Facultad de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DEL SOFTWARE

Aplicación móvil para la gestión de listas de la compra entre varios usuarios

Estudiante: Jorge Casal Munín
Dirección: Juan Raposo Santiago

A Coruña, February de 2023.

Agradecimientos

Agradezco a mi familia y amigos por apoyarme en todo momento, ayudarme y estar a mi lado no solo en los buenos momentos si no también en los malos. Agradecer también a mi tutor porque sin su ayuda no podría haber sacado adelante este proyecto.

Resumen

Este proyecto tiene como objetivo desarrollar una aplicación para gestionar cómo realizan la compra varios usuarios, de esta forma se consigue agilizar y automatizar dicho proceso, así como llevar un control de las compras que se han realizado, quién o quiénes han realizado dichas compras, y cómo y cuándo se han realizado las mismas.

Para conseguir esto, cada usuario tiene varias listas de la compra que pueden ser o bien individuales o bien compartidas entre varios usuarios. Estas listas están compuestas a su vez por listas de productos, en las cuales se encuentran los productos que vamos a comprar. Estas listas de productos contienen a uno o varios usuarios, que son los que forman parte de la lista de la compra. Esto lo hacemos así para que en los registros de las compras podamos saber qué productos de los que se han comprado pertenecen a cada persona.

Estos productos se podrán añadir a la lista de productos a través de un buscador, en el que podremos encontrar los productos que estén previamente creados en nuestra base de datos o crear productos personalizados en caso de que no existan. Estos productos se podrán modificar o eliminar de la lista de productos y también marcar como adquiridos. Cuando se marcan como adquiridos, se añaden a una cesta en la que tenemos todos los productos que vamos a comprar. Cuando finalicemos la compra, esta cesta se vaciará y se creará un registro de esta compra. Este registro contendrá los productos que teníamos previamente en la cesta, organizados en las listas de productos que pertenecen a los diferentes usuarios.

Fuera de lo que es la propia gestión de las compras, el usuario contará con dos pantallas extra: una para ver y gestionar las invitaciones a las distintas listas de la compra, y otra para modificar sus datos de usuario.

Abstract

The aim of this project is to develop an application for shopping management, speeding and automatising the process. This app also seeks to keep track of all the purchases made, the people behind those purchases and the date of acquisition of the products.

In order to achieve this, each user has several shopping lists that can be either personal or shared with other users. In turn, these lists consist of products lists, where we will find the products to be acquired. Products lists have one or more users, that also belong in the shopping list. The reason behind this is to make sure we know which of the products acquired belongs to each person.

These products can be added to the product list through a search engine. In this search engine, we will find products previously created in our database, as well as personalised products created by the users, in case they are not in our database. These products can be modified or removed from the product list, or marked as acquired. Once they have been marked as acquired, they are added to a shopping basket where we will find all the products we wish to buy. Once the purchase is finished, this basket will empty and a register of the purchase will be created. In this register, we will find the products that were previously in the shopping basket, organised according to the products lists to which the users belong.

Aside from shopping management, users will have two more screens: one that will allow them to see and manage shopping list invitations, and another that will allow them to change their personal data.

Palabras clave:

- Aplicación móvil
- Firebase
- No-SQL
- Base de datos en tiempo real
- Flutter
- Cloud functions
- JavaScript
- Metodología ágil
- Gestión de productos
- Control de usuarios
- Gestión de listas

Keywords:

- Mobile app
- Firebase
- No-SQL
- Real time database
- Flutter
- Cloud functions
- JavaScript
- Agile methodology
- Product management
- User control
- List management

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Con la llegada a nuestras vidas de las nuevas tecnologías la forma en la que realizamos distintas tareas ha comenzado a cambiar. Tareas que antes se realizaban a mano y requerían un esfuerzo, tiempo y recursos que aunque no fuesen mucho si se hacían muchas veces acababan siendo notables. Hoy en día las podemos hacer en unos pocos minutos y en cualquier lugar, siempre que contemos con un teléfono móvil y conexión a Internet (elementos que hoy en día en nuestro país están prácticamente al alcance de todo el mundo).

Y es que, si antes para realizar un proceso como era hacer la compra, había que apuntar en un papel las cosas que hacía falta comprar, ahora con las nuevas tecnologías, se puede hacer más rápido y sin necesidad de tener a mano un papel para anotar, ni de memorizarlo y que corramos el riesgo de olvidarnos de algo. Además, en caso de que nos hayamos olvidado de apuntar algo en la lista de la compra y solamente nos demos cuenta al llegar a casa, gracias a las aplicaciones de comunicación actuales, si nos olvidamos algo, una persona nos puede enviar un mensaje para recordarnos qué hay que comprar. Asimismo, antes si queríamos saber cuándo y qué producto compramos, necesitábamos guardar el tique (y seguramente una vez tuviéramos unas cuantas compras hechas, perderíamos alguno, además de que llevar un fajo de tiques encima resulta incómodo). En cambio, actualmente, podemos hacer que el móvil lleve por nosotros ese historial con toda la información y no tener que andar nosotros recordando dónde tenemos los tiques y poder dedicar así nuestra cabeza a otras cosas.

Además, cuando se hace una compra en grupo, a la hora de repartir los productos, saber qué producto pertenece a cada persona puede acabar resultando molesto y tedioso. Por razones como estas, surgen las aplicaciones para gestionar listas de la compra que nos ayudan a agilizar todo este proceso y nos permiten que hacer la compra sea una tarea mucho más llevadera.

1.1 Estado del arte

Actualmente existe un amplio abanico de aplicaciones dedicadas a hacer listas, algunas incluso enfocadas específicamente a hacer listas de la compra. Entre las aplicaciones ya existentes podemos destacar SoftList [1], Bring! [2] y Listonic [3]:

- **SoftList**: Es una aplicación especializada en gestionar listas de la compra. Incluye un catálogo de productos y una función de autocompletar para agilizar el proceso de realizar las compras. Además, permite una alta personalización de las listas, ya que es posible añadir precios, unidades de medida, observaciones y fotografías. En caso de añadir los precios, la aplicación calcula el total y mantiene un historial para hacer un análisis de gastos y comparativa de precios (Figura 1.1). Además, cuando separamos los productos en categorías, es capaz de indicarnos en qué categorías estamos gastando más. Lo malo de esta aplicación es que solamente está disponible para Android.

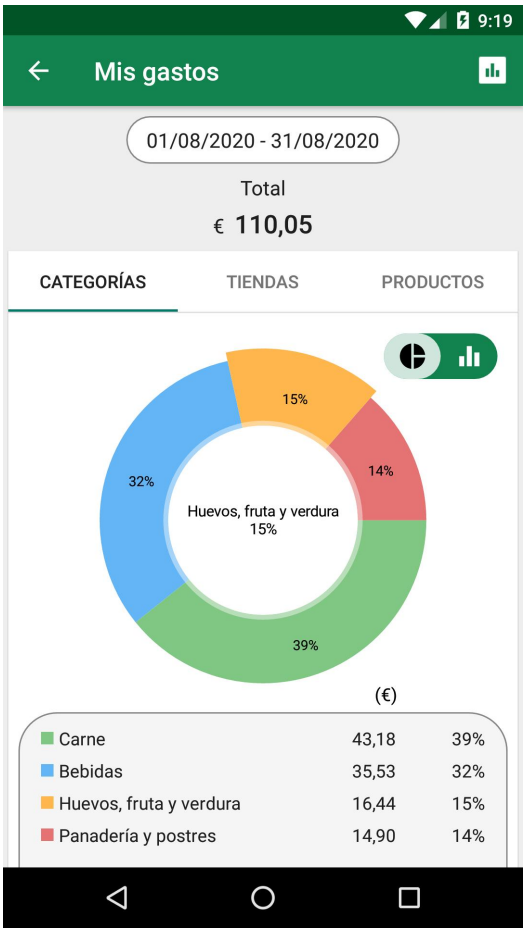

Figura 1.1: Captura de pantalla de la aplicación SoftList.

- **Bring!**: Esta aplicación nos permite crear listas de la compra y compartirlas entre varios usuarios. Las listas de la compra tienen productos, ya sean productos comunes o productos creados por nosotros mismos. Estos productos, una vez se han comprado, se pueden seleccionar y eliminar de la lista, además de poder modificar sus detalles para poner alguna anotación. También nos proporciona una lista de recetas para inspirarnos en caso de que no sepamos qué cocinar (Figura 1.2). Como se puede ver en la imagen, cuando buscamos los productos en una categoría, estos productos se marcan en un color diferente en caso de que ya estén en la lista de la compra.

Figura 1.2: Captura de pantalla de la aplicación Bring!.

- **Listonic**: Listonic nos permite crear listas de la compra que se van guardando en nuestra propia base de datos para, posteriormente, sugerirnos otros productos en función de nuestras compras anteriores. Además, posibilita compartir listas con otros usuarios, tiene control de gastos, nos permite añadir productos por voz y es compatible con Android Wear por lo que es bastante versátil. Una vez marquemos los productos a través de una barra de progreso, se nos indicará cuántos productos llevamos comprados y el precio total de los productos marcados (Figura 1.3). Uno de sus inconvenientes es que no permite la creación de categorías personalizadas.

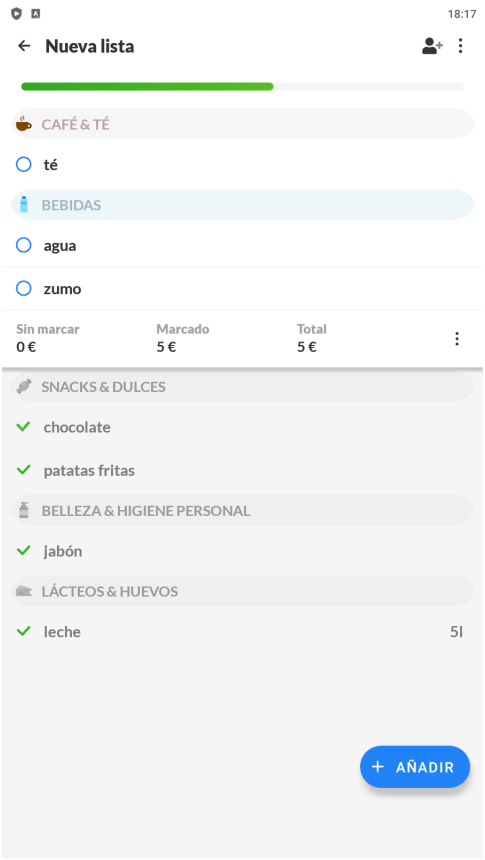

Figura 1.3: Captura de pantalla de la aplicación Listonic.

Como podemos ver, estas aplicaciones disponen de buenas características y un gran potencial. Con todo, somos conscientes de que todas presentan pequeñas carencias. Este es precisamente uno de los objetivos de la aplicación presentada en este trabajo: nuestra intención es conseguir la combinación de estas aplicaciones con el fin de intentar eliminar aquellos pequeños déficits de los que estas adolecen. Para ello, uno de los principales aspectos que distinguen a esta aplicación es la capacidad de compartir las listas entre múltiples usuarios en tiempo real y clasificar los productos añadidos entre estos usuarios.

1.2 Objetivos

El objetivo principal de este proyecto es la creación de una aplicación que permita crear y gestionar listas de la compra de forma colaborativa entre varios usuarios.

Para conseguir esto, cada usuario tiene varias listas de la compra que pueden ser o bien individuales o bien compartidas entre varios usuarios. Estas **listas de la compra** están compuestas a su vez por **listas de productos**. Cada lista de productos contiene productos y un

subconjunto de los usuarios de la lista de la compra. Es decir, una lista de productos representa un conjunto de productos, que quieren comprar un subconjunto de los usuarios de la lista de la compra. Por ejemplo, si varias personas que comparten piso hacen una compra común pero no todos necesitan los mismos productos, cada lista de productos contendría una serie de productos y tendría solamente a los usuarios que quieren adquirir dichos productos.

Cuando creamos una lista de la compra, esta va a tener una lista de productos común, de la que formarán parte todos los usuarios que hayamos invitado a nuestra lista de la compra. El usuario que crea la lista de la compra será su administrador, lo que le otorga el privilegio de eliminar usuarios de la lista de la compra o eliminar la lista de la compra permanentemente. El resto de usuarios no pueden eliminar a otros usuarios de la lista de la compra ni borrarla, pero sí que pueden abandonar esa lista, o invitar a otros usuarios, ya que, si solamente el administrador pudiera invitar a otras personas a la lista, el proceso resultaría tedioso y lento al depender en todo momento del administrador.

Todos los usuarios pueden crear listas de productos. Cualquier usuario puede añadir un producto a cualquier lista de productos sin necesidad de pertenecer a ella, de esta forma si un usuario no pudiera añadir un producto a su lista de productos en un momento dado, otro usuario podría compensar esta necesidad. El motivo para que las listas de productos contengan a los usuarios radica en que posteriormente en el historial de compras, cuando veamos las compras, podremos ver en cada lista los productos que se compraron, así como los usuarios a los que forma parte de esta lista, para así ulteriormente no tener problemas para saber para qué usuarios se compró cada producto a la hora de repartirlos. Ya que, solamente tendríamos que mirar en los registros de las compras, qué productos se compraron, a qué listas de productos pertenecen estos productos y que usuarios contienen estas listas. Asimismo, para facilitar la tarea de añadir productos, en caso de que un usuario tenga una lista de productos en otra lista de la compra y quiera añadir los productos de esa lista a una nueva lista conjunta entre varios usuarios, podrá hacer una copia de esa lista de productos a la nueva lista de la compra. Esta lista de productos contendrá los productos que había en la lista que se ha copiado y contendrá al usuario que la copió.

Los productos de las listas de productos se añaden a través de un buscador, en el que aparecerán divididos por categorías para ver solamente los productos de esa categoría que contengan las letras indicadas. Además, en caso de que el producto no esté previamente registrado en la base de datos, se podrá crear un nuevo producto con los datos que nosotros queramos. Estos productos se podrán modificar posteriormente. A parte de actualizarse, estos productos se pueden añadir a una cesta de la que forman parte todos los productos que marquemos como comprados, o eliminarlos de la lista de productos en caso de que los hayamos añadido por equivocación. Ya que uno de los principales objetivos y motivaciones de la aplicación es que múltiples personas puedan ir a hacer la compra de manera simultánea,

los productos que se vayan añadiendo a la cesta, así como cualquier modificación en las listas debe verse reflejada en tiempo real.

Una vez finalicemos la compra, nos quedará un registro con el nombre de la compra que hayamos indicado, así como la fecha en la que se realizó dicha compra. Este registro estará formado por los productos que estaban en la cesta en el momento de la finalización de la compra. Dichos productos aparecerán clasificados según la lista de productos a la que pertenezcan. Con esto podremos saber a quién pertenecen concretamente los productos que hemos comprado. Asimismo, se podrá modificar su descripción para añadir alguna anotación y su imagen, por si queremos poner una foto del producto concreto que hemos comprado.

1.3 Estructura de la memoria

En este apartado se muestran los capítulos que se encuentran en la memoria con una breve descripción de cada uno de ellos:

- **Introducción**: Se indican los motivos y los objetivos de este proyecto.
- **Tecnologías**: Se comentan todos los lenguajes, frameworks y herramientas utilizadas para el desarrollo del trabajo, así como su justificación.
- **Metodología**: Se expone la metodología que se ha decidido seguir en este proyecto.
- **Análisis**: Se definen los actores, sus roles y las historias de usuarios así como los sprints que se van a utilizar.
- **Diseño**: Se explica la arquitectura que se ha seguido, así como los patrones que se han utilizado y por qué.
- **Implementación**: Se especifica cómo se ha llevado acabo el diseño indicado anteriormente.
- **Pruebas**: Se explican las pruebas que se han realizado.
- **Planificación y evaluación de costes**: Se muestran las fases del proyecto y sus costes.
- **Conclusiones del proyecto**: Se comentan las conclusiones finales y posibles líneas de trabajo futuras.

Capítulo 2

Tecnologías

En este capítulo se presentarán los diversos lenguajes utilizados durante la implementación y desarrollo del proyecto. También se muestran las tecnologías, las APIs y los servicios incluidos en cada parte del trabajo.

2.1 Lenguajes y Frameworks

En esta sección se presentan los lenguajes y framework utilizados en el proyecto.

2.1.1 Dart

Dart [4] es un lenguaje de código abierto desarrollado por Google con el objetivo de permitir a los desarrolladores de software utilizar un lenguaje orientado a objetos y con análisis estático de tipo.

El principal motivo por el que hemos escogido Dart es por su soporte a la programación asíncrona, que es fundamental para la comunicación con Firebase, a través de los tipos de **Future** y **Stream**, mediante los cuales podemos obtener los datos de Firestore sin necesidad de esperar al resto de operaciones y estar conectados en todo momento para poder reaccionar a los cambios en Firestore en tiempo real.

2.1.2 Flutter

Flutter [5] es un framework desarrollado por Google que usa el lenguaje de Dart. Se ha optado por esta opción debido a su desarrollo cómodo, rápido y simple a la hora de desarrollar interfaces, así como a que nos permite desplegar nuestro proyecto en dispositivos Android e iOS sin necesidad de añadir código adicional.

Sus interfaces prediseñadas así como su capacidad de personalización fueron extremadamente útiles a la hora de diseñar la interfaz de una forma rápida y versátil. Además su **Stateful**

Hot Reload, el cual nos permite realizar cambios en nuestro código que se reflejan instantáneamente, aumentó nuestro nivel de productividad al no tener que recargar la aplicación con cada cambio que hagamos.

Cabe destacar también que es un framework con el que ya estábamos algo familiarizados y al ser de código abierto presenta una gran documentación y en nuestro caso nos permitió realizar una fácil integración con Firebase.

2.1.3 JavaScript

Hemos utilizado JavaScript [6] para programar las Cloud Funtions. Se ha escogido este lenguaje porque además de ya estar familiarizados con él nos permite utilizar *event-driven* a partir del cual reaccionamos a los diferentes eventos que suceden en nuestra aplicación, como por ejemplo la creación de un documento, el borrado de este u otras operaciones. Además, es el lenguaje más utilizado en Firebase por lo que la documentación era muy extensa y proporcionaba un gran número de librerías y otras herramientas.

Como entorno de ejecución para JavaScript se ha utilizado **Node.js** [7], ya que al permitirnos manejar un gran número de datos de forma simultánea es útil en nuestra aplicación que necesita manejar datos en tiempo real y reaccionar a eventos.

2.2 Firebase

Firebase [8] es una plataforma de desarrollo de aplicaciones diseñada por Google. En este apartado se muestran las distintas librerías y servicios de Firebase que se han utilizado para desarrollar la aplicación.

2.2.1 Cloud Firestore

Cloud Firestore [9] es una base de datos NoSQL en la nube. Sus datos se almacenan en documentos organizados en colecciones, estos documentos a mayores de campos con datos pueden contener otras subcolecciones. Las principales razones por las que hemos escogido Cloud Firestore son:

- **Actualizaciones en tiempo real**: Firestore usa una sincronización en tiempo real para actualizar sus datos en tiempo real, de manera que cuando realicemos un cambio a un documento de Firestore, este cambio se notificará al instante al resto de dispositivos conectados. Lo cual es muy importante en la aplicación ya que queremos que cuando un usuario añada un producto o modifique una lista se notifique instantáneamente al resto de usuarios sin tener la necesidad de recargar la lista.

- **Compatibilidad**: Presenta una fácil integración con otros SDKs como Java, Python o en nuestro caso Flutter. Además de proporcionar un buen soporte y documentación a estos SDKs lo cual facilita la implementación de la aplicación.

- **Disponibilidad**: Firestore nos permite almacenar nuestros datos de forma local en caché, por lo que en caso de que no tengamos conexión a Internet esos datos se almacenarán en nuestra caché y cuando volvamos a tener conexión se sincronizarán automáticamente a Firestore.

2.2.2 Firebase Auth

Firebase Auth [10] es un servicio de Firebase que permite que los usuarios se autentiquen más fácilmente además de almacenar ciertos datos del usuario como son su email, nombre o foto de perfil. Hemos utilizado Firebase Auth para el servicio de autenticación ya que es la opción óptima de registrar usuarios en Firebase. Además también nos permite conocer datos personales del usuario que está actualmente logueado, evitando así tener que arrastrar variables innecesarias a lo largo de la aplicación, o tener que almacenar estos datos en la base de datos de Firestore y luego leerlos provocando así lecturas y escrituras redundantes.

2.2.3 Firestore security rules

Las Firestore security rules [11] son una serie de condiciones para determinar quién tiene acceso a qué datos. Las hemos utilizado para ayudarnos en la seguridad y evitar que un usuario pueda por ejemplo ver o modificar listas de la compra de las que él no forma parte, o para operaciones que solamente queremos que un usuario con el atributo de administrador pueda acceder.

2.2.4 Cloud Storage for Firebase

Cloud Storage for Firebase [12] es un servicio de almacenamiento de datos. Lo hemos utilizado para almacenar las diferentes imágenes de nuestros productos y las listas de la compra de los usuarios.

2.2.5 Cloud Functions

Las Cloud Functions [13] nos permiten realizar las operaciones del backend en la nube. En nuestro proyecto hemos utilizado las Google Functions para realizar peticiones *https* así como *triggers* que reaccionan a cambios en la base de datos de Cloud Firestore.

2.2.6 Firebase Local Emulator

Firebase Local Emulator [14] es un conjunto de herramientas que nos permiten trabajar con Firebase en un entorno Local sin conexión a Internet. Lo hemos utilizado principalmente a la hora de probar nuestra aplicación en un entorno de desarrollo antes del paso a producción.

2.2.7 Algolia

Es una plataforma que ofrece búsquedas de texto y un sistema de recomendaciones. Firestore no admite la búsqueda de campos de texto completo en los documentos, por lo que para poder conseguir esto hemos utilizado Algolia [15] la cual podemos vincular a nuestra base de datos Firestore a través de una extensión.

2.3 Herramientas de desarrollo

En este apartado mostramos las herramientas que hemos utilizado, desde los IDEs para programar la aplicación hasta las herramientas de control de versiones en donde almacenamos nuestros cambios del proyecto.

2.3.1 Android Studio

Para desarrollar nuestra aplicación hemos utilizado Android Studio [16] ya que es el IDE oficial para Android e incluye extensiones que nos permiten trabajar cómodamente con Flutter y con dispositivos móviles a través de su emulador. Entre las ventajas que nos proporciona cabe destacar su rápida compilación y no tener que parar y volver a ejecutar la aplicación con cada cambio que realicemos en nuestro código.

2.3.2 Visual Studio Code

Visual Studio Code [17] es el editor de código que hemos utilizado para programar las Cloud Functions necesarias para la parte del backend de nuestra aplicación. Hemos escogido esta opción debido a la facilidad de programar JavaScript (nuestras Cloud Functions están programadas en JavaScript), además de su extensa cantidad de extensiones, entre las cuales cabe destacar la extensión de Firebase, la cual agilizó el proceso de escritura de código. También nos fue de ayuda a la hora de debuggear las Cloud Functions usando el Firebase Local Emulator.

2.3.3 GitHub

GitHub [18] es un servicio basado en la nube que aloja un sistema de control de versiones llamado Git. Hemos utilizado esta opción ya que es la herramienta de control de versiones

que más hemos utilizado a lo largo de la carrera, debido a su facilidad a la hora de compartir el código, así como la posibilidad de revertir los cambios y volver a una versión previa del código en caso de que este ya no funcione. Además facilita la división de las funcionalidades y la organización del proyecto a través de la creación de ramas.

2.3.4 Firebase Console

Firebase Console [19] es una herramienta de desarrollo de Firebase que nos proporciona una web la cual hemos utilizado, para controlar los diferentes servicios de Firebase, desde Firestore, las Cloud Functions, el Cloud Storage, etc. Y comprobar así que podíamos acceder a estos servicios comprobando que las lecturas y escrituras en Firestore eran correctas, que teníamos las colecciones adecuadas creadas en Firestore o las Cloud Functions eran las que habíamos creado y se lanzaban cuando nosotros queríamos para evitar un mal uso que nos podría conllevar a un mal funcionamiento de la aplicación y a un costo monetario significante.

2.4 Otros servicios

Aquí se muestran los servicios que se han utilizado para el manejo de información, diseño de diagramas, wireframes, etc.

2.4.1 Google Drive

Se ha utilizado Google Drive [20]para almacenar copias los diferentes archivos, como los diagramas, la memoria, los wireframes, guardar notas de la aplicación, etc. Hemos decidido utilizar esta opción ya que nos permite acceder desde múltiples dispositivos y nos permite acceder a versiones antiguas de los documentos evitando así problemas de modificar un archivo y no poder volver hacia atrás.

2.4.2 Overleaf

Es un editor de texto LaTeX online con entorno web que permite trabajar de forma colaborativa. Lo hemos utilizado para redactar la memoria. Hemos escogido esta opción por su alta personalización y que nos permite que más de una persona acceda y modifique el proyecto. [21]

2.4.3 Balsamiq

Esta herramienta la hemos utilizado para desarrollar los wireframes de la aplicación. [22]

Capítulo 3

Metodología

Una metodología es un conjunto de técnicas y métodos que nos permite plantear de forma uniforme las actividades del ciclo de vida del proyecto. Una metodología comprende los procesos a seguir para diseñar, implementar y mantener un producto software desde que surge la necesidad del producto hasta que se cumple el objetivo por el cual fue creado.

3.1 Metodologías ágiles

Las metodologías ágiles son aquellas que permiten adaptar la forma del trabajo a las condiciones del proyecto consiguiendo flexibilidad e inmediatez para amoldar el proyecto y su desarrollo a las circunstancias específicas del entorno. Un modelo de desarrollo ágil generalmente es un proceso **incremental** (entregas frecuentes con ciclos rápidos), **cooperativo** (constante comunicación entre clientes y desarrolladores), **sencillo** (el método es fácil de aprender y modificar para el equipo) y **adaptativo** (se pueden hacer cambios en el último momento). Las metodologías ágiles proporcionan una serie de pautas y principios que hacen que la entrega del proyecto sea menos complicada y más satisfactoria tanto para clientes como para los equipos de trabajo, evitando los caminos burocráticos de las metodologías tradicionales, generando poca documentación y no haciendo uso de métodos formales. Algunas de las ventajas de las metodologías ágiles son:

- **Mayor satisfacción del cliente**: el cliente está más satisfecho al estar involucrado directamente y poder opinar durante el proceso de desarrollo a través de las diferentes reuniones.
- **Mayor motivación de los trabajadores**: los equipos de trabajo autogestionados, facilitan el desarrollo de la capacidad creativa y de innovación entre sus miembros
- **Trabajo colaborativo**: dividir el trabajo en equipos y roles además de las reuniones frecuentes facilita la organización del trabajo.

- **Uso de métricas más relevantes**: las métricas utilizadas para estimar parámetros como tiempo, coste, rendimiento, etc. son normalmente más reales en proyectos ágiles que en los tradicionales. Además al dividir el proyecto en pequeños equipos y fases podemos ser más conscientes de lo que está sucediendo.

- **Mayor control y capacidad de predicción**: la oportunidad de revisar y adaptar el producto a lo largo del proceso ágil, permite a todos los miembros del proyecto ejercer un mayor control sobre su trabajo, cosa que permite mejorar la capacidad de predicción en tiempo y costes.

- **Reducción de costes**: al llevar todo el proyecto al día los errores se identifican y corrigen durante el desarrollo en vez de al final cuando el producto esté finalizado y la inversión ya está realizada.

En nuestro proyecto las metodologías ágiles planteadas fueron **Scrum** y **Kanban**

3.2 Scrum

Se basa en una estructura de desarrollo incremental. Las interacciones del proceso se conocen como **Sprints**, es decir las diferentes entregas regulares y parciales del producto final. Permite abordar proyectos complejos desarrollados en entornos dinámicos y cambiantes de forma flexible. En Scrum se aplican de forma regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible. Estas prácticas se ayudan unas a otras y su origen tiene lugar en estudios sobre como trabajan otros equipos altamente productivos. En Scrum se realizan entregas parciales las cuales se priorizan en función del beneficio al proyecto. Por eso Scrum es indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o no están correctamente definidos al principio, y en donde la innovación, la competitividad, la flexibilidad y la producción son primordiales.

Los fundamentos de Scrum son:

- El desarrollo incremental de los requisitos del proyecto en bloques temporales cortos y fijos.

- La priorización de los requisitos en función del valor que presenten para el cliente y el coste en cada iteración.

- El control del proyecto a partir de la experiencia y de la observación de como suceden los cambios en el proyecto.

- La potenciación del equipo al comprometerse a entregar unos requisitos y por lo tanto se le otorga la autorizad necesaria para realizar el trabajo.
- La sistematización de la colaboración y la comunicación tanto entre el equipo como con el cliente.
- El timeboxing (establecer un cantidad estimada de tiempo) de las actividades del proyecto, para ayudar a la toma de decisiones y conseguir resultados.

En Scrum destacamos 3 roles, estos son:

- **Product Owner**: es el encargado de decidir los objetivos que se van a realizar y maximizar el valor del producto, mediante el uso del Product Backlog, esto es el listado de tareas que se pretende hacer durante el proyecto. Además, es el encargado de comunicarse con los **stakeholders** (las personas u organizaciones que tienen una relación directa o indirecta con el proyecto) y el cliente.
- **Scrum Master**: es la persona que lidera al equipo intentando cumplir una serie de buenas prácticas y eliminando los impedimentos que puedan afectar a la entrega del producto. Además, se encarga de las tareas de formación y de facilitar reuniones y eventos en caso de ser necesario.
- **Equipo de desarrollo**: son los encargados de desarrollar el producto, auto-organizándose y auto-gestionándose para conseguir entregar un incremento del software al final del equipo de desarrollo.

3.3 Kanban

Kanban se basa en la idea de que el trabajo en curso debería limitarse, y solo deberíamos empezar con algo nuevo cuando hemos terminado el trabajo previo. En la metodología Kanban se emplea un mecanismo de control visual para hacer el seguimiento del trabajo. Normalmente, se emplea una pizarra con notas adhesivas o un panel electrónico de tarjetas para gestionar el flujo de trabajo y las asignaciones.

El aporte principal de Kanban a las metodologías ágiles es que a través de los mecanismos visuales se consigue una mayor transparencia del proceso, ya que su flujo de trabajo expone la variabilidad, los desperdicios a lo largo del tiempo y todas las cosas que afectan al rendimiento de la organización en términos de cantidad de trabajo entregado y el tiempo requerido para realizarlo. La visibilidad de estos problemas también promueve la discusión sobre las posibles mejoras, y hace que los equipos de trabajo comiencen rápidamente a implementar mejoras en su proceso.

Los principios del método Kanban son:

- **Calidad garantizada**: todo lo que se hace debe salir bien a la primera, no hay margen de error. Para conseguir esto no se premia la rapidez, sino la calidad final de las tareas realizadas.
- **Reducción del desperdicio**: se basa en hacer solamente lo justo y necesario, para garantizar que lo que se haga se haga bien. Esto implica la reducción del trabajo superficial o secundario.
- **Mejora continua**: se establece perseguir el cambio incremental y evolutivo. No es simplemente un método de gestión, sino también un sistema de mejora en el desarrollo de proyectos según los objetivos a conseguir.
- **Flexibilidad**: es necesario ser capaz de priorizar las tareas según las necesidades que tengamos en cada momento y tener la capacidad de dar respuesta a estas tareas imprevistas.

3.4 Metodología escogida en el proyecto

En nuestro caso hemos escogido **Scrum** con algunos detalles de **Kanban** como metodologías a aplicar en nuestro proyecto. Al ser un proyecto realizado por una única persona hay ciertos puntos de las metodología ágiles que no hemos podido aplicar como son las prácticas colaborativas entre varios miembros del equipo o la comunicación con el cliente, sin embargo si que se han hecho reuniones con el director del proyecto en las que se han definido los objetivos del proyecto, así como un análisis y diseño de este. Además, la parte del cliente se ha auto asumido estableciendo nosotros mismos los objetivos del proyecto así como determinando al final de cada Sprint como iban avanzando esos objetivos si se habían completado o cosas a modificar.

El proyecto se ha dividido en diferentes Sprints, en cada Sprint se han realizado las fases de un ciclo de vida tradicional: análisis, diseño, implementación y pruebas. Al final de cada Sprint se han revisado las historias de usuario (una explicación general de una función en la que se describe el resultado deseado) para ver si hemos cumplido con el esfuerzo estimado y ver si el proyecto iba avanzando al ritmo que nosotros esperábamos. Uno de los principales motivos por los que hemos escogido una metodología ágil sobre una tradicional es la capacidad de cambiar el proyecto en función de cambios posteriores. Esto se debe a que los requisitos fueron cambiados a medida que realizábamos el proyecto debido al uso de nuevas tecnologías que no se habían contemplado previamente, o a un nuevo enfoque del proyecto sobre como trabajar con Firebase, ya que al ser relativamente reciente esta sujeto a cambios constantemente.

Aunque ha habido tareas que hemos tenido que revisar y rehacer, si que hemos seguido la parte Kanban en la que no comenzamos con una tarea hasta haber terminado con la actual,

además, hemos ido priorizando las tareas en función de las necesidades que tuviésemos en ese momento. Como por ejemplo, lo primero que priorizamos al implementar la aplicación fue la creación de los usuarios, ya que queríamos que cuando creáramos las listas de la compra estas ya estuvieran asignadas a unos usuarios. También nos hemos concentrado en hacer solamente lo justo y necesario en algunos casos de uso para evitar que las historias se alarguen demasiado y que no cumplan los plazos estimados en el Sprint. Por ejemplo, en el caso de crear la lista de la compra esta va a tener una imagen, pero para eso vamos a necesitar usar el servicio de Cloud Storage. Para evitar mayores complicaciones en esta historia hemos decidido, crear un Sprint a parte en el que trabajamos lo relacionado a Cloud Storage.

En lo referente a los roles de Scrum. El **product owner** se ha asignado tanto al director como al alumno, ya que los objetivos se han marcado a través de reuniones al inicio del proyecto. El rol de **Scrum master** lo ha asumido el director del proyecto ya que ha sido el que ha ido guiando al alumno en las diferentes pautas a seguir a la hora de realizar el proyecto, así como la gestión de las reuniones. Por último el alumno se ha hecho cargo del rol de **equipo de desarrollo** ya que ha sido el que ha ido implementando el proyecto y ha respondido a los diferentes cambios que han sucedido en este, intentando auto-organizarse y auto-gestionarse de la mejor forma posible.

Capítulo 4

Análisis

En este capítulo hablaremos sobre los actores que hemos definido en nuestra aplicación, así como las historias de usuarios y los Sprints que engloban estas historias.

4.1 Actores

En esta aplicación distinguimos dos actores principales:

- **Usuario**: cualquier persona que se haya registrado en la aplicación. Este actor podrá crear listas de la compra, aceptar invitaciones a otras listas de la compra y modificar los datos de su cuenta. Hay dos tipos de usuario:
  - **Usuario invitado**: el usuario que ha sido invitado a la lista de la compra por un administrador y ha aceptado dicha invitación. Este actor puede crear o eliminar listas de productos. También puede añadir productos a dichas listas (estos productos pueden estar ya registrados en nuestra base de datos o pueden ser productos que cree el propio usuario). Asimismo, podrá invitar a otros usuarios a unirse a la lista de la compra, así como modificar los usuarios que forman parte de una lista de productos (estos usuarios tienen que formar parte previamente de la lista de la compra). También es capaz de marcar los productos como adquiridos y añadirlos a la cesta, eliminar dichos productos o modificarlos para concretar sus detalles. Además, puede terminar la compra cuando lo considere oportuno y consultar el historial de dicha lista de la compra.
  - **Usuario administrador**: el usuario que ha creado la lista de la compra. Aparte de poder realizar todas las funciones que realiza un usuario invitado, este es capaz de eliminar de forma permanente la lista de la compra, a diferencia de los invitados, que solamente pueden abandonar dicha lista. Además, el administrador puede eliminar a los usuarios de la lista de la compra.

Cualquier usuario invitado puede ser administrador y viceversa. Esto se debe a que cualquier usuario puede crear sus propias listas de la compra y, asimismo, pertenecer a las listas de la compra de otros usuarios. Sin embargo, una lista de la compra solo puede tener a un usuario administrador.

- **Administrador de la aplicación**: el actor encargado de añadir productos comunes a la base de datos, de forma que estos productos sean accesibles para todos los usuarios. Por el momento, este actor no forma parte de las historias actuales, pero está prevista su incorporación en trabajos futuros cuando se elabore una web de administración. En estos momentos, la incorporación y eliminación de productos se lleva a cabo directamente desde la consola de Firebase.

4.2 Historias de usuario

En este apartado se indican las diferentes historias de usuario asociadas a los casos de uso de la aplicación. Cada historia tiene un identificador, un nombre y un valor, este valor nos indica la importancia que tiene esa funcionalidad dentro de nuestra aplicación. Estas historias se han dividido en diferentes Sprints según sus funcionalidades. A excepción de las historias relacionadas con expulsar usuarios de la lista de la compra, eliminar la lista de la compra y abandonar la lista de la compra, que dependen de si el usuario es el administrador o el invitado de la lista de la compra, el resto de las historias son comunes para ambos usuarios, por lo tanto, el rol indicado para estas historias será usuario sin diferenciar si es invitado o administrador.

ID	Nombre	Valor
1	Como usuario, quiero poder registrarme en la aplicación	30
2	Como usuario, quiero poder iniciar sesión en la aplicación	30
3	Como usuario, quiero poder cerrar mi sesión	5
4	Como usuario, quiero poder actualizar mis datos de usuario	15
5	Como usuario, quiero ser capaz de verificar mi correo electrónico	10
6	Como usuario, quiero poder reiniciar mi contraseña en caso de haberme olvidado de ella	5
7	Como usuario, quiero ser capaz de crear mis listas de la compra	40
8	Como usuario administrador, quiero poder eliminar mis listas de la compra	10
9	Como usuario, quiero poder obtener las listas de la compra de las que formo parte	40
10	Como usuario, quiero poder crear listas de productos dentro de una lista de la compra	30
		215

Tabla 4.1: Historias de usuario primera parte

ID	Nombre	Valor
11	Como usuario, quiero poder obtener los productos de una lista de productos	30
12	Como usuario, quiero ser capaz de obtener los productos de la cesta	20
13	Como usuario, quiero poder añadir un producto a la lista de productos	25
14	Como usuario, quiero poder marcar un producto como adquirido, para así añadirlo a la cesta	20
15	Como usuario, quiero ser capaz de finalizar la compra	20
16	Como usuario, quiero ser capaz de consultar el historial	20
17	Como usuario, quiero ser capaz de crear mis propios productos	25
18	Como usuario, quiero poder actualizar un producto que se encuentre en una lista de productos	10
19	Como usuario, quiero ser capaz de eliminar los productos de una lista de productos	15
20	Como usuario, quiero ser capaz de buscar los productos por nombre	15
21	Como usuario, quiero poder desmarcar los productos como adquiridos	20
22	Como usuario, quiero poder filtrar los productos por categoría	15
23	Como usuario invitado, quiero ser capaz de abandonar una lista de la compra de la que no soy administrador	15
24	Como usuario, quiero ser capaz de ver los detalles de una entrada del historial	15
25	Como usuario, quiero ser capaz de poder obtener las imágenes de mis listas de la compra	10
26	Como usuario, quiero ser capaz de obtener la imagen de un producto	15
27	Como usuario, quiero ser capaz de consultar mis detalles de usuario	15
28	Como usuario, quiero ser capaz de actualizar los detalles de un producto en el historial	10
29	Como usuario, quiero poder eliminar una lista de productos	10
30	Como usuario, quiero poder importar una lista de productos	20
31	Como usuario, quiero poder invitar a otros usuarios a unirse a la lista de la compra de la cual formo parte	25
32	Como usuario, quiero ser capaz de gestionar mis invitaciones a listas de la compra	20
33	Como usuario, quiero poder ver mis invitaciones a listas de la compra	20
34	Como usuario, quiero ser capaz de gestionar a los usuarios que forman parte de una lista de productos	20
35	Como usuario, quiero ser capaz de obtener los usuarios que pertenecen a una lista de productos	20
36	Como usuario administrador, quiero ser capaz de expulsar a los usuarios invitados que forman parte de mi lista de la compra	10
		460

Tabla 4.2: Historias de usuario segunda parte

4.3 Sprints

El proyecto se ha dividido en 10 Sprints. En general, para cada Sprint, hemos estimado una duración de 2-3 semanas dependiendo del número de historias con las que cuente y de la complejidad de las mismas. Con todo, los tres primeros Sprints presentan una duración superior que alcanza las 4 semanas. Esto se debe a que en estos primeros Sprints estamos empezando a trabajar con Firebase, por lo que al inicio no tenemos tanta soltura como en Sprints posteriores donde ya hemos llevado a cabo varias historias relacionadas con Firebase y conocemos mejor esta plataforma. Además, estos Sprints cuentan con un mayor número de historias que el resto.

Los Sprints se han agrupado según sus funcionalidades relacionadas. A la hora de establecer el orden los Sprints, se ha dado prioridad a aquellos que tenían historias de las cuales dependerían otros Sprints en un futuro. Por ejemplo, añadir productos a las listas de productos es una tarea muy importante, ya que no tendría sentido tener una aplicación sobre gestión de compras sin productos que comprar. Sin embargo, para poder añadir un producto, necesitamos haber creado previamente una lista de productos a la que añadir dichos productos, así como una lista de la compra a la que pertenezca esta lista de productos.

4.3.1 Sprint 0

En este Sprint se han realizado las principales tareas de usuarios. Este es fundamental ya que queremos que, en las futuras operaciones de la aplicación, las mismas sean llevadas a cabo por un usuario que ya se haya registrado y logueado. Para realizar estas historias es clave el uso del servicio Firebase Auth. Las historias de este Sprint son las siguientes:

ID	Nombre	Valor
1	Como usuario, quiero poder registrarme en la aplicación	30
2	Como usuario, quiero poder iniciar sesión en la aplicación	30
3	Como usuario, quiero poder cerrar mi sesión	5
5	Como usuario, quiero ser capaz de verificar mi correo electrónico	10
		75

Tabla 4.3: Sprint 0

En este caso, aunque no se le dé mucho valor, la historia número 2 es importante para evitar problemas de seguridad y asegurarnos de que ese correo existe.

4.3.2 Sprint 1

Este Sprint se corresponde con las principales tareas de creación de la lista de la compra. Esta parte es fundamental, ya que la aplicación gira en torno a las listas de la compra, donde se almacena todo nuestro contenido. En este caso, necesitamos estar logueados previamente, dado que la primera tarea que se debe llevar a cabo es la asignación de la lista de la compra al usuario que la ha creado. De esta forma, a la hora de obtener las listas de la compra de un usuario, solamente tenemos que filtrar las listas de la compra por el identificador del usuario al que pertenecen, evitando así ver y acceder a aquellas listas en las que el usuario no está involucrado.

ID	Nombre	Valor
7	Como usuario, quiero ser capaz de crear mis listas de la compra	40
8	Como usuario administrador, quiero poder eliminar mis listas de la compra	10
9	Como usuario, quiero poder obtener las listas de la compra de las que formo parte	40
		90

Tabla 4.4: Sprint 1

4.3.3 Sprint 2

Este Sprint es el que más historias contiene debido a que es el más importante del proyecto. Esto se debe a que, si en una aplicación sobre gestión de listas de la compra no contamos con productos y solamente tenemos listas, la aplicación no tendría ningún uso, es más, sería mejor simplemente una aplicación en la que apareciesen productos pero no listas que clasificasen dichos productos. Con todo, a pesar de tratarse del Sprint más importante de todo el proyecto, hemos decidido que este ocupe la tercera posición porque queríamos que los usuarios tuviesen que iniciar sesión previamente para poder manejar las listas. Además, como estas listas de productos forman parte de listas de la compra, carecería de sentido crear las listas de productos antes que las listas de la compra. En este Sprint se abordan las historias que involucran tanto la creación y supresión de las listas de productos, como la adición y eliminación de productos en estas listas. Además, también está presente la historia que permite obtener los productos de una lista de productos para comprobar de esta forma que los productos se añaden y visualizan correctamente.

ID	Nombre	Valor
10	Como usuario, quiero poder crear listas de productos dentro de una lista de la compra	30
29	Como usuario, quiero poder eliminar una lista de productos	10
11	Como usuario, quiero poder obtener los productos de una lista de productos	30
13	Como usuario, quiero poder añadir un producto a la lista de productos	25
19	Como usuario, quiero ser capaz de eliminar los productos de una lista de productos	15
20	Como usuario, quiero ser capaz de buscar los productos por nombre	15
30	Como usuario, quiero poder importar una lista de productos	20
		145

Tabla 4.5: Sprint 2

Para poder añadir los productos es necesario haberlos buscado previamente en un buscador en donde estén los productos de nuestra base de datos. En cuanto a la historia de relacionada con la importación de una lista de productos, esta se encuentra en este Sprint debido a que esta tarea se basa en crear una copia de la lista de productos seleccionada en otra lista de la compra.

4.3.4 Sprint 3

El objetivo de este Sprint es garantizar la realización de todas las historias relacionadas con la cesta, a saber: obtener los productos que contiene la cesta, marcar los productos de las listas de productos como adquiridos para así añadirlos a la cesta y desmarcar estos productos para devolverlos a su lista original. La cesta ya se ha creado previamente cuando creamos la lista de la compra. Al igual que sucedía con los Sprints anteriores, para poder realizar este Sprint es necesario haber llevado a cabo un Sprint en el que creamos listas de productos de las cuales obtenemos los productos que se van a añadir a la cesta. Asimismo, es a estas listas de productos a donde devolvemos los productos de la cesta en caso de desmarcarlos como adquiridos.

ID	Nombre	Valor
12	Como usuario, quiero ser capaz de obtener los productos de la cesta	20
14	Como usuario, quiero poder marcar un producto como adquirido, para así añadirlo a la cesta	20
21	Como usuario, quiero poder desmarcar los productos como adquiridos	20
		60

Tabla 4.6: Sprint 3

4.3.5 Sprint 4

En este Sprint se tratan las partes de *finalizar la compra*, *visualizar el historial* y *ver detalles del historial*. Estas tres tareas se realizan de forma conjunta, ya que todas están relacionadas entre sí. De esta forma, una vez finalicemos la compra, debemos poder ver si en el historial se ha creado una nueva entrada y si podemos acceder a esta entrada para comprobar que los productos se han añadido correctamente.

ID	Nombre	Valor
15	Como usuario, quiero ser capaz de finalizar la compra	20
16	Como usuario, quiero ser capaz de consultar el historial	20
24	Como usuario, quiero ser capaz de ver los detalles de una entrada del historial	15
		55

Tabla 4.7: Sprint 4

4.3.6 Sprint 5

Este Sprint contiene menos tareas debido a que trabajaremos con un nuevo servicio, Cloud Storage for Firebase, el cual utilizaremos para almacenar y obtener las imágenes de nuestra aplicación. Consideramos que manejar este servicio puede acarrear una serie de dificultades, ya que está relacionado con la manera en que hemos creado la lista de la compra y cómo se almacenan los productos en nuestra base de datos. Así, esto podría resultar en la modificación de alguna historia previa. Por lo tanto, hemos decidido definir estas historias en un Sprint aparte.

ID	Nombre	Valor
25	Como usuario, quiero ser capaz de poder obtener las imágenes de mis listas de la compra	10
26	Como usuario, quiero ser capaz de obtener la imagen de un producto	15
		25

Tabla 4.8: Sprint 5

No se le ha dado un valor muy alto a *obtener imágenes* ya que, si los productos o las listas no tuvieran imágenes, la aplicación seguiría funcionando igual, simplemente podría resultar un poco menos atractiva a nivel visual.

4.3.7 Sprint 6

El objetivo de este Sprint es ocuparnos de las tareas de creación de nuevos productos que no estaban definidos en nuestra base de datos previamente. También nos permite actualizar los

productos. Estos productos pueden haber sido creados por nosotros mismos o existir previamente en la propia base de datos. Hay que tener en cuenta que cuando añadimos un producto que ya existe en la base de datos de Firestore a una lista de productos, se crea una copia de ese producto en nuestra lista de productos. Por lo tanto, cuando modificamos el producto en la lista de productos no se modifica el producto que ya existe en Firestore, sino la copia que hemos creado en la lista de productos. Lo mismo sucede con los productos del historial cuando los actualizamos, ese producto es una copia de los productos ya existentes, por lo tanto modificamos la copia y no el original.

ID	Nombre	Valor
17	Como usuario, quiero ser capaz de crear mis propios productos	25
18	Como usuario, quiero poder actualizar un producto que se encuentre en una lista de productos	10
28	Como usuario, quiero ser capaz de actualizar los detalles de un producto en el historial	10
		45

Tabla 4.9: Sprint 6

4.3.8 Sprint 7

En esta iteración se trabaja todo lo relativo a la gestión de usuarios de nuestras listas de la compra, ya que no debemos olvidar que nuestras listas de la compra tienen usuarios. Así, en este Sprint, se tratan las partes de invitar a los usuarios a nuestra lista de la compra, que los usuarios puedan tener una pantalla en la que puedan ver sus invitaciones, así como que puedan gestionar dichas invitaciones, es decir, decidir si las aceptan o las rechazan.

ID	Nombre	Valor
31	Como usuario, quiero poder invitar a otros usuarios a unirse a la lista de la compra de la cual formo parte	25
32	Como usuario, quiero ser capaz de gestionar mis invitaciones a listas de la compra	20
33	Como usuario, quiero poder ver mis invitaciones a listas de la compra	20
		65

Tabla 4.10: Sprint 7

4.3.9 Sprint 8

En este Sprint, como ya hemos abordado el tema de los usuarios en las listas de la compra en el Sprint anterior, tratamos la parte del manejo de usuarios en las listas de productos que forman parte de la lista de la compra. Para ello, afrontamos las historias de cómo obtener los usuarios que pertenecen a una lista de la compra y cómo ver a los usuarios que forman parte de una lista de productos. En caso de ser administrador, también hemos contemplado el caso de uso de expulsar a usuarios. Por otro lado, en caso de no ser administrador, un usuario debe poder abandonar la lista de la compra.

ID	Nombre	Valor
23	Como usuario invitado, quiero ser capaz de abandonar una lista de la	15
34	Como usuario, quiero ser capaz de gestionar a los usuarios que forman parte de una lista de productos	20
35	Como usuario, quiero ser capaz de obtener los usuarios que pertenecen a una lista de productos	20
36	Como usuario administrador, quiero ser capaz de expulsar a los usuarios invitados que forman parte de mi lista de la compra	10
		65

Tabla 4.11: Sprint 8

4.3.10 Sprint 9

En este Sprint se tratan las historias de las que no dependen otras historias. En primer lugar, tenemos la actualización de los datos de los usuarios. De igual modo, para ello, necesitamos también poder obtener los detalles de usuario. Asimismo, en este Sprint también encontramos qué hacer en caso de que el usuario olvide su contraseña. Por último, abordamos el tema de cómo filtrar los productos por categoría a la hora de añadirlos a una lista de productos.

Estas historias no se han tratado en el primer Sprint, ya que no consideramos que sean fundamentales para la aplicación, ni que de ellas dependan otras historias. Además, para poder obtener los detalles de un usuario se necesita acceso a su foto de perfil, por lo que necesitaríamos haber completado primero el Sprint 5, en el cual trabajamos con el servicio de Cloud Storage. Del mismo modo, para proceder al filtrado de productos es necesario haber pasado antes por el Sprint 2.

ID	Nombre	Valor
4	Como usuario, quiero poder actualizar mis datos de usuario	15
27	Como usuario, quiero ser capaz de consultar mis detalles de usuario	15
22	Como usuario, quiero poder filtrar los productos por categoría	15
6	Como usuario, quiero poder reiniciar mi contraseña en caso de haberme olvidado de ella	5
		50

Tabla 4.12: Sprint 9

Así, para no añadir estas historias a los Sprints mencionados anteriormente haciéndolos más complicados y largos se ha decidido agrupar estas historias en un Sprint final.

Capítulo 5

Diseño

En este capítulo hablamos sobre la arquitectura de la aplicación y sobre los patrones utilizados. En nuestro caso, nuestra arquitectura se basa principalmente en el patrón **BLoC** [23]. Además, también comentamos los diferentes diagramas realizados para nuestra aplicación, así como la estructura de nuestra base de datos Cloud Firestore.

5.1 Patrón BLoC

El patrón BLoC nos permite separar la capa de lógica de negocio de la capa de presentación de la capa de acceso a datos. BLoC significa Business Logic Component. Los dos conceptos más importantes del patrón BLoC son el **estado** y los **eventos**. Los principales objetivos de este patrón son:

- **Centralizar la lógica de negocio**. La idea es extraer todo lo que no es la renderización de la vista, como por ejemplo la lógica de negocio, en unas clases llamadas BLoC. De esta forma, si hacemos que una clase BLoC solamente almacene la lógica, facilitaremos la implementación de futuros cambios o el uso de diferentes tecnologías.

- **Centralizar cambios de estado**. Las clases BLoC van a centralizar los cambios de estado. Estas clases se encargarán de recibir las acciones o eventos que se producen en la aplicación y que modifican el estado. Con este patrón conseguimos, además, que solamente tengamos que renderizar los widgets suscritos a determinados cambios de estado en BLoC, en vez de tener que renderizar todos los widgets descendientes.

- **Mapear al formato que necesita la vista**. Los BLoCs también se van a encargar de formatear los datos de la manera en que lo necesitan las vistas. De esta forma, esta lógica de presentación de formateo es también reutilizable.

Para modificar el estado, primero manejamos los eventos en el widget y, a continuación, en

función del evento que recibamos, modificaremos el estado de diferente manera en el propio BLoC.

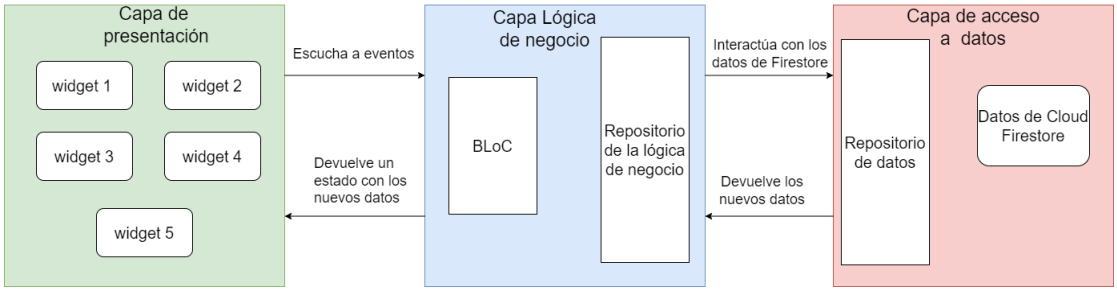

Figura 5.1: Capas del BLoC.

5.2 Patrón BLoC en nuestro proyecto

Nuestra aplicación se divide en 3 capas (Figura 5.1) :

- **Capa de presentación**: esta capa está conformada por los widgets que puede ver el usuario en la interfaz. Cuando el usuario interactúa con estos widgets se genera un evento que manejará el BLoC de la capa lógica de negocio.
- **Capa lógica de negocio**: está formada por los BLoCs y los distintos repositorios que tenemos. Los BLoCs lo que hacen es capturar los eventos que se producen cuando el usuario interactúa con los widgets. El BLoC llamará a un método u otro del repositorio dependiendo del evento obtenido. Este repositorio será el que haga la lógica de la aplicación y se encargue de llamar a la capa de acceso a datos.
- **Capa de acceso a datos**: esta capa está compuesta por un repositorio y los datos que se almacenan en Firestore. El repositorio lo que hace es, captura una petición que le ha enviado el repositorio de la capa lógica de negocio, el repositorio de la capa de acceso a datos interactúa con los datos que tenemos en Firestore (colecciones y documentos) y devuelve los datos solicitados. A partir de aquí el repositorio de la capa de acceso a datos envía los datos al repositorio de la capa lógica de negocio. Este repositorio se comunica con el BLoC que devuelve un estado a la capa de presentación, a partir de los valores de este estado se actualizarán los widgets.

Los principales pasos que realizamos en nuestra aplicación son:

1. La capa de presentación activa un evento cuando se interactúa con un widget.
2. El BLoC recibe y maneja este evento.
3. El BLoC llama a un método en el repositorio de la lógica de negocio para realizar una operación.
4. El repositorio de la lógica de negocio llama a un método del repositorio de datos, en nuestro caso es Firestore. Este método puede ser por ejemplo hacer un update o un get.
5. El repositorio de datos se comunica con los datos para llevar a cabo la operación.
6. Devolvemos los datos en caso de ser necesario al repositorio de la lógica de negocio.
7. Este último repositorio devuelve el resultado al BLoC.
8. El BLoC actualiza el estado con los nuevos datos y se lo envía a la capa de presentación para que actualice la interfaz en caso de ser necesario.

A la hora de como realizamos las operaciones en nuestro proyecto, cabe diferenciar entre las operaciones de lectura y las de escritura. Al trabajar con Firestore y la comunicación asíncrona en este proyecto, estaremos operando con dos tipos de datos los **Stream** y los **Future**.

Operaciones de lectura

Los Stream serán empleados en operaciones de lectura y muestrearán los cambios reflejados en la pantalla de forma que tengamos un evento que escuche los cambios en nuestra aplicación y, en función de esos cambios, nos devuelva solamente los datos que se han modificado. Por ejemplo, a la hora obtener las listas de productos de una lista de la compra, lo que tendremos será un Stream que nos devuelva esas listas de productos. Y, si otro usuario crea una nueva lista de productos, el Stream nos notificaría los cambios y Firestore nos devolvería solamente la lista que se ha añadido para evitar así lecturas innecesarias, al no tener que leer de nuevo todas las listas de productos ya existentes en esa lista de productos.

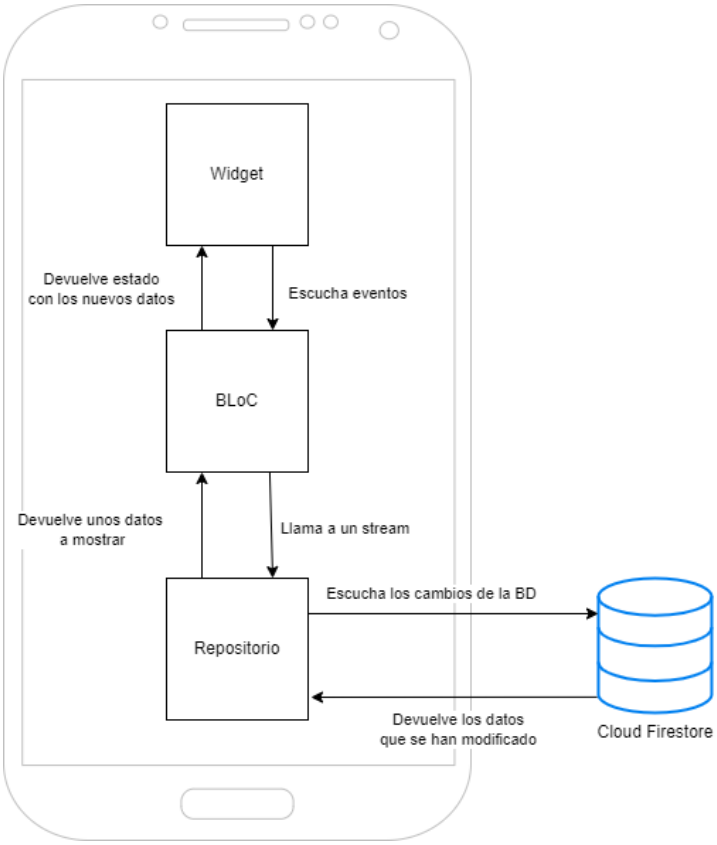

Figura 5.2: BLoC lectura.

operaciones de escritura

A la hora de la escritura, estaríamos ante un proceso similar. Esta vez utilizaremos los tipos Future. Así, cuando pulsemos un botón, se generará un evento que interpretará nuestro BLoC. Este llamará a un método donde llamemos a las Cloud Functions y, en caso de que ese método devuelva algo, modificaremos el estado actual. Al estar utilizando Firebase, las Cloud Functions se ejecutan en un servidor de Google, mientras que el resto del código se ejecuta en nuestro dispositivo, en el lado cliente.

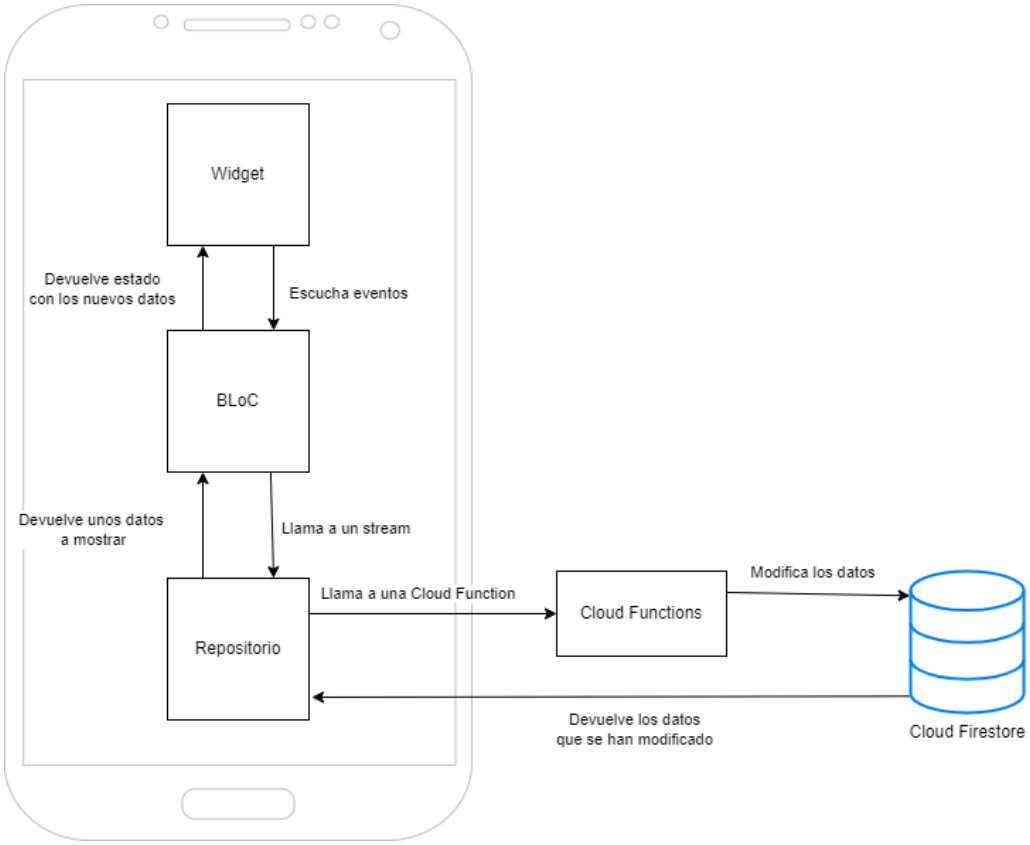

Figura 5.3: BLoC escritura.

5.3 Estructura de Cloud Firestore

En nuestro caso utilizamos una base de datos NoSQL la cual contiene colecciones, de las cuales forman parte unos documentos y estos documentos a su vez pueden contener valores u otras colecciones, creando de esta forma una jerarquía de datos. Unos de los principios de las bases de datos NoSQL que hemos aplicado ha sido la **de-normalización**, la cual consiste en duplicar algunos datos para optimizar las consultas. De esta forma, cuando queramos acceder a un documento, en vez de tener una referencia a otra colección en la que guardemos ese documento, podemos simplemente duplicar una parte de ese documento para incrementar la velocidad a la que obtenemos nuestros datos. Este punto es bastante importante a la hora de realizar operaciones de lectura en Firestore, las cuales son más frecuentes que las operaciones de escritura.

En nuestro caso, queremos de-normalizar principalmente los productos y las listas de productos. Esto se debe a que en una lista de la compra vamos a tener varias listas de productos, y estas van a contener varios productos. Por lo tanto, al final en una única lista de la compra vamos a estar leyendo en tiempo real una cantidad considerable de productos. Así, es nece-

sario duplicar cierta información de estos productos, porque, si almacenamos todos los datos de un producto en una misma colección, cada vez que queramos acceder a dichos datos será necesario hacer solicitudes adicionales. Por ejemplo, si en una lista de productos no almacenáramos una subcolección con los productos que pertenecen a esa lista de productos, y en vez de eso almacenáramos referencias a los productos que se encuentran en otra colección llamada "productos", entonces necesitaríamos hacer una petición para cada referencia. Esto resultaría muy costoso y lento. En nuestro caso, queremos obtener los productos en tiempo real con la mayor velocidad posible, por eso hemos decidido duplicar ciertos datos. Somos conscientes de que esto puede suponer un problema de almacenamiento ya que estamos duplicando una gran cantidad de datos, pero hemos decidido centrarnos en la rapidez de las consultas, ya que estamos trabajando en tiempo real.

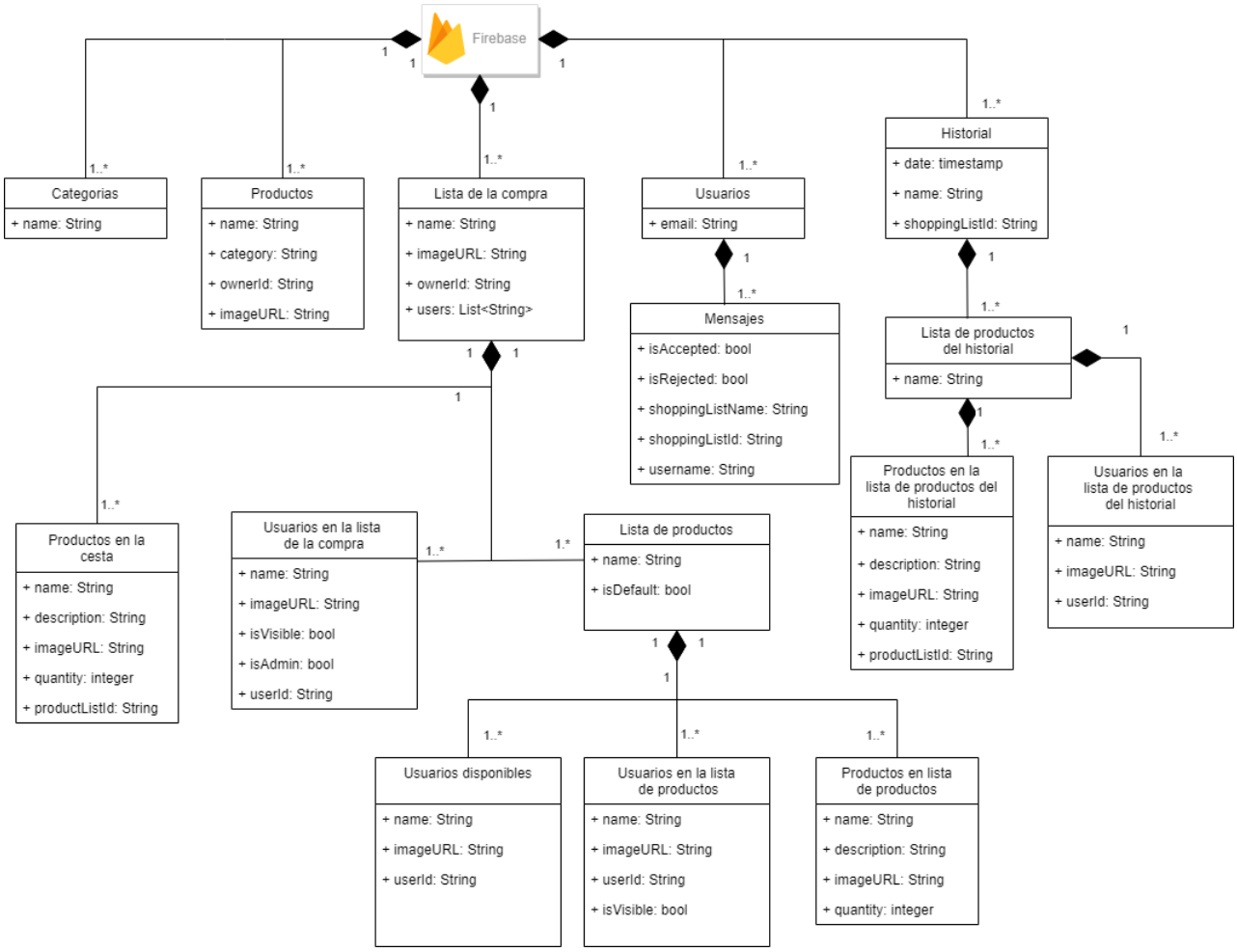

Figura 5.4: Colecciones Firestore.

Como se puede observar (Figura 5.4), las colecciones que hemos duplicado han sido princi-

palmente las de productos debido a lo que hemos comentado anteriormente. Además, tenemos diferentes tipos de usuarios con diferentes datos. Por eso, hemos decidido diferenciar estas colecciones, en vez de tratarlas como una única colección con múltiples atributos y referencias. En este caso también hemos considerado que no debería haber un gran número de usuarios, por lo que duplicar dichos datos no debería afectar demasiado al almacenamiento de la aplicación. Cabe aclarar, para futuras referencias, que en nuestro código la colección **lista de la compra** aparece definida como **shoppingList**; la **lista de productos**, como **productList**; y la **cesta**, como **cart**. Queremos hacer énfasis en estos conceptos para facilitar su entendimiento cuando hagamos referencias en el código.

5.3.1 Colecciones de Firestore

En este apartado hablaremos de las diferentes colecciones, sus atributos y a qué hacen referencia. Estas colecciones se corresponden con los datos de Cloud Firestore de la Figura 5.1.

Lista de la compra

En esta colección tenemos todas las listas de la compra de la aplicación. Están clasificadas por usuarios. De esta forma, el **ownerId** representa al usuario que creó dicha lista de la compra, es decir, es el administrador. El atributo **users** es la lista de los usuarios que forman parte de la lista de la compra. Además, estas listas de la compra también tienen un nombre y una imagen. Esta colección está formada por tres subcolecciones: los usuarios de la propia lista de la compra, las diferentes listas de productos que pertenecen a esa lista de la compra y la cesta de esta lista de la compra.

Usuarios en la lista de la compra

Estos son los usuarios que forman parte de la propia lista de la compra. En la colección anterior ya teníamos a estos usuarios representados en una lista de Strings, pero ahora vamos a detallarlos. De esta forma, cada usuario tendrá un nombre, que es el que se mostrará en la lista de la compra. También contará con una imagen, que será su foto de perfil. Asimismo, dispondrá de un atributo *isVisible* para saber si el usuario ha aceptado la invitación a la propia lista de la compra y se puede trabajar con él. El funcionamiento de las invitaciones se explica en la entidad de los mensajes de usuario. Por último, también tenemos un atributo *isAdmin* que nos permite saber si ese usuario es el administrador de la lista de la compra.

Productos en la cesta

Estos son los productos que forman parte de la cesta de una lista de la compra. Se trata de los productos que se van a añadir al historial una vez finalicemos la compra. Los productos de las listas de productos se pueden marcar como adquiridos para, de esta forma, añadirlos a la cesta. Estos constan de un nombre, una descripción, una imagen, la cantidad que vamos a comprar de dicho producto y el identificador de la lista de productos a la que hacen referencia. Este identificador de la lista de productos es importante, ya que en caso de que hayamos añadido un producto a la cesta por equivocación, o lo queramos devolver, podemos volver a añadir el producto a su lista de productos original a partir de este identificador. Una vez finalicemos la compra, se eliminarán los productos en la cesta.

Lista de productos

Esta subcolección representa a las diferentes listas de productos que forman parte de una lista de la compra. Tiene dos atributos: un nombre y un atributo *isDefault*, el cual nos sirve para diferenciar si esta lista es la lista por defecto que se crea al inicio. Queremos diferenciar la lista por defecto de las demás listas de productos, porque esta siempre va a contener a todos los usuarios de la lista de la compra y no se puede eliminar. Los documentos de esta subcolección contienen a su vez otras tres subcolecciones que son: **usuarios disponibles**, **usuarios en la lista de productos** y **productos en la lista de productos**.

Usuarios disponibles

Estos son los usuarios que están en nuestra lista de la compra, pero no forman parte de la lista de productos actualmente. Por ejemplo, si tenemos dos usuarios en la lista de la compra, usuario A y usuario B. Si el usuario A forma parte de una lista de productos, pero el usuario B no forma parte de esa lista de productos, entonces el usuario B sería un usuario disponible, mientras que el usuario A sería el usuario que contiene dicha lista de productos. Hemos decidido separar los usuarios en dos colecciones diferentes, porque, como se ha explicado anteriormente, queremos que nuestras consultas sean lo más rápidas posibles. Estos usuarios tienen un nombre, una imagen y su identificador de usuario.

Usuarios en la lista de productos

Como se indicó en el ejemplo anterior, son los usuarios los que forman parte de una la lista de productos. Estos usuarios tienen un nombre, una imagen, el identificador de acceso del usuario y un atributo *isVisible* que se modificará cuando el usuario acepte la invitación a la lista.

Productos en la lista de productos

Esta subcolección representa a los productos que forman parte de una lista de productos. Estos productos son una copia de los productos que tenemos almacenados en nuestra base de datos. También tienen atributos personalizados que estaban en la colección que representa a los productos de nuestra base de datos Cloud Firestore, como son la descripción o la cantidad. La razón de que esta entidad sea una copia de productos es que, aparte de optimizar las consultas mediante la de-normalización, también nos permite modificar datos en esta entidad sin cambiarlos en la original, como son el nombre y la imagen. Esto se debe a que, si modificamos la descripción de un producto de nuestra lista de productos que ya está presente en nuestra base de datos, no tendría sentido que un usuario que no está en nuestra lista de la compra pudiese ver y modificar esa descripción.

Categorías

En esta colección tenemos todas las categorías en las que se clasifican los diferentes productos. Estas categorías únicamente tienen un nombre. Los productos van a tener un atributo categoría para saber a que categoría pertenecen, en vez de hacer que la colección categoría tenga una subcolección productos. Lo hacemos de esta forma porque únicamente nos interesa saber a qué categoría pertenecen los productos a la hora de buscarlos por categoría. Por lo tanto, no tendría sentido que cada vez que quisiéramos obtener un producto tuviéramos que acceder antes a esta colección. Y lo mismo sucede con los productos que tienen un atributo categoría, este atributo lo usamos para clasificar los productos en categorías. Sin embargo, si solamente tuviéramos este atributo y no la propia colección de categorías, entonces para obtener todas las categorías tendríamos que obtener todos los productos, lo cual sería un proceso costoso.

Productos

Se trata de los productos comunes que existen en nuestra base de datos. Estos solamente los puede modificar el administrador de la aplicación directamente, ya que modificar estos productos afecta todos los usuarios de la aplicación. Por ello, el resto de usuarios que no sean administradores de la aplicación solamente podrán modificar copias. Estos productos tienen un nombre, una categoría de la cual forman parte y que usaremos a la hora de filtrar los productos por categoría, un *ownerId*, que es el identificador del usuario que creó dicho producto, y una imagen.

Historial

En esta colección cada documento se corresponde con una entrada del historial. Cada una de estas entradas tiene una fecha, que es la fecha en la que se terminó dicha compra; un nombre, que es el que hemos decidido ponerle a esa compra; y el identificador de la lista de la compra a la que pertenece ese historial. De esta forma, cuando queramos obtener el historial de una lista de la compra, solamente tendremos que buscar los documentos cuyo *shoppingListId* coincida con el que buscamos.

Lista de productos del historial

Estamos ante las listas de productos en las cuales se dividen los productos una vez finalizada la compra. Cuando terminemos la compra, los productos que estén en la cesta se volverán a clasificar en listas de productos a partir del *productListId* que tenían. De esta forma, conseguimos volver a organizar nuestros productos una vez finalicemos la compra, para así poder dividir la compra en función de los productos que contenga cada lista de productos. Estas listas de productos del historial solamente tendrán un nombre, que es el nombre que tenían cuando se finalizó la compra. El atributo *isDefault* que tenían las listas de productos cuando estaban en la lista de la compra ya no será necesario, porque las listas de productos ya no se podrán modificar.

Productos en la lista de productos del historial

Se trata de los productos de las listas mencionadas anteriormente. Estos productos tendrán un nombre, una descripción, una imagen, una cantidad y el identificador de la lista de productos a la que hacen referencia. La imagen y la descripción serán modificables. Esto nos permitirá añadir anotaciones en las que demos más detalles del producto o una fotografía del producto concreto que hemos comprado. No será posible la modificación del nombre y la cantidad, ya que esto inutilizaría la lista de la compra, al no saber qué hemos comprado o en qué cantidad.

Usuarios en la lista de productos del historial

En esta subcolección, tenemos los documentos que hacen referencia a los usuarios que forman parte de las listas de productos. Como ya se ha mencionado anteriormente uno de los objetivos de la aplicación es facilitar el reparto de los productos una vez finalizada la compra. Para ello, en las entradas del historial dividimos los productos en listas de productos y estas listas, a su vez, contienen usuarios. Así, para saber qué productos ha comprado un usuario, solamente tendremos que ver las listas de productos a las que pertenece este usuario. Los atributos de estos usuarios son el nombre, la imagen y el identificador de usuario al que hace

referencia. En este caso, no tenemos un atributo *isVisible* porque solamente los usuarios que estén presentes en el momento de finalizar la compra, es decir, que tengan el atributo *isVisible* como true, podrán aparecer en el historial.

Usuarios

Esta colección representa a los usuarios de la aplicación. Hemos decidido no almacenar muchos datos de los usuarios en esta colección, ya que la usaremos a la hora de manejar las invitaciones a las listas de un usuario. Por lo tanto, dispondremos simplemente del correo electrónico para poder filtrar las invitaciones por correo ya que este es único. Otros parámetros, como son la foto, el username o la contraseña, son guardados directamente en el servicie de Firebase Auth para mayor comodidad.

Mensajes

Representan las diferentes invitaciones a listas de la compra que tiene un usuario. Estas invitaciones pueden estar pendientes de aceptar o rechazar, aceptadas o rechazadas. Estos mensajes cuentan con varios atributos, el *shoppingListName*, el cual representa el nombre de la lista de la compra a la que invitan al usuario; el *shoppingListId*, es decir, el identificador de dicha lista, el cual vamos a necesitar para poder acceder a esa lista; y un *username*, que es el nombre del usuario que nos envía la invitación. Asimismo, cuenta con otros dos atributos: *isAccepted*, que nos indica si la invitación ya ha sido aceptada, e *isRejected*, que muestra si la invitación ha sido rechazada. Si en un mensaje tanto el atributo *isAccepted* como el atributo *isRejected* son falsos, entonces la invitación está en estado pendiente. Una vez se acepte o rechace la invitación, uno de los dos atributos pasará a ser verdadero.

5.4 Repositorio

En esta sección indicamos los diferentes métodos que hemos utilizado para desarrollar los casos de uso. Así, hemos diseñado tres clases abstractas: una para los métodos que utilizan Firebase Auth, otra para los que usan Cloud Firestore y, por último, una interfaz para los que utilizan el servicio de Cloud Storage. Al estar trabajando en todo momento con datos asíncronos, devolveremos **Future** y **Streams** en la mayoría de los casos. Estos servicios están relacionados con el repositorio de la lógica de negocio en la figura 5.1.

5.4.1 Firebase Auth

Aquí hablaremos de los métodos relacionados con la autenticación del usuario.

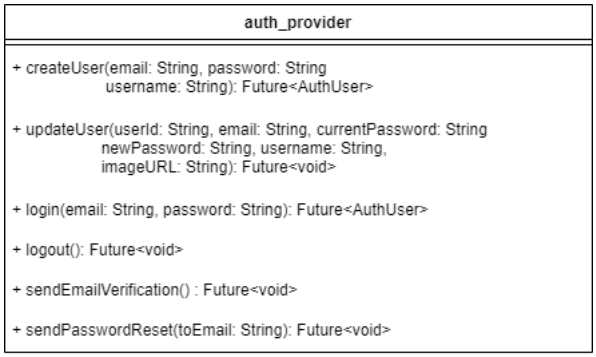

Figura 5.5: Métodos Firebase Auth.

createUser

Este método permite crear un usuario a partir de su correo electrónico y la contraseña. El username lo necesitamos para actualizar los datos del usuario, ya que recordemos que en Firebase Auth podemos guardar el nombre de usuario, aparte de su correo electrónico y contraseña. El correo es único para cada usuario. Sin embargo, el nombre de usuario no tiene por qué, ya que este nombre es el que van a ver el resto de usuarios en las listas de la compra. Así, no tendría sentido que una persona a la cual conocen por su nombre propio, se tenga que llamar de otra forma porque ese nombre ya se encuentra en uso.

updateUser

Este método nos permite actualizar los datos de un usuario. Podremos modificar su nombre, su correo electrónico, su imagen de perfil y la contraseña. Para ello comprobaremos si la contraseña antigua coincide con la que nos envía el usuario.

login

Iniciaremos sesión utilizando el correo electrónico, que es único para cada usuario, y la contraseña. Devolveremos un *AuthUser* que se corresponde con el usuario que acaba de iniciar sesión.

logout

Simplemente cerraremos sesión.

sendEmailVerification

Cuando creamos la cuenta para poder acceder es necesario verificar el correo. En caso de que no nos llegue un mensaje de verificación a nuestro correo electrónico, podemos volver a

utilizar este método cuántas veces sea necesario hasta que nos llegue un mensaje. No es necesario pasar por parámetro el correo, ya que el mensaje se envía al correo que se ha registrado cuando creamos el usuario.

sentPasswordResetEmail

Recurriremos a este método cuando el usuario haya olvidado su contraseña. En este caso introduciremos el correo electrónico al que queremos enviar el mensaje para reiniciar la contraseña.

5.4.2 Cloud Firestore

Aquí trataremos los métodos que interactúan principalmente con el servicio de Cloud Firestore. Cabe destacar que muchos de los valores de retorno son clases que hemos creado nosotros, por ejemplo *CloudShoppingList*. Hemos decidido incluir ”Cloud” en el nombre para hacer referencia a que estamos tratando con elementos en la nube. Estas clases son variaciones de los documentos que pertenecen a las colecciones que hemos definido anteriormente. Por ejemplo, *CloudShoppingList* hace referencia a la colección lista de productos. Hemos decidido crear múltiples clases para las distintas colecciones, en vez de hacer varios constructores en una misma clase, para así hacer más entendible nuestro proyecto al crear nuevas clases en función de la funcionalidad.

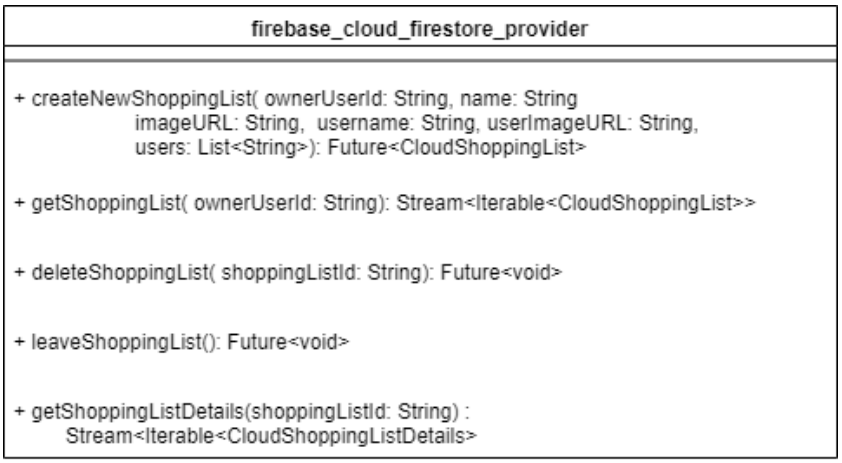

Figura 5.6: Métodos Firestore 1

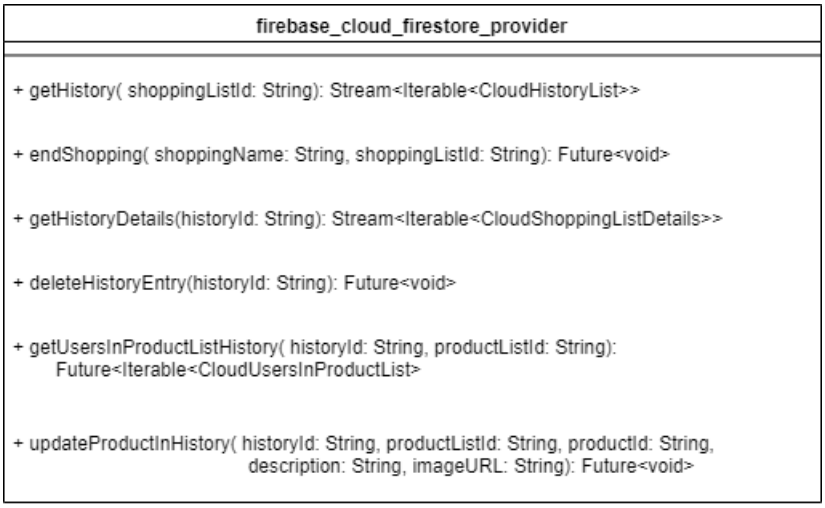

Figura 5.7: Métodos Firestore 3

Figura 5.8: Métodos Firestore 2

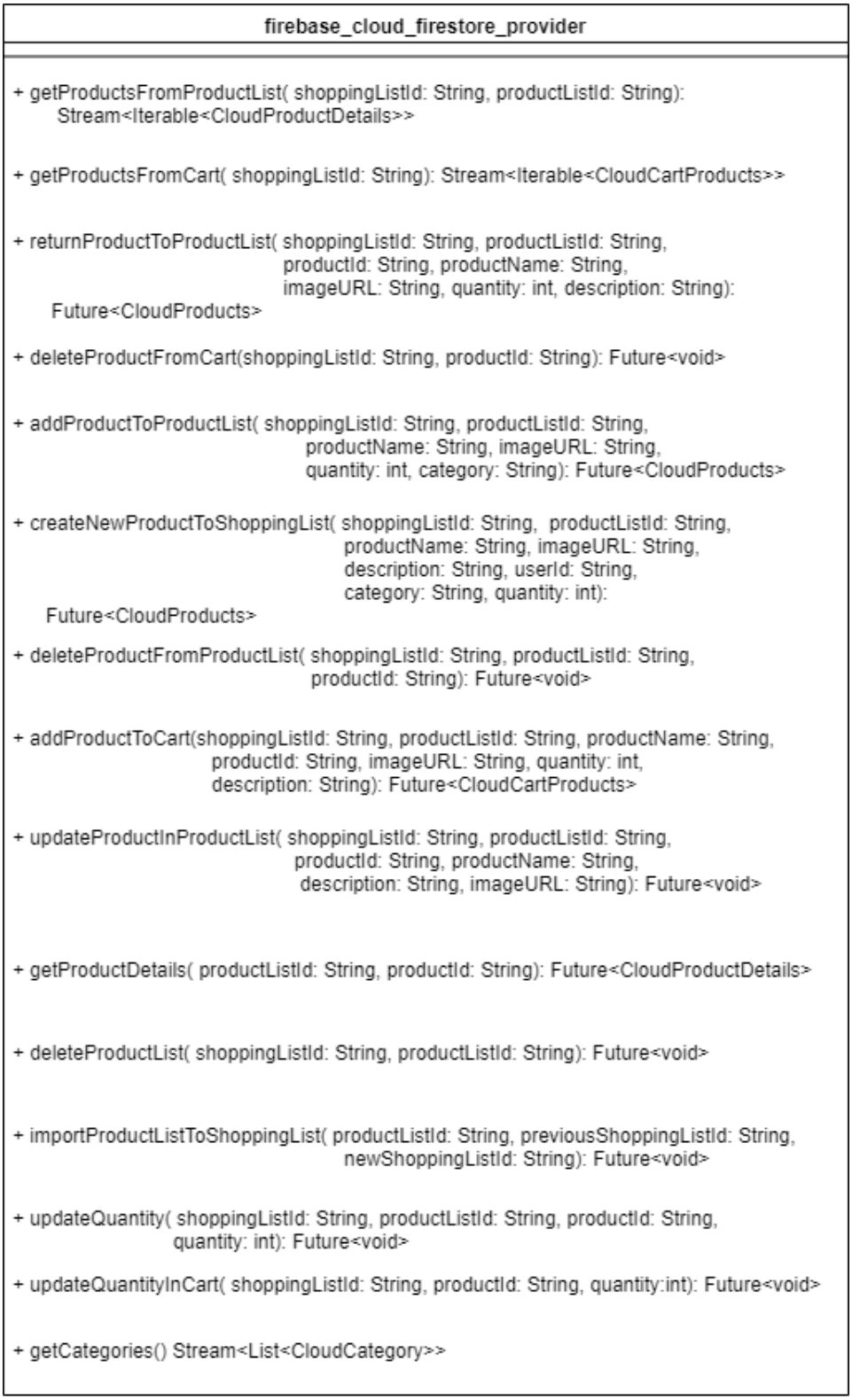

Figura 5.9: Métodos Firestore 4

Los apartados siguientes se corresponden con la figura 5.6.

createNewShoppingList

Este método permite crear una nueva lista de la compra. El *ownerUserId* representa al usuario que crea esa lista de la compra y, por lo tanto, va a ser el administrador. El *name* y el *imageURL* son el nombre y la imagen de la lista respectivamente. El *username* es el nombre que tiene el usuario de la lista de la compra y *userImageURL*, su imagen. *Users* son todos los usuarios a los que se les ha enviado una invitación para unirse a la lista de la compra.

getShoppingLists

Este método permite obtener todas las listas de la compra que pertenecen a un usuario. Para ello, introducimos el identificador del usuario del cual queremos obtener las listas de la compra. Esto nos devuelve un Stream con las listas de la compra que recuperamos. Cabe recordar que usamos Stream porque estamos trabajando con información en tiempo real, la cual se puede modificar en cualquier momento.

deleteShoppingList

Permite borrar una lista de la compra. Este método solo lo puede utilizar el administrador de la lista de la compra.

leaveShoppingList

Se utiliza cuando un usuario abandona la lista de la compra. Esta función solamente la pueden utilizar aquellos usuarios que no son administradores de la lista de la compra que quieren abandonar.

Los apartados siguientes se corresponden con la figura 5.8.

getAllUserInvitations

A partir de esta función obtenemos todas las invitaciones para unirse a listas de la compra que tiene un usuario. Hemos decidido que esto sea un Stream<Iterable> porque así será posible recibir más invitaciones en tiempo real mientras estamos consultando las propias invitaciones.

manageListInvitation

A partir de este método manejaremos las diferentes invitaciones a listas de la compra. Si hemos aceptado la invitación, el atributo *isAccepted* será cierto. Por el contrario, si es falso, significará que hemos rechazado dicha invitación. Con este método modificaremos la subcolección de mensajes para saber si un usuario ha aceptado o rechazado una invitación.

sendUserInvitation

Este método permite invitar a otros usuarios a nuestra lista de la compra. Cualquier usuario que pertenezca a la lista de la compra será capaz de invitar a nuevos usuarios a unirse a esta.

addUserToProductList y addAvailableUserToProductList

En una lista de productos tendremos dos tipos de usuarios. El primer tipo se corresponde con los usuarios que ya están en la lista de productos; el segundo tipo hace referencia a aquellos usuarios que no están en la lista de productos pero sí se encuentran en la lista de la compra, esto últimos se denominan *usuarios disponibles*. *addUserToProductList* permite añadir usuarios que están disponibles a la lista de la compra. *addAvailableUserToProductList* añade usuarios que ya se encuentran en la lista de productos a la lista de usuarios disponibles. Cuando un usuario está disponible deja de estar en la lista de productos y viceversa.

getUsersInProductList y getAvailableUsersInProductList

Como se ha comentado anteriormente, tenemos dos tipos de usuarios. Estos dos métodos nos permitirán visualizar a esos dos tipos de usuarios y, así, poder interactuar con ellos.

removeUserFromShoppingList

Utilizando este método el administrador será capaz de eliminar a un usuario de la lista de la compra.

Los apartados siguientes se corresponden con la figura 5.7.

endShopping

Este método permite finalizar el proceso de la compra y crear una nueva entrada en el historial. Esa nueva entrada del historial puede tener un nombre, además de la fecha.

getUsersInProductListHistory

Cada entrada del historial se corresponde con una compra. Cada compra tiene productos organizados en listas de productos, y estas listas, a su vez, tienen usuarios. Con este método podemos obtener los usuarios que forman parte de estas listas. En este caso, es del tipo Future<Iterable> porque se trata de una lista que no se va a modificar en tiempo real, porque una vez finalizada la compra los usuarios que participaron en ella no cambian. Por eso, empleamos un Future y no un Stream.

A continuación se explican los métodos más importantes del diagrama de la figura5.9.

returnProductToProductList

Este método permite devolver un producto de la cesta a la lista de productos de la cual forma parte. Para ello, introducimos los datos del producto que queremos devolver y el identificador de la lista de productos a la que vamos a devolver dicho producto.

addProductToProductList

Este método permite añadir un producto ya existente en nuestra base de datos a una lista de productos. Para ello, introducimos los datos que tendrá este producto, a saber: un nombre, una imagen, una cantidad y la categoría a la que pertenecen.

createNewProductToShoppingList

Este caso es similar al anterior pero aquí estamos creando un nuevo producto que no existe previamente en la base de datos. A diferencia del anterior, en este caso introduciremos una descripción cuando creemos el producto.

addProducToCart

Este método nos permite marcar un producto como adquirido y añadirlo a la cesta. Para ello, introducimos los detalles del producto que queremos añadir, la lista de productos a la que pertenece y la lista de la compra de la cual forma parte la cesta.

importProductListToShoppingList

Creamos una copia de la lista de productos que vamos a importar a otra lista de la compra que seleccionemos. En la nueva lista de la compra tendremos la lista de productos que hayamos

importado con sus respectivos productos. Esta lista al principio solamente tendrá un usuario y es el que ha importado dicha lista. Para conseguir esto, debemos indicar el identificador de la lista de productos que vamos a importar; la lista de la compra en la que nos encontramos, porque para acceder a una lista de productos necesitamos saber el identificador de la lista de la compra; y, por último, el identificador de la nueva lista de la compra a la cual vamos a importar la lista de productos.

updateQuantity y updateQuantityInCart

Los productos van a tener una cantidad, con este método podremos actualizar esa cantidad. Estos dos métodos se podrían haber combinado en uno solo e introducir un booleano como parámetro para saber si es necesario actualizar la cantidad de un producto de una lista de productos o de un producto de la cesta. Con todo, preferimos emplear dos métodos diferentes para facilitar el entendimiento de los mismos. Además, en caso de que en el futuro hicieran falta más de dos métodos, sería más fácil implementar nuevo código.

5.4.3 Cloud Storage

En este apartado hablaremos de los métodos que se especializan en acceder al servicio de Cloud Storage.

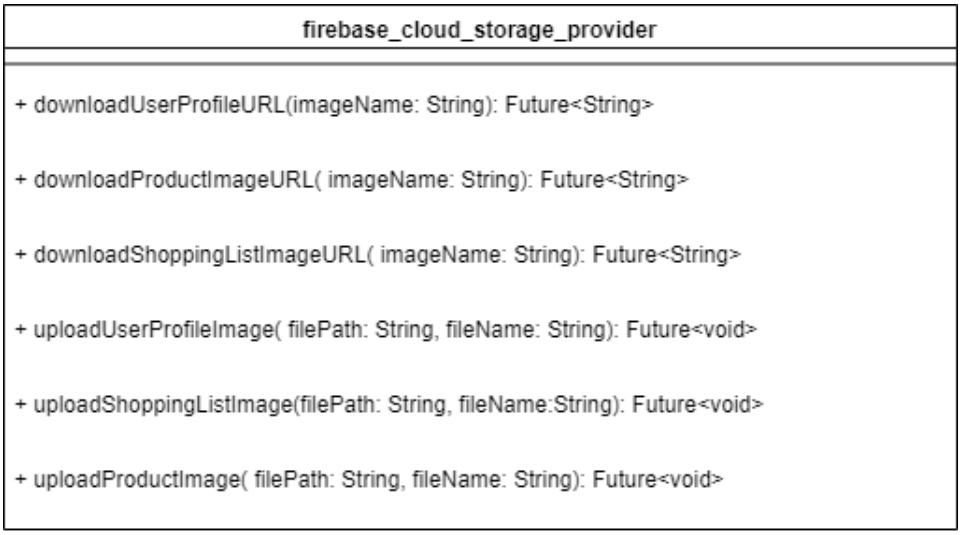

Figura 5.10: Métodos Storage

downloadUserProfileURL, downloadProductImageURL, downloadShoppingListNameURL

En estos casos descargaremos la imagen del Cloud Storage que tenga el nombre que nosotros introducimos como parámetro. La diferencia entre estos tres métodos es la carpeta a la que acceden.

uploadUserProfileImage, uploadShoppingListImage, uploadProductImage

Estos casos son similares al apartado anterior, pero, en esta ocasión, en vez de descargar las imágenes las subiremos a Cloud Storage a su correspondiente carpeta.

5.5 Interfaz

En este apartado se muestran los wireframes de la aplicación correspondientes a la vista de una lista de la compra. Se ha decidido mostrar solamente estos wireframes para que este apartado no resulte demasiado extenso, además de ser la pantalla más importante de la aplicación. Estos wireframes se han realizado utilizando la herramienta de Balsamiq. Asimismo, se explican las razones que se tuvieron en cuenta a la hora de diseñar la interfaz.

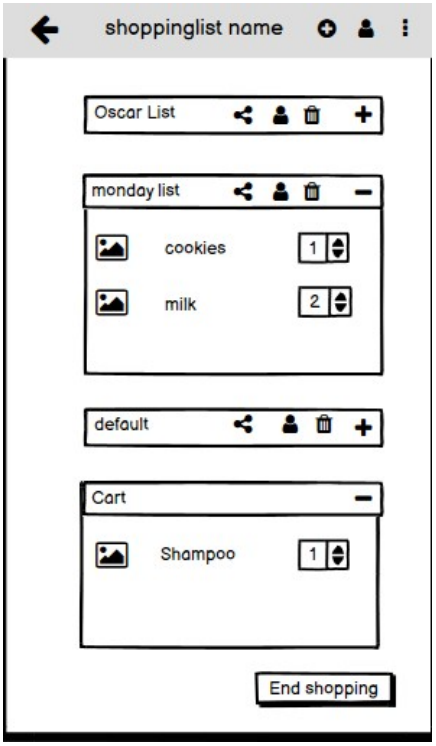

Figura 5.11: Wireframe de la aplicación 1

Como se puede ver en la figura 5.11 una lista de la compra contiene varias listas de productos y una cesta. En este caso tenemos tres listas productos (Oscar List, monday list y la lista default), la default es la que tienen todos los usuarios y que no se puede eliminar. Tampoco es posible añadir o quitar usuarios de esta lista. La listas de productos tienen un nombre y cuatro iconos. Los iconos + y - permiten maximizar y minimizar la lista de productos respectivamente. El icono de compartir hace posible importar esa lista de productos a otra lista de la compra, como se ha explicado anteriormente. El icono del usuario en la lista de productos posibilita ver los usuarios que forman parte de esa lista de productos. Recordemos que estos usuarios pueden estar disponibles para añadirlos a la lista de productos o estar ya en la propia lista de productos. En esa pantalla en la que vemos los usuarios, podremos añadirlos o quitarlos de la lista de productos. Por último, el icono del contenedor de basura permite eliminar dicha lista.

Respecto a la lista de la compra tenemos un nombre y un icono de flecha para volver hacia atrás, a la vista de todas las listas de la compra de un usuario. El icono del + con un círculo hace posible añadir una nueva lista de productos a la lista de la compra. El icono de usuarios nos permite añadir nuevos usuarios a la lista de la compra y, en el caso de un administrador, también nos permite eliminar a los usuarios actuales de la lista. El icono con los tres puntos en vertical nos mostrará un diálogo para acceder al historial o cerrar sesión. Aparte de las listas de productos, también tenemos una cesta que podremos maximizar o minimizar con los productos que marcamos como adquiridos. Por último, disponemos del botón End shopping con el que finalizamos la compra y creamos una entrada en el historial con los productos de la cesta y las listas de productos que contienen esos productos.

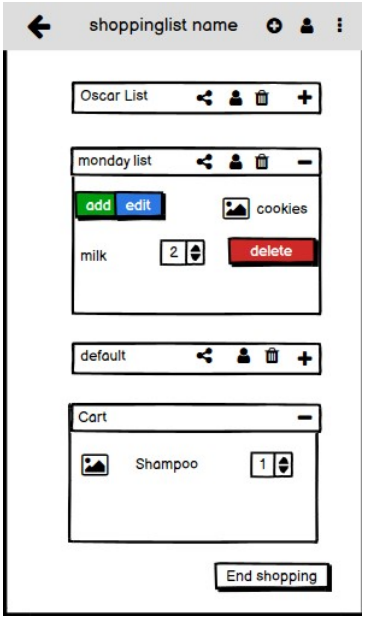

Figura 5.12: Wireframe de la aplicación 2

En lo referente a los productos, estos cuentan con una imagen, un nombre y una cantidad a simple vista. Hemos decidido solamente mostrar estas tres partes en un primer momento para no sobresaturar la pantalla, evitando así una interfaz más molesta visualmente y que el usuario se pueda equivocar al pulsar un icono. Para ello, hemos decidido separar las diferentes operaciones que podemos realizar mediante slides, como se puede observar en la figura 5.12. De esta forma, cuando queramos borrar de forma permanente un producto, deslizaremos el producto a la izquierda y seleccionaremos dicha opción. Por otro lado, en caso de que deseemos editar y añadir a la cesta, deslizaremos el producto a la derecha. Hemos decidido mostrar la imagen y nombre ya que hacen el producto más comprensible a simple vista. La cantidad se muestra porque consideramos que será uno de los atributos que más utilicemos y al ponerlo directamente en esta pantalla evitamos tener que acceder a sus detalles para modificar desde ahí la cantidad. Hubo un momento que nos planteamos que, al mantener pulsado, pudiéramos acceder a los detalles del producto, que al pulsar el producto se añadiera a la cesta y contar con un botón para eliminar el producto. Finalmente, decidimos descartar dicha opción para intentar mantener uniformes las diferentes funcionalidades.

Nuestra interfaz está dividida en diferentes widgets, algunos de ellos han sido creado por nosotros mismos para poder reutilizarlos en el futuro. Es el caso de los slides que podemos ver en la figura 5.12. Estos slides no solo están en los productos de las listas de productos, también están en los productos de la cesta, por eso decidimos crear nuestro propio widget slide que contenga los 3 tipos de slide, el de añadir a la cesta, el de actualizar el producto y el de borrar el producto.

Capítulo 6

Implementación

En este capítulo explicaremos en detalle cómo hemos realizado los diferentes métodos definidos previamente en el diseño, así como la implementación del patrón BLoC. También hablaremos sobre las diferentes reglas de seguridad que hemos utilizado en nuestra aplicación. Para aclarar todos estos conceptos mostraremos capturas del código implementado.

6.1 Repositorio

En esta sección, con el fin de explicar los métodos utilizados en los casos de uso, volveremos a dividirlos en función del servicio que utilicen, igual que hicimos en el diseño. Estos servicios son **Firebase Auth**, **Cloud Firestore** y **Cloud Storage**. También hablaremos de las **Cloud Functions** y **Algolia**, ya que, hemos utilizado ambas para desarrollar dichos servicios.

6.1.1 Firebase Auth

Aquí abordaremos todos los casos de uso relacionados con el servicio de autenticación de Firebase. Estos casos de uso se caracterizan principalmente por utilizar métodos predefinidos en Firebase Auth.

createUser

Firebase Auth ya tiene un método para crear usuarios utilizando un email y una contraseña, por lo que en este caso de uso lo que hacemos es llamar a este método. Una vez creamos el usuario, obtendremos una referencia a este usuario utilizando el método *currentUser* de Firebase Auth. A partir de esta referencia, actualizaremos el nombre de usuario que se almacena en Firebase Auth, porque recordemos que Firebase Auth puede almacenar el nombre del usuario, su imagen, su email y su contraseña. Por último, añadiremos un documento a la colección de usuarios. Aunque en este caso de uso estemos tratando principalmente con el servicio de Firebase Auth, para conseguir crear el documento tendremos que utilizar Cloud Firestore.

updateUser

Al igual que en el caso anterior, obtendremos los detalles de un usuario utilizando el método *currentUser* de Firebase Auth. Si modificamos las credenciales actuales, el token que estamos usando en la sesión actual caducará y nos cerrará la sesión una vez actualicemos los datos. Para evitar esto, lo que haremos será obtener las credenciales de la aplicación, para que posteriormente, una vez hayamos actualizado los datos, nos volvamos a autenticar con estas credenciales. De esta forma lo que haremos será modificar los datos que se almacenan en Firebase Auth: la foto de perfil, el nombre, etc. Asimismo, modificaremos la colección de usuarios, por lo que tendremos que volver a acceder a Firestore.

El resto de métodos se realizan utilizando métodos predefinidos en Firebase Auth como, por ejemplo, *signInWithEmailAndPassword* para iniciar sesión o *signOut* para cerrar sesión.

6.1.2 Cloud Firestore

En este apartado explicaremos cómo hemos implementado métodos relacionados con Cloud Firestore. Para ello, primero abordaremos el funcionamiento de los métodos que devuelven un Stream, ya que todos ellos se realizan de la misma forma.

Streams

En nuestra aplicación utilizamos Streams para manejar los datos en tiempo real. De esta forma, cuando se modifiquen los datos Firestore, obtendremos los nuevos datos modificados a través de Firestore. A la hora de trabajar con los Stream, tenemos que destacar otro concepto: el **snapshot**. Un snapshot es una representación de los datos obtenidos en una consulta de Firestore. Cuando establecemos un listener por primera vez, en nuestro caso utilizando un Stream, Cloud Firestore nos envía un snapshot inicial con los datos que coinciden con la consulta realizada y, después, nos enviará un snapshot adicional cada vez que se produzca un cambio en los datos que estamos escuchando. Los snapshot contienen un atributo de tipo Map, que representa los datos obtenidos en la consulta. Estos datos se pueden leer o modificar igual que el resto de Map.

```
Stream<Iterable<CloudShoppingList>> getShoppingList({required String ownerUserId}) {

  final allShoppingLists = firestore.collection(shoppingListCollectionName)
      .where(usersField, arrayContainsAny: [ownerUserId] ).snapshots().map((event) =>
      event.docs
          .map((doc) => CloudShoppingList.fromSnapshot(doc)));
  return allShoppingLists;
}
```

Figura 6.1: Stream getShoppingList

En este método 6.1, obtendremos las listas de la compra de las que forma parte un usuario, quien no tiene porqué ser el administrador. Para ello, tenemos que acceder a la colección lista de la compra (shoppingList) y buscar todas las listas de la compra en las que el atributo *users* contenga el *ownerUserId* que hemos introducido. Recordemos que el atributo *users* es una lista con los de identificadores de todos los usuarios que forman parte de esa lista de la compra. El Stream se crea a partir de los datos obtenidos en el método *snapshots*, que en este caso está escuchando los cambios de la colección lista de la compra. A continuación, utilizamos el primer map para transformar cada snapshot en un Iterable<CloudShoppingList>. El *event* representa a un snapshot, con *docs* obtenemos los documentos de ese snapshot. A partir de ahí, mapeamos cada documento a una CloudShoppingList utilizando el método *fromSnapshot* 6.2. De esta forma, podemos crear nuestro *Iterable<CloudShoppingList>*.

```
CloudShoppingList.fromSnapshot(QueryDocumentSnapshot<Map<String, dynamic>> snapshot)
    : shoppingCartListId = snapshot.id,
      users = List<String>.from(snapshot.data()[usersField] ?? []),
      name = snapshot.data()[nameShoppingListField] as String,
      imageURL = snapshot.data()[imageURLField] ?? '',
      ownerUserId = snapshot.data()[ownerUserIdFieldName] ?? '';
}
```

Figura 6.2: FromSnapshot CloudShoppingList

El resto de métodos que devuelven *Stream<Iterable>* son prácticamente idénticos, solamente cambia la forma de mapear. Por eso, hemos preferido no detallar más métodos de este tipo.

Batch

Antes de explicar más en detalle los métodos de escritura como son la actualización, creación o borrado, explicaremos en qué consiste la funcionalidad **Batch**. Batch es un conjunto de operaciones de escritura que se ejecutan en una única transacción, lo que nos permite realizar operaciones atómicas. Hemos utilizado Batch para los casos en los que necesitábamos llevar a cabo más de una operación sin interrupción, como por ejemplo devolver un producto de la cesta a una lista de productos.

```
final db = firestore;
final batch = db.batch();

var productListRef = firestore.collection(shoppingListCollectionName)
    .doc(shoppingListId).collection(productCollectionName).doc(productListId)
    .collection(productsInProductListCollectionName).doc(productId);
var cartProductsRef = firestore.collection(shoppingListCollectionName).doc(shoppingListId)
    .collection(cartCollectionName).doc(productId);

batch.set(productListRef,
    {
      nameShoppingCartField: productName,
      imageURLField: imageURL,
      quantityField: quantity,
      descriptionField: description,
    }, SetOptions(merge: true));
batch.delete(cartProductsRef);

batch.commit();
```

Figura 6.3: Return product to product list

En este caso, en primer lugar creamos un batch. Seguidamente, establecemos unas referencias a los documentos que queremos modificar, en este caso, esos documentos se encuentran en las colecciones de productos en la lista de productos y en la colección de productos en la cesta. El método *set* nos permite crear o modificar un documento, pero necesitamos especificar previamente la dirección de ese documento. La opción *merge: true* hace posible actualizar los atributos del documento. En caso de ser false, se sobrescribe el documento. El método *delete* es empleado para borrar. En este caso, creamos un nuevo producto en la lista de productos. Para ello, modificamos la colección de productos en la lista de productos y borramos el producto de la colección productos en la cesta. Cuando llamemos a *batch.commit* lo que sucederá es que se enviará una solicitud al servidor Firestore que incluirá todas las operaciones de escritura que hemos especificado en el batch.

add y update

A la hora de crear documentos en una colección utilizamos el método *add*. A este método le pasamos como parámetro un Map con los atributos que queremos que tenga el documento que creamos. Este método nos devuelve el identificador del documento creado. Se trata de método asíncrono.

El *update* nos permite actualizar los atributos de un documento. A diferencia del *set*, en el *update*, si el documento no existe, se produce un error. El *update map* recibe como parámetro un *map* con los atributos que queremos modificar, solamente actualizará los atributos que le enviemos.

Utilizando la terminología anterior hemos implementado todos los casos de uso de Firestore. De esta forma, el único término que quedaría por explicar sería el de las Cloud Functions. No obstante, por motivos de claridad, hemos decido recoger dicha explicación en un apartado posterior.

6.1.3 Cloud Storage

En esta sección hablaremos de cómo hemos implementado los métodos relacionados con el Cloud Storage. Estos se dividen en 2 tipos: los métodos download y los métodos upload.

downloadUserProfileURL, downloadProductImageURL, downloadShoppingListImageURL

```
Future<String> downloadProductImageURL(String imageName) async {
  String downLoadUrl = await FirebaseStorage.instance
      .ref('productsImage/$imageName').getDownloadURL();

  return downLoadUrl;
}
```

Figura 6.4: Download product image

Los tres métodos de descarga de archivos son iguales, únicamente cambia la carpeta a la que se accede. En este caso accedemos a la carpeta productsImage e introducimos el nombre que queremos obtener. El método *donwloadURL* nos devuelve la URL de la imagen para que podamos acceder a ella posteriormente.

uploadUserProfileImage, uploadShoppingListImage, uploadProductImage

```
Future<void> uploadProductImage(String filePath, String fileName) async {
  File file = File(filePath);
  try {
    await FirebaseStorage.instance.ref('productsImage/$fileName').putFile(file);
  } on FirebaseException catch (e) {
    print(e);
  }
}
```

Figura 6.5: Upload product image

filePath es el path local del archivo que queremos subir y *fileName* es el nombre que tendrá cuando lo subamos a Firebase Storage. En este caso, creamos una referencia para el archivo en Firebase Storage y, a continuación, utilizando *putFile* subimos ese archivo a Firebase Storage.

6.1.4 Cloud functions for firebase

Estas funciones nos permiten ejecutar código de backend en respuesta a eventos (triggers), autenticación y a peticiones HTTPS. Estas funciones, una vez escritas, empiezan a ser administradas por los servidores de Google, por lo que no necesitamos usar nuestro propio servidor.

En nuestro caso hemos utilizado dos tipos de Cloud functions for Firebase: aquellas que reaccionan a peticiones https y los Firestore Triggers, que se activan cuando modificamos un documento concreto en Firestore.

Peticiones https

```
exports.importProductList = functions.region('europe-west1').https.onCall(async(snap, context) => {

  const db = firestore;
  const previousShoppingListId = snap.previousShoppingListId;
  const newShoppingListId = snap.newShoppingListId;
  const previousProductListId = snap.previousProductListId;
  const newProductListId = snap.newProductListId;

  if (!(context.auth)) {
    throw new functions.https.HttpsError(
      'permission-denied',
      'The user must be authenticated'
    );
  }
  try {

  const shoppingListRefPrevious = db.collection('shoppingLists').doc(previousShoppingListId)
    .collection('shoppingList').doc(previousProductListId).collection('products');
  const shoppingListRefNew = db.collection('shoppingLists').doc(newShoppingListId)
    .collection('shoppingList').doc(newProductListId).collection('products');
  const snapshot = await shoppingListRefPrevious.get();

  if(!snapshot.empty){
    snapshot.forEach(async doc =>{

      var productName = doc.data()['name'];
      var imageURL = doc.data()['imageURL'];
      var quantity = doc.data()['quantity'];
      var description = doc.data()['description'];

      if (!imageURL){
        imageURL='';
      }
      if (!quantity){
        quantity=1;
      }
      if (!description){
        description='';
      }
      shoppingListRefNew.add({'name': productName, 'imageURL': imageURL, 'quantity': quantity, 'description': description});
    });
}

  } catch (err){
    console.log('Error when importing products: ' + err);
  }
});
```

Figura 6.6: Https Cloud Function

Términos que debemos tener en cuenta:

- **region**: indica la región en la que se encuentra nuestra base de datos. En nuestro caso se trata de europe-west1.

- **snap**: contiene los datos que introducimos cuando llamamos a la Cloud Function. Estos datos se almacenan en formato JSON. Para acceder a los mismos, hacemos *snap.nombreDelDato*.
- **context**: contiene información sobre la autenticación del usuario, en caso de haberla, así como información sobre los datos del recurso con el que estamos operando, como, por ejemplo, su path.
- **snapshot**: contiene todos los documentos de la colección a la que queremos acceder.

En nuestro caso, primero establecemos la base de datos a Firestore. A continuación, obtenemos los valores de los atributos que le pasamos en el snap. Seguidamente, comprobamos que el usuario está autenticado, ya que la función *importProductList* solo la pueden realizar los usuarios autenticados. Después, establecemos referencias a la lista de productos que vamos a importar y a la nueva lista de productos que vamos a crear. Por último, con el *snapshot* obtenemos todos los documentos (productos) de la lista de productos que vamos a importar y los almacenamos en el *snapshot*. En caso de que tengamos algún producto, añadiremos cada producto a la lista de productos.

Firestore Triggers

Hay tres tipos de triggers en Firestore: *onCreate*, que se activa cada vez que creamos un documento en una colección; *onUpdate*, que se activa cuando actualizamos un documento existente en Firestore; y *onDelete*, que se activa cuando borramos un documento de una colección.

```
exports.onUserUpdatedInShoppingCart = functions.region('europe-west1').firestore
.document('shoppingLists/{shoppingListId}/userList/{userId}').onUpdate(async (change, context) => {
    const newUser = change.after.data();
    const shoppingListId = context.params.shoppingListId;
    const userId = context.params.userId;

    const productLists = await firestore.collection('shoppingList').doc(shoppingListId).collection('productList').get();

    productLists.forEach(async (productListDoc) => {
        const availableUsers = await productListDoc.ref.collection('availableUsers').where('user_id', '==', userId).get();
        availableUsers.forEach(async (doc) => {
            if (doc.exists) {
                await doc.ref.update(newUser);
            }
        });
        const userList = await productListDoc.ref.collection('userList').where('user_id', '==', userId).get();
        userList.forEach(async (doc) => {
            if (doc.exists) {
                await doc.ref.update(newUser);
            }
        });
    });
});
```

Figura 6.7: Cloud function onUpdate

Términos que debemos tener en cuenta:

- **change**: contiene información sobre el documento que se acaba de actualizar. El *before.data* incluye los datos del documento antes de actualizarse. El *after.data* incluye los nuevos datos del documento.

- **context.params**: recoge los valores del path del documento, es decir, contiene el valor del *shoppingListId* y el *userId*. Estos dos atributos son wildcards, es decir, pueden tener cualquier valor. Con esto, estamos consiguiendo que al actualizar un *userId* de cualquier shoppingList se active esta función. Y es que, si especificáramos el *shoppingListId*, esta solo se activaría cuando modificáramos los usuarios de esa lista de la compra en concreto.

Nuestro objetivo con esta función 6.7 es que, cuando modifiquemos los usuarios en la lista de la compra, se modifiquen también los usuarios en las listas de productos. Para ello, en el *newUser* guardamos los nuevos datos del usuarios. El *shoppingListId* representa el identificador de la lista de la compra que contiene los usuarios. El *userId* representa el identificador del usuario que acabamos de actualizar. Entonces, en *productLists*, obtenemos todos los documentos de las listas de productos. Recordemos que estas listas de productos tienen dos tipos de usuarios: los que ya se encuentran en dicha lista de productos y los que están en la lista de la compra, pero no en esa lista de productos. Obtenemos los usuarios cuyo identificador coincide con el identificador del usuario de la lista de la compra que acabamos de actualizar y actualizamos estos usuarios con los nuevos datos.

6.1.5 Algolia

Firestore no nos permite realizar búsquedas de texto libre. Para poder hacer estas búsquedas hemos utilizado Algolia. Para vincular Algolia y Firestore hemos recurrido a una extensión. Además, para sincronizar los datos de nuestra base de datos Firestore y Algolia hacemos uso de los triggers que hemos implementado usando Cloud Functions. En nuestro caso, estas búsquedas las hemos llevado a cabo a la hora de buscar productos existentes para añadirlos a nuestras listas de productos. Gracias a las Cloud Functions, conseguimos que, cuando modifiquemos nuestra colección Productos, también se modifiquen los datos que tenemos almacenados en Algolia.

```
AlgoliaQuery algoliaQuery = algoliaInstance
    .query(query);
AlgoliaQuerySnapshot snapshot = await algoliaQuery.getObjects();
final rawHits = snapshot.toMap()['hits'] as List;
final hits = List<CloudProducts>.from(rawHits.map((hit) => CloudProducts
    .fromJson(hit, _selectedCategories)));
if (hits.isNotEmpty && hits.elementAt(0).productId != ''){

  _productsList = hits;

} else {
  _productsList.clear();
}
```

Figura 6.8: Algolia

En este código, la query son los caracteres por los que vamos a buscar los productos. En los hits, obtenemos todos los productos que estén almacenados en Algolia y que concuerden con nuestra query. El atributo *selectedCategories* es empleado para filtrar por categorías.

6.2 Firestore Security Rules

Se trata de un conjunto de condiciones que nos ayudan a establecer qué usuarios pueden acceder a los datos almacenados en Firestore. Estas condiciones se aplican a todas las peticiones a nuestra base de datos. Un aspecto que debemos tener en cuenta antes de trabajar con las Security Rules es que estas no son filtros. De esta manera, si queremos obtener las listas de la compra de las cuales un usuario es administrador, no podemos establecer una regla que prohíba el acceso a todas las listas de la compra en las que el identificador del usuario no coincida con el identificador del atributo administrador. Para conseguir esto tendremos que hacer un *where* en nuestro código para filtrar las listas. Hay dos tipos de security rules: **read** y **write**. Las *read* se dividen en: get y list. Las *write* se dividen en: create, update, delete. get nos otorga acceso a un documento, mientras que list nos confiere acceso a un conjunto de documentos. Por su parte, el atributo **request** representa la petición del usuario para leer o escribir datos en Firestore. Asimismo, *request* contiene información sobre el estado de autenticación del cliente, los datos que se leen o escriben y dónde se localizan dichos datos en Firestore. Por último, el atributo **resource** representa el documento al que estamos intentando acceder.

```
match /databases/{database}/documents {


  match /shoppingLists/{shoppingListId=**} {
        allow read: if isSignedIn() && request.auth.uid in resource.data.users;
        allow update, delete: if isSignedIn() && isOwnerId(resouce.data.ownerId);
        allow create: if isSignedIn();
  }

  function isSignedIn(){
      return request.auth != null;
  }

  function isOwnerId(ownerId){
      return request.auth.uid = ownerId;
  }
```

Figura 6.9: Security rules

Con el primer *match* hacemos referencia a la base de datos Firestore, indicando que estas reglas se aplican a toda la base de datos. Disponemos de dos funciones: *isSignedIn* e *isOwnerId*. Con *isSignedIn* comprobamos si el usuario está autenticado. Para ello, utilizamos el *request*, que contiene información del cliente, y comprobamos que no sea nulo. Con *isOwnerId* verificamos que el identificador del cliente sea igual al que hemos introducido. En el segundo *match* estamos aplicando esas reglas de seguridad a la colección de las lista de la compra. Al introducir **, indicamos que estas reglas se aplican a todas sus subcolecciones. Las reglas aplicadas en las subcolecciones sobreescriben a las de la colección predecesora. En este segundo *match* solamente permitimos que un usuario cree listas de la compra si está autenticado. Un usuario solamente puede actualizar o borrar las listas de la compra de las que es administrador. Para ello, es necesario introducir el identificador del administrador, el cual almacenamos en el *resource.data.ownerId*, y compararlo con el del usuario autenticado. Un usuario solamente debe poder leer las listas de la compra de las que forma parte. Por lo tanto, en el *allow read* comprobamos que esté autenticado y que el *userId* del usuario autenticado se encuentre dentro de la lista de usuarios que pertenecen a dicha lista de la compra.

6.3 BLoC

En el BLoC en la mayor parte de métodos hemos seguido el mismo proceso: obtenemos los atributos del evento que ha ocurrido, llamamos al método del repositorio con dicho método y actualizamos el estado con los nuevos datos en caso de haberlos.

Los únicos que hemos hecho de forma algo diferente son los relacionados con *Stream<Iterable>*.

```
on<ProductSubscriptionRequestedEvent>((event, emit) {
  String shoppingListId = event.shoppingListId;
  String productListId = event.productListId;
  _productsInShoppingListSubscription = provider.getProductsFromProductList(
      shoppingListId: shoppingListId,
      productListId: productListId
  )
      .listen((products) => add(UpdateProductStreamEvent(products)) );
});
on<UpdateProductStreamEvent>((event, emit){
  emit(ProductsStreamState(productList: event.products));
});
```

Figura 6.10: BLoC Products

En este caso, primero obtenemos los valores del evento que capturamos. Después, introducimos esos valores en el método de nuestro repositorio. El método *listen* llama a una función callback, en la cual obtenemos los productos del Stream y modificamos el estado al llamar a *addUpdateProductStreamEvent*. Esto nos permite mantener los cambios en tiempo real, ya que el método *getProductsFromProductList*, al estar vinculado con Firestore, nos devolverá nuevos datos cada vez que se produzca un cambio en Firestore. Así, conseguimos que se actualice el estado con esos nuevos datos y, por ende, la interfaz.

6.4 Widgets

Como se indicó al final del capítulo de diseño, cuando hablamos de la interfaz, hemos creado nuestros propios widgets para poder reutilizarlos posteriormente. Uno de los widgets que hemos personalizado ha sido el de slidable:

```dart
@override
Widget build(BuildContext context) {
  return Slidable(
      startActionPane: ActionPane(
        motion: const StretchMotion(),
        children: [
          SlidableAction(
            borderRadius: BorderRadius.circular(5),
            backgroundColor: Colors.green,
            foregroundColor: Colors.white,
            onPressed: (BuildContext context) {...},
            icon: isAdd ? Icons.remove_shopping_cart : Icons.add_shopping_cart,
            label: isAdd ? 'add' : 'remove',
          ),  // SlidableAction
          SlidableAction(
            borderRadius: BorderRadius.circular(5),
            backgroundColor: Colors.blue,
            foregroundColor: Colors.white,
            onPressed: (BuildContext context) {...},
            icon: Icons.edit,
            label: 'edit',
          )  // SlidableAction
        ],
      ),  // ActionPane
      child: child,
    endActionPane: ActionPane(...),);  // ActionPane, Slidable
}
```

Figura 6.11: Slidable widget

Este widget recibe los datos necesarios para poder añadir o quitar un producto de/a la cesta, borrarlo o actualizarlo. Además, recibe un boolean que nos indica si el producto está presente en la cesta, también recibe un child que es otro widget. En este widget que hemos creado nosotros tenemos 3 Slidable Action. Si deslizamos a la derecha, podremos ver un Slidable en azul para actualizar y uno en verde para añadir o quitar de la cesta, esto depende de si el boolean mencionado antes es true or false. Si deslizamos a la izquierda podremos ver un Slidable rojo, este nos permite borrar el producto de forma permanente.

Capítulo 7

Pruebas

Hemos decidido dividir las pruebas según como hemos implementado dichas pruebas. En nuestro caso tenemos 3 tipos de pruebas: las pruebas que usan el Firebase Local Emulator, las pruebas que utilizan mock para Firestore y las pruebas de aceptación.

7.1 Firebase Local Emulator

Hemos utilizado el Firebase Local Emulator para testear nuestras security rules. De esta forma, hemos comprobado si se cumplían las condiciones establecidas y si solamente podían acceder a las colecciones los usuarios que nosotros queríamos que accedieran. Para ejecutar estos test hemos utilizado Mocha [24]. Estos test son test de integración, ya que comprobamos la interacción entre las security rules y el Firebase local emulator.

```
it('Create and read shopping lists', async () => {
    const db = firebase.initializeTestApp({ projectId: MY_PROJECT_ID, auth: testUser.uid })
                       .firestore();

    const docRef = db.collection(shoppingListCollection).doc(shoppingListId);
    await firebase.assertSucceeds(
            docRef.set({name: shoppingListName, imageURL: '', users: [testUser.uid, userId1, userId2],
            ownerId: testUser.uid })
    );

    const user1 = firebase.initializeTestApp({
      projectId: MY_PROJECT_ID,
      auth: { uid: userId1 },
    }).firestore();
    await firebase.assertSucceeds(user1.doc(docRef.path).get());

    const user3 = firebase.initializeTestApp({
        projectId: MY_PROJECT_ID,
        auth: { uid: 'user3' },
    }).firestore();
    await firebase.assertFails(user3.doc(docRef.path).get());

    const userNotAuthenticated = firebase.initializeTestApp({
        projectId: MY_PROJECT_ID,
    }).firestore();
    await firebase.assertFails(userNotAuthenticated.doc(docRef.path).get());
});
```

Figura 7.1: Security rules test

En este ejemplo 7.1, comprobamos si podemos crear nuevas listas de la compra y acceder a las listas a las que pertenezcamos. Para ello, primero, en *db*, inicializamos la base de datos y nos conectamos como un usuario, en este caso *testUser*. Una vez hecho esto, intentaremos crear una nueva lista de la compra usando el método set. Hemos preferido no utilizar aquí nuestros propios métodos de la lógica de negocio, para separar los diferentes test. Esta operación debería tener éxito, ya que el usuario está autenticado y este es el único requisito para crear una lista de la compra. Cabe destacar que esta lista de la compra tiene a otros 2 usuarios, aparte del usuario que la creó.

La siguiente comprobación que llevaremos a cabo es verificar que un usuario puede visualizar las listas de la compra de las cuales forma parte. En este caso, el usuario con identificador *userId1* forma parte de los usuarios que conforman la lista de la compra. Por lo tanto, esta comprobación debería ser exitosa. La tercera comprobación debería fallar, debido a que el usuario con *userId3* no está en el array users de la lista de la compra. La cuarta comprobación también debería fallar, debido a que no hemos autenticado al usuario.

7.2 Mock test

En nuestro caso, para realizar mocks hemos utilizado FakeFirebaseFirestore. FakeFirebaseFirestore funciona exactamente igual que una base de datos Firestore, pero mantendrá nuestros datos en memoria en vez de almacenarlos en la nube.

```
  FirebaseFirestore firestore = FakeFirebaseFirestore();
  FirebaseCloudFirestoreProvider firebaseCloudFirestore =
      FirebaseCloudFirestoreProvider(firestore: firestore);
  await firebaseCloudFirestore.createNewShoppingList(ownerUserId: userId1, name: shoppingListName,
  imageURL: '', users: [], username: username, userImageURL: '');

  Iterable<CloudShoppingList> shoppingLists =
      await firebaseCloudFirestore.getShoppingList(ownerUserId: userId1).first;
  expect(shoppingLists.first.name, shoppingListName);
  expect(shoppingLists.length, 1);

  await firebaseCloudFirestore.deleteShoppingList(
      shoppingListId: shoppingLists.first.shoppingListId);
  Iterable<CloudShoppingList> shoppingLists2 = await firebaseCloudFirestore
      .getShoppingList(ownerUserId: userId1).first;
  expect(shoppingLists2.length, 0);
});
```

Figura 7.2: Fake Firebase Firestore

En este caso, inicializamos la base de datos FakeFirebaseFirestore. A continuación, indicamos a la clase donde tenemos los métodos de nuestra lógica de negocio, que queremos que

se ejecute utilizando FakeFirebaseFirestore. Posteriormente, creamos una lista de la compra llamando al método de la lógica de negocio. Seguidamente, obtenemos las listas de la compra de las que formamos parte y comprobamos si coinciden los valores obtenidos con los esperados. Después, borramos la lista de la compra y comprobamos que la longitud del iterable es uno menos que antes. Si esto se cumple, la lista de la compra ha sido borrada con éxito. Estos test son test unitarios, en los que vamos probando las diferentes funcionalidades de nuestro código.

7.3 Pruebas de aceptación

En estas pruebas comprobamos que todas las funcionalidades responden como deberían en un entorno real. Con esto hemos comprobado factores como, por ejemplo, la visibilidad de las diferentes pantallas que conforman nuestra aplicación, la velocidad (un aspecto importante al estar trabajando con datos en tiempo real) o la facilidad de uso, comprobando si la aplicación era lo suficientemente intuitiva. Estas pruebas son las que más hemos realizado a lo largo de nuestra aplicación. Esto se debe a que, al trabajar con Flutter y Android Studio, teníamos la opción de que el código que modificáramos se mostrase rápidamente, sin necesidad de volver a ejecutar todo el código.

Capítulo 8

Planificación y evaluación de costes

En este capítulo se explica la planificación estimada de cada Sprint y los diferentes motivos que llevaron a que la planificación real fuera diferente de la estimada. Al final se muestra un diagrama de Gantt con ambas duraciones. También se estiman los costes según las horas dedicadas a cada Sprint y a otro tipo de gastos, como los gastos materiales.

8.1 Sprints

A pesar de que la duración de los Sprints se mida en semanas, debido a la dificultad de algunos Sprints, se han hecho más horas por semana de las esperadas.

8.1.1 Sprint 0

Este Sprint tenía una duración estimada de 4 semanas. La duración real fue de 5 semanas, ya que en este Sprint fue donde vinculamos la aplicación con Firebase y, al no estar familiarizados con Firebase, tuvimos ciertos problemas al inicio.

8.1.2 Sprint 1

Este Sprint tenía una duración estimada de 4 semanas al igual que el anterior. Sin embargo, la duración de este Sprint fue de casi 6 semanas. Esto se debe a que no estábamos familiarizados con Firestore y la programación asíncrona. Así, la curva de aprendizaje fue más alta de lo que esperábamos. En concreto, tuvimos problemas a la hora de obtener las listas de la compra debido a las Security Rules. Además, también requerimos más tiempo del esperado para realizar las pruebas de estas Security Rules.

8.1.3 Sprint 2

Este Sprint fue el más largo del proyecto. Estimamos 4 semanas, pero tuvo una duración de 6 semanas y media. En este caso, habíamos estimado que llegados a este Sprint ya estaríamos más familiarizados con Firestore y los casos de uso serían realmente similares a los anteriores. Sin embargo, debido al uso de Cloud Functions, encontramos bastantes problemas, ya que tuvimos que comprar un plan de prepago y hacer una nueva configuración en nuestro proyecto para poder utilizar estas Cloud Functions.

8.1.4 Sprint 3

Este Sprint tenía una duración estimada de 2-3 semanas. En este caso sí que cumplimos la estimación, ya que la duración real de este Sprint fueron 2 semanas. Esto se debe a que obtener los productos de la cesta era similar a obtener los productos de las listas de productos. Con todo, marcar y desmarcar productos nos llevó ligeramente más de lo esperado. Sin embargo, una vez hecho el de marcar, el de desmarcar era prácticamente idéntico, por lo que no tuvo demasiada dificultad.

8.1.5 Sprint 4

Este Sprint tiene una duración estimada de 2-3 semanas. La duración final fue de 3 semanas, por lo que cumplimos la previsión. La mayor complicación en este caso fue a la hora de finalizar la compra a través de una Cloud Function. Asimismo, también tuvimos algún problema con las security rules a la hora de ver los detalles del historial.

8.1.6 Sprint 5

En este Sprint estimamos una duración de 2-3 semanas. La duración real fueron 2 semanas. Si bien tuvimos algún problema a la hora de utilizar el Cloud Storage y el patrón BLoC, al tratarse solamente de dos historias de usuarios, pudimos realizarlas en las 2 semanas previstas.

8.1.7 Sprint 6

Este Sprint tenía una duración estimada de 2-3 semanas. Finalmente, la duración real fue de 4 semanas. Esto se debe a que para crear los nuevos productos utilizamos un buscador por nombre, pero Firebase no nos permite realizar este tipo de búsquedas. Por lo tanto, para conseguir esto tuvimos que utilizar una extensión llamada Algolia, lo cual incrementó la duración debido a las dificultades que encontramos para vincularla con nuestro proyecto. Además, para utilizar Algolia tuvimos que crear varias Cloud Functions, que eran triggers que reaccionaban a modificaciones en nuestra colección productos. Así, cuando modificábamos la colección

productos, se tenía que activar una función que modificase también Algolia. Esto fue lo que principalmente aumentó la dificultad de este Sprint.

8.1.8 Sprint 7

Este Sprint tenía una duración estimada de 2-3 semanas. La duración real fueron 5 semanas. El motivo detrás de este retraso fue que pensamos que Firebase tenía un servicio que nos permitía enviar mensajes a otros usuarios y almacenarlos como si fuera una colección. Después de estar un tiempo buscando este servicio, al final decidimos implementar nosotros una alternativa. La razón por la que buscamos primero el servicio y no implementamos directamente la alternativa fue que queríamos intentar utilizar lo máximo posible Firebase, ya que nuestro proyecto gira en torno a ello.

8.1.9 Sprint 8

Este Sprint tenía una duración estimada de 2-3 semanas. La duración real fueron 4 semanas. El motivo de esta demora fue que tuvimos problemas a la hora de gestionar los usuarios que pertenecían a cada lista de productos y cómo añadir usuarios que estaban en la lista de la compra pero no en la lista de productos. La solución a esto fue crear una nueva colección en la lista de la compra en la que almacenáramos a estos usuarios.

8.1.10 Sprint 9

Para este Sprint estimamos 2-3 semanas. Sin embargo, este Sprint no llegó a las 2 semanas. El motivo por el que este Sprint se adelantó a lo previsto fue que la mayor parte de las historias de este Sprint eran pinceladas a funcionalidades anteriores. Por ejemplo, la parte de las categorías era simplemente añadir un filtro a la búsqueda de productos. En este Sprint, la historia más compleja fue la actualización de los datos del usuario, ya que esto modificaba el token del usuario y teníamos que volver a iniciar sesión.

8.2 Duración esperada

La duración estimada para la aplicación era de 33 semanas, para las cuales se esperaban 424 horas/hombre de esfuerzo (Figura 8.1). Para obtener este número de horas, estimamos un trabajo de 2.5 horas al día, 5 días a la semana durante 33 semanas. A esta estimación, le sumamos unas horas adicionales en función de la dificultad de la tarea. En cada Sprint realizamos las diferentes fases de un ciclo de vida tradicional: análisis, diseño, implementación y pruebas. Por lo tanto, si desglosamos el esfuerzo realizado en cada Sprint, nos queda 8.1:

Sprint	Análisis (h)	Diseño (h)	Implementación (h)	Pruebas (h)	Total (h)
0	8	12	33.5	6.5	60
1	9	10	31	6	56
2	9	8	42	5	64
3	4	5	19	2	30
4	2	4	22	2	30
5	0.5	2	30.5	2	35
6	2	2	34.5	3.5	42
7	3	4	34	4	45
8	4	5.5	27.5	5	42
9	0.5	1	17	1.5	20
	42	53.5	291	37.5	424

Tabla 8.1: Esfuerzo esperado

8.3 Duración real

Debido a los motivos comentados anteriormente y a problemas personales, la duración fueron 39 semanas. El esfuerzo real fue de 500 horas/hombre (Figura: 8.2). Intentamos mantener la misma línea que en la estimación: 2.5 horas al día, 5 días a la semana. No obstante, esto no se cumplió en todas las tareas de nuestro proyecto. Si desglosamos los Sprints en función de las diferentes fases del ciclo de vida, nos queda:

Sprint	Análisis (h)	Diseño (h)	Implementación (h)	Pruebas (h)	Total (h)
0	12	15	36.5	8.5	72
1	14	16.3	42.9	12.8	86
2	9	17	52.6	15.4	94
3	1.5	5.5	9.8	5.2	22
4	1	7.3	15.9	3.8	28
5	0.5	2	19	2.5	24
6	1	5.8	37.3	5.9	50
7	2.5	8.5	40.8	8.2	60
8	5	11.2	22.3	9.5	48
9	0.5	1.5	11.3	2.7	16
	47	90.1	288.4	74.5	500

Tabla 8.2: Esfuerzo real dedicado

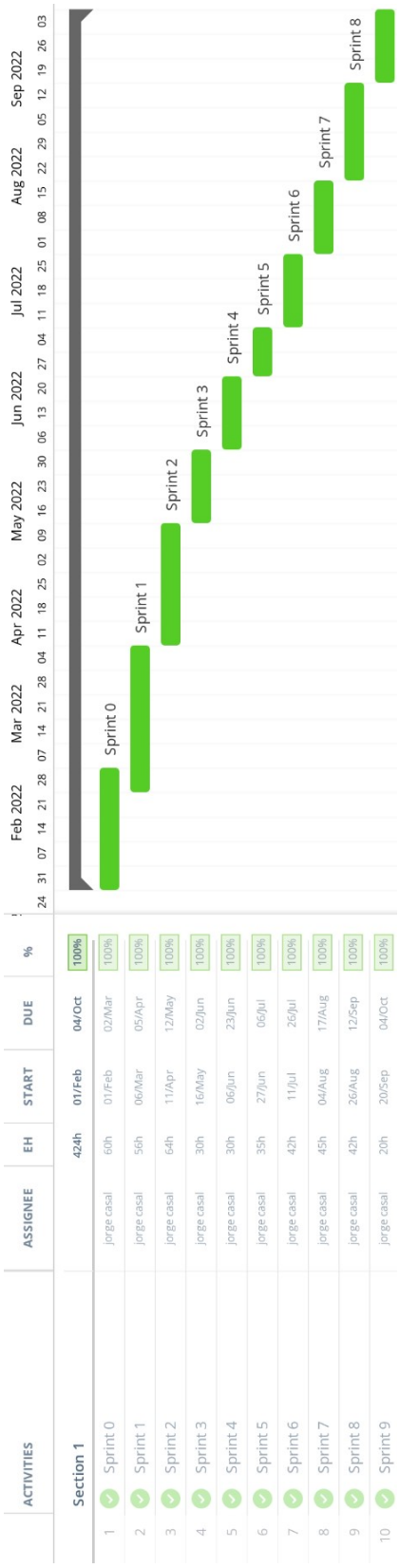

Figura 8.1: Diagrama de Gantt de la planificación estimada

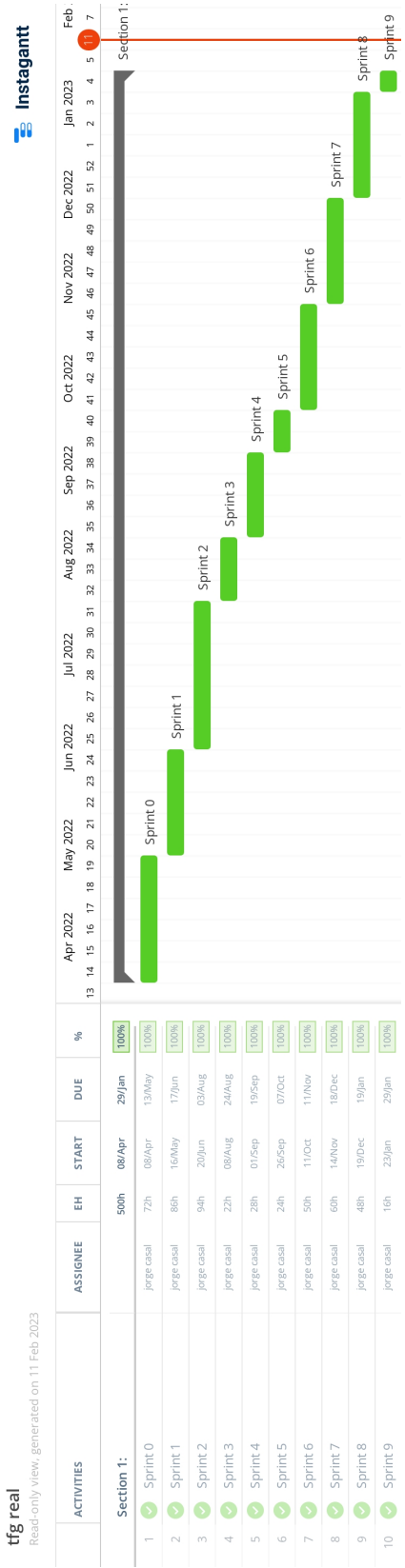

Figura 8.2: Diagrama de Gantt de la planificación real

8.4 Evaluación de costes

Para calcular el coste del proyecto, hemos tenido en cuenta las horas de trabajo, tanto del alumno, como del director. El alumno asume el trabajo de análisis, diseño, implementación y pruebas. El director asume el rol de jefe de proyecto.

Los salarios de las diferentes personas que trabajan en el proyecto se han obtenido a través de búsquedas en diferentes páginas web de empleo.

- Salario jefe de proyecto: 55€/hora.
- Salario analista junior: 18€/hora.
- Salario programador junior: 15€/hora.

Asimismo, se ha añadido el coste de material, que incluye el coste de un portátil valorado en 1000 €.

8.4.1 Coste previsto

Partiendo de la tabla 8.1 habíamos esperado un coste de:

Rol	€/h	Total horas	Total €
Jefe de proyecto	55	25	1.375,00
Analista junior	18	42	756,00
Programador junior	15	382	5.730,00
		449	7.861,00

Tabla 8.3: Coste esperado

A esto hay que sumarle el valor del portátil de 1000 €. Teniendo en cuenta que el portátil tiene un tiempo de vida útil de 4 años y que el proyecto se estimó que tendría una duración de 8 meses aproximadamente, el coste material sería 160 € aproximadamente. A esto tenemos que sumarle el IVA del 21 %, por lo que el coste final estimado serían 9.705,41 €.

8.4.2 Coste real

Debido a los imprevistos mencionados anteriormente, la duración del proyecto se incrementó en 6 semanas, esta demora provocó un incremento en el gasto final. A partir de los datos obtenidos en la figura 8.2 el coste real es:

Rol	€/h	Total horas	Total €
Jefe de proyecto	55	25	1.375,00
Analista junior	18	47	846,00
Programador junior	15	453	6.795,00
		525	9.016,00

Tabla 8.4: Coste real

Además, como este proyecto duró más de lo esperado, también consumió más vida útil que el proyecto anterior, aproximadamente 9 meses. Por lo tanto, el coste material es de 190 €. Si a todo lo anterior le sumamos el IVA del 21 %, obtenemos un resultado final de 10.909,36€.

Capítulo 9

Conclusiones del proyecto

Para terminar, abordaremos las conclusiones obtenidas y el posible trabajo futuro.

9.1 Objetivos del proyecto

El objetivo de este trabajo de fin de grado era desarrollar una aplicación para gestionar las listas de la compra entre múltiples usuarios. Para ello, queríamos que nuestra aplicación permitiese a los usuarios compartir listas de la compra. Además, dentro de estas listas de la compra habría otras listas, que serían las listas de productos, a las cuales añadiríamos los productos. Todo este proceso de añadir productos, queríamos que fuera en tiempo real, para que así no hubiera necesidad de andar recargando la aplicación con cada cambio que hiciéramos. Para ello, decidimos utilizar Firebase, ya que cuenta con una base de datos Firestore, que permite manejar datos en tiempo real, mediante el uso de Streams.

Otro de los objetivos de la aplicación, aparte de que almacenase productos, era el manejo de múltiples usuarios. Queríamos que las listas de la compra tuvieran usuarios y que los usuarios que pertenecían a estas listas de la compra también pudieran formar parte de las listas de productos. Esto surgió debido a que, al compartir piso entre varios estudiantes, puede haber problemas a la hora de repartir los productos una vez comprados. Esto se debe a que, con el paso del tiempo, es posible olvidar quién ha comprado cada producto. Para evitar esto, se decidió crear listas de productos que tuvieran usuarios, además de elaborar un historial para las compras realizadas.

9.2 Lecciones aprendidas

A nivel académico, el motivo que nos han llevado a escoger todas estas tecnologías ha sido principalmente el apredizaje, ya que apenas habíamos trabajado con estas tecnologías, lo único con lo que se traía un conocimiento previo era Flutter y Javascript. Por lo tanto, lo

que buscábamos a la hora de hacer este proyecto no era simplemente hacer la aplicación, sino también aprender cómo funcionaban los servicios de la nube de Google. Por eso, decidimos escoger Firebase frente a otras opciones, como por ejemplo Amazon Web Services.

Además de las tecnologías, también hemos aprendido a desarrollar un proyecto desde cero y a establecer nosotros mismos lo que queríamos hacer, dándonos cuenta de las dificultades que esto conlleva. Durante la carrera, hemos realizado proyectos con compañeros, pero al hacer un proyecto desde cero de forma "individual" (contábamos con la ayuda del director, lo cual a la hora de guiarnos cuando no sabíamos muy bien que hacer fue extremadamente útil), nos permitió darnos cuenta de que hay que conocer muy bien cada parte del proyecto.

Los errores del proyecto fueron realmente necesarios, ya que gracias a ellos pudimos ver qué era lo que nos faltaba. Además, es mejor que nos demos cuenta de los errores aquí a hacerlo posteriormente en un entorno de trabajo real.

9.3 Líneas de trabajo futuras

En esta sección abordamos las diferentes funcionalidades que podemos implementar para mejorar nuestra aplicación.

9.3.1 Página web de administración

Es posible crear una página web en la que dispongamos de un usuario administrador que pueda crear nuevos productos para todos los usuarios, así como modificar o borrar los ya existentes. Además, también podría encargarse del manejo de usuarios. Ahora mismo, todo esto se realiza directamente desde la página web de Firebase. Es cierto que su interfaz es visualmente bonita y fácil de manejar, pero una interfaz personalizada para nuestro aplicación sería mejor. Esta página web se podría desarrollar utilizando o bien React, que es un framework con el que ya estamos familiarizados por uso durante la carrera, o bien Flutter web, para seguir con el mismo framework que en la aplicación móvil.

9.3.2 Control de gastos

También, se podría hacer que cada producto tuviera un precio. Una vez finalicemos la compra, las listas de productos tendrían el precio total de los productos que contienen, y los usuarios que forman parte de esa lista de productos se repartirían los gastos.

9.3.3 Chat

Se podría crear también un chat o un bloc de notas en la aplicación, donde los usuarios puedan escribir sus comentarios o hablar entre ellos. Si bien es cierto que para comunicarse

los usuarios de la aplicación probablemente usarían otras aplicaciones de mensajería, como por ejemplo whatsapp o telegram, nunca es malo ofrecer una alternativa.

9.3.4 Mapa de supermercados

Otra opción que deberíamos tener en cuenta es crear un mapa con los supermercados cercanos. De esta forma, cuando un usuario vaya a hacer la compra puede localizar dichos supermercados. Otra posible funcionalidad sería que, al seleccionar los supermercados, la aplicación nos redirija a su página web, en caso de tenerla. De este modo, podríamos ver los productos de ese supermercado. Asimismo, podríamos incluso asociar las compras en el historial a los diferentes supermercados, de forma que podamos indicar que la compra del viernes la hemos hecho en este supermercado concreto.

9.3.5 IA para escanear tiques

También es posible intentar crear una inteligencia artificial, que pueda escanear los productos del tique y modificar los datos del historial. De esta manera, al escanear el tique, podríamos, por ejemplo, asociar directamente los precios a los productos. Si bien es cierto que esta opción es muy compleja y costosa, es posible que en un futuro se pueda realizar.

Apéndices

Apéndice A

Manual de usuario

Aquí explicamos cómo utilizar la aplicación mediante capturas de pantalla. Hemos decidido explicar solamente las funcionalidades más importantes.

A.1 Pantalla ver mis listas de la compra

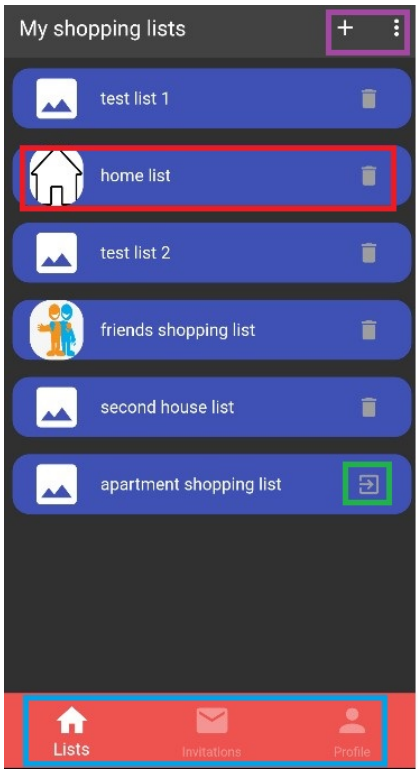

Figura A.1: Pantalla ver mis listas de la compra.

En esta pantalla A.1, podemos visualizar las listas de la compra de las que formamos parte. Como podemos observar en el recuadro rojo, una lista de la compra se compone de una imagen y un nombre a simple vista. También tiene un icono, con forma de papelera, para eliminar la lista de la compra, en caso de que seamos los administradores de dicha lista. En el recuadro verde, podemos ver otro icono, este se nos mostrará en las listas de la compra de las cuales formamos parte, pero en las que no somos administradores, este icono nos permite abandonar dicha lista de la compra. En la parte superior derecha podemos ver un recuadro morado con dos iconos, el icono del + nos permite crear listas de la compra, los tres puntos en vertical nos mostrarán un diálogo para cerrar sesión. En la parte inferior podemos ver un recuadro azul, aquí podemos ver las diferentes opciones entre las cuales navegar en nuestra aplicación. Disponemos de la opción de visualizar nuestras listas de la compra; la opción de ver las invitaciones a diferentes listas de la compra; y la opción de ver nuestro perfil para poder editarlo.

A.2 Pantalla crear lista de la compra

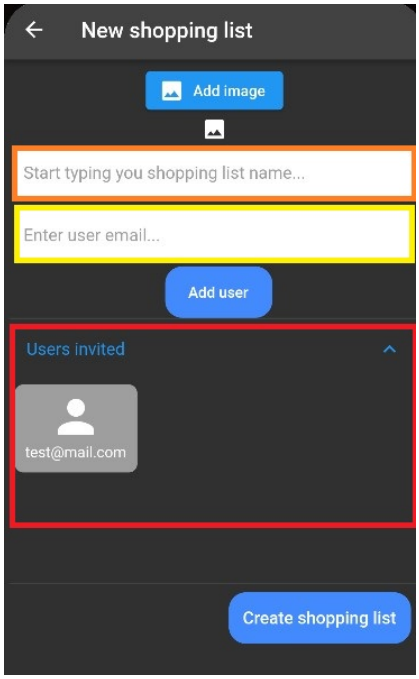

Figura A.2: Pantalla de crear lista de la compra

En esta pantalla A.2, tenemos un botón para añadir una imagen a la lista de la compra, esta imagen la seleccionamos de nuestra galería. En el recuadro naranja le indicamos el nombre de la lista de la compra. En el recuadro amarillo añadimos los correos de los usuarios que vamos

a invitar. Estos usuarios aparecen en el recuadro rojo. Una vez pulsemos el botón de create shopping list se creará la lista de la compra.

A.3 Pantalla de los detalles de la lista de la compra

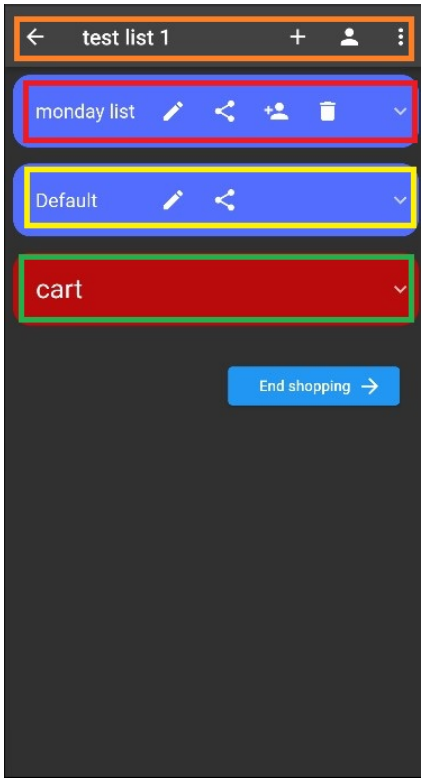

Figura A.3: Pantalla de ver los detalles de la lista de la compra.

En esta pantalla A.3, podemos ver cuatro recuadros de colores diferentes.

En la parte superior, en el recuadro naranja, tenemos el nombre de la lista de la compra. Además contamos con cuatro opciones, de izquierda a derecha: la flecha hacia atrás nos permite volver a la pantalla de ver nuestras listas de la compra; el icono del + nos permite crear nuevas listas de productos; el icono del usuario nos permite gestionar los diferentes usuarios que forman parte de nuestra lista de la compra, es decir, invitar a nuevos usuarios, además en caso de ser administrador, también se puede eliminar a los actuales; y los tres puntos en forma vertical nos muestran un diálogo para ver el historial.

En el recuadro en rojo tenemos la lista de productos, esta lista consta de un nombre, cuatro opciones y un icono para desplegar la lista de productos y poder ver así sus productos. Las opciones son de izquierda a derecha: el lápiz nos permite modificar el nombre de la lista de productos; el icono de compartir hace posible importar la lista de productos a otra lista de la

compra, al hacer esto se crea una copia de la lista de productos actual en la lista de la compra seleccionada, esta compra incluye los productos de la lista de productos y el usuario de esa lista es el usuario que la copió; el icono de añadir usuarios nos permite gestionar los usuarios de esa lista de productos; y el icono de la papelera posibilita eliminar dicha lista de productos.

En el recuadro amarillo podemos ver la lista de productos por defecto. El lápiz y el icono de compartir tienen las mismas funcionalidades que se han indicado antes. No se ha mostrado ni el icono de los usuarios, ni el de la papelera, porque esta lista siempre va a contener a todos los usuarios de la lista de la compra, ya que es una lista común y no se puede eliminar.

Por último, en el recuadro verde, podemos ver la cesta. No es posible ni modificar su nombre ni eliminarla, solo manejar sus productos. A continuación, tenemos un botón 'End shopping' que nos permite finalizar la compra, vaciando de esta forma la cesta y creando un registro en el historial con los productos que haya en la cesta clasificados según las listas de productos a las que pertenecen.

A.4 Pantalla de ver los detalles de una lista de productos

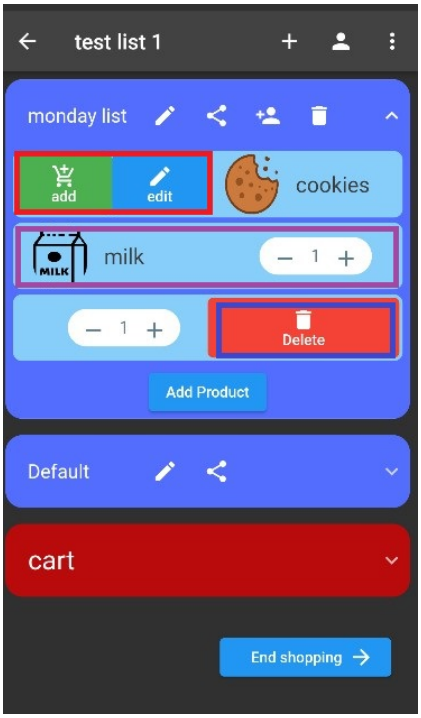

Figura A.4: Pantalla de ver los detalles de una lista de productos

En esta imagen, podemos ver una lista de productos que contiene productos. En este caso tiene 3 productos. En el recuadro morado podemos ver la vista estándar de un producto. De

un producto a simple vista podemos ver: su imagen, su nombre y una cantidad asociada al producto. Después tenemos 2 opciones: deslizar a la derecha o deslizar a la izquierda. Si deslizamos a la derecha podremos ver las opciones indicadas en el recuadro rojo. Tenemos una opción para añadir producto a la cesta, y otra opción para editar los detalles de un producto. Si deslizamos a la izquierda podremos ver la opción indicada con un recuadro azul oscuro. Esta opción nos permite borrar el producto de forma permanente. Los productos de la cesta se visualizan igual que los de las listas de productos, la única diferencia es que en vez de la opción de añadir a la cesta es una opción de retirar de la cesta. Después de todos los productos tenemos una opción add product, que nos permite añadir productos a la lista de productos.

A.5 Pantalla de añadir usuarios a lista de la compra

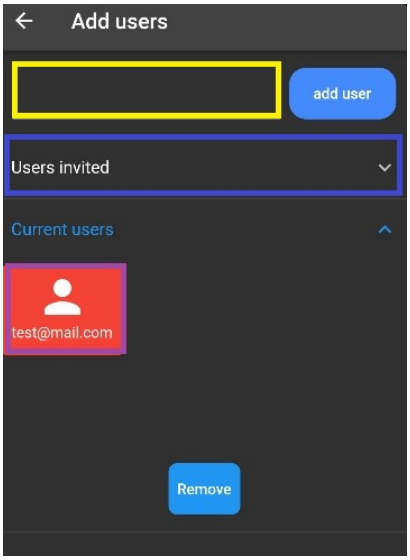

Figura A.5: Pantalla de añadir usuarios a lista de la compra

En esta pantalla tenemos 3 recuadros: uno amarillo, uno azul oscuro y uno morado.

En el amarillo introducimos el correo del usuario que queremos añadir a nuestra lista de la compra. Cuando pulsemos el botón de add user le enviaremos una invitación a nuestra lista.

En el azul aparecen todos los usuarios que hemos invitado a la lista de la compra pero todavía no han aceptado la invitación.

En el desplegable denominado current users tenemos a los usuarios que están en la lista de la compra. Un usuario administrador puede seleccionar a estos usuarios para expulsarlos de la lista de la compra.

A.6 Pantalla de ver y gestionar los usuarios de una lista de productos

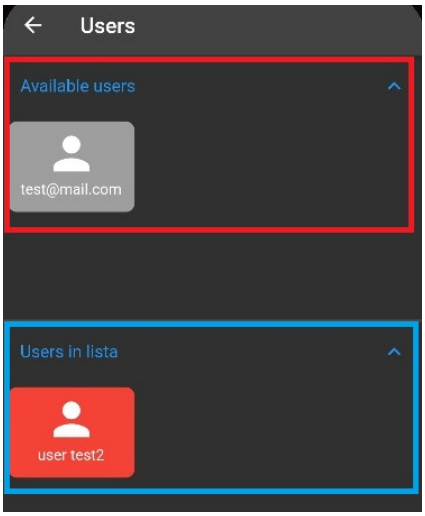

Figura A.6: Pantalla de ver y gestionar los usuarios de una lista de productos

En esta pantalla A.6 podemos ver como añadir y quitar usuarios de una lista de productos. En el recuadro rojo podemos ver los usuarios disponibles, es decir los que pertenecen a la lista de la compra pero no a la de productos. En el recuadro azul podemos ver los usuarios que pertenecen a la lista de productos. Si pulsamos sobre uno de los 2 usuarios se moverá a la otra lista. Es decir, si queremos añadir un usuario de la lista de usuarios disponibles a la lista de productos, simplemente tenemos que seleccionar al usuario en cuestión. Y lo mismo si queremos eliminar a un usuario de la lista de productos.

A.7 Pantalla de invitaciones

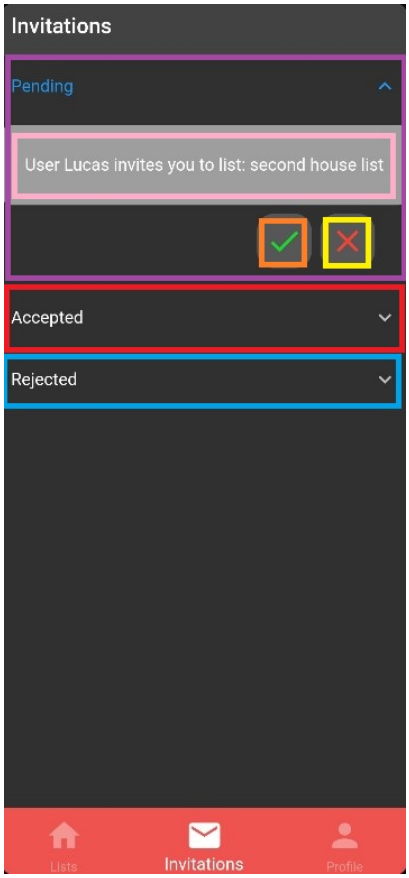

Figura A.7: Pantalla de ver invitaciones

En esta imagen A.7 podemos ver las diferentes invitaciones que ha recibido un usuario.

En morado podemos ver las invitaciones que están pendientes de resolución. Estas invitaciones las podemos aceptar pulsando el botón del recuadro naranja, o rechazar pulsando el botón del recuadro amarillo. En el recuadro rosa podemos ver la invitación en sí, quien nos envía la invitación y a que lista de la compra nos invita. Las invitaciones que ya hemos aceptado se muestran en el desplegable indicado en rojo, las rechazadas en el desplegable del recuadro azul.

A.8 Pantalla de buscar productos

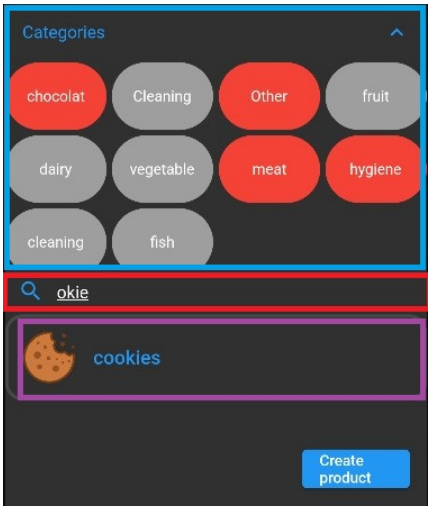

Figura A.8: Pantalla de buscar productos

Para añadir un producto a la lista de productos tendremos un buscador para buscar los productos por nombre y categoría. En el recuadro azul podemos ver las diferentes categorías, de las cuales podemos seleccionar más de una. En el recuadro rojo introducimos el nombre del producto que queremos añadir. En el recuadro morado nos aparecen las sugerencias a partir del nombre y las categorías indicadas. Por último en la esquina inferior derecha tenemos un botón para crear nuestros propios productos.

A.9 Pantalla de crear un producto

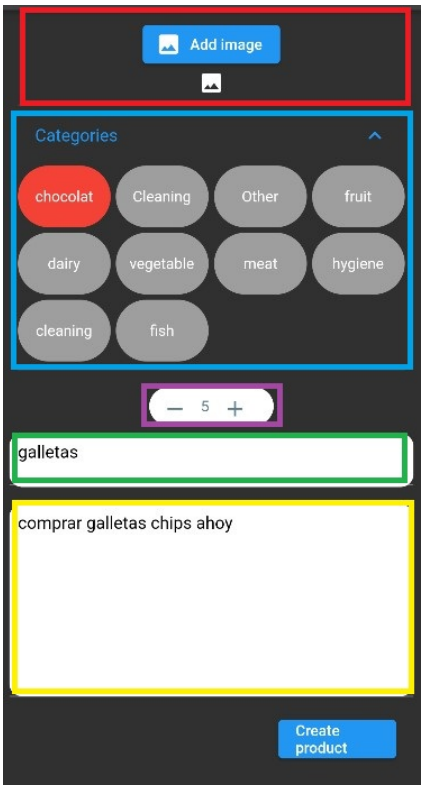

Figura A.9: Pantalla de crear producto

La imagen A.9 tiene 5 recuadros. El recuadro rojo nos permite añadir una imagen de nuestra galería al producto. En el recuadro azul indicamos la categoría a la que va a pertenecer este producto. En el recuadro morado indicamos la cantidad que queremos que tenga inicialmente ese producto. En el recuadro verde indicamos el nombre del producto. Y, en el recuadro amarillo su descripción.

A.10 Pantalla del historial

Figura A.10: Pantalla del historial

A la hora de visualizar el historial tenemos las diferentes entradas, como se puede ver en el recuadro rojo A.10, estas entradas tienen un nombre que se le dio al finalizar la compra y una fecha. En el recuadro morado se puede ver la opción para eliminar una entrada del historial. En el recuadro amarillo podemos ver la opción para ver los detalles de esa entrada del historial.

A.11 Pantalla de una entrada del historial

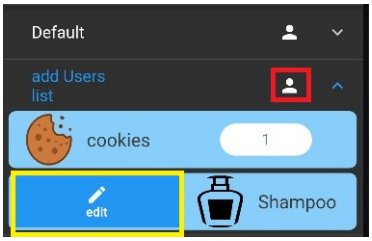

Figura A.11: Pantalla de una entrada del historial

En esta imagen A.11 podemos ver los detalles de una entrada del historial. Es similar a una lista de la compra, tiene varias listas de productos y cada una tiene unos productos. Con el icono marcado con el recuadro rojo podemos ver los usuarios que forman parte de esa lista de productos. Con la opción edit, indicada a través del rectángulo amarillo, podemos ver los detalles del producto para así editarlo.

Lista de acrónimos

APIs Application Programming Interface. 7

BLoC Business Logic Component. 29

HTTPS Hypertext Transfer Protocol Secure. 58

IDE integrated development environment. 10

iOS iPhone Operating System. 7

IVA impuesto al valor agregado. 75

JSON JavaScript Object Notation. 59

NoSQL Not Only Structured Query Language. 8

SDKs Software Development Kit. 9

Bibliografía

[1] "Softlist," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://bloygo.yoigo.com/apps-lista-de-la-compra/

[2] "Bring!" 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://isenacode.com/bring-la-app-definitiva-para-hacer-la-compra-en-el-super/

[3] "Listonic," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://listonic.com/es/

[4] "Dart," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://dart.dev/overview

[5] "Flutter," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://docs.flutter.dev/

[6] "Javascript," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Web/JavaScript

[7] "Node.js," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://nodejs.org/es/

[8] "Firebase," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://firebase.google.com/?hl=es

[9] "Firestore," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://firebase.google.com/docs/firestore

[10] "Firebase auth," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://firebase.google.com/docs/auth

[11] "Security rules," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://firebase.google.com/docs/firestore/security/overview

[12] "Cloud storage," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://firebase.google.com/docs/storage

[13] "Cloud functions," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://firebase.google.com/docs/functions

[14] "Firebase emulator," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://firebase.google.com/docs/emulator-suite

[15] "Algolia," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://www.algolia.com/developers/firebase-search-extension/

[16] "Android studio," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://developer.android.com/studio

[17] "Visual studio code," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://developer.android.com/studio

[18] "Git hub," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://github.com/

[19] "Firebase console," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://console.firebase.google.com/

[20] "Google drive," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://drive.google.com/

[21] "Overleaf," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://www.overleaf.com/

[22] "Balsamiq," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://balsamiq.com/

[23] "Bloc," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://bloclibrary.dev/#/

[24] "Mocha," 2023, consultado el 15 de febrero de 2023. [En línea]. Disponible en: https://mochajs.org/