-9f501.appspot.com",
  messagingSenderId: "469540300909",
  appId: "1:469540300909:web:8d50242f8afee8cc917764",
};

const app = initializeApp(firebaseConfig);
```

```
export const auth = initializeAuth(app, {
  persistence: getReactNativePersistence(ReactNativeAsyncStorage)
});
export const db = getFirestore(app);
export const storage = getStorage(app)
```

En las líneas anteriores se encuentran las credenciales del proyecto y la configuración inicial para contar con los servicios de Firestore y Authentication.

6.9.1 FIRESTORE

Firestore [2] es uno de los servicios que proporciona Firebase. Consiste en una base de datos no relacional, es decir NoSQL en tiempo real. El esquema conceptual es en términos generales similar al de cualquier otra base de datos, con ciertas particularidades. FireStore es una base de datos con escalabilidad automática que permite consultas complejas con numerosos filtros. Se compone de colecciones, que son agrupaciones de documentos. Los documentos son contenedores de pares clave-valor. Hay muchos tipos de datos almacenables en las claves de los documentos, suficientes como para cubrir las necesidades básicas. Aunque no es de obligatorio cumplimiento, las colecciones suelen contener documentos con un formato parecido. Las consultas de datos se hacen sobre una colección en particular.

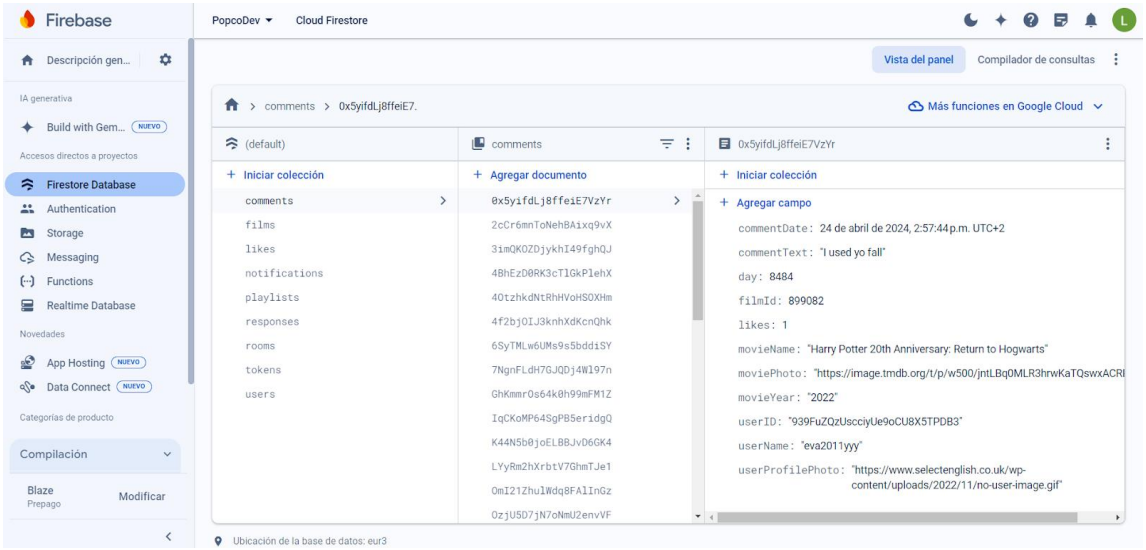


Ilustración 38. Panel de desarrollador de Firebase en el apartado de Firestore

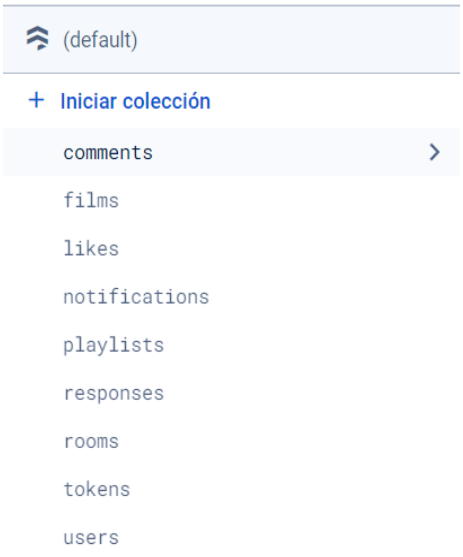


Ilustración 39. Todas las colecciones de en Firestore de Popco

En la visual superior aparece seleccionada la colección de *comments*, agrupación correspondiente a las reseñas de los usuarios, entre el resto de las colecciones. En dicha visual se pueden ver todas las colecciones de Popco.

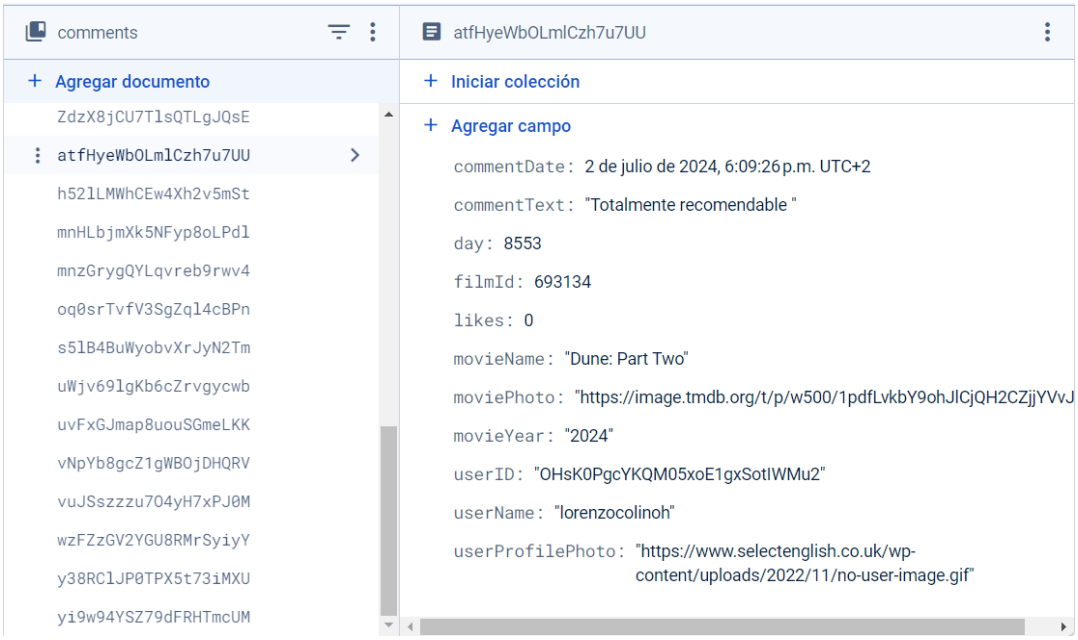


Ilustración 40. Panel de la colección seleccionada: comments

En la *Ilustración 40* solo se representa la parte del panel referente a la colección seleccionada, en este caso, *comments*. Y en la parte derecha se encuentran todos los pares clave-valor de un documento aleatoriamente seleccionado, que lógicamente corresponde a un comentario.

Aunque no haya obligación alguna de mantener una estructura fija que cumplan a rajatabla todos los documentos, se procura que mantengan dicha solidez en pos de información ordenada y fácilmente gestionable.

La estructura de los diferentes documentos, es decir las propiedades que componen los diferentes documentos se ven en las siguientes ilustraciones:

Ilustración 41. Estructura de la colección de comments

Ilustración 42. Estructura de la colección de likes

Ilustración 43. Estructura de la colección de las notificaciones

Ilustración 44. Estructura de la colección de PlayLists

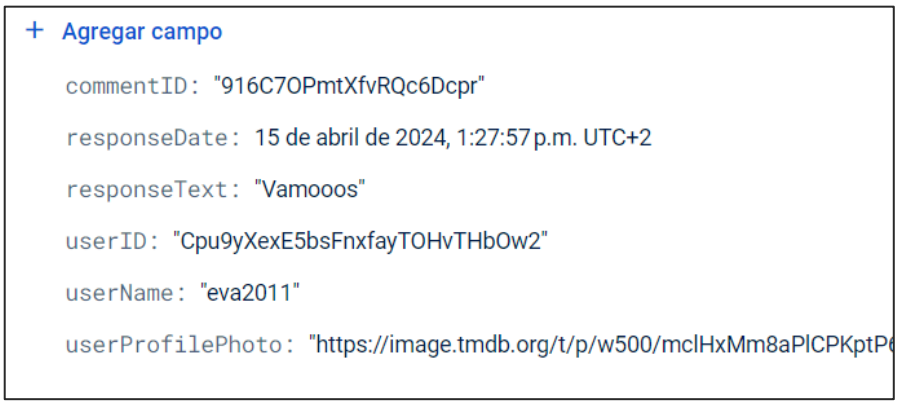


Ilustración 45. Estructura de la colección de respuestas a comentarios

Ilustración 46. Estructura de la colección de tokens

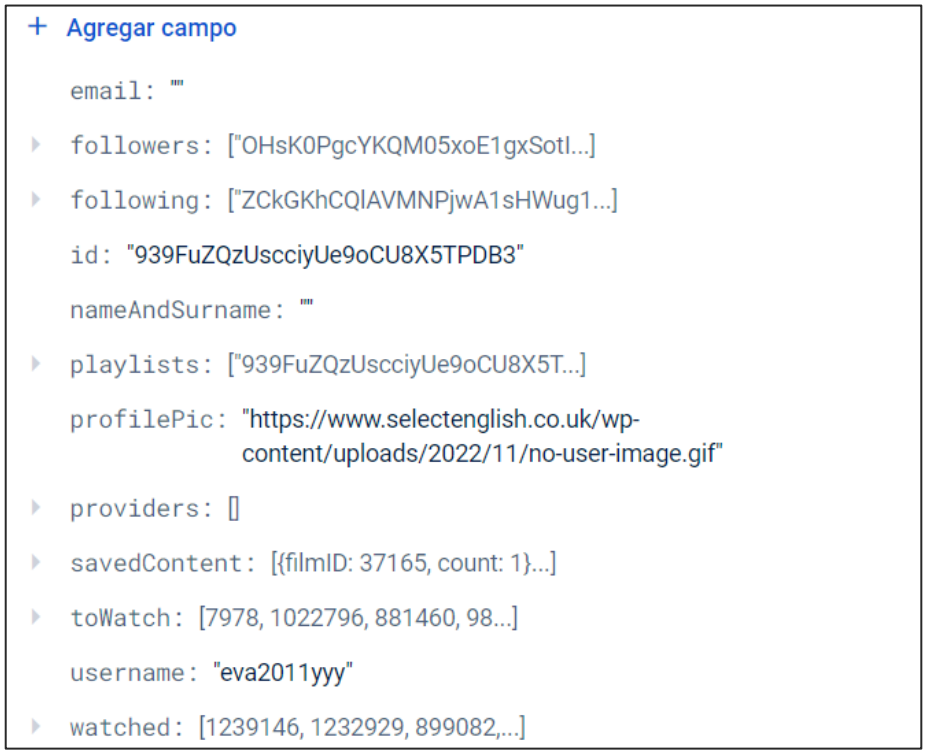


Ilustración 47. Estructura de la colección de usuarios

6.9.2 USO DE FIRESTORE

Los datos se almacenan en el *backend* pero emanan del usuario que opera desde la aplicación. Es indispensable por tanto crear caminos que conecten bidireccionalmente aplicación y *backend*.

Los datos se pueden pedir, se pueden modificar se pueden crear y eliminar. (CRUD)

Para llevar a cabo todas ellas Firestore ofrece un arsenal de posibilidades en forma de métodos. No bastará con usar los métodos, sino que habrá que complementarlos con lógica para que el flujo de la aplicación funcione correctamente.

Se procede a exponer el uso e influencia de Firestore en el proyecto, para poder explicar el funcionamiento de la aplicación por detrás, más allá de lo visualmente apreciable.

Denominador común en todos los métodos – las referencias

En el ecosistema firestore cada pieza de información tiene una dirección, por tanto, para la obtención de un documento específico, se deberá primero obtener la dirección de la colección donde reside el documento.

Cada documento tiene, si no se le indica lo contrario, una dirección irrepetible que lo identifica dentro de su colección automáticamente generada por Firestore cuando se crea el documento.

Para obtener la dirección completa hará falta la combinación de ambas partes. El nombre técnico asociado a estas direcciones son referencias. Estas son imprescindibles, ya que son la forma de explicitar el documento que se desea leer, modificar o eliminar, ninguna de las posibilidades anteriores se podrá llevar a cabo sin una referencia que indique a Firestore dónde se encuentra dicho documento.

Para conseguir las referencias solo harán falta dos métodos y el objeto *database*.

El objeto *database* se exporta en el documento de configuración de Firebase.

Los métodos son *doc* y *collection*.

Collection recibe dos parámetros, el objeto *database* y el nombre de la colección cuya referencia se quiera obtener. *Collection* devuelve la referencia de la *collection* en cuestión.

Doc es el análogo de *collection* para documentos. Son dependientes en tanto en cuanto se necesita la referencia de la colección en la que se pretende buscar para obtener la dirección del documento.

Nuevamente y de forma análoga, con la diferencia de que *doc* cuenta con tres argumentos, se introduce primeramente el objeto *database*, como segundo argumento la dirección de la colección y finalmente el nombre o código identificativo del documento. *Doc* devuelve la referencia del documento deseado.

Ejemplo de referencia de colección:

```
const commentsCollectionRef = collection(db, "comments");
```

Ejemplo de referencia de documento:

```
const userCollectionRef = collection(db, users)
const userRef = doc(db, userCollectionRef, auth.currentUser.uid);
```

Métodos asíncronos

Una petición contra una base de datos ha de viajar desde la aplicación hasta el servidor y la respuesta ha de hacer exactamente lo mismo. La inmediatez presente en cualquier operación local no existe cuando se hacen peticiones y por tanto se trabajará con Promesas.

Todos los métodos que se esclarecerán a continuación son asíncronos y por tanto devuelven promesas.

Las promesas se tratarán con la estructura *async-await* disponible en Javascript. Su uso ya ha sido explicitado con anterioridad y se da por descontado su entendimiento en los apartados venideros.

6.9.3 GET - RETRIEVE

La obtención de datos se hará a través de getDoc y getDocs.

- getDoc

getDoc es un método que recibe como argumento la dirección de un documento específico. getDoc devuelve un objeto que cuenta, entre otros, con un método data() que devuelve la información del documento en forma de objeto Javascript.

Ejemplo: AuthProvider.

```
const getProfile = async () => {
    const userRef = doc(db, "users", auth.currentUser.uid);
    const userData = (await getDoc(userRef)).data();
    setProfile(userData);
  };
```

AuthProvider es un componente jerárquicamente importante, de hecho se encuentra en la raíz del proyecto, por tanto se ejecutará todo lo que contenga inmediatamente.

AuthProvider es el componente encargado de manejar el estado global de la aplicación, es decir, todas aquellas variables cuyo uso se de con frecuencia.

Entre dichas variables se encuentra la información del usuario. Para recabar la información de perfil referente al usuario se ejecuta el método getProfile.

En la primera línea se define la referencia del documento. En la segunda se hace uso de getDoc, introduciendo como parámetro la recién definida referencia del documento, este devolverá el la promesa que luego se resolverá resultando en el objeto user.

Cabe recordar la importancia del await que precede al getDoc. getDoc es un método asíncrono e inevitablemente devolverá una promesa que hay que tratar con el await.

Finalmente, se configura el estado global definiendo la variable compartida profile, disponible a lo largo de toda la aplicación. El empleo de esta variable es frecuentado por páginas como *My Profile*, listas de seguidores, *My playlists*, entre otras. De forma que se evita tener que hacer múltiples veces una petición.

getDoc sirve por tanto para conseguir la información de un documento específico almacenado en la base de datos. Su uso es frecuente, pero resultaría redundante exponer todas las ocasiones en las que se usa. Este ejemplo es muy representativo y tremendamente similar a cualquier otro uso del getDoc.

- getDocs

getDocs es un método que recibe como único argumento una *query*. Una *query* es una consulta a la base de datos que permite opcionalmente filtrado de datos, ordenación o limitació entre otras.

Primero habrá que definir la consulta a través del método *query*. Es un método con un número indeterminado de argumentos, en la medida que se puede modificar al antojo del desarrollador qué filtros aplicar.

Obligatoriamente se deberá especificar qué colección se está atacando, a través de la referencia que habrá de estar en el primer argumento.

A continuación, se explicarán las consultas más frecuentadas dentro de una *query* y cómo llevarlas a cabo programáticamente: *where, limit* y *orderby*.

El *where* consiste en una cláusula donde se especifica qué valor se desea que tenga una clave específica.

```
                    where("userID", "==", auth.currentUser.uid),
```

En la línea anterior se especifica que todos los resultados que devuelva dicha *query* han de ser usuarios cuya propiedad o clave userID coincida con el valor de la variable introducida como tercer argumento.

El *limit* recibe únicamente como argumento un entero, sirve para limitar el número de resultados de una consulta.

Por último, el *orderBy* sirve para ordenar los resultados de la consulta bajo el criterio especificado en el argumento.

getDocs es un método asíncrono, análogamente de getDoc devuelve una promesa, la única diferencia es que la resolución devuelve una lista con todas las coincidencias que se hayan encontrado.

A continuación, algunos ejemplos de uso de getDocs e inevitablemente *query*.

- CheckIfLiked (comments)

```
const checkIfLiked = async (commentID) => {
    const likesCollectionRef = collection(db, "likes");
    const q = query(
      likesCollectionRef,
      where("userID", "==", auth.currentUser.uid),
      where("commentID", "==", commentID),
      limit(1)
    );
    const querySnapshot = await getDocs(q);

    if (querySnapshot.docs.length !== 0) {
      return true;
    } else {
      return false;
    }
  };
```

CheckIfLiked es un método que tiene como propósito devolver un booleano que indique si el usuario ha dado *like* anteriormente a un comentario concreto.

Primeramente, se hace uso del método *collection*, para así conseguir la referencia de la colección que se desea atacar.

Seguidamente, se define la *query*. Se desea comprobar si existe algún *like* cuyo usuario, emisor del *like*, coincida con el usuario usando la aplicación y que además tenga como propiedad "commentID" el código de identificación del comentario que se está evaluando.

Finalmente se ejecuta el método getDocs que materializa la búsqueda de aquellos *likes* que cumplan con la *query*.

Si existe alguna coincidencia, que en términos programáticos consiste en comprobar la longitud de una lista, entonces ese comentario ya tiene un *like* de ese usuario. En su defecto no lo tendrá.

- FriendsThatWatchedTheMovie.js

```
const getFriendsWhoWatchedMovie = async () => {
    const usersCollectionRef = collection(db, "users");

    const q = query(
      usersCollectionRef,
      where("following", "array-contains", auth.currentUser.uid),
      where(`watchedObj.${movieID}`, "==", "true"),
      limit(5)
    );

    const querySnapshot = await getDocs(q);
    const userWhoWatchedMovie = [];

    for (const doc of querySnapshot.docs) {
      userWhoWatchedMovie.push({...doc.data(), id: doc.id});
    }

    return userWhoWatchedMovie;
  };
```

El método getFriendsWhoWatchedTheMovie devuelve todos los amigos de un usuario que hayan visto una película en particular.

Este método es parte de un componente independiente integrado en MovieScreen, es decir, la página de información básica de una película.

En primer lugar, se define la referencia de la colección que atacaremos, que es lógicamente la de *users*.

En segundo lugar, la *query* debe consultar aquellos usuarios de cuya lista de *following* formas parte y que hayan visualizado la película en cuestión. El documento *user* cuenta con una propiedad llamada watchedObj, cuyo valor corresponde a un objeto que tiene por propiedades todas las películas que ha visualizado un usuario. De forma que solo aquellos usuarios que siendo seguidores tuyos y tengan como propiedad de watchedObj el id de la película en cuestión, pasarán el filtro.

Finalmente se ejecuta getDocs que recibe como argumento la consulta y devuelve la lista de usuarios a los que sigues y que además han visualizado la película.

- Comments.js

```
const commentsCollectionRef = collection(db, "comments");
    const q = query(
      commentsCollectionRef,
      orderBy("likes", "desc"),
      and(
        where("likes", ">=", 0),
        or(
          where("day", "==", diasDesde31Enero2001()),
          where("day", "==", diasDesde31Enero2001() - 1),
          where("day", "==", diasDesde31Enero2001() - 2),
          where("day", "==", diasDesde31Enero2001() - 3),
          where("day", "==", diasDesde31Enero2001() - 4),
          where("day", "==", diasDesde31Enero2001() - 5),
          where("day", "==", diasDesde31Enero2001() - 6)
        )
      ),
```

```
      limit(10)
    );

    const querySnapshot = await getDocs(q);
```

Esta es una consulta perteneciente al componente Comments.js. Es el componente que puebla la página de comentarios populares. A modo de recordatorio, la página comentarios populares es una lista con las reseñas emitidas la última semana ordenadas por popularidad.

La consulta es compleja, pero si se desgrana minuciosamente se comprobará que está compuesta de herramientas simples.

Lo primero que se hace es ordenar la consulta por *likes* en sentido descendente, de manera que aquellas reseñas más populares, sinónimo de muchos *likes*, aparecerán primero.

Las consultas *where* anidadas dentro del *or*, devuelven un booleano. Los documentos de comentarios cuentan con una propiedad llamada *day*, que es precisamente la que se compara en el *where*.

Esta propiedad se define en la emisión del comentario o reseña y perdura inamovible, se refiere al número de días entre el 31 de enero de 2001, fecha aleatoria, y el día de la emisión.

El método diasDesde31Enero2001() devuelve la diferencia de días entre la fecha actual y el 31 de enero de 2001. Si la propiedad *day* de algún comentario coincide con algún día de la última semana, entonces se agregará dicho comentario a los resultados de la consulta.

6.9.4 POST - CREATE

La creación de documentos es una tarea relativamente más sencilla que la composición de consultas. El requerimiento para poder llevar a cabo la creación de nuevos documentos es nuevamente la previa obtención de la referencia de la colección donde se desee meter el documento.

Fundamentalmente se hace uso del método addDoc. Este método recibe únicamente dos argumentos, la referencia de la colección y el objeto Javascript.

Para poder visualizar el concepto, se presentará un caso práctico de uso:

- Comentarios.js

```
const onComment = async () => {
    const commentCollectionRef = collection(db, "comments");
    const newCommentData = {
      filmId: film?.id,
      commentText: newComment,
      moviePhoto: `https://image.tmdb.org/t/p/w500${film?.poster_path}`,
      movieName: film?.title,
      movieYear: film?.release_date.slice(0, 4),
      userProfilePhoto: profile?.profilePic,
      userName: profile?.username,
      userID: auth.currentUser.uid,
      likes: 0,
      commentDate: Timestamp.fromDate(new Date()),
      day: diasDesde31Enero2001(),
    };
    Keyboard.dismiss();
    flatListRef.current?.scrollToOffset({ animated: true });
    await addDoc(commentCollectionRef, newCommentData);
    setNewComment("");
    setNumber((prev) => {
      return prev + 1;
    });
  };
```

Comentarios.js es el componente renderizado en el BottomSheet desplegable en MovieScreen.js. Es decir, es el componente que habilita a los usuarios a emitir reseñas, y consecuentemente cuenta con un método específicamente para cargar dicho comentario con sus respectivas propiedades, en la base de datos.

La función onComment no sirve únicamente para enviar la petición POST; esta se desencadena cuando el usuario clica el botón enviar, siempre y cuando el mensaje contenga al menos un carácter.

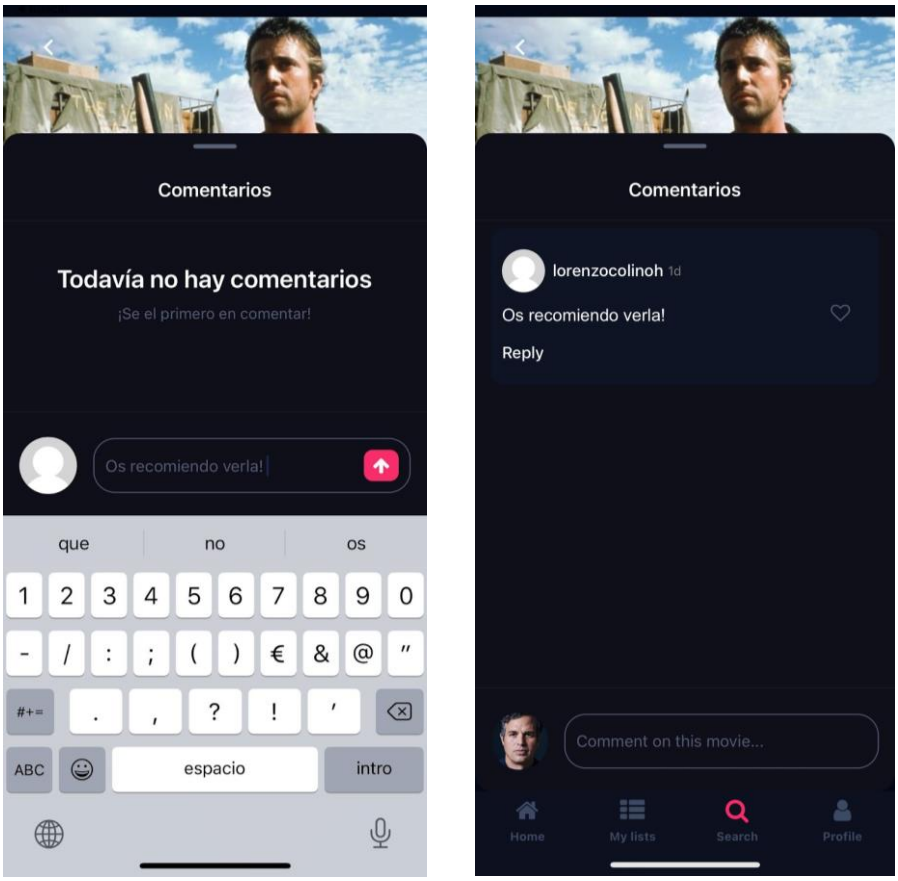


Ilustración 48. Un usuario publica un comentario

La función onComment limpia el estado de la variable que contiene el mensaje, para que en posteriores comentarios el usuario no tenga que borrar manualmente el mensaje, mejorando la calidad de la experiencia de uso. Además, cierra el teclado, nuevamente para que el usuario no tenga que hacerlo.

Finalmente desencadena una animación que desliza al usuario a lo más alto de la FlatList, allí donde se encuentra el comentario recién emitido.

Sin embargo, el propósito fundamental de onComment es cargar el comentario en la base de datos. En las líneas de código dispuestas anteriormente se puede apreciar primero la definición de la referencia a la colección de comentarios, y posteriormente el uso de addDoc con el contenido de la reseña.

- Contenido del comentario:

```
const newCommentData = {
      filmId: film?.id,
      commentText: newComment,
      moviePhoto: `https://image.tmdb.org/t/p/w500${film?.poster_path}`,
      movieName: film?.title,
      movieYear: film?.release_date.slice(0, 4),
      userProfilePhoto: profile?.profilePic,
      userName: profile?.username,
      userID: auth.currentUser.uid,
      likes: 0,
      commentDate: Timestamp.fromDate(new Date()),
      day: diasDesde31Enero2001(),
    };
```

➢ setDoc

Firestore provee de una alternativa al uso de addDoc. setDoc es en términos prácticos análogo a addDoc, en tanto en cuanto sirven el mismo propósito, crear un nuevo documento. Sin embargo, setDoc es ligeramente más configurable.

SetDoc recibe como segundo argumento lo mismo, el contenido del documento a cargar, no obstante, el primero es la dirección del documento.

Esto da la posibilidad al desarrollador, no solo de elegir la colección de almacenamiento, sino el nombre específico con el que se requiera guardar el documento.

De manera que, el documento no tendrá un código de identificación automáticamente generado y totalmente inteligible sino uno diseñado por el desarrollador.

- Ejemplos prácticos:

```
const newUserName = doc(db, "users", uid);
          userSchema = {
            followers: [],
            following: [],
            watched: [],
            toWatch: [],
            description: "Who does not like cinema man?",
            username: username.toLowerCase(),
            profilePic: "https://www.selectenglish.co.uk/wp-
content/uploads/2022/11/no-user-image.gif",
            nameAndSurname: "",
            playlists: [uid],
            savedContent: [],
            activity: [],
            id: auth.currentUser.uid,
            providers: [],
            email: ""
             }
await setDoc(newUserName, userSchema)
```

Cuando un usuario ingresa en la aplicación y completa el registro, dicha información se almacena en un documento dentro de la colección de usuarios.

Las líneas anteriores representan el objeto usuario, poblado con las credenciales introducidas por el usuario en la interfaz, como el nombre de usuario. Seguidamente se hace uso de setDoc para guardar el usuario recién creado.

6.9.5 UPDATE

- updateDoc

El único método a disposición del desarrollador para modificar los documentos existentes es updateDoc. updateDoc es un método que ha de recibir la referencia del documento que se aspire a cambiar, asimismo se habrá de introducir aquellas propiedades que se desee cambiar con sus respectivos valores, todas ellas dentro de un objeto.

```
                await updateDoc(userRef, { watched: newWatched });
```

En este caso se desea cambiar la propiedad watched de un usuario.

Como se puede apreciar, updateDoc no exige la reescritura al completo del documento a cambiar, solo aquella propiedad que se quiera modificar. UpdateDoc sirve también para la adición de nuevas propiedades anteriormente inexistentes.

Haciendo gala de la estructura seguida en los apartados anteriores de la presente sección, se expondrán ejemplos prácticos del uso de updateDoc.

- Settings.js

Settings.js es el componente renderizado en la página de Configuración, desde la cual se puede modificar cualquier aspecto del perfil del usuario.

Lógicamente estas modificaciones habrán de ser reflejadas en el documento del usuario en la base de datos, dicho proceso se llevará a cabo a través del método updateDoc.

- Cambio de username y nameAndSurname:

```
const makeChanges = async (newUsername, nameAndSurname) => {
    const usersCollection = collection(db, "users");
    const me = doc(usersCollection, auth.currentUser.uid);
    if (!!newUsername) {
      const q = query(usersCollection, where("username", "==", newUsername));
      const snapshot = await getCountFromServer(q);
      if (snapshot.data().count === 0) {
        await updateDoc(me, { username: newUsername });
      }
    }
    if (!!nameAndSurname) {
      await updateDoc(me, { nameAndSurname });
    }
  };
```

makeChanges es el nombre determinado para el método encargado de actualizar el estado de la información en Firestore.

En primer lugar, se define como de costumbre la referencia del documento a modificar, en este caso dicho documento será el usuario.

Seguidamente si el nuevo nombre de usuario tiene caracteres, es decir, no está vacío, se procederá a ejecutar updateDoc. El mismo procedimiento se lleva a cabo con el nameAndSurname.

- Providers.js

Desde la página de configuración también se pueden seleccionar las plataformas a las que el usuario está suscrito. De forma que la aplicación solo le recomiende películas y series disponibles en aquellas plataformas a las que puede acceder.

El documento usuario, correspondiente a cada usuario y localizable en la colección usuarios, posee una propiedad denominada *providers*, una lista con los códigos identificativos de cada uno de los proveedores que haya seleccionado.

La lógica que sostiene esta funcionalidad es la siguiente:

```
const addProviders = async () => {
    if (selectedForDelete.length === 0) {
      navigation.push("AddProviders");
    } else {
      setSelectedForDelete([]);
      const newProviders = profile?.providers?.filter(
        ({ provider_id }) => !providerExists(selectedForDelete, provider_id)
      );
      setProfile({ ...profile, providers: newProviders });
      await updateDoc(me, { providers: newProviders });
      updateProfile();
    }
  };
```

6.9.6 TRANSACCIONES

La consistencia en una base de datos es generalmente recomendable. Hay casos donde puede no ser imprescindible y por el contrario hay casos donde se convierte en una exigencia. Por

ejemplo, si un usuario decide convertirse en seguidor de otro modificando su lista de seguidores, este segundo deberá también modificar su lista de seguidos. Realizar la primera, pero no la segunda podría resultar en un grave problema de inconsistencia y una grieta que podría expandirse deteriorando consigo la aplicación.

Un problema latente en Firestore es que los documentos no pueden ser editados en espacios temporales cercanos. Si un documento es editado por dos usuarios a la vez los cambios del primero pueden no verse reflejados y sobrescritos por el segundo. Para evitar este tipo de casuísticas, Firestore provee de una herramienta que mitiga cualquier inconsistencia, las transacciones.

Las transacciones sirven para ejecutar simultáneamente conjuntos de operaciones codependientes. Hay operaciones que han de hacerse conjuntamente, en su defecto perderían el sentido.

Es por ello, que las transacciones en Firestore bloquean momentáneamente el documento que se quiera modificar para que el usuario pueda hacer las operaciones y dan el registro más reciente de los datos del documento. De esta manera, si otro usuario casualmente modifica el mismo documento, esperará a que la transacción haya finalizado para hacer efectiva la nueva modificación.

Con esta herramienta el desarrollador se podrá despreocupar de una posible coincidencia de intenciones entre dos usuarios. Las transacciones aseguran la exitosa ejecución de las operaciones que se quieran hacer dentro de la misma.

Un ejemplo de transacción dentro de Popco son los *likes*. Un comentario es público de manera que cualquier usuario dentro de la aplicación tiene accesibilidad completa al mismo. Un comentario puede recibir múltiples *likes*, lo que podría resultar potencialmente peligroso si dos usuarios deciden dar *like* al mismo comentario simultáneamente.

Cuando se da un *like*, no sólo se suma una unidad a la cantidad de *likes* que tiene un comentario, propiedad numérica de los comentarios, sino que además se crea un objeto *like*.

Sin la protección de las transacciones si dos usuarios le dan a *like* a la vez al mismo comentario, el primero crearía el registro de *like* y sumaría una unidad a la agregación de *likes* del comentario. El segundo haría lo mismo sin embargo al llegar en términos temporales, infinitesimalmente después, crearía el *like* y además agregaría una unidad a la propiedad *likes* del comentario sin tener en cuenta el *like* del primer usuario.

El comentario concluiría estas operaciones con solo una unidad más de *likes*, en vez de las dos que le corresponde y la colección de *likes* contaría con dos *likes* más.

El usuario segundo sobrescribe no intencionadamente el *like* del primero, convirtiendo a este último virtualmente inexistente y creando una inconsistencia entre colecciones.

Para aplacar este problema se ha hecho uso del poder de las transacciones de la siguiente manera:

```
const handlePress = () => {
    if (timerId) {
      clearTimeout(timerId);
    }

    setIsLiked((prev) => {
      if (prev) {
        setNumSum((prev2) => {
          if (prev2 === 0) {
            return -1;
          }
          else if (prev2 === 1){
            return 0
          }
        });
      } else {
        setNumSum((prev2) => {
          if (prev2 === 0) {
            return 1;
```

```
        }
        else if (prev2 === -1){
          return 0
        }
      });
    }
    return !prev;
  });
  const isLikedByMeRef = doc(
    collection(db, "likes"),
    `${commentID}${auth.currentUser.uid}`
  );
  const commentRef = doc(collection(db, "comments"), commentID);
  const newTimerId = setTimeout(async () => {
    await runTransaction(db, async (transaction) => {
      const likedByMeDoc = await transaction.get(isLikedByMeRef);
      const commentData = await transaction.get(commentRef);
      if (likedByMeDoc.exists() && isLiked) {
        const newLikes = commentData.data().likes - 1;
        transaction.update(commentRef, { likes: newLikes });
        transaction.delete(isLikedByMeRef);
        queryClient.setQueryData(["isCommentLiked", commentID], false);
      } else if (!likedByMeDoc.exists() && !isLiked) {
        const newLikes = commentData.data().likes + 1;
        transaction.update(commentRef, { likes: newLikes });
        const newLike = doc(
          db,
          "likes",
          `${commentID}${auth.currentUser.uid}`
        );
        const newLikeData = {
          userID: auth.currentUser.uid,
          commentID: commentID,
        };
        transaction.set(newLike, newLikeData);
        queryClient.setQueryData(["isCommentLiked", commentID], true);
      } else if (
        (!likedByMeDoc.exists() && isLiked) ||
        (likedByMeDoc.exists() && !isLiked)
      ) {
        console.log("No haremos nada");
      }
    });
  }, 3000);
  setTimerId(newTimerId);
};
```

También se ha hecho uso de las transacciones en el sistema de solicitudes de amistad, para lograr consistencia entre las listas de *following* y de *followers*.

6.9.7 AUTHENTICATION

Firebase Authentication es un servicio ofrecido por Firebase que permite gestionar el proceso de verificación de usuarios de forma sencilla y segura. Este servicio proporciona diversos métodos de autenticación y facilita la integración con otros servicios de Firebase.

Hasta la fecha solo se ha implementado la autenticación con correo y contraseña.

La seguridad es automáticamente gestionada por Firebase, de forma que el desarrollador se puede despreocupar, fundamentalmente delegar la encriptación de las contraseñas. Además de la propia seguridad que proveen los servidores de Google.

Además, ofrece un almacenamiento independiente de todos los usuarios registrados.

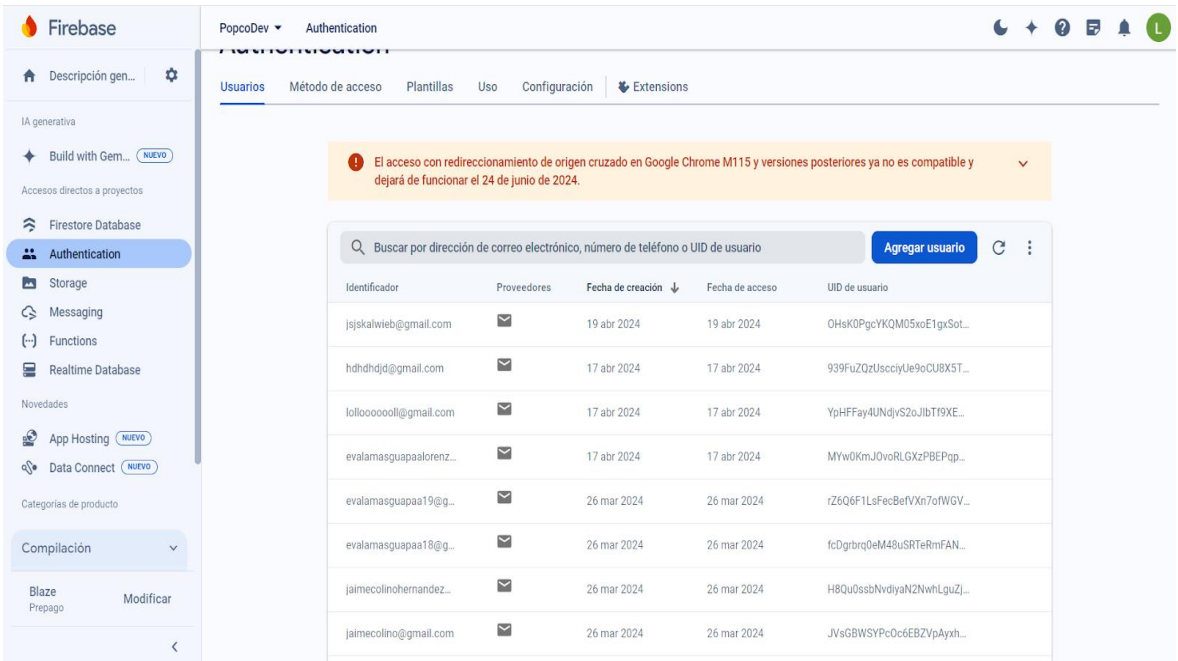


Ilustración 49. Panel de control de autenticación

Firebase provee al desarrollador de ciertos métodos para la conexión entre interfaz y backend.

Los usuarios han de registrarse e iniciar sesión desde el frontend y mandar las credenciales al backend para poder contrastarlas y aceptarlas en caso de que sean correctas.

Estos métodos son singInWithEmailAndPassword y createUserWithEmailAndPassowrd

Para registrar a los usuarios se hará uso de createUserWithEmailAndPassowrd que recibe el mail y la contraseña.

Para iniciar sesión se utilizará el método singInWithEmailAndPassword que recibe exactamente los mismos parámetros.

A continuación, la implementación de estos métodos a las páginas de Register.js y Login.js respectivamente.

```
const handleLogin = async () => {
      const user = await getDoc(doc(db, "users", username))
      try{
        if (!user.data()){
          const credencial = await
createUserWithEmailAndPassword(auth,email,password)
          const uid = credencial.user.uid
          const newUserName = doc(db, "users", uid);
          userSchema = {
.
            followers: [],
            following: [],
            watched: [],
            toWatch: [],
            description: "Who does not like cinema man?",
            username: username.toLowerCase(),
            profilePic: "https://www.selectenglish.co.uk/wp-
content/uploads/2022/11/no-user-image.gif",
            nameAndSurname: "",
            playlists: [uid],
            savedContent: [],
            activity: [],
            id: auth.currentUser.uid,
            providers: [],
            email: ""
```

```
        }
        await setDoc(newUserName, userSchema)

        const newPlaylist = doc(db, "playlists", uid);

        const playlistSchema = {
          name: "Watched films",
          description: "",
          photo: "",
          visibilidad: "Private",
          default: true,
          participants: {[auth.currentUser.uid]: "creator"},
          films: [],
          follows: 0,

        }
        await setDoc(newPlaylist, playlistSchema)
      }
      else {
        Alert.alert("Este usuario o email ya existen")
      }
    }
    catch(err){
      console.log(err)
    }
  }
```

En el método *register* se aprovecha, en términos logísticos, para crear el documento usuario. A pesar de tener una base de datos en la sección de autenticación, en esta solo es posible almacenar las credenciales del usuario exclusivamente, es por ello que se crea simultáneamente un documento almacenado en la colección de usuarios donde se poblará con información privativa de cada usuario.

```
const handleLogin = async () => {
      await signInWithEmailAndPassword(auth,email,password)
  }
```

Capítulo 7. ANÁLISIS DE RESULTADOS

El propósito final de cualquier proyecto es materializar la idea inicial en un producto útil. No solo se han cumplido con creces las expectativas establecidas en la concepción inicial del proyecto, con alguna salvedad, sino que la aplicación ya es una realidad, disponible para su descarga en cualquier dispositivo. El hecho de poder utilizarla personalmente representa un completo éxito, validando el esfuerzo y la dedicación invertidos en su desarrollo.

7.1 FRONTEND

La interfaz de usuario de la aplicación, inicialmente esbozada en un boceto, se ha materializado de manera correcta. A pesar de la complejidad de la propuesta, se han cumplido los objetivos, replicando los bocetos con precisión. Popco ofrece una apariencia visual que proyecta seriedad y profesionalismo. Este éxito en el *frontend* se debe fundamentalmente al estudio exhaustivo de las bases de React Native y JSX, así como de sus componentes nativos. La interfaz no solo es visualmente atractiva, sino también funcional y accesible, lo que mejora significativamente la experiencia del usuario.

7.2 BACKEND

El desarrollo del *backend* para una aplicación tan extensa y completa presentó desafíos significativos, desde la elección de la infraestructura hasta su implementación. Optar por un *Backend as a Service* (BaaS) en lugar de una configuración manual y personalizada ha sido una decisión acertada. A pesar de la pérdida de flexibilidad, Firebase ha proporcionado un conjunto de herramientas robustas y adaptables a cualquier proyecto. La principal desventaja es el potencial aumento en los costes si la aplicación atrae a un gran número de usuarios, debido a la estructura de costes escalables de los BaaS. No obstante, esta limitación no ha afectado al proyecto, ya que actualmente se encuentra en versión beta. Esta elección ha

permitido un desarrollo más rápido y eficiente, lo que ha sido crucial para el éxito inicial del proyecto.

Esto no ha llegado a ser un problema, ya que el proyecto sigue siendo una versión beta. Estas razones hacen de la decisión de optar por un BaaS un completo acierto.

El esquema final de la aplicación se expone en la siguiente imagen. El usuario interactúa directamente con el *frontend* que está conectado con el *backend*, compuesto por la API, desde donde emana la información cinematográfica y el BaaS, Firebase, donde se almacena toda la información del servicio. Las conexiones son bidireccionales, el usuario interactúa con el *frontend* y a su vez este reacciona a las interacciones enseñando visualmente los cambios. El *frontend* tiene el rol de intermediación entre el usuario y el *backend*. Desde el *frontend* el usuario manda peticiones al *backend* que posteriormente el *backend* responderá, devolviendo la información al *frontend*. Finalmente el *frontend* hace llegar la información de forma visual al usuario.

Ilustración 50. Esquema general de la aplicación

Capítulo 8. CONCLUSIONES Y TRABAJOS FUTUROS

El mayor logro de este proyecto ha sido la consecución del objetivo principal: crear una red social dedicada a los cinéfilos. He desarrollado una plataforma que no solo permite a los usuarios buscar y obtener información detallada sobre películas, sino que también les ofrece un espacio para interactuar, compartir opiniones y conectar con otros entusiastas del cine. Esta red social proporciona perfiles personalizados, seguimiento de películas, listas temáticas y un sistema de recomendaciones personalizadas entre otras.

A lo largo del desarrollo de Popco, he adquirido una perspectiva panorámica sobre la conceptualización y materialización de una aplicación móvil. Este proyecto ha sido una experiencia reveladora que me ha permitido entender mejor los múltiples aspectos y etapas involucradas en el desarrollo de una aplicación, especialmente una con objetivos comerciales. Desde la idea inicial hasta la implementación final, cada fase ha requerido una atención meticulosa y un compromiso constante con la excelencia.

Un proyecto de esta envergadura y ambición ha nutrido en mí un espíritu perfeccionista. Este enfoque me ha impulsado a profundizar en el aprendizaje de todas las tecnologías disponibles, asegurando que cada decisión tomada y cada funcionalidad implementada estuvieran alineadas con los más altos estándares de calidad. He aprendido a valorar la importancia de mantenerse al día con las últimas herramientas y técnicas, así como a aplicar estas tecnologías de manera efectiva para resolver problemas específicos y mejorar la experiencia del usuario.

En términos metodológicos, este proyecto me ha enseñado la importancia de una planificación y conceptualización detalladas antes de comenzar con la programación. Diseñar y conceptualizar en profundidad ha demostrado ser mucho más productivo que lanzarse a programar sin directrices claras. Este enfoque no solo facilita la identificación de posibles obstáculos y la formulación de soluciones antes de que surjan problemas, sino que

también asegura que el producto final sea coherente y cumpla con los objetivos establecidos desde el principio.

La fase de diseño me permitió explorar y definir las funcionalidades clave de la aplicación, establecer la estructura de la base de datos, y diseñar una interfaz de usuario intuitiva y atractiva. La conceptualización en profundidad me ayudó a visualizar el producto final y a anticipar las necesidades y expectativas de los usuarios. Esta preparación meticulosa fue crucial para mantener el proyecto en el camino correcto y evitar desviaciones que pudieran comprometer la calidad y la coherencia de la aplicación.

Además, este proyecto ha subrayado la importancia de la iteración y la retroalimentación continua. A lo largo del desarrollo, se realizaron múltiples pruebas y revisiones, lo que permitió ajustar y mejorar la aplicación en base a las observaciones y comentarios recibidos. Este ciclo iterativo de desarrollo y evaluación asegura que el producto final no solo cumpla con los estándares técnicos, sino que también satisfaga las expectativas de los usuarios y ofrezca una experiencia óptima.

Igualmente, a lo largo del desarrollo de esta aplicación, he adquirido una notable capacidad y soltura para aprender y adaptarme a la documentación de las diversas tecnologías involucradas. Este proyecto me ha obligado a sumergirme en una variedad de recursos documentales, desde guías de usuario y manuales técnicos hasta foros de discusión y tutoriales en YouTube. Esta experiencia me ha enseñado a navegar eficientemente por la documentación técnica, a extraer la información relevante rápidamente y a aplicar estos conocimientos de manera práctica en el desarrollo de la aplicación. La habilidad de comprender y utilizar la documentación de manera efectiva ha sido fundamental para resolver problemas técnicos, implementar nuevas funcionalidades y optimizar el rendimiento de la aplicación, consolidando así una competencia esencial para cualquier desarrollador.

Popco es un proyecto con un largo recorrido por delante. Es una aplicación en un sector que clama la presencia de una red social como Popco. Los agentes de la industria del cine tanto

usuarios y consumidores como productores, distribuidoras o plataformas de video on demand estarían profundamente interesados en la existencia de Popco, cada uno por sus particulares razones.

Esta demanda supone una interesante oportunidad que pasa por la continuación en el desarrollo y mantenimiento de la aplicación en una primera instancia y una planificación en ámbitos no solamente relacionados con el software, sino con la comercialización y marketing del proyecto. Fundamentalmente implicaría contratar a desarrolladores de software para una mejora cualitativa del proyecto. Un departamento de marketing para impulsar la popularidad de la aplicación, preponderantemente a través de redes sociales, y finalmente un departamento de ventas esencialmente dedicado a la promoción de un espacio publicitario en la aplicación. Como se ha expuesto en la memoria, Popco reunirá potencialmente a millares de cinéfilos y por tanto se convierte en un escaparate perfecto para promocionar películas. El departamento de ventas facilita la promoción de cara a productoras, distribuidoras e incluso plataformas de *video on demand* o cadenas de televisión, todos ellos agentes de la industria interesados en dar a conocer sus películas.

En resumen, este proyecto me ha brindado una visión integral del proceso de desarrollo de aplicaciones móviles, destacando la importancia de una planificación meticulosa, el aprendizaje continuo y la adaptación a nuevas tecnologías. Estos conocimientos y habilidades adquiridos son de gran valor y me servirán de guía en proyectos similares y en el prometedor futuro de Popco, asegurando que aborde cada nuevo desafío con una sólida base de experiencia y un enfoque estratégico bien definido.

Capítulo 9. BIBLIOGRAFÍA

[1] API The Movie Data Base (TMDB). Recuperado de https://www.themoviedb.org/

[2] Firebase y Firestore. Recuperado de https://firebase.google.com/docs/firestore?hl=es-419

[3] React. Recuperado de https://es.react.dev/

[4] React Native. Recuperado de https://reactnative.dev/

[5] Librería asyncStorage. Recuperado de https://reactnative.dev/docs/asyncstorage

[6] Librería Expo. Recuperado de https://expo.dev/

[7] Librería Navigation. Recuperado de https://reactnavigation.org/

[8] React Query - TanStack Query. Recuperado de https://tanstack.com/query/latest/docs/framework/react/overview

[9] Librería pushNotifications. Recuperado de https://reactnative.dev/docs/pushnotificationios

[10] TestFlight. Recuperado de https://www.google.com/search?q=testflight&rlz=1C1ONGR_esES987ES987&oq=test&gs_lcrp=EgZjaHJvbWUqDggAEEUYJxg7GIAEGIoFMg4IABBFGCcYOxiABBiKBTIRCAEQRRg5GEMYsQMYgAQYigUyBggCEEUYQDIGCAMQIxgnMgwIBBAAGEMYgAQYigUyBggFEEUYPDIGCAYQRRg9MgYIBxBFGDzSAQgzNDE3ajBqN6gCALACAA&sourceid=chrome&ie=UTF-8

[11] NPM (Node Packet Manager). Recuperado de https://www.npmjs.com/

[12] Caballero, D. (2024, mayo 8). Prime Video lidera el streaming en España y HBO Max se dispara tras el acuerdo con Movistar. ADSLZone. Recuperado de https://www.adslzone.net/noticias/streaming-tv/plataformas-streaming-mas-suscriptores-0524/

[13] Fernández, S. (2024, abril 19). Más de 9 millones de personas se suscribieron a Netflix en el primer trimestre de 2024. La Vanguardia. Recuperado de https://www.lavanguardia.com/andro4all/series/mas-de-9-millones-de-personas-se-suscribieron-a-netflix-en-el-primer-trimestre-de-2024#:~=Netflix%20ya%20tiene%20m%C3%A1s%20de,tres%20primeros%20meses%20de%202024

ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS

Después de un análisis de todos los objetivos de desarrollo sostenible se ha llegado a la conclusión de que la aplicación cumplirá con muchos de ellos de manera directa e indirecta.

El título del objetivo número cuatro se llama "Educación de calidad", y recita la importancia de una enseñanza útil y al alcance de todos. Uno de los objetivos de Popco es fomentar la cultura cinematográfica, a través de críticos y una comunidad de cine. De esta forma PopCo fomentaría la educación y la cultura general involucrándose en el desarrollo educativo de los usuarios disfruten de dicha funcionalidad.

PopCo tiene que objetivo ulterior una mejora sustancial de la experiencia cinematográfica en todos sus formatos, es por ello por lo que parece indiscutible que PopCo acabará mejorando la industria, esta implicación casa muy bien con el objetivo número 9, cuyo título es "Trabajo Decente y Crecimiento Económico".

Como se comentará más adelante, específicamente en la sección de recursos, PopCo es una aplicación que pretende crecer con tecnologías punteras. Modernizar el sector del cine no es tarea fácil, pero en caso de lograr integrar todas las funcionalidades mencionadas con las tecnologías incipientes, se modernizará con creces el sector, cumpliendo con el objetivo número 9, Industria, Innovación e Infraestructura.

El apartado que te permite ver la disponibilidad de las películas en los VOD cumple una doble función en miras al cumplimiento de los Objetivos de Desarrollo Sostenible. Que los usuarios sean conocedores de las mejores alternativas a la hora de buscar la película que desean ver, casa a la perfección con el objetivo número 12 ("Producción y Consumo

Responsables"), proporcionando una información que facilita la elección final del consumidor promoviendo prácticas, como el consumo responsable.