Facultade de Informática
UNIVERSIDADE DA CORUÑA

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

Gestor social de flujo de tareas

Estudiante: Alejandro Díaz Pallas
Dirección: Fernando Bellas Permuy

A Coruña, febreiro de 2021.

Resumen

El objetivo de este proyecto consiste en el diseño e implementación de un sistema de gestión de tareas basado en tableros Kanban.
La aplicación debe permitir crear, editar o eliminar tableros, columnas y tarjetas, además de añadir usuarios a los tableros deseados para hacer posible la colaboración en un mismo tablero o incluso la sincronización de columnas entre tableros, sean de un mismo usuario o de diferentes usuarios.

La lógica de negocio es accedida a través de una API REST desarrollada en Node.js utilizando la tecnología serverless de Firebase Cloud Functions y persistiendo los datos en la base de datos Firestore, una base de datos NoSQL basada en colecciones.

El componente principal del proyecto es su frontend, una aplicación móvil escrita en Dart que usa el framework Flutter, a través del cual se puede desplegar tanto a iOS como a Android.

Abstract

This project's purpose is to design and implement a task management system based on Kanban.
The application must allow a user to create, edit or delete boards, columns and cards, besides adding users to the chosen boards, making collaboration in one specific board or even synchronizing columns between boards possible, belonging those boards to the same or different users.

A REST API developed in Node.js serves as an entrypoint to interact with the business logic, which is hosted in the serverless technology provided by the Firebase Cloud Functions and storing the application data in the Firestore database, a NoSQL collection based database.

The project's main part is the frontend, a mobile application written in Dart and using the framework Flutter, through which allows to deploy the application both on iOS and Android.

Palabras clave:

- Serverless
- Firestore
- Flutter
- Stream
- Tablero
- Columna
- Tarjeta

Keywords:

- Serverless
- Firestore
- Flutter
- Stream
- Board
- Column
- Card

Índice general

Índice de figuras

Índice de tablas

Capítulo 1
Introducción

La gestión de tareas es una problemática que se ha afrontado de diversas maneras a lo largo del tiempo. Desde el nacimiento del software se han creado soluciones digitales para aportar formas diferentes a las tradicionales de interactuar con tareas, proyectos, objetivos, etc.
Las herramientas software han avanzado desde meras notas en archivos de texto hasta aplicaciones locales completas y, por último, sistemas multiplataforma sofisticados de colaboración online. Debido a la popularidad de su uso concretamente en ámbitos de desarrollo de software, muchas herramientas facilitan el trabajo con metodologías de desarrollo como Scrum[11] o Kanban[12].

1.1 Kanban

Mientras que actualmente Kanban se suele asociar con la metodología de desarrollo de software, nació para cubrir una necesidad de organización de la manufactorización de vehículos en Toyota para mejorar la calidad de los productos, de los procesos y del flujo entre éstos.

Con el tiempo Kanban se ha visto introducido en las metodologías ágiles, tanto como una metodología en sí misma como en conjunto con otras, típicamente Scrum.

1.1.1 Funcionamiento

Kanban se basa en el uso de tableros que contienen columnas que, a su vez, contienen tarjetas.

- Un **tablero** (Figura 1.1) suele representar un proyecto, es decir, una agrupación de responsabilidades relacionadas, sujeto a un flujo de trabajo establecido.
- Una **columna** refleja el estado en el que se puede encontrar una o varias tareas dentro del flujo.

- Una **tarjeta** define una tarea.
- El flujo de trabajo es la secuencia de estados (columnas) que una tarea (tarjeta) debe seguir hasta el estado final representado en el tablero.

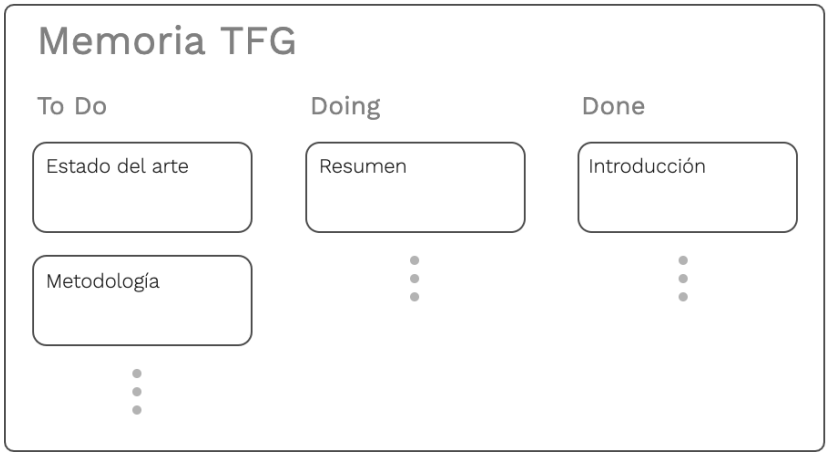


Figura 1.1: Ejemplo de un tablero Kanban para la redacción de una memoria de un TFG.

Entre las ventajas principales que aporta Kanban a un sistema de gestión de tareas se pueden citar:

- De un vistazo se puede ver fácilmente el estado de las tareas.
- Sigue un flujo intuitivo y que se puede usar tanto para problemas simples como para complejos.

La metodología Kanban tiene muchos otros componentes que forman parte esencial de su funcionamiento, pero no se entrará en detalle al respecto debido a que no forman parte del ámbito del proyecto ni de la aplicación, que no pretende realizar una implementación de Kanban en sí sino basarse en un subconjunto de sus características para aportar un uso sencillo e intuitivo.

1.2 Problema

Actualmente la mayoría de soluciones son simples pero cubren pocas necesidades o son muy completas pero, a su vez, complejas.

En general, encontrar una solución completa que permita un control extenso sobre el proyecto que se quiera abarcar recae en una configuración avanzada previa necesaria.
La mayoría de aplicaciones que realmente cubren unas necesidades por encima de las básicas son orientadas a negocio y, en consecuencia, están llenas de opciones y complejidades que, a

menos que se utilicen para proyectos con una envergadura que justifique el uso de la herramienta, sería más adecuado optar por una aplicación más sencilla y probablemente, menos costosa.
En caso de sólo utilizar un subconjunto de las posibilidades de la herramienta, sin embargo, tanto los usuarios como los administradores ven una gran cantidad de botones y opciones a su alcance que nunca usarán.

En el otro lado de la balanza se encuentran las aplicaciones que son fáciles de usar y que no mantienen una interfaz en exceso poblada, sino que muestran lo justo y necesario para interactuar con la misma en el momento adecuado.
Sin embargo, habitualmente las herramientas simples lo son porque ofrecen pocas opciones y raramente soportan formas de llevar más allá la colaboración con otros usuarios sin disponer de soluciones de terceros o tener que mantener la información en ubicaciones diferentes y sin forma de automatizar el cambio de ésta para cubrir distintas áreas del proyecto o proyectos.

De acuerdo con los escenarios observados, el menos cubierto es, por tanto, el de una solución que afronte una complejidad y potencia medias, sin tratar de ser un gestor para grandes proyectos pero que sea suficiente para hacer sencilla una colaboración medianamente compleja.

1.3 Objetivo

El objetivo del proyecto es la creación de un sistema de gestión de tareas que afronte una complejidad media, donde además de cubrir las necesidades básicas de cualquier sistema, como puede ser la toma de notas o el marcar una tarea como completada, debe posibilitar asignar tareas, comentar en las mismas y darles un estado acorde a su progreso.

La aplicación debe ser sencilla de utilizar y a la vez permitir enfrentarse a complejidades de cierta envergadura para evitar ser un estorbo a la hora de colaborar en múltiples proyectos.

Además, debido a la naturaleza colaborativa de la herramienta, la información presente en ésta debe reflejar en todo momento la información almacenada en la base de datos y reaccionar a los cambios que se produzcan en el ámbito en el que el usuario se encuentra en un momento dado.

Para evitar cuellos de botella y complejidades innecesarias, todos los usuarios tienen los mismos privilegios sobre los recursos a los que tienen acceso, buscando una visión de colaboración horizontal más que vertical.

La funcionalidad clave que permitirá darle un uso más complejo a la aplicación será la de **sincronizar columnas** entre diferentes tableros.

Para permitir modelar flujos de trabajo complejos entre distintos tableros de un mismo usuario o de usuarios diferentes, la aplicación permitirá sincronizar columnas de diferentes

tableros, tanto de un mismo usuario como de usuarios diferentes.

La sincronización se puede dar unidireccional o bidireccionalmente, es decir, desde una columna CA de un tablero TA se puede realizar una petición de sincronización a una columna CB de un tablero TB. Esta sincronización permite que:

- CA obtenga todas las tarjetas de CB desde ese punto en adelante, incluyendo las que CB ya tuviera en el momento de aceptar la sincronización
- La dirección opuesta al punto anterior, es decir, CB recibe las tarjetas de CA.
- Una sincronización completa, es decir, CB y CA contienen las mismas tarjetas y los cambios en una columna se ven reflejados en la otra.

La sincronización se explicará en la sección 3.2.

1.4 Visión global del sistema

El sistema (Figura 1.2) está compuesto de una aplicación móvil como frontend y una API REST[13] como backend que contendrá la lógica de negocio e interactuará con la base de datos.
Sin embargo, para permitir la reactividad de la aplicación ante cambios en la base de datos, también se abrirá una vía de comunicación entre la base de datos y el frontend sin pasar por el servicio REST. Esta comunicación será de sólo lectura, es decir, el frontend permanecerá atento a cambios que se produzcan en la base de datos para refrescar la vista en el momento en el que ocurran, pero no podrá realizar cambios en la base de datos, sino que delegará en el servicio REST para tal tarea.

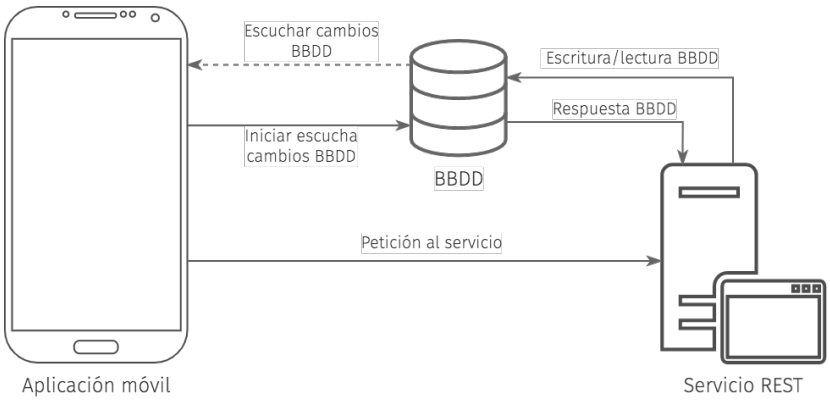


Figura 1.2: Arquitectura general del proyecto.

1.4.1 Servicio REST

El servicio REST, ilustrado en la figura 1.3, sigue una arquitectura en capas alterada para aprovechar la versatilidad de Javascript, pudiendo inyectar implementaciones diferentes o genéricas dependiendo de la necesidad del caso de uso concreto implementado.

Se ha utilizado el servicio Firebase Cloud Functions[4] usando Express[3] y Node.js[2] para el enrutamiento de las peticiones, aprovechando la flexibilidad que aporta para garantizar un mayor desacoplamiento del punto de entrada al servicio REST con las operaciones que están implementadas en el mismo.

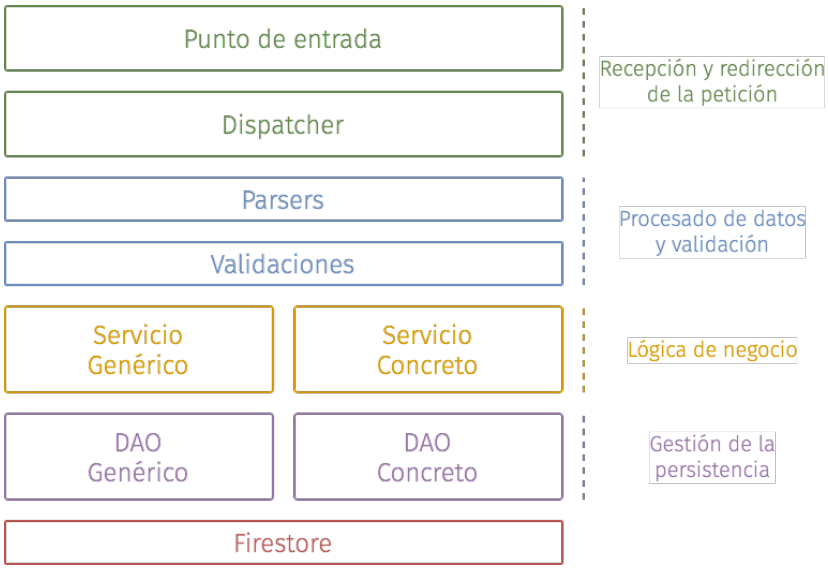


Figura 1.3: Vistazo de la arquitectura de capas del servicio REST.

1.4.2 Aplicación móvil

El frontend sigue una estructura que separa la lógica de los componentes puramente visuales, los cuales se agrupan por páginas o en un directorio de utilidades en caso de ser más genéricos.

La lógica y relación con tanto la base de datos como con el backend se realizará en unos componentes llamados BLoC, que se describirán en detalle en la sección 8.7.1. Estos componentes lógicos formarán la pieza que conecta los datos con los componentes visuales.

Por una parte, ante ciertos eventos como guardar los datos de un formulario, los componentes visuales se comunicarán con el componente lógico asociado que realizará una o varias peticiones al backend, que se encargará de realizar las operaciones de escritura de base de datos necesarias.

Por otra parte, cuando se carga una página, este evento es comunicado a uno o varios

BLoCs asociados a los componentes que haya en la vista para que realicen un inicio de escucha de cambios de la base de datos. De esta forma los componentes en la vista se mantienen actualizados con respecto a la información que haya en Firestore. Este flujo se ilustra en la figura 1.4.

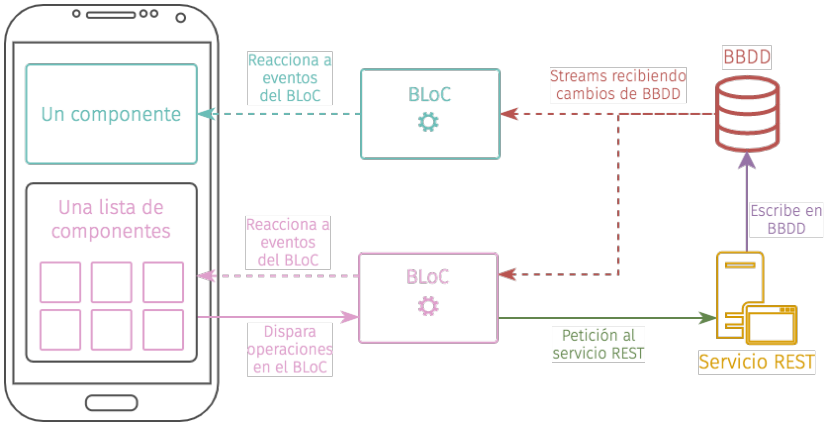


Figura 1.4: Visión general de la arquitectura basada en el uso del patrón BLoC con Flutter.

1.4.3 Firestore

Firestore[1] es una base de datos NoSQL[14] basada en colecciones. La base de datos es una buena opción debido a que las colecciones no mantienen unas relaciones muy estrechas en este proyecto y que no es necesaria la creación de DTOs muy distintos a los documentos que representan los objetos del modelo. Firestore permite interactuar con ella muy fácilmente mediante los SDKs provistos tanto para Dart como para Node.js, que además presentan una forma sencilla de estar atento a eventos de la base de datos.

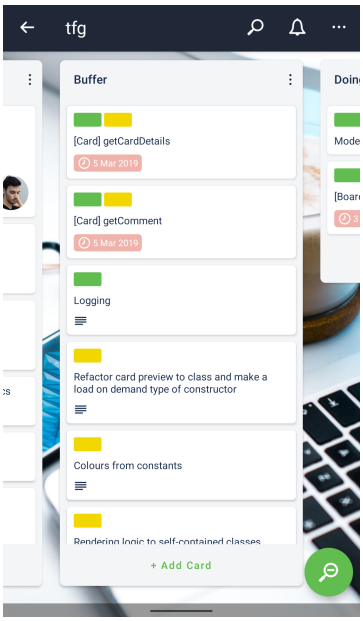
Capítulo 2

Estado del arte

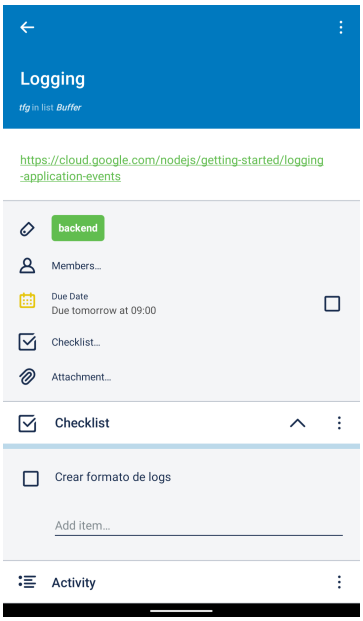
Actualmente no existen herramientas que provean las funcionalidades que se abordan en este proyecto, no sin ser necesario el paso por herramientas de terceros. No obstante, hay muchas otras aplicaciones con grandes similitudes o con otras funcionalides que se pueden utilizar para el mismo fin que las de esta aplicación.

2.1 Trello

Una de las aplicaciones más conocidas y similares a este proyecto es Trello[15].



(a) Vista de un tablero en Trello.



(b) Vista de una tarjeta en Trello.

También basado en Kanban, Trello permite que el usuario cree tableros, columnas y tarjetas, además de incluir a otros usuarios en los tableros para colaborar y asignarlos a las tarjetas.

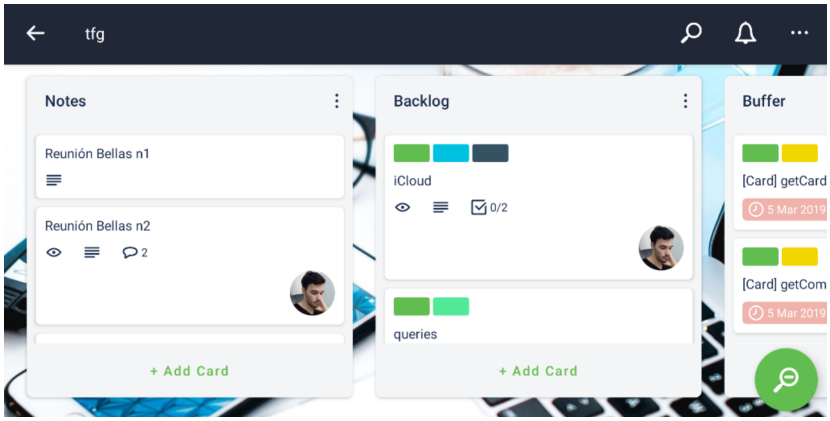


Figura 2.2: Vista de varias columnas en un tablero de Trello.

Es posible asociar etiquetas de colores con nombre a las tarjetas, aunque en la versión móvil sólo se ve el color de las mismas desde la previsualización de las tarjetas en la vista del tablero tal y como se muestra en las figuras 2.1a y 2.2.

Cada tarjeta (Figura 2.1b) puede tener descripción, *checklists*, fecha de finalización de la tarea, miembros, etiquetas y actividad, donde se pueden ver los comentarios y cualquier movimiento que haya tenido la tarjeta.

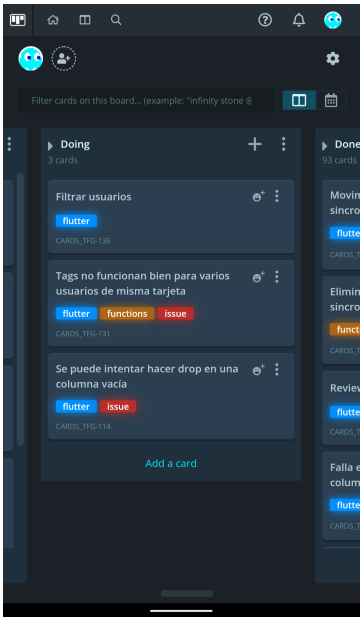
Trello intenta que la responsabilidad de gestiones más complejas recaiga en integraciones de terceros, llamados *power ups*, que interactúan con los tableros y columnas de los usuarios con diversas finalidades.

2.2 GitKraken Boards

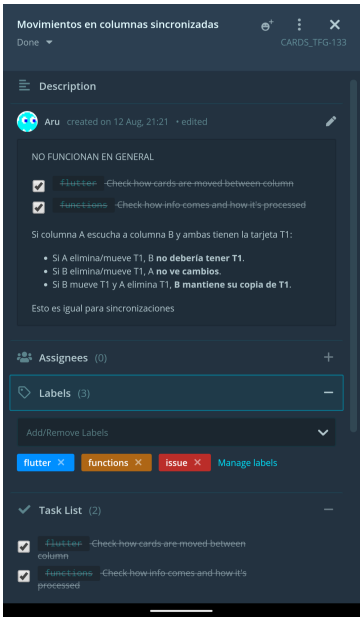
Recientemente ha surgido como alternativa a Trello la aplicación GitKraken Boards[16], también basado en Kanban pero que toma una aproximación asociada directamente al desarrollo de software debido a que es creada como aplicación asociada a GitKraken GUI, una interfaz gráfica para Git[8].

Mientras que Trello tiene más integraciones con aplicaciones de terceros, GitKraken Boards ofrece más funcionalidades de base y evoluciona más rápidamente, haciéndose con una base de clientes grande a pesar de tener un competidor directo que lleva en el mercado mucho más tiempo.

La apariencia es similar a la de Trello con respecto a la organización de los recursos, como se puede ver en las figuras 2.3a, 2.3b y 2.4.



(a) Vista de un tablero en GitKraken Boards.



(b) Vista de una tarjeta en GitKraken Boards.

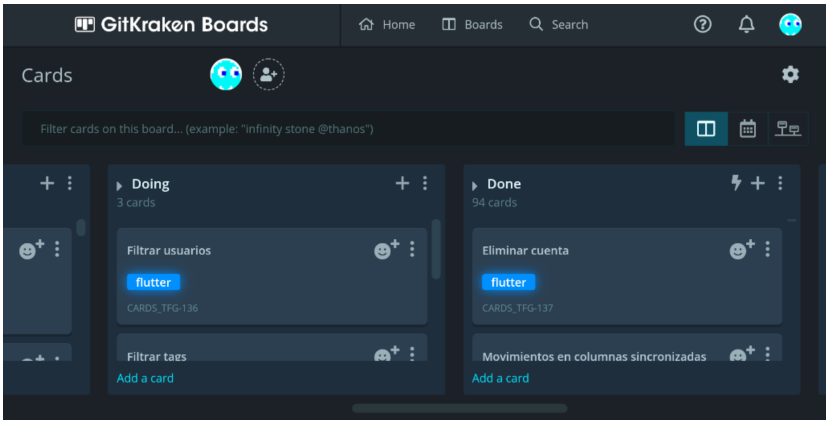
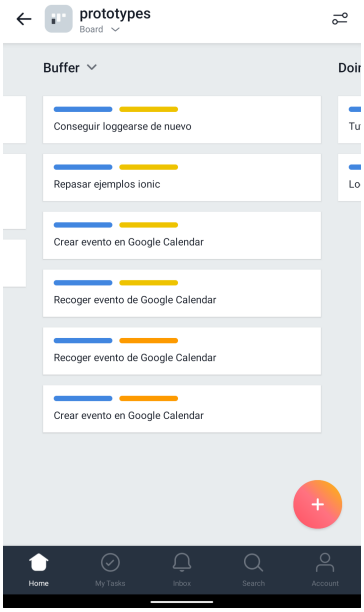


Figura 2.4: Vista de varias columnas en un tablero de GitKraken Boards.

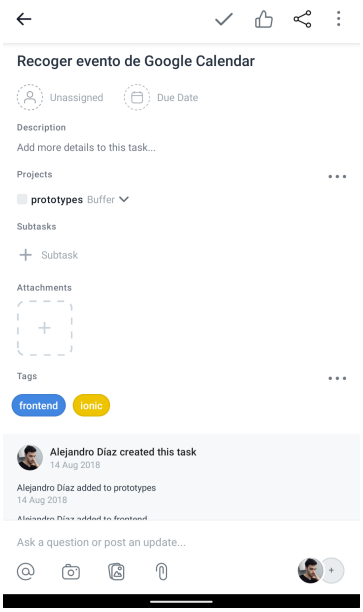
2.3 Asana

Asana ofrece múltiples perspectivas que permiten observar un mismo proyecto como una lista de tareas, un tablero Kanban, un calendario o una visión general del progreso.

Asana (Figuras 2.5a,2.5b y 2.6) es una aplicación muy establecida como herramienta de colaboración de equipos de desarrollo de software, ofreciendo un punto intermedio como herramienta simple pero de aspecto y utilidad corporativa.



(a) Vista de un tablero en Asana.



(b) Vista de una tarjeta en Asana.

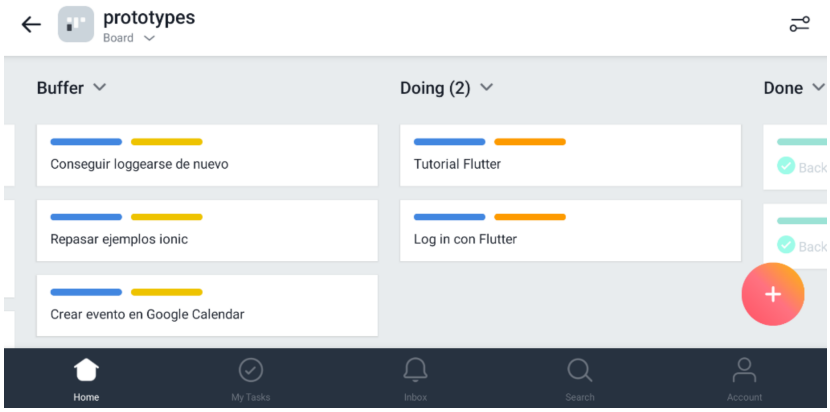


Figura 2.6: Vista de varias columnas en un tablero de Asana.

2.4 Jira

Una de las aplicaciones más conocidas y utilizadas para la gestión de tareas en el desarrollo de software es Jira[17].

Jira ofrece una perspectiva más profesional y completa, pero a cambio de ser más compleja. Comprende gestión de estado de las tareas, creación de subtareas, asignación, seguimiento de tareas, imputación de tiempo y muchas perspectivas diferentes que ofrecen distintos puntos de vista dependiendo de la finalidad del que lo use.

Jira además ofrece una integración con Trello dentro de la propia herramienta debido a

que Atlassian, la corporación a la que pertenece, adquirió Trello.
También permite la integración con repositorios Git mediante Bitbucket, también perteneciente a Atlassian.

A pesar de todas sus ventajas, Jira puede ser una solución excesivamente compleja para organizaciones pequeñas o proyectos independientes que no requieran de mucho control de permisos o de un seguimiento exhaustivo de tareas.

Capítulo 3

Funcionamiento

3.1 Modelo

Una vez entendidas las bases de la metodología Kanban en la que la aplicación se basa, y otras aplicaciones similares como las mostradas, se puede ahondar en qué es cada parte de la aplicación para comprender los siguientes puntos del documento.

En realidad, parte del interés de una aplicación de este tipo es la versatilidad que ofrece, por lo que se describirán los casos más habituales para los que se usarían los elementos de la misma, pero entendiendo que pueden ser utilizados de muchas otras formas.

3.1.1 Tablero

Un tablero suele representar un proyecto o conjunto de tareas relacionadas que han de atravesar un flujo de estados o un repositorio de tareas de distintos tipos.

En un tablero se pueden filtrar, mediante una búsqueda, las tarjetas que se quieren ver por texto, etiquetas o miembros de cada tarjeta.
Además, en un tablero puede haber uno o más miembros. Todos los miembros tienen los mismos permisos sobre el tablero. Cualquier miembro del tablero puede crear, eliminar y ver tanto columnas como tarjetas. Sin embargo, las etiquetas son personales de cada usuario. Si un usuario tiene la etiqueta *miEtiqueta* en una tarjeta y otro usuario no, el segundo usuario **no verá la etiqueta ni la podrá usar como criterio de búsqueda**.

Un tablero puede tener un número cualquiera de columnas, y cada columna puede tener un número cualquiera de tarjetas.

3.1.2 Columna

Una columna es normalmente utilizada como estado de las tareas que contiene o para agruparlas en un tipo de tareas. Por ejemplo, un usuario podría crear unas columnas en su

tablero para representar *Sin empezar*, *En curso* y *Terminadas* y otra persona podría tener un tablero que discrimina las columnas como de tipos de proyecto: *Proyecto Web*, *Proyecto Móvil*.

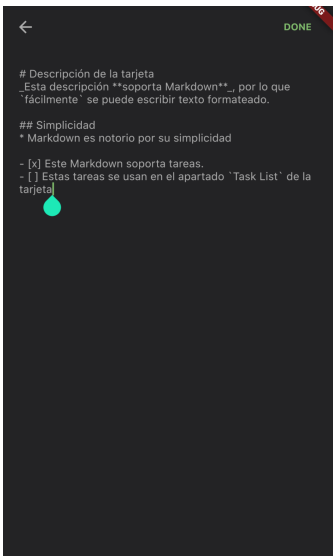
3.1.3 Tarjeta

Las tarjetas son la parte central de la aplicación ya que son las que definen las tareas. Todas las tarjetas tienen que tener título, mientras que los demás campos son opcionales.

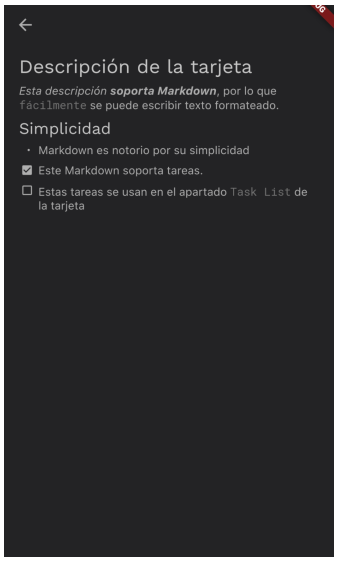
Una tarjeta puede tener:

Descripción

Describe qué hay que hacer en la tarea y soporta Markdown[18], específicamente el formato estándar con la inclusión de checkboxes (Figuras 3.1a y 3.1b).



(a) Descripción en proceso de edición.



(b) Vista formateada de la descripción.

Etiquetas

Las etiquetas tienen como únicos usos:

- El filtrado mediante búsqueda por texto, utilizando el carácter # como precedente a la etiqueta buscada.
- Facilitar la visualización de tarjetas que las contengan en la vista de tablero (Figura 3.2).

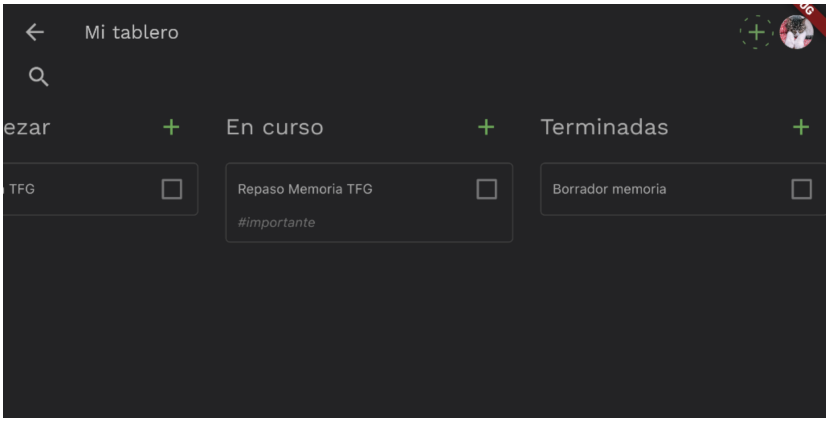


Figura 3.2: Un tablero con varias columnas y una tarjeta etiquetada.

Miembros o asignados de la tarjeta

Se pueden asignar o asociar miembros a una tarjeta para filtrar y ver quién debería estar involucrado en una tarea.

Fechas de comienzo y fin de la tarea

La fecha de fin no puede ser anterior a la fecha de comienzo si la hay. Se pueden definir ambas, una o ninguna de las fechas.

Lista de tareas

La vista de tareas refleja en modo de *checklist* las tareas que hay en la descripción precedidas con "`- [ ]`" o "`- [x]`", en caso de estar pendientes o terminadas respectivamente. Aunque las tareas se pueden crear, editar o eliminar en la propia descripción, también se pueden hacer dichas operaciones en la sección de lista de tareas de la tarjeta.

Comentarios

Cualquiera que pueda ver la tarjeta puede escribir un comentario. Al igual que la descripción, los comentarios soportan Markdown.

3.2 Sincronización

PARA permitir un flujo de tareas realmente flexible entre los usuarios se propone la idea de sincronización entre columnas.

3.3 Descripción con ejemplos

A continuación se mostrarán varios casos de uso para ilustrar la sincronización. Cada caso de uso tendrá asociada una figura (figuras 3.3, 3.4, y 3.5) para complementar la explicación. Los tres casos de uso mostrados son ejemplos de la **sincronicación unilateral**. Los tableros comparten ciertas características a tener en cuenta para comprender los ejemplos:

- El propietario o propietarios de cada tablero se muestra en la esquina superior derecha de cada tablero.
- Los flujos de escritura que relacionan columnas de varios tableros están representados por flechas, indicando la direccionalidad y qué columnas relacionan.
- Las etiquetas son propiedad de cada usuario y, por tanto, cada usuario ve las suyas en las tarjetas que etiqueten pero no la de los demás, estén éstas compartidas o no.
- Las barras de progreso muestran la cantidad de subtareas terminadas dentro de una tarjeta.

CU 1: Gestión de tableros (Figura 3.3)

En este caso sólo hay un usuario: Luis. Luis quiere utilizar la aplicación para ver todas las tareas de un curso académico en un tablero.

Luis tiene dos asignaturas de tercer curso y crea un tablero para cada una, resultando en los tableros ISD y XP.

A continuación crea un tablero llamado Tercero, en el que crea las columnas **To Do**, **Doing** y **Done**. En ISD y en XP crea otras columnas de propiedades similares.

Para las columnas de creación de tareas, tanto de XP como de ISD, genera un flujo de sincronización en dirección a la columna **To Do** del tablero Tercero. De esta forma, todas las tareas creadas en ISD y en XP aparecerán en Tercero.

Para aquellas que tienen actividad, se crea una sincronización en dirección a la columna **Doing** del tablero Tercero. En este caso, en ISD hay varias columnas que se adhieren al criterio, es decir, el flujo ha sido creado tanto para ambas columnas, **Doing** y **Testing**, pero con una misma columna objetivo. Esto permite que el mover una tarea de **Doing** a **Testing** en ISD mantenga la tarea en la columna Doing de Tercero.

Para finalizar, sincroniza las columnas de término de tarea de los tableros fuente con la del tablero repositorio.

CU 2: Proceso en pipeline (Figura 3.4)

En este caso hay tres tableros, **cada uno perteneciente a un usuario diferente**.

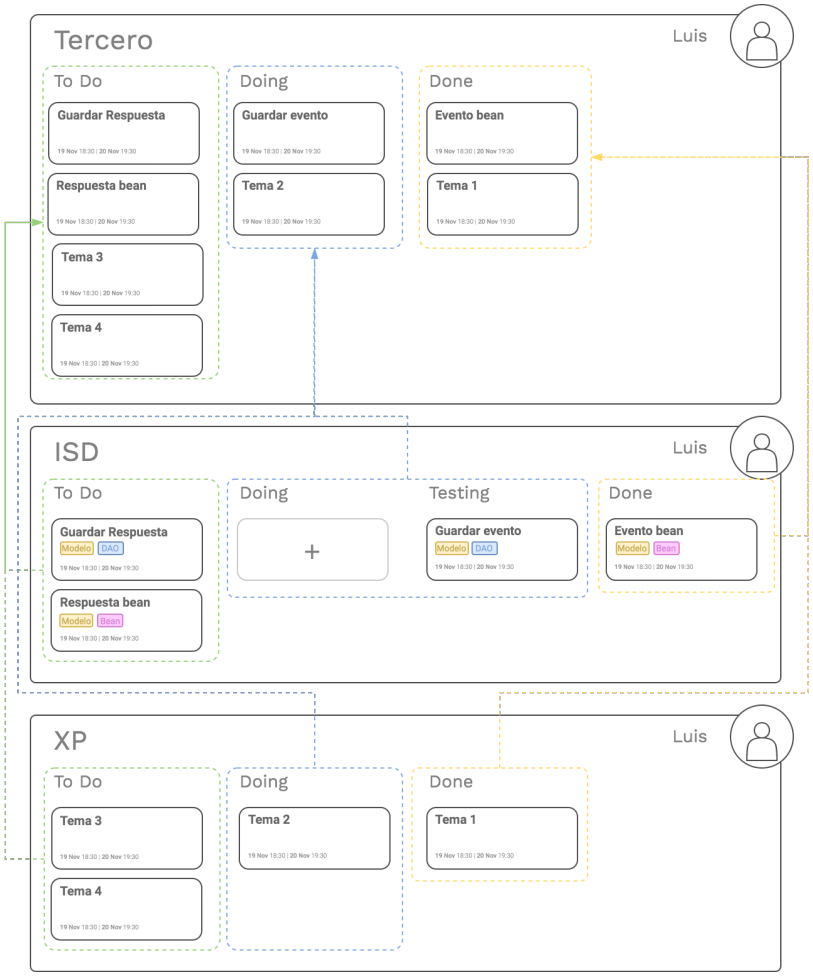


Figura 3.3: Caso de uso de un único usuario usando un tablero como repositorio común.

El guionista es el que comienza el flujo. En el momento en el que introduce tarjetas en la columna **Para dibujar**, éstas son compartidas **o sincronizadas** a la columna **Guiones** del tablero DIBUJO SPIDEY, propiedad del dibujante.

El dibujante entonces puede mover las tarjetas en su propio tablero. Cuando acaba su proceso, pone las tarjetas en **Terminado**, apareciendo éstas a continuación en el tablero del colorista en la columna **Dibujado**.

Cuando éste termina, por último, las sitúa en **Para entregar**.

CU 3: Seguimiento de equipos (Figuras 3.5 y 3.6)

Hay cuatro tableros, uno para cada equipo: ANDROID (Figura 3.6), IOS, WEB y BACKEND. Como en los ejemplos anteriores, cada uno define una columna que será el objetivo de las columnas mostradas en el tablero que se ve en la figura 3.5. Por ejemplo, la columna **Android** del

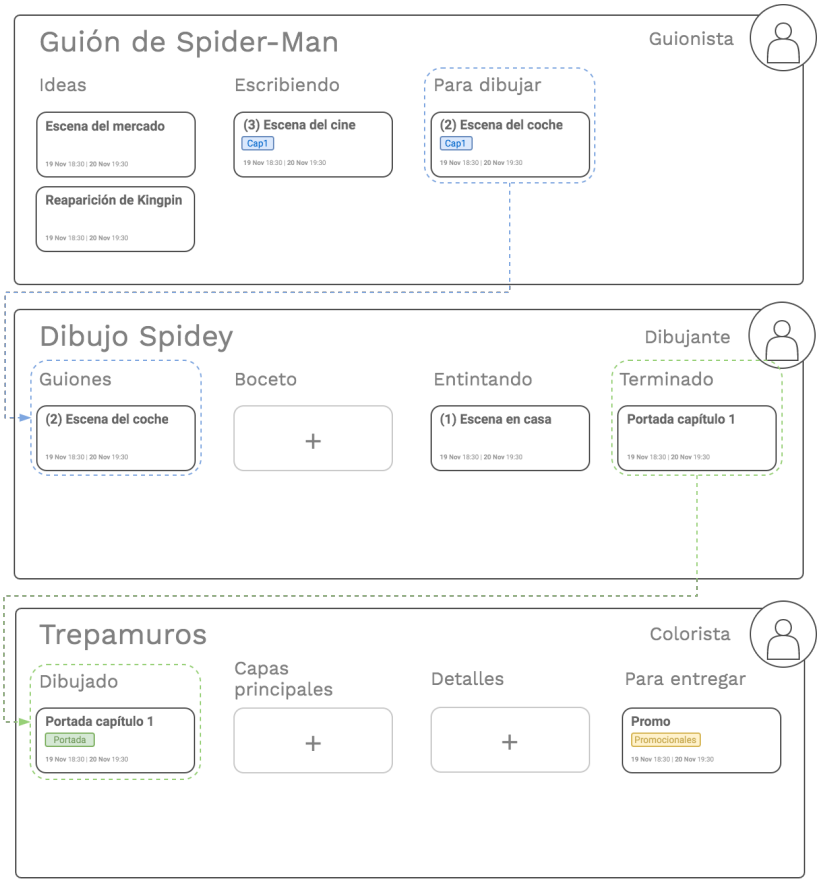


Figura 3.4: Caso de uso de varios usuarios utilizando los tableros como partes de un proceso continuo.

Figura 3.5: Ejemplo de gestión y monitorización de un equipo.

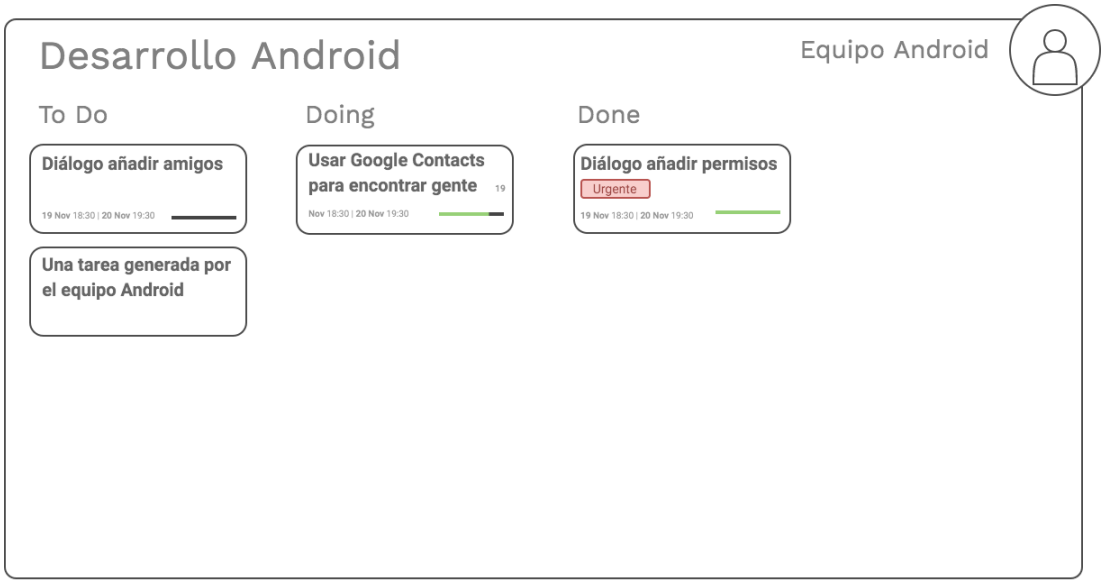


Figura 3.6: Tablero ejemplo del equipo de Android.

tablero Proyecto de Tarjetas se sincroniza como fuente de una columna en el tablero Desarrollo Android. Esto representa que las tarjetas situadas en esta columna serán enviadas al equipo encargado de desarrollo en Android.

Cada uno de los tableros que reciben tareas del tablero Proyecto de Tarjetas en el ejemplo puede tener la organización interna que deseen, sea una o varias columnas, tal y como se muestra en la figura 3.6, que ilustra cómo podría ser el tablero que usa el equipo de Android.

Debido a que la sincronización es unilateral, aunque los equipos que reciben las tarjetas de las columnas sincronizadas muevan sus tarjetas a columnas no sincronizadas, las tarjetas no desaparecen de las columnas de Proyecto de tarjetas.

Como las tarjetas no son copias sino enlaces o referencias a las tarjetas, en el tablero Proyecto de tarjetas se podrán ver los cambios realizados en ellas en otros tableros. Por ejemplo, al marcar una tarea de la checklist en la tarjeta *Usar Google Contacts para encontrar gente* dentro de la columna **Doing** en el tablero Desarrollo Android, la barra de progreso aumentará tanto en este tablero como en la tarjeta de mismo nombre que se ve en la columna **Android** en el tablero Proyecto de Tarjetas.

De esta forma se puede ver en ellas la actividad de cada una de las tarjetas como, por ejemplo, su barra de progreso si tiene una lista de tareas, o el checkbox marcado si se marca como hecha.

Esto permite que el jefe de proyecto pueda monitorizar los cuatro equipos y su progreso de un vistazo.

El seguimiento se haría aún más sencillo aplicando filtros que, por ejemplo, dejen de mos-

trar las tareas que tienen la etiqueta *Aprobado*, para que se puedan ver sin molestias sólo las tareas sin aprobar. O quizás se use un filtro que sólo muestre las no completadas, etc.

3.3.1 Descripción teórica

Una vez se ha visto la sincronización en funcionamiento, el comportamiento es previsible.

Flujos

Una sincronización puede ser:

- **Unilateral**
  Las tarjetas de una columna (columna escritora) se copian a la segunda columna (columna receptora). Sin embargo, si esta segunda columna crea nuevas tarjetas o mueve una de las tarjetas compartidas a otra columna, la columna escritora **no varía**. Si una tarjeta se mueve de la columna escritora o se elimina, la tarjeta desaparece de la columna receptora.
  Visto de otro modo: A escribe en B pero B no escribe en A.

- **Bilateral**
  Ambas columnas contienen las mismas tarjetas en todo momento, independientemente de dónde se creen.
  Visto de otro modo: contenido(A) = contenido(B).

La sincronización ocurre entre dos columnas de dos tableros diferentes. La relación de sincronización de estas columnas puede ser creada por un solo usuario en caso de que éste sea miembro de ambos tableros o por dos usuarios en caso contrario.
En el caso de que un usuario quiera sincronizar una columna de un tablero del que es miembro con otra de un tablero del que no lo es, el usuario realiza una petición de sincronización hacia otro usuario, que será el que defina la naturaleza de la suscripción. Para clarificar la explicación, se propone el siguiente ejemplo:

Luis quiere colaborar con Marta en un proyecto, pero cada uno tiene su propio tablero. Luis quiere recibir tareas de Marta, para lo que necesita crear una sincronización de su tablero al de Marta.
Luis selecciona la columna que quiere que mantenga esta sincronización y acciona la compartición indicando que la invitación debe ser recibida por Marta.
Marta, en su sesión, abre el tablero en el que tiene la columna que quiere que tenga la relación de sincronización con la columna de Luis. En este tablero, selecciona la columna y crea una suscripción. En las opciones de posibles suscripciones encuentra la de ”Columnas compartidas conmigo”, donde se encuentra la columna que le compartió Luis.

Marta selecciona la columna compartida por Luis y acciona la relación "'Mi columna' *escribe en* 'Columna de Luis'".

Compartición de tarjetas

Los flujos tienen ciertas implicaciones. Realmente las copias que aparecen en las columnas mediante una sincronización no son copias, sino las propias tarjetas. De esta forma, el flujo unilateral se convierte en una herramienta versátil para la delegación y seguimiento de tareas. El ejemplo de la sección 3.3 se describe más en profundidad las consecuencias de este comportamiento.

Capítulo 4

Metodología

Una metodología es un conjunto de prácticas, guías y formas de actuar a lo largo del desarrollo de un proyecto.

Las metodologías valoradas inicialmente para este proyecto fueron Scrum y Kanban.

4.1 Scrum

Scrum es una metodología de desarrollo ágil iterativa basada en la planificación de períodos breves de desarrollo de un número definido de funcionalidades, llamados *sprints*. Las funcionalidades, o historias de usuario, serán compuestas de un número de tareas a decidir durante el *sprint planning*, que se lleva a cabo una vez por sprint.

En Scrum existen diferentes roles[19], siendo los más habituales los miembros del equipo, los product owner y el scrum master:

- **Miembros del equipo**
  Los miembros del equipo son aquéllos que se van a encargar de crear el producto, es decir, no son necesaria o únicamente desarrolladores.

- **Product Owner**
  La misión del product owner es mantener una dirección clara para el equipo, sabiendo qué funciones necesita cubrir el producto a desarrollar y sus prioridades. El product owner valora los requisitos, sus prioridades y los agrega al backlog.

- **Scrum Master**
  Se preocupa de que el Scrum se esté realizando de la forma correcta, guía al equipo para no caer en malas prácticas como equipo ágil y trata de ayudar al equipo a que sea capaz de gestionar las tareas, de centrarse en los resultados y minimizar los blockers.

En Scrum hay varios tipos de reuniones[20] a tener en cuenta durante cada iteración:

- **Diarias**
  En estas reuniones de duración breve el equipo se suele tratar qué hizo cada uno el día anterior, qué se va a hacer a continuación y si hay algo que impida el progreso de la tarea.

- **Sprint planning**
  En el sprint planning se realizan estimaciones sobre las historias de usuario y se crean las tareas que se van a realizar para llevarlas a cabo.

- **Sprint review**
  Durante esta reunión se revisan si las funcionalidades cumplen las expectativas originales, si se deben realizar alteraciones y, en general, se observan en detalle las tareas llevadas a cabo durante el sprint.

- **Retrospectiva**
  Su objetivo es reflexionar sobre el sprint, observar qué ha funcionado, qué no y qué se puede mejorar de cara al futuro.

4.1.1 Scrum para este proyecto

La ventaja principal de Scrum sobre Kanban es que ofrece objetivos cerrados por sprint, evitando el sumergirse en tareas fuera de ámbito del mismo y manteniendo un conjunto de tareas por sprint que debería ser alcanzable.
Sin embargo, hay varias características que hacen imposible su implementación para este proyecto:

- **Cada sprint debe tener una duración cerrada**
  Los sprints deberían tener duraciones definidas determinadas al principio del proyecto. Habitualmente se escoge dos semanas para cada sprint. Sin embargo no era posible comprometerse a fechas de entrega debido a restricciones de tiempo laborales y personales que impidieron un flujo de trabajo continuo en el proyecto.

- **Reuniones**
  Por los mismos impedimentos mencionados en el punto anterior, resultaba imposible realizar reuniones frecuentes con el director del proyecto, por lo que era importante utilizar una metodología que ofrezca un flujo de trabajo más independiente.

- **Las tareas deben ser estimadas**
  Aunque esto es considerado habitualmente como una gran ventaja para Scrum, el desconocimiento de las tecnologías usadas en el proyecto hacían muy complicada la estimación fiable de la duración de las tareas durante gran parte de la duración del mismo.

Evidentemente se podría haber utilizado un subconjunto de las características de Scrum que no incluyeran los puntos problemáticas, pero al ser puntos centrales en la metodologia se ha considerado que escoger un Scrum sin fechas, reuniones ni estimaciones no tenía sentido para este proyecto.

4.2 Kanban

Kanban, o parte del mismo, es a menudo incorporado dentro de otras metodologías ágiles por su forma de disponer de las tareas en tableros y columnas de forma que se pueda comprobar el estado del proyecto en un vistazo.

Kanban está basado en las siguientes prácticas:

- **Visualización**
  En todo momento se debe poder visualizar qué está haciendo todo el equipo. Esto se consigue mediante el tablero.
- **Limitación del trabajo en curso**
  Cada columna debe tener un límite de tareas que pueden estar en curso al mismo tiempo.
- **Flujo mejorado**
  Cada vez que se termine una tarea, se incorpora al flujo en progreso la siguiente tarea más prioritaria del backlog.
- **Procesos explícitos**
  Los procesos deben ser claros y públicos para que pueda ser usado por todo el equipo y sujeto a crítica y, por tanto, mejora de cualquiera.
- **Bucles de feedback**
  En Kanban existe un sistema de reuniones similar al de Scrum, pero menos estricto. Se suelen tener reuniones diarias, de revisión de riesgos, estrategia, entrega, etc. Exceptuando las reuniones diarias, las demás no es necesario que tengan una frecuencia exacta, sino que basta con que sean regulares y que tengan sentido para el proyecto. Las reuniones deben tener un horario cerrado y concisas.
- **Mejora colaborativa**
  Se debe tratar de que el equipo comparta la visión de las metas, flujo de trabajo, procesos y riesgos del proyecto para que todos aporten de forma colaborativa en su mejor resolución.

En Kanban no hay roles predefinidos y no se trabaja en sprints, sino que el desarrollo es continuo, simplemente sacando nuevas tareas del backlog según éstas se van terminando.

Como no hay sprints, se pueden realizar cambios en cualquier momento. Estos cambios pueden ser de prioridad, tareas que se quitan o que se añaden al backlog. Por esto es especialmente importante que siempre se continúe con la tarea de más prioridad cuando se busca un nuevo trabajo, ya que pudo haber variado durante la realización de la anterior tarea.

Una forma simplística de ver Kanban es como un flujo de trabajo por el que pasan unas tareas.

En Kanban, al igual que en Scrum, existen ciertas restricciones que se han de tener en cuenta si se quiere utilizar de forma purista:

- **Máximo de tareas por estado**
  En cada columna se debe marcar un máximo de tareas. Habitualmente se anota en la cabecera de la columna como se muestra en la figura 4.1.

- **Cambio de tarea**
  Sólo se puede empezar una nueva tarea una vez se ha terminado la anterior. La tarea, su alcance y definición no varía durante la realización de la misma, pero esta restricción no aplica a las tareas no activas del tablero.

Además, Kanban permite utilizar no sólo columnas sino también fast lanes, es decir, para tareas que puedan estar asociadas a una emergencia, haciendo más flexible el límite de las columnas.

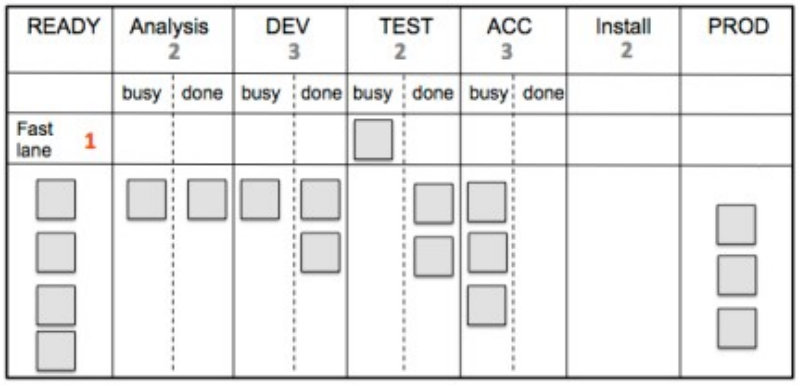


Figura 4.1: Ejemplo de un tablero Kanban con una fast lane.

4.3 Metodología a usar en el proyecto

En este proyecto se ha utilizado una adaptación de Kanban a las necesidades del mismo. En este caso sólo se restringe el número de tareas en la columna de **"En progreso"** a tres, mientras que el resto de tareas no tiene ningún límite.

4.3.1 Motivación para usar Kanban

Las razones para escoger Kanban frente a Scrum fueron las siguientes:

- Kanban permite un flujo más libre de las tareas, optando por una perspectiva de marcar una tarea cuando esté terminada de forma satisfactoria en lugar de marcar fechas para el término de una o más tareas.
- Al no tener claras las estimaciones de las tareas y, en muchos casos, el alcance de las mismas, se ha preferido utilizar una metodología que no tiene un número determinado de tareas por sprint, llegando a romper en ocasiones algunas tareas en tareas menores en el momento de sacarlas del backlog al comprobar que la complejidad para ellas en un principio era mayor que el esperado en el momento de la creación de la tarea.
- Kanban permite la reorganización de prioridades de tareas inactivas en cualquier momento, dando más libertad que Scrum frente al desarrollo de un tipo de aplicación y con tecnologías sobre las que no había experiencia previa al proyecto.

4.3.2 Características del Kanban implementado en el proyecto

Durante el desarrollo del proyecto se ha utilizado la herramientoa GitKraken Boards[16] como tablero Kanban. Esta herramienta permite el uso de etiquetas para las tareas, que serán de gran importancia para definir el flujo de trabajo del proyecto.
En el caso de este proyecto, las etiquetas discriminan tipo de tarea, mientras que las columnas definen el estado en el que se encuentran.

Durante la implementación de una tarea de la columna *Doing* se puede tanto crear nuevas tareas en *Backlog*, como reordenarlas según varíen prioridades, como enviarlas a la columna *Rejected* si se volvía irrelevante, innecesario o inapropiado para el proyecto.

Estructura de las columnas

En un proyecto de una única persona podría tener sentido que la columna *Doing* sólo tuviera una tarea de máximo. Sin embargo, se ha valorado usar tres debido a que una misma tarea puede englobar tanto el servicio REST como el frontend y, a menos que fuera una tarea de naturaleza trivial, se ha preferido separar dichas tareas para controlarlas con mayor granularidad.
La tercera tarea está habitualmente reservada para tareas con las etiquetas `#issue` o `#review`, es decir, tareas en las que hay que arreglar o comprobar por completo su funcionamiento.
Debido a que se han planeado las tareas por hito, normalmente aquéllas activas estaban relacionadas durante su desarrollo, por lo que la revisión de las etiquetadas con `#review` forma

parte natural del ciclo de desarrollo de una tarea directamente relacionada.
Se podría defender la creación de una columna *Review* en lugar de usar la etiqueta mencionada para este tipo de tarea, pero se ha decidido que el uso de la columna *Doing* con la etiqueta `#review` da una mayor visibilidad de la tarea a revisar. Debido a que además no todas las tareas pasan por esa fase sino que sólo se marcan aquéllas que una vez finalizada la tarea se ha considerado que resultaría útil mantenerla a la vista durante un tiempo para comprobar más posibles errores para la tarea implementada.

En este caso, las tareas a arreglar etiquetadas con *`#issue`* no representan ningún tipo de urgencia por lo que no se ha implementado ningún tipo de representación del fast lane. Sin embargo, es posible cambiar las prioridades de la cola de la columna *Backlog* con el objetivo de favorecer el paso de un *`#issue`* si éste es considerado crucial o que tiene importancia para las siguientes tareas a abarcar.

Uso de etiquetas

Además de las ya mencionadas, se han utilizado las siguientes etiquetas:

- **`#flutter`**
  Tareas de frontend.

- **`#functions`**
  Tareas de backend.

- **`#refactor`**
  Refactorización de código, habitualmente se agrega una descripción detallando la naturaleza de la misma.

- **`#hold`**
  Tareas pausadas por alguna causa. Se ha intentado evitar el uso de esta etiqueta debido a que una vez se pone una tarea en espera, ésta no se puede sacar de **"En progreso"**, lo cual perjudica la restricción de tareas máximas de la columna.

- **`#maybe`**
  Tareas anotadas como recordatorio para valorar más tarde si se realizarán o no o para pensar cómo adaptarlas al proyecto.

- **`#story`**
  Tareas que representan casos de uso descritos de forma más general previamente a desgranarlas en tareas más pequeñas.

Iteraciones

A pesar de que en Kanban no hay iteraciones de forma natural ya que promueve un flujo continuo, en este proyecto se han agrupado los casos de uso en hitos para tener una imagen inicial a gran escala del proyecto. Dichos hitos no reflejan una naturaleza iterativa sino puntos clave del desarrollo o un roadmap que seguir a lo largo de la duración del proyecto. Este roadmap se verá más a fondo en el capítulo 6.
Estos hitos se han utilizado como puntos donde en una metodología Kanban real se trataría una reunión de revisión y de estrategia, es decir, se han tratado como feedback loops desde los cuales generar las siguientes tareas a abarcar para continuar el camino marcado por el siguiente hito.

Eso implica que en el contexto de este documento cuando se mencionen iteraciones no se referirá a iteraciones como parte de una metodología sino a un período enfocado al trabajo en una parte del desarrollo, que se puede ver afectada en diferentes períodos o iteraciones.

Capítulo 5

Análisis de requisitos global

Inicialmente, los requisitos del sistema se han definido como hitos para desgranarlos a historias de usuario y, posteriormente, definir las tareas que componen las historias de usuario.

Debido a que los hitos son demasiado globales y las tareas demasiado detalladas, se especificarán los requisitos usando las historias de usuario definidas para el proyecto.

En este proyecto no existen diferentes roles: todos los usuarios tienen las mismas capacidades, por lo que quién activa los casos de uso siempre es el mismo rol. Por tanto, las descripciones de las historias de usuario no siguen la convención de Scrum: ”Como *rol*, quiero *hacer algo*”.

La única diferencia de roles es la de usuario **no autenticado** frente a usuario **autenticado**, por lo que sí se mostrará de forma explícita en los casos de **usuario no autenticado** quién activa la historia en favor de que haya más claridad en el documento.

5.1 Historias de usuario

Se presenta una visión general de las historias de usuario en la tabla 5.1.

5.1.1 Cuenta

Registro

Un **usuario no registrado** se puede registrar utilizando su correo electrónico y una contraseña o su cuenta de Google. Una vez registrado, la sesión es iniciada automáticamente y el usuario queda autenticado hasta que cierre la sesión.

En caso de que el usuario introduzca datos incorrectos se mostrará un mensaje de error indicando el problema.

Vista	Historia de usuario	Descripción
Cuenta	Registro	Un usuario no autenticado debe poder registrarse en la aplicación mediante correo y contraseña o su cuenta de Google.
	Autenticación	Un usuario no autenticado y registrado debe poder autenticarse en la aplicación.
	Editar datos de la cuenta	Un usuario debe poder editar los datos relacionados con su cuenta.
	Cerrar sesión	Un usuario debe poder cerrar la sesión.
	Eliminar cuenta	Un usuario debe poder eliminar su cuenta.
Tableros	Crear tablero	Un usuario debe poder crear un tablero.
	Editar tablero	Un usuario debe poder editar un tablero al que esté asociado.
	Eliminar tablero	Un usuario debe poder eliminar un tablero al que esté asociado.
Columnas	Crear columna	Un usuario debe poder crear una columna en un tablero al que esté asociado.
	Editar columna	Un usuario debe poder editar una columna en un tablero al que esté asociado.
	Eliminar columna	Un usuario debe poder eliminar una columna en un tablero al que esté asociado.
	Filtrar tarjetas	Un usuario debe poder filtrar las tarjetas mostradas en un tablero al que esté asociado.
	Sincronizar columnas	Un usuario debe poder realizar una petición de sincronización de una columna en un tablero al que esté asociado.
Tarjetas	Crear tarjeta	Un usuario debe poder crear una tarjeta en una columna de un tablero al que esté asociado.
	Mover tarjeta	Un usuario debe poder mover una tarjeta en una columna o de una columna a otra en un tablero al que esté asociado.
	Eliminar suscripción a una tarjeta	Un usuario debe poder eliminar la suscripción a una tarjeta de una columna en un tablero al que esté asociado.
	Editar tarjeta	Un usuario debe poder editar una tarjeta de una columna en un tablero al que esté asociado.

Tabla 5.1: Historias de usuario de la aplicación

Autenticación

Un **usuario registrado pero no autenticado** se puede autenticar utilizando las credenciales con las que se ha registrado previamente, sea correo electrónico y contraseña o la cuenta

de Google. Una vez autenticado el usuario tiene acceso a los casos de uso para usuarios autenticados.

En caso de introducir datos incorrectos se mostrará un mensaje de error indicando el problema.

Editar datos de la cuenta

Un usuario puede editar datos asociados a su cuenta, específicamente nombre, apellidos y avatar. En caso de introducir datos incorrectos se mostrará mensaje de error.

Cerrar sesión

Un usuario puede cerrar sesión. Una vez cerrada la sesión, el usuario necesitará autenticarse de nuevo para acceder a los casos de uso de un usuario autenticado.

Eliminar cuenta

Un usuario puede eliminar su cuenta. Una vez eliminada la cuenta, el usuario no podrá autenticarse con las credenciales que una vez estuvieron asociadas a la misma sin registrarse de nuevo.

5.1.2 Tableros

Crear tablero

Un usuario puede crear un tablero, al que es automáticamente asociado una vez creado.

Editar tablero

Un usuario puede editar un tablero al que esté asociado, pudiendo cambiarle el nombre o editar los miembros asociados al tablero.

Eliminar tablero

Un usuario puede eliminar su asociación a un tablero. Si es el último usuario asociado al tablero, el tablero será eliminado de la base de datos, incluyendo las columnas pertenecientes a éste.

5.1.3 Columnas

Crear columna

Un usuario puede crear una columna en un tablero al que esté asociado.

Editar columna

Un usuario puede editar el nombre de una columna de un tablero al que esté asociado.

Eliminar columna

Un usuario puede eliminar una columna de un tablero al que esté asociado.

Filtrar tarjetas

Un usuario puede filtrar las tarjetas puntualmente mostradas en un tablero al que esté asociado según diversos criterios:

- **Por título**
  Si se incluye en el filtro un fragmento de texto, éste será utilizado para mostrar sólo las tarjetas que tengan parte del texto como título. En caso de ser un texto que contenga varias palabras, sólo se mostrarán las tarjetas que contengan todas las palabras buscadas, sin importar el orden.

- **Por miembros**
  Si se incluyen uno o más miembros con la notación `@nombrecompleto`, se mostrarán las tarjetas que contengan todos los usuarios por los que se filtra.

- **Por etiquetas**
  Si se incluyen una o más etiquetas con la notación `#etiqueta`, se mostrarán las tarjetas que contengan todas las usuarias por las que se filtra.

Sincronizar columnas

Este caso de uso trata sobre la implementación de la sincronización de columnas, descrito previamente en la sección 3.2.

5.1.4 Tarjetas

Para que la sincronización de las columnas tenga sentido, todas las columnas que vean una tarjeta deben acceder a la información de dicha tarjeta en cualquier momento dado, es decir, si dos columnas sincronizadas ven una misma tarjeta, no tienen dos copias diferentes de dicha tarjeta, sino que ambas ven e interactúan con los mismos datos.
Para realizar esto, se propone un sistema de suscripciones, donde una columna no tiene una relación de posesión con una tarjeta sino de suscripción a la misma.
Las tarjetas que se ven, por tanto, en un tablero son aquéllas a las que las columnas del tablero están suscritas.

Crear tarjeta

Un usuario puede crear tarjetas en columnas pertenecientes a tableros a los que esté asociado. Una vez creada esta tarjeta, la columna desde la que se inicia la creación de la tarjeta añade una suscripción a la tarjeta recién creada. Esto permite que a partir de este momento se pueda ver la tarjeta asociada a la columna.

Mover tarjeta

Un usuario puede mover tarjetas en columnas o entre columnas pertenecientes a tableros a los que esté asociado. Mover una tarjeta de una columna A a una columna B implica eliminar la suscripción de la columna A a la tarjeta y la suscripción de la columna B a la tarjeta a continuación.

Eliminar suscripción a una tarjeta

Un usuario puede eliminar una suscripción de una columna a una tarjeta cuando dicha columna pertenece a un tablero al que esté asociado. Una vez cancelada la suscripción, la tarjeta ya no se mostrará en la columna.
Si ésta era la última columna suscrita a la tarjeta, la tarjeta será eliminada de base de datos ya que nadie puede visualizarla.

Editar tarjeta

Un usuario puede editar una tarjeta de una columna perteneciente a un tablero al que esté asociado. Cualquier cambio realizado en la tarjeta se verá reflejado en todas las demás columnas suscritas a dicha tarjeta.

Capítulo 6

Planificación

Debido a las circunstancias mencionadas en el capítulo 4 no fue posible realizar un plan rígido con fechas cerradas para cada una de las iteraciones, por lo que se creó un *roadmap* de hitos que documenta a grandes rasgos los pasos a seguir por el desarrollo sin dictaminar tiempos concretos para cada uno.

En este capítulo se observará la estimación de las horas y fechas aproximadas en las que se alcanzaron los hitos.

En la figura 6.1 se representa el roadmap a seguir por el desarrollo de la aplicación. Cada bloque en este roadmap es un hito y estos hitos están marcados con colores que reflejan su posición en una fase del desarrollo. Estos hitos se explicarán en detalle en el capítulo 8.

Para entender bien la figura, antes hay que explicar qué se quiere decir con hitos y fases del desarrollo:

- **Hito**
  Un hito es un punto marcado con cierta importancia en el desarrollo. En este caso marcan puntos de avance durante el desarrollo. Sin embargo, fácilmente se puede ver que hay diferentes tipos de hito, según el tipo de funcionalidades o la naturaleza de tareas con las que estén relacionados.

- **Fase**
  En este contexto se ha denominado fase a un conjunto de hitos que tratan sobre un mismo tema o que están relacionado:

  - **Azul**
    Los primeros hitos corresponden a una fase inicial, de análisis, estudio y decisión de las tecnologías a utilizar mediante prototipado y refinamiento del concepto del producto.
  - **Púrpura**

A continuación, los siguientes hitos tratan de darle la funcionalidad básica al producto.

- **Rosa**
  Una vez se realiza la primera aproximación de la vista y del servicio REST, se añade y propaga lógica de autenticación por la aplicación, implicando también la posiblidad de crear, editar y eliminar usuarios.
- **Rojo**
  Esta última es la fase en la que se completan los hitos que se apoyan en todas las fases anteriores, pudiéndose ver como la fase final.

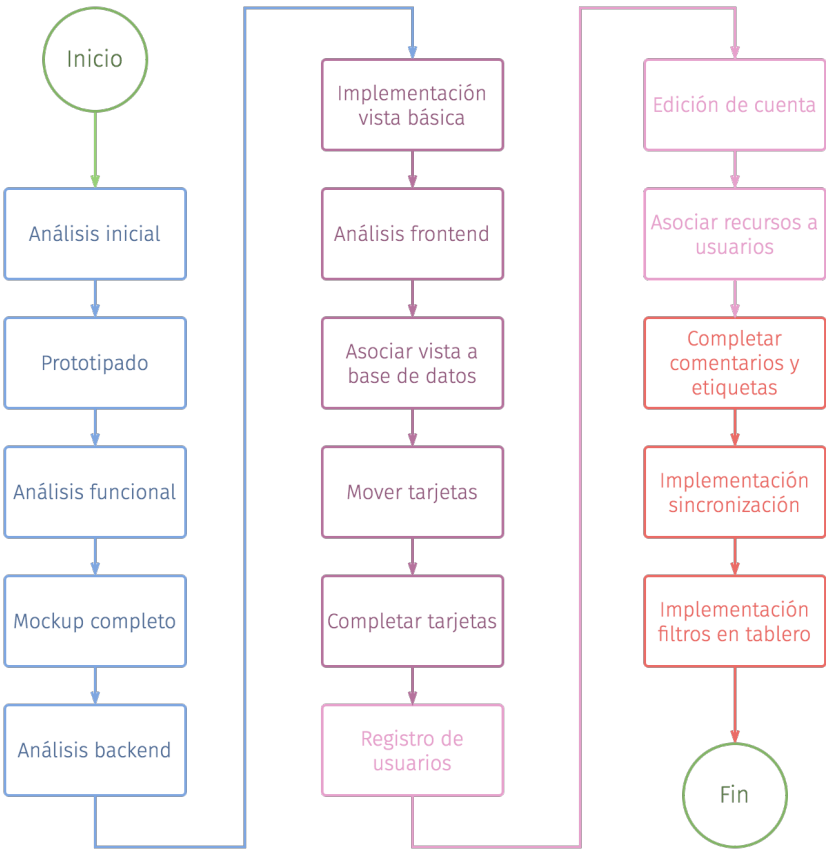


Figura 6.1: Disposición de los hitos planeados para el proyecto.

Hito	Fecha inicial	Fecha fin	Horas
Análisis inicial	20/05/18	14/08/18	40
Prototipado	15/10/18	27/12/18	100
Análisis funcional	04/02/19	10/03/19	20
Mockup completo	11/03/19	18/03/19	10
Análisis backend	19/03/19	30/03/19	20
Implementación vista básica	10/04/19	15/06/19	80
Análisis frontend	20/06/19	02/07/19	20
Asociar vistas a base de datos	03/07/19	15/07/19	40
Mover tarjetas	20/01/20	16/02/20	120
Completar tarjetas	25/02/20	15/03/20	80
Registro de usuarios	15/03/20	17/03/20	20
Edición de cuenta	18/03/20	22/03/20	20
Asociar recursos a usuarios	10/04/20	15/04/20	30
Completar comentarios y etiquetas	10/05/20	28/05/20	40
Implementación sincronización	06/07/20	12/08/20	100
Redacción de la memoria	14/08/20	30/08/20	100
Total	20/05/18	30/08/20	840

Tabla 6.1: Estimación de horas y fechas de los hitos.

Capítulo 7

Fundamentos tecnológicos

PARA este proyecto se han utilizado tecnologías que no se han visto en la formación académica con el fin de que fuera más didáctico.

7.1 Tecnologías comunes

7.1.1 Cloud Firestore[1]

Google define la base de datos como *"una base de datos flexible y escalable para desarrollo móvil, web y de servidor"*.

Firestore es una base de datos NoSQL basada en colecciones, ofreciendo soporte para escribir localmente datos cuando el dispositivo está offline y sincronizándolos con el servidor cuando vuelve a estar online mediante una comparación de las diferencias en los datos de una forma similar a los commits usados en Git.

Las características de interés crítico para el proyecto son las siguientes:

- **Reactiva**
  Firestore permite escuchar cambios a través de websockets, haciendo posible que el frontend reaccione a éstos y mantenga la vista siempre actualizada con respecto a la información presente en la base de datos.

- **SDKs**
  Firestore presenta SDKs tanto para Node.js como para Dart, lenguajes utilizados para el backend y frontend respectivamente, haciendo su uso cómodo y fácilmente integrable en el desarrollo de la aplicación.

Debido a que la aplicación no tiene una lógica de negocios excesivamente compleja, que su mayor reto es la colaboración y que varios usuarios vean la información al mismo tiempo, Firestore es una elección muy adecuada para este fin.

Además, Firestore se basa en la creación de reglas para acceder a sus colecciones y documentos, de forma que se permite el acceso a la base de datos incluso desde una web de forma segura.

En este proyecto, sin embargo, las escrituras de datos se harán a través del servicio backend para aplicar un servicio REST al uso.

7.2 Tecnologías del backend

7.2.1 Node.js[2]

Node.js es un runtime de JavaScript que permite utilizar JavaScript como lenguaje de servidor. Se ha escogido para poder aprovechar las facilidades que aportan las Cloud Functions para el proyecto.

7.2.2 Express.js[3]

Express.js es un framework para Node.js utilizado para enrutar las peticiones que llegan al servidor, pudiendo crear múltiples enrutadores o *aplicaciones* y escoger cuáles exportar, cuáles no, o incluso juntarlas.

Express.js facilita el enrutamiento de URLs anidadas y con parámetros, además de diferenciar las peticiones por método HTTP.

7.2.3 Cloud Functions para Firebase[4]

Google las describe como *"un framework serverless que permite ejecutar código backend automáticamente en respuesta a eventos disparados por peticiones HTTPs y funcionalidades de Firebase"*.

En este proyecto en concreto sólo se utilizan las peticiones HTTPs para que el flujo sea tal como sería en un servicio REST tradicional.

Las Cloud Functions se pueden desarrollar en JavaScript o TypeScript, código que se guarda en la nube de Google y se ejecuta en un entorno controlado por la organización. De este modo, la escalabilidad es manejada por Google dependiendo de cuánto uso se le esté dando en un momento concreto a las Cloud Functions.

En este caso se ha utilizado Express.js en conjunto con Node.js usando JavaScript.

A pesar de que el SDK de Google de las Cloud Functions permite enrutar peticiones de forma directa a las funciones que se quieren ejecutar, el uso de Express.js posibilita enrutar las URLs disponibles programáticamente y la creación de URLs con variables insertadas de forma muy simple.

7.2.4 JavaScript[5]

JavaScript es un lenguaje de tipado dinámico habitualmente utilizado en el lado cliente de las aplicaciones, normalmente en un navegador. Sin embargo, con el nacimiento de Node.js, JavaScript ha ganado peso en el desarrollo de backend a través de la creación de aplicaciones fullstack, permitiendo a desarrolladores web pasar al lado del servidor más fácilmente que con otros lenguajes con los que no están familiarizados.

Este proyecto trata de sacar partido del tipado dinámico de JavaScript para evitar la repetición de código, permitiendo abstraerse más de cualquier limitación que se podría observar con un lenguaje de tipado fuerte.

7.3 Tecnologías frontend

7.3.1 Dart[6]

Dart es un lenguaje compilado fuertemente tipado y de asignación dinámica que se puede utilizar tanto en cliente como en servidor, en cuyo caso la forma de compilación varía según el dispositivo objetivo[21].

En este caso se utiliza Dart Native, que emplea ambos tipos soportados de compilación, JIT y AOT, para el desarrollo y despliegue respectivamente.

Dart soporta nativamente el uso de streams y programación asíncrona[22], haciendo posible la implementación de funcionalidades reactivas de forma natural y expresiva. En este caso, las operaciones asíncronas no resueltas se llaman `Future`, tal y como en JavaScript existen las promesas[23].

7.3.2 Flutter[7]

Flutter es un framework para crear aplicaciones móviles, web y de escritorio[1] utilizando el mismo código. Este proyecto se centra en el despliegue a **aplicaciones móvil**.

Mientras que es un framework muy joven (lanzamiento oficial el 4 de diciembre de 2018) ha conseguido un gran número de usuarios debido a la facilidad con la que se desarrolla en él, la calidad de la documentación y el soporte dado por el propio equipo a los IDEs VSCode, IntelliJ y Android Studio, recomendando entre éstas Android Studio.

Flutter, al igual que otras tecnologías utilizadas para el frontend actuales como React.js o Vue.js, utiliza un paradigma de **programación orientada a componentes**[24].

En Flutter todos los componentes a mostrar heredan de la clase `Widget`, pudiendo ser `StatefulWidget` o `StatelessWidget` dependiendo de si las propiedades del widget varían o no, respectivamente.

[1]La versión web está en fase beta y el despliegue a aplicaciones nativas de macOS y Linux está en alpha.

Los widgets siguen una estructura tal y como describe el patrón composición[25], es decir, pueden ser nodos u hojas. De esta forma, la información puede pasar unidireccionalmente por el árbol de widgets y los widgets superiores pueden reaccionar a cambios producidos en las ramas hijas, por ejemplo, mediante el uso de eventos.

Los puntos a destacar son los siguientes:

- Con Flutter es posible desarrollar desplegando directamente contra un dispositivo móvil o contra un emulador y realizar *hot reload* para que la aplicación desplegada cargue los cambios realizados en el proyecto automáticamente una vez se guardan los cambios, sin tener que volver a desplegar la aplicación.
- Flutter provee de widgets que soportan de forma nativa el uso de streams y futures. De esta forma se puede introducir un future o stream de datos a un widget que desempaquetará la información cuando esté disponible con el tipo adecuado haciendo uso de generics[26] de forma que los widgets contenidos en éste puedan utilizar los datos de forma síncrona.
- Gracias a la asignación dinámica de Dart, en conjunto con el tipado fuerte, es muy sencillo la serialización y deserialización de información de datos en formato JSON[27].
- El utilizar Firestore y Firebase facilita tareas como la autenticación y mantener la sesión utilizando los SDKs provistos para Dart.

7.4 Tecnologías utilizadas para el desarrollo

7.4.1 GitKraken Boards

Para hacer el seguimiento de las tareas y planificación del proyecto se ha utilizado GitKraken Boards, ya descrito en la sección 2.2.

7.4.2 Git[8]

Git es un sistema de versiones descentralizado en el que cada usuario mantiene una copia local del repositorio, pudiendo hacer commits locales, ramas, y otras gestiones que facilitan el desarrollo.

7.4.3 GitLab[9]

Este proyecto se ha alojado en un repositorio en GitLab, plataforma que mantiene repositorios Git tanto públicos como privados de forma gratuita.

7.4.4 VSCode[10]

VSCode es un editor open source con funcionalidades de IDE desarrollado por Microsoft. VSCode provee de una plataforma de descarga de extensiones para añadir elementos que no existan en VSCode por defecto. Entre éstas se encuentran las extensiones de soporte de Flutter y Dart para el desarrollo de aplicaciones que utilizan estas tecnologías en VSCode.

Además de funcionalidades propias, VSCode tiene extensiones que mejoran la experiencia de usuario con respecto a Git. Entre ellas se encuentran la posibilidad de realizar comparaciones con versiones anteriores, ver cuándo y en qué commit se ha modificado el fichero línea a línea, editor de merges, etc.

Capítulo 8
Desarrollo

En este capítulo se tratarán los aspectos de análisis, diseño e implementación de los hitos mencionados con anterioridad.

8.1 Análisis inicial

El análisis inicial trató casi exclusivamente las funcionalidades que formarían parte del alcance del proyecto, cómo se podría abordar y qué alcance debería tener.

Durante el esta etapa se pensaron diferentes modelos cuyo objetivo siempre era el de abordar un flujo de trabajo de forma colaborativa. Tras el diseño de varios modelos que afrontaban la sincronización de formas distintas, se decidió por utilizar el diseño basado en Kanban que se presenta en este documento.

8.2 Prototipado

En la fase de prototipado se tuvieron en cuenta diversas tecnologías para frontend y para backend, manteniendo desde un principio el uso de Cloud Firestore como base de datos por la sencillez que implicaría su uso para aplicaciones distribuidas.

Se crearon prototipos utilizando React Native, Ionic y Flutter para frontend y Go y Node.js para backend.

Se decidió a favor de Flutter y Node.js por las ventajas descritas en el capítulo de **Fundamentos tecnológicos** 7.

8.3 Análisis funcional

Una vez decididas las tecnologías se planearon las funcionalidades que debería tener la aplicación y el alcance aproximado. Tal y como se sugiere en una metodología ágil no se tu-

vieron en cuenta los detalles concretos para las funcionalidades. Las decisiones tomadas en este punto estarían sujetas a cambios según la evolución ya que debido al poco conocimiento de las tecnologías a utilizar para el desarrollo no era posible estimar el coste real de la implementación del diseño.

8.4 Mockup completo

En esta fase se realizaron los mockups principales para un diseño inicial de la aplicación, es decir, la vista de tableros, la de un tablero y la de una tarjeta, que se presentan en las imágenes 8.1, 8.2 y 8.3.

En algunas imágenes se puede observar cómo se concebían algunas características que al final no fueron implementadas. También se puede comprobar la falta de algunos componentes que aparecieron más tarde.

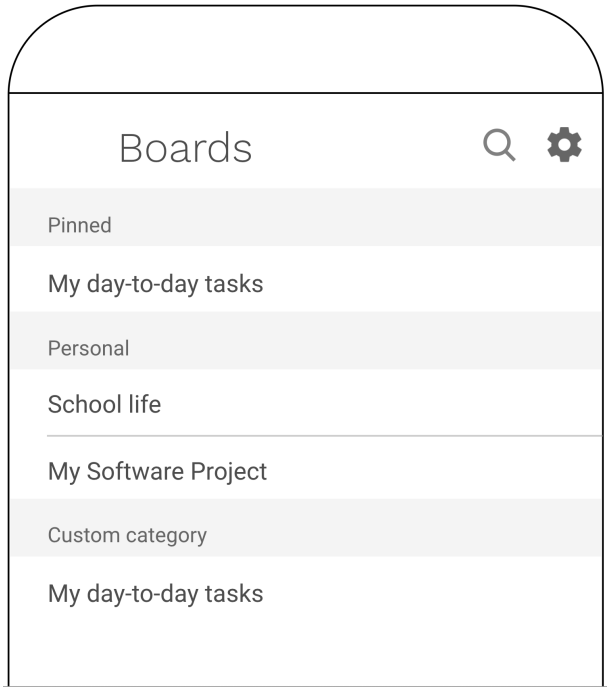


Figura 8.1: Vista de los tableros de un usuario.

8.5 Análisis backend

Debido a que la formación académica se centra sobre todo en tecnologías backend, en este proyecto se trató de darle más peso al frontend, por lo que el backend se realizó utilizando

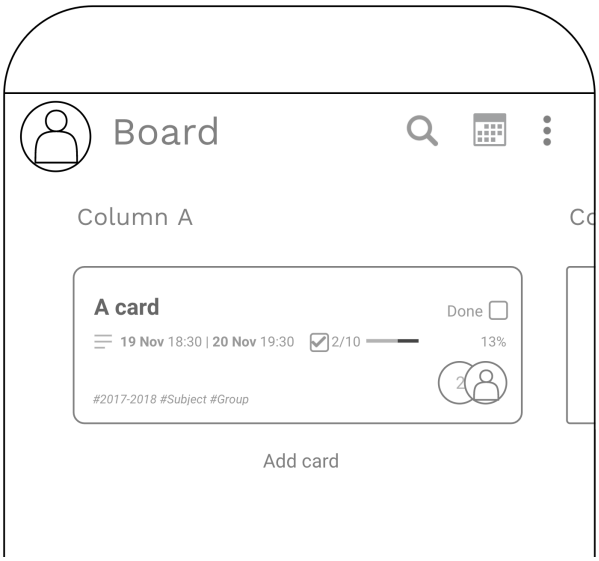


Figura 8.2: Vista de un tablero concreto de un usuario.

una estructura de capas habitual alterada para evitar la repetición de código gracias al tipado dinámico de JavaScript.

8.5.1 Diseño

Observando los casos de uso y limitando la interacción del frontend con el backend para operaciones de escritura se tuvieron en cuenta las operaciones de creación, actualización y borrado de los documentos como procesos genéricos en un principio y después se añadirían los cambios necesarios a cada caso particular para partir desde un punto de vista común a todas las entidades.

Este punto tiene sentido para este proyecto debido a que hay pocas entidades y comparten poca relación entre sí.

Además de los casos de escritura o eliminación generales, es necesaria la implementación de casos particulares dependiendo de la entidad que se esté escribiendo. Por ejemplo, cuando se crea una tarjeta (caso general) también se debe añadir una suscripción a dicha tarjeta desde la columna desde la que se ha creado (caso particular).

Para llevar a cabo el desarrollo del sistema descrito por la figura 1.3 se diseña un sistema presentado en la figura 8.4.

8.5.2 Implementación

En el denominado como `Entrypoint` será donde se reciban las peticiones HTTP. Las URL publicadas en Functions se obtienen programáticamente a través de una lista de `Http-`

Figura 8.3: Vista de una tarjeta de un usuario.

`Functions`, que contendrá, a su vez, uno o más `HttpMethods`. Cada `HttpMethod` contiene la URL parcial a partir de la `HttpFunction`.

De este modo añadir una nueva URL base es tan sencillo como añadir una nueva clase que hereda de `HttpFunction` y publicarla en el `Entrypoint`, que en el momento de desplegar la aplicación la exportará como una función de Firebase Cloud Functions.

Los `HttpMethods` a su vez determinan qué `Parsers` y `Validators` utilizar en caso de que se use alguno en la gestión de la petición.

Una vez tratados los datos enviados en la petición se continúa con la ejecución de la ló-

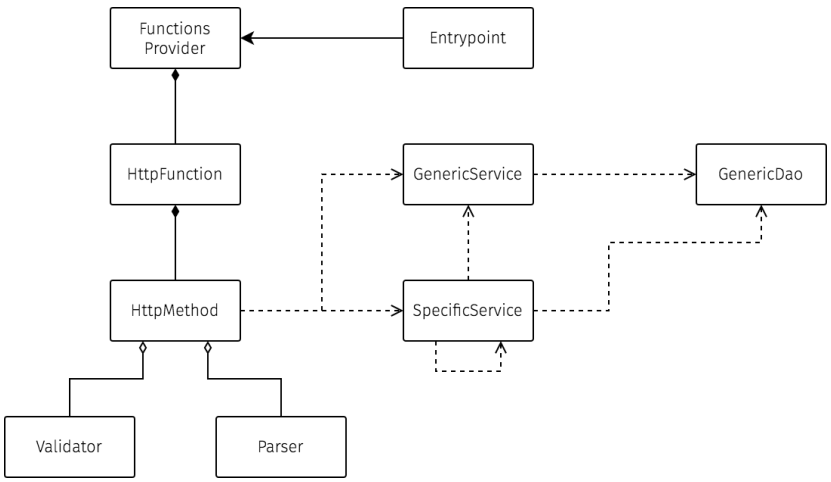


Figura 8.4: Estructura del servicio REST.

gica de negocio. Para ello, dependiendo de la operación que está siendo ejecutada, se utiliza el `GenericService` o un `SpecificService`. El primero tiene métodos genéricos de guardado, sobreescritura y borrado de cualquier tipo de entidad; el segundo representa cualquier servicio que se salga de dicha norma.

Para que el `GenericDao` sirva para los usos más habituales se crea una clase, `Entity`, de la que heredan todas las clases que representan objetos de la base de datos o *entidades*.

Aunque en JavaScript no exista el concepto de clase abstracta, se puede crear una clase base para que otras la hereden e implementen el comportamiento concreto. Éste es el caso de las entidades, `Entity`, que define métodos a implementar por las clases hija siguiendo el patrón plantilla[28]. En caso de no implementarlos en la clase hija, JavaScript utilizará el método de la clase base, que se encargará de lanzar una excepción alertando de que falta implementación.

De este modo todas las clases tienen que ser capaces de crear una representación en forma de objeto JavaScript (mapa o diccionario) y viceversa.

La clase `Entity` es capaz de cargar un documento de base de datos a través de su identificador, de eliminar un documento o de guardarlo.

Para los casos que sea necesario, también existen formas de guardar múltiples documentos como un grupo de escrituras para evitar posibles discrepancias.

Firestore no obliga a tener una estructura de tipos de datos concreto por documento en una colección, pero la figura 8.5 muestra los datos que pueden llegar a tener los documentos para cada una de las colecciones. Cada una de las cajas representa una entidad de una colección excepto `Comment`, que se indica como objeto asociado a `Card` por simplicidad, debido a que la estructura de un documento en Firestore es la de un objeto JSON.

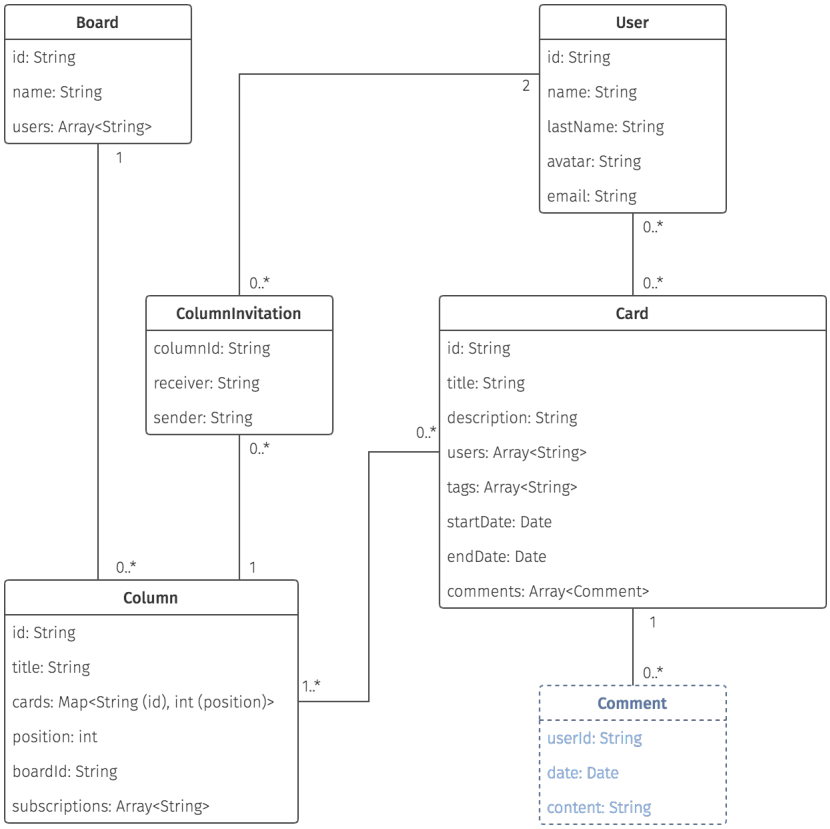


Figura 8.5: Modelo de datos.

8.6 Implementación vista básica

Una vez definida la estructura a seguir por el backend y el modelo de datos que a utilizar, el siguiente paso fue la creación de las vistas con Flutter adhiriéndose lo máximo posible a los mockups como se puede observar en las figuras 8.6a, 8.6b y 8.7.

Esta fase serviría como una fase de formación para aprender a usar Flutter, patrones en el mismo y expresiones en Dart.

8.6.1 Implementación

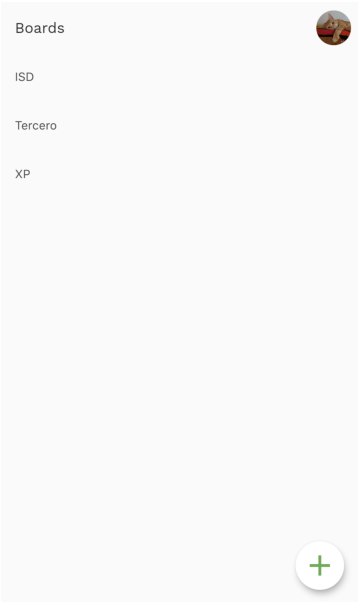
Las páginas a implementar en esta primera iteración de la vista fueron las siguientes:

- Página de tableros.
- Página de un tablero concreto.
- Página de detalles de una tarjeta.
- Página de visualización de la descripción de una tarjeta.

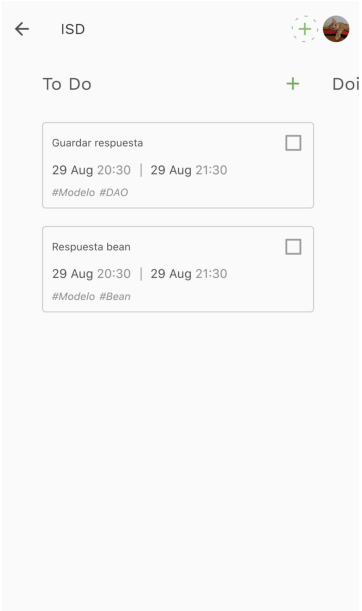
En esta implementación no se utilizó en ningún momento el patrón BLoC que después se usaría en la mayoría de páginas.

En esta primera fase se utilizó simplemente el cambio de estado desde los propios elementos de Flutter. Además, los datos utilizados en los componentes eran obtenidos como datos de muestra en las propias clases en métodos privados que se suplantarían en futuras iteraciones con los datos reales. De esta forma se podría trabajar cómodamente en Flutter y utilizar este hito como punto de arranque en Dart y Flutter, sin tener que abarcar al mismo tiempo Node.js.

Gracias a que los componentes se estructuran igual que clases en un lenguaje orientado a objetos fue posible aislar las partes que más susceptibles serían de cambiar o los componentes más comunes desde un punto muy temprano en el proyecto.



(a) Página de los tableros del usuario.



(b) Página de un tablero.

8.7 Análisis frontend

Al terminar la primera fase, se pudo continuar con el análisis de cómo seguir con el desarrollo en Flutter.

8.7.1 Análisis

En este punto se hizo hincapié en ver los patrones más usados en Flutter. El patrón que más destaca y que se puede utilizar de forma escalable tanto para operaciones sencillas como complejas es el patrón BLoC.

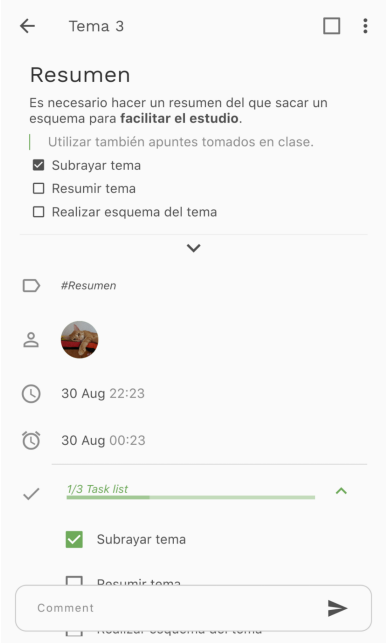


Figura 8.7: Página de una tarjeta.

Este patrón separa la lógica de la vista, manteniendo habitualmente uno o varios streams con los cambios según se actualice el estado de la aplicación. La vista observa los streams y reconstruye las partes necesarias en base a esto, creando un modelo reactivo.

El patrón BLoC[29]

El patrón BLoC consiste, principalmente, en mantener los datos actualizados y disponibles en streams[30] para que los componentes visuales se actualicen en base a cambios producidos en los streams.

Los BLoCs realizarán consultas a BBDD a través de los DAOs presentes en el frontend para mantener streams abiertos que escuchen eventos disparados por cambios en la BBDD y así mantener la vista siempre actualizada.
Además de ello, la vista capturará los eventos de interacción del usuario disparando las operaciones adecuadas en el BLoC, que a su vez lanzará las peticiones pertinentes al servicio REST.

`setState` vs BLoC

Utilizar `setState` es más sencillo que utilizar el patrón **BLoC** a primera vista, pero puede tener un coste potencialmente alto.

Mientras que el patrón BLoC se basa en el uso de streams para el control del estado,

`setState` llama al ciclo de vida de un componente para cambiar el estado y realizar una petición de reconstrucción. Esta reconstrucción involucra a todo el árbol de componentes, desde el que se ha llamado `setState` a todos los hijos que dependen de él.

```
class Component extends StatefulWidget {
  @override
  _ComponentState createState() => _ComponentState();
}

class _ComponentState extends State<Component> {

  Property property;

  @override
  Widget build() => Column(
        children: [
          // When _updateProperty is used, every component in
          // _ComponentState is redrawn
          ComponentThatUsesProperty(
            prop: property,
            onChanged: _updateProperty,
          ),
          ComponentThatDoesntUseProperty(),
        ],
      );

  void _updateProperty(PropertyData propertyData) {
    setState(() {
      // Update Widget
      property.data = propertyData;
    });
  }
}
```

Sin embargo, el patrón BLoC no altera directamente el estado, sino que guarda datos en un stream y los componentes reaccionan a estos cambios. Como no cambian el estado, los componentes hijos no necesitan ser reconstruidos.

```
class Component extends StatefulWidget {
  @override
  _ComponentState createState() => _ComponentState();
}

class _ComponentState extends State<Component> {

  Bloc _bloc;

```

```
  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    _bloc = BlocProvider.of(context);
  }

  @override
  Widget build() => _buil;

  Widget build() => Column(
        children: [
          // Component is rebuilt when bloc.propertyStream fires
    events
          StreamBuilder<Property>(
            stream: bloc.propertyStream,
            builder: (context, propertySnapshot) {
              return ComponentThatUsesProperty(
                prop: propertySnapshot,
                onChanged: _updateProperty,
              )
            }
          ),
          // Component doesn't need to be rebuild based on parent
    events
          ComponentThatDoesntUseProperty(),
        ],
      );

  void _updateProperty(PropertyData propertyData) {
    // BLoC handles logic
    bloc.updateProperty(propertyData);
  }
}
```

También se puede hacer uso de una propiedad stream sin delegar lógica en el BLoC si es trivial, siguiendo el patrón de una forma simplificada.

8.7.2 Diseño

En esta fase se definieron a grandes rasgos los componentes que se implementarían en las siguientes iteraciones, entre ellos las páginas que serían las vistas céntricas de la aplicación:

- Página de inicio de sesión y registro de usuario.
- Página de la cuenta.

- Cambio de página de visualización de la descripción a edición de la descripción.

Además, debido a que la interacción con el backend sería limitada, se definió la clase `Repository` que funcionaría como fachada ntre la vista y clases de comunicación con el backend y el frontend.

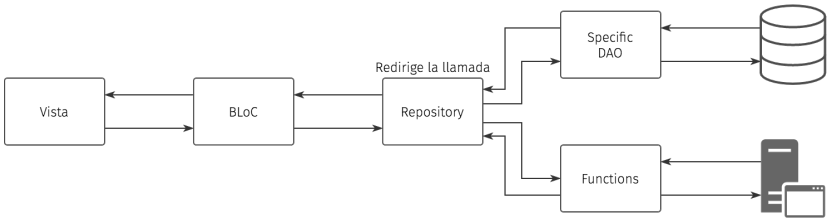


Figura 8.8: Implementación del patrón fachada mediante la clase `Repository`.

8.8 Asociar vistas a base de datos

La implementación de este hito, debido a cómo están organizadas la escritura y lectura de la base de datos en este proyecto, se realizará en dos fases por colección (de base de datos) con la que interactuar: fase de lectura desde Flutter y fase de escritura a través de las Cloud Functions.

Debido a dicha organización, la problemática se afronta desde dos puntos de vista diferentes para cada subapartado. Se creará una clase por tipo de documento que acepte los datos de Firestore en forma de mapa en un constructor para crear nuevas instancias para que la transformación de mapa a objeto sea responsabilidad de cada clase, siguiendo un patrón similar al que se sigue en el backend. En este caso la escritura y lectura no se realiza utilizando dichos objetos debido a las necesidades muy diferentes que surgen del uso de streams y de functions.

8.8.1 Análisis

Lectura

Los BLoCs se sitúan como componentes que forman parte del árbol de componentes en Flutter. Cualquier componente puede obtener un BLoC situado en un nodo ancestro si existe, siguiendo un patrón similar al Singleton usado, por ejemplo, en Spring, donde se crea instancia por componente en lugar de instancia por aplicación.

En el caso de este proyecto, sin embargo, debido a que no hay ningún sitio donde pueda haber potencialmente múltiples BLoCs de un mismo tipo, los BLoCs se sitúan por simplicidad en la raíz del árbol.

En este punto del desarrollo todavía no hay gestión de autenticación, por lo que los BLoCs no tienen que ser conscientes de a quién pertenecen los documentos y, por tanto, las consultas no incluyen ese filtro.

Escritura

Al igual que ocurre con la lectura, todavía no existe sesión, por lo que no se envía un token de identificación en las cabeceras HTTP.

El backend está diseñado de forma que responde a las llamadas con los datos pertinentes a la petición realizada de manera que se puedan añadir clientes a la API exportada. Sin embargo, debido a que ya existe una reacción a los cambios de base de datos por parte de los streams, la respuesta sólo se utilizará para tener en cuenta cuándo termina la acción, pero no se parsearán los datos de la respuesta HTTP y simplemente se reaccionará a los datos a través del stream de Firestore como muestra la figura 8.10.

8.8.2 Diseño

Lectura

El patrón BLoC mencionado en la sección anterior se adapta de forma natural a los websockets que Firestore permite crear a partir de consultas concretas a la base de datos, por lo que se puede suscribir a los cambios que ocurran frente a una consulta concreta (Figura 8.9).

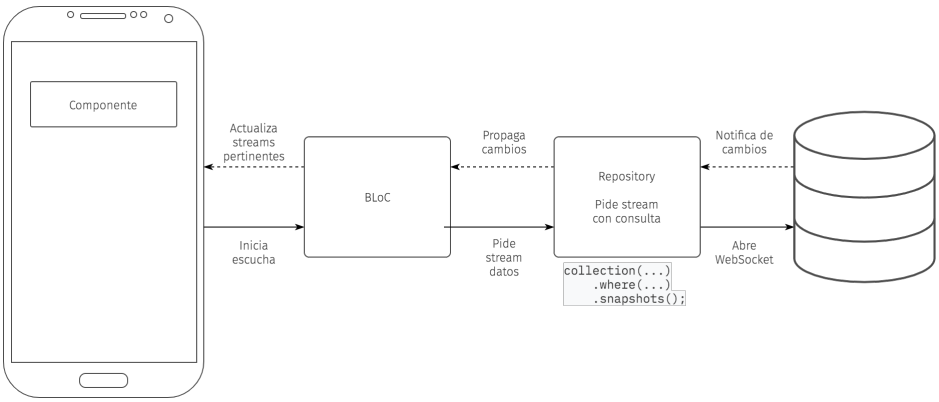


Figura 8.9: Esquema de funcionamiento de streams con Firestore.

Cada página tiene asociada al menos un BLoC del que obtiene los datos necesarios para presentar los componentes en la vista. Los BLoCs inician la escucha a través de la clase `Repository`, que interactúa con los DAOs necesarios según el tipo de dato que se quiera obtener, y éstos a su vez crean el websocket a través del que se escucha a eventos de la consulta concreta que se desee utilizando el SDK de Firestore para Dart.

Una vez iniciados, los `StreamBuilder` dispuestos en las páginas reaccionarán a los eventos del stream mostrando los cambios necesarios en la vista.

Escritura

La escritura sigue un patrón similar pero, en lugar de reaccionar a eventos dispuestos desde un stream pasivo, sigue un flujo disparado por un evento con la interfaz y que finaliza cuando el backend responde como se ve en la figura 8.10.

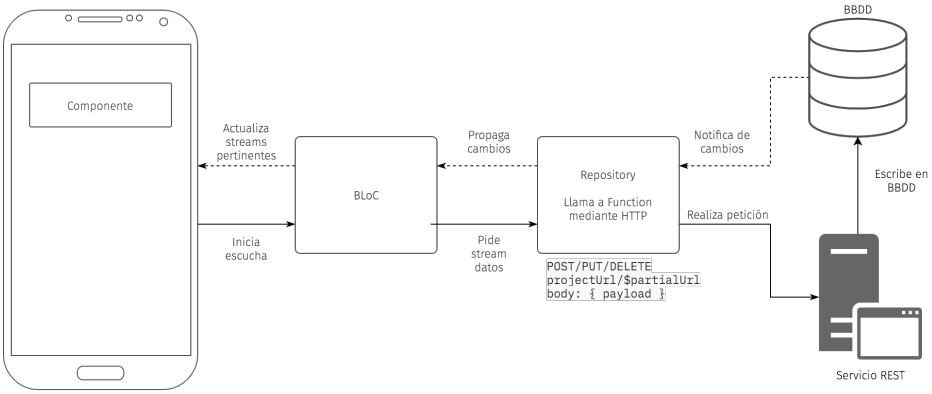


Figura 8.10: Esquema de comunicación con backend para la escritura.

8.8.3 Implementación

En este punto se crea una primera aproximación del backend con las entidades básicas necesarias y con parsers y validadores que todavía no tienen en cuenta muchos atributos que terminarán existiendo debido a que hay muchas partes sin implementar en la vista.

La mayoría de la tarjeta, por ejemplo, no se guarda en este punto, prevaleciendo una mezcla de datos de prueba y reales en la vista que se resolverá en futuras iteraciones.

8.9 Mover tarjetas

8.9.1 Diseño

Para mover tarjetas dentro de una columna o de una columna a otra, es necesaria la implementación de dos funcionalidades grandes dentro de este hito.

Drag and drop

Drag and drop es un tipo de componente en el que se coge y arrastra otro componente dentro de éste y se suelta en otro designado como objetivo o drag target. Debido a que no hay

un componente propio en Flutter que realice esto, hay que crearlo, siendo un componente bastante complejo, que incluye animaciones, cambios de estado, manejo de scroll, etc.

Sobreescribir los datos

En este punto, ésta es la primera interacción que causará la escritura de múltiples documentos, por lo que habrá que realizar una observación en el backend de cómo realizarlo.

Esto implicará la implementación de una forma de escribir varios documentos en una transacción y de un servicio distinto al genérico, tal y como se muestra en la figura 1.3.

8.9.2 Implementación

Drag and drop

Para crear el componente drag and drop se parte de una implementación de una lista única que existe en la librería de Flutter cuyo código es accesible gracias a ser open source. Sin embargo, hay que adaptar el diseño para soportar una lista múltiple en lugar de realizar el drag and drop en una única lista como soporta dicho componente.

Guardado de múltiples documentos

Firestore soporta la agrupación de escrituras de documentos en *batch*[31]. También soporta escritura y lectura en transacción, pero dado que la necesidad es sólo la de escritura de varios documentos sin leer documentos entre escrituras, es suficiente con la implementación de escrituras en *batch*.

Para ello se añadirá un método `saveInBatch` en la clase `Entity` del backend. No obstante, para que tenga sentido sentido con el diseño de las entidades y con las necesidades a cubrir, se limitarán las escrituras en *batch* a documentos **de la misma colección**.

8.10 Completar tarjetas

Al poder editar los datos reales se continuó con la edición de las tarjetas. La edición en la página de las tarjetas se consideró un hito en sí mismo debido a que es la página con más tipos de componentes diferentes, implicando el manejo de diferentes tipos de eventos, datos y formas de comunicarse con el BLoC dedicado a guardar los datos de las tarjetas.

8.10.1 Análisis

Cada sección de la tarjeta se incorporaría de forma aislada, considerándose una tarea por sección a completar, incluyendo la edición de texto de la descripción.

Ante el guardado de una tarjeta el backend se tiene que ocupar de obtener las etiquetas de los demás usuarios antes de sobreescribirla debido a que el frontend sólo mantiene las etiquetas del usuario autenticado. A causa de ese desconocimiento, el único cambio para el frontend es laincorporación de los nuevos valores en la petición al backend y en el mapeo de base de datos a objeto Dart. El backend, sin embargo, tendrá que realizar una operación en un servicio específico, es decir, no sirve la clase `GenericService` para guardar las etiquetas asociadas al usuario sin alterar las de los demás usuarios.

8.10.2 Implementación

Se sigue manteniendo a grandes rasgos el diseño del mockup, diferenciándose sobre todo en las fechas de inicio y fin que se separan en dos secciones diferentes y eliminando la opción *More information* para la actividad, reduciendo a mostrar sólo comentarios.

Mientras que la mayoría de secciones fueron implementaciones bastante directas, los componentes de descripción y la barra de escribir comentarios tuvieron una complejidad inesperada, ralentizando el desarrollo de este hito.

- Para que la altura de la vista previa de la descripción se adaptara al contenido de ésta hasta un máximo de un 40% de la pantalla fue necesario realizar un gran número de pruebas, dejando tal flexibilidad como una tarea pendiente mientras se continuaba con las siguientes para no perder demasiado tiempo en eso.

- Para la implementación de la caja de comentarios fue necesario utilizar un tipo de componente llamado `OverlayEntry`, que se sitúa por encima del resto de componentes y al que se le tiene que indicar cuándo aparecer y cuándo desaparecer teniendo en cuenta el ciclo de vida de los componentes en Flutter para que no se den discrepancias, como que el componente permaneciera una vez se cambiaba de página.
  Además, para mejorar la usabilidad, se decidió esconder y volver a mostrar la caja de comentarios cada vez que se editase algo en la página de la tarjeta ya que en caso contrario entre el teclado táctil y la caja se ocupaba demasiada pantalla, haciendo inviable la navegación por la tarjeta o la consulta de datos mientra se introducía el texto pertinente.

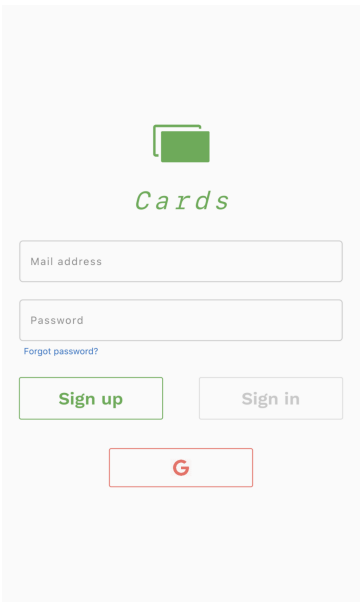
Para la adición de usuarios y etiquetas se creó un componente que sale de la parte inferior de la pantalla que serviría para ser reutilizado en diversas páginas.

8.11 Registro de usuarios

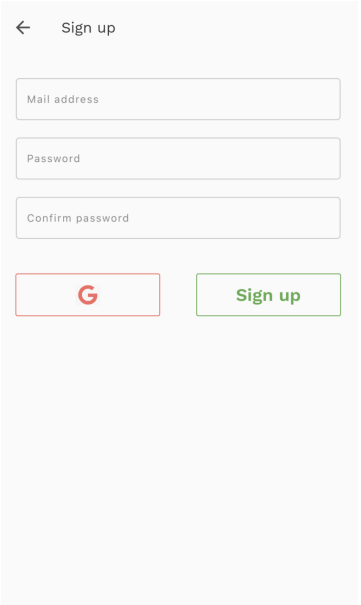
8.11.1 Diseño e implementación

Firebase facilita la autenticación mediante OAuth[32], por lo que se habilitó inicialmente la autenticación por correo electrónico y contraseña o vía Google.

En las figuras 8.11a y 8.11b se muestran las páginas de inicio de sesión y registro de usuario. Los formularios tienen control de errores para dar información al usuario de si ha introducido mal la contraseña, si la forma del correo es incorrecta, etc.



(a) Página de inicio de sesión.



(b) Página de registro.

8.12 Edición de cuenta

8.12.1 Análisis

Una vez creada la cuenta es sencillo utilizarla mediante Firebase para actualizarla. Google provee de un SDK, **Firebase Auth**, que permite crear cuentas, eliminarlas, iniciar sesión y otras operaciones relacionadas con la autorización de un usuario (Figura 8.12).

Sin embargo, lo que no es posible es añadir datos al usuario utilizando Firebase Auth, por lo que es necesario crear una colección propia de usuarios como se muestra en 8.5.

8.12.2 Diseño e implementación

Debido a que los usuarios no necesitan ninguna gestión en particular con respecto a guardado o eliminación en la base de datos, simplemente se utilizan las clases genéricas creadas para ese propósito en el backend, además de añadir la function necesaria al servicio.

En el frontend, sin embargo, es necesario:

- Eliminar la cuenta.

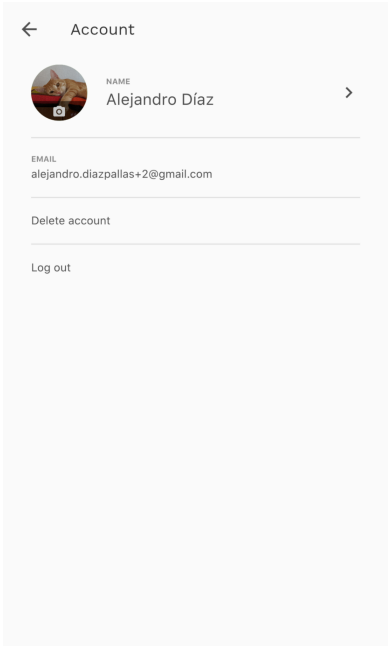


Figura 8.12: Página de configuración de la cuenta del usuario.

- Gestionar la sesión.
- Enviar la petición HTTP DELETE al backend del usuario para eliminar el documento de base de datos.

8.13 Asociar recursos a usuarios

En este punto es necesario asociar los recursos, como tableros, tarjetas, comentarios y etiquetas, a los usuarios que los creen o a los que sean asignados.

8.13.1 Diseño e implementación

En esta iteración es imprescindible comprobar qué recursos deben ser asociados a usuarios directamente, todavía sin asociar partes más indirectas como comentarios o etiquetas a los usuarios.

Para llevar a cabo el desarrollo planteado durante el análisis hay que cambiar las consultas de base de datos que se vean afectadas tanto en el backend como en el frontend. En el frontend se introduce el token de autenticación en las cabeceras HTTP y en el backend se obtiene dicho token y se usa el SDK de JavaScript de Firebase Auth para recuperar el usuario a través del mismo.

A partir de este momento se puede utilizar el usuario obtenido para escribir los documentos

de la forma adecuada incluyéndolo que ha realizado la petición.

8.14 Completar comentarios y etiquetas

8.14.1 Análisis

Al tener usuarios ya se pueden crear etiquetas y comentarios asociadas al usuario que esté actualmente autenticado en la sesión. Para ambos se crea un componente que se despliega de la parte inferior de la pantalla como se ve en la figura 8.13.

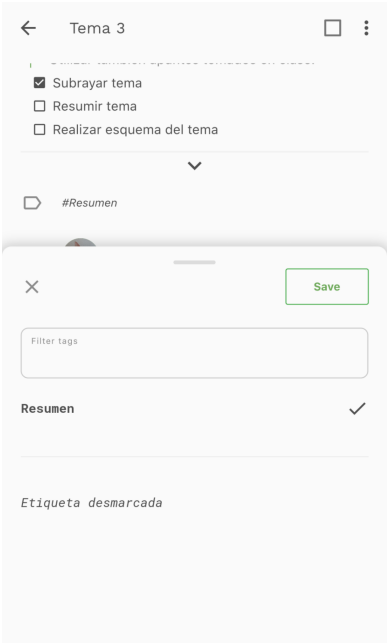


Figura 8.13: Edición de etiquetas en una tarjeta.

8.15 Implementación sincronización

8.15.1 Análisis

La sincronización implicará crear métodos específicos en el backend que realicen las suscripciones o las eliminen y los componentes necesarios en el frontend.

La sincronización realmente se basa en dos tipos de suscripción: de columna a columna y de columna a tarjetas.

De columna a columna

Una columna puede sincronizarse con otra. Dicha sincronización puede ser de escritura de la columna iniciadora en la columna objetivo, de lectura de la columna objetivo desde la columna iniciadora o de un flujo de datos en ambas direcciones.

Realmente, la escritura de una en otra se utiliza en forma de suscripción de la columna que lee a la columna que escribe y, para lograr la sincronización, ambas columnas se suscriben la una a la otra.

Una suscripción de una columna a otra equivale a abrir un disparador que cada vez que se añade o elimina una tarjeta en la columna escritora dicho cambio se reproduce en la columna lectora.

De columna a tarjetas

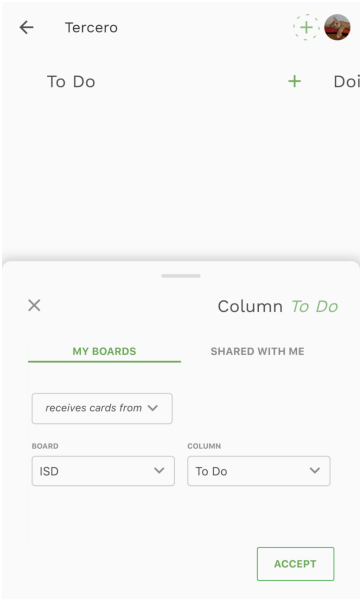
Cuando se crea una tarjeta, una columna se suscribe automáticamente a ésta haciendo que después se muestre en la vista en la posición adecuada.
Aún así, no hay una relación de posesión de una columna a una tarjeta y, debido a esto, una tarjeta puede ser la receptora de múltiples suscripciones, por lo que sólo se elimina en el momento en el que se elimina la última suscripción vigente.

8.15.2 Diseño e implementación

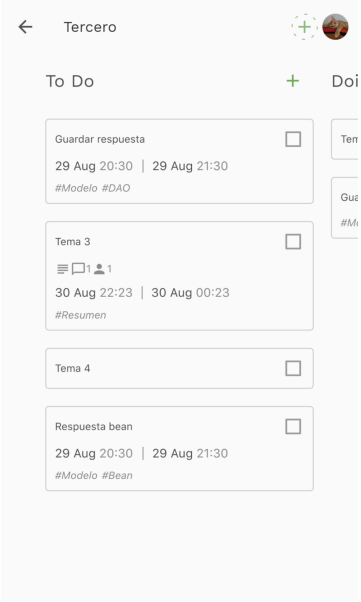
Para realizar la sincronización se creó un componente que mostraría la información necesaria para saber cómo sería el flujo de la datos indicando la columna iniciadora, el tablero de la columna objetivo, la columna objetivo y la dirección del flujo.

La direccionalidad de la suscripción se envía al backend, que realiza las suscripciones de la columna iniciadora y columna objetivo en la dirección necesaria y las suscripciones de la columna lectora a las tarjetas que tiene en ese momento la columna objetivo.

Las figuras 8.14a y 8.14b muestran el menú de suscripción y las columnas que reciben los datos una vez suscritas, ilustrando así el ejemplo de caso de uso provisto en la figura 3.3.



(a) Menú de sincronización.



(b) Tarjetas en la columna una vez se ha suscrito.

Capítulo 9

Conclusiones y trabajo

Finalizado el proyecto se considera que los objetivos planteados inicialmente se han llevado a cabo con éxito.

9.1 Conclusiones

9.1.1 Objetivos del proyecto

El objetivo principal de este proyecto era crear una herramienta alternativa simple cuyos flujos de trabajo estuvieran basados en la sincronización. La definición de completitud de este proyecto se basó en los casos de uso mostrados como ejemplos en la sección 3.2. Para definirlo como completado se requería que la aplicación cubriera al menos los dos primeros casos, pero en este caso se ha llegado a cumplir el tercero, con lo que el resultado se considera satisfactorio.

Otro objetivo importante del proyecto era conseguir una reactividad de las vistas a los cambios producidos en la base de datos, haciendo que la aplicación sea dinámica y esté siempre actualizada con la última información disponible.

Gracias a Firestore y al soporte de streams ofrecido por Flutter, la obtención de datos es instantánea, con lo que el intercambio de datos entre usuarios es fluido.

9.1.2 Objetivos académicos

Gran parte de la motivación de escoger las tecnologías empleadas en este proyecto era el obtener conocimientos de lenguajes y frameworks que no se abarcan en la carrera.

Debido a que se hace más hincapié en la parte backend que en el frontend durante la formación académica, este proyecto ha tratado de enfocarse más en el frontend para tratar problemáticas menos vistas durante la formación.

Para el desarrollo del frontend fue necesario aprender el framework Flutter y el lenguaje Dart, del que no se tenía conocimiento previo, además del uso de streams.

En la parte del backend fue posible trabajar en profundidad con JavaScript, utilizar superficialmente Express.js y ahondar en el uso de un lenguaje de programación dinámico e interpretado.

Debido a que tanto JavaScript como Dart disponen de operaciones asíncronas también hubo que aprender la forma de usarlas y los errores comunes que se producen durante su uso.

A lo largo del proyecto, pero especialmente al inicio, hubo muchas variaciones entre lo que se estimó que iban a durar las tareas y lo que realmente costaría llevarlas a cabo debido al desconocimiento de las tecnologías y a la forma de trabajar con ellas.

Sin embargo, una vez entendidos ciertos conceptos la experiencia ha sido positiva con respecto a todos los aspectos del desarrollo, tanto en la parte frontend como en la parte backend.

9.2 Trabajo futuro

En un futuro hay ciertas funcionalidades de la aplicación que habría que cambiar o evolucionar:

9.2.1 Implementación de un modelo de permisos

Para que la aplicación tenga más salida como gestor de proyectos en un ambiente de aspecto más corporativo que el público inicialmente objetivo sería interesante crear un modelo de permisos a nivel, como mínimo, de tablero.

9.2.2 Modelo de invitaciones

El modelo actual para crear las sincronizaciones entre usuarios es muy primitivo y el objetivo sería crear un sistema de invitaciones entre usuarios.

En el momento de compartir una columna a un usuario se permitiría escribir un mensaje con el fin de transmitir con qué objetivo se envía la invitación, además de introducir en ese mismo mensaje la direccionalidad que se quiere dar a la sincronización.

Una vez aceptada o denegada la invitación ésta caducaría.

9.2.3 Calendario

Una de las propuestas que quedó fuera del alcance del proyecto es la de incorporar una vista de calendario y sincronizar las tareas con servicios como pueden ser Google Calendar o Apple Calendar. Desde el calendario también se podrían crear tareas y entrar en las tarjetas, que se verían en dicha vista en forma de eventos.

9.2.4 Web

Uno de los siguientes pasos sería la implementación de una vista web. Debido a que Flutter también sirve para la creación de páginas web se podría reutilizar la mayoría del código.

9.2.5 Posible migración de lógica del backend al frontend

El backend se realizó como un servicio REST por dos razones:

- **Objetivo didáctico**
  Para ahondar en los conceptos vistos durante la carrera.
- **Facilita la migración a otro servicio**
  La mayoría del servicio no es consciente ni de estar embebido en las Cloud Functions ni de utilizar Firestore, ya que se han desacoplado estas responsabilidades para facilitar posibles migraciones.

Durante el uso de las functions se ha visto que para un servicio en producción no son la mejor opción porque el precio del uso de las funciones es mayor que acceder directamente a la base de datos.

Habitualmente el acceso directo a la base de datos sin pasar por un backend se consideraría una mala práctica, pero para estas tecnologías puede tener sentido por las siguientes razones:

- Firestore tiene un sistema de reglas muy elaborado para crear, modificar, leer y eliminar documentos, con el uso desde la web en mente, para que sea lo más segura posible. Esto, en combinación con su SDK de Firebase Auth, facilita el desarrollo y agiliza el servicio.
- Debido a que Flutter se puede usar también para la web, se puede aislar la lógica en una librería que se importe tanto en el componente móvil como en el componente web, eliminando la duplicación de lógica y los errores que acarrea.

Apéndices

Lista de acrónimos

AOT Ahead Of Time. 43

BBDD Base de Datos. 54

BLoC Business Logic Component. 54, 55

CU Caso de Uso. 16

IDE Integrated Development Environment. 45

JIT Just In Time. 43

SDK Software Development Kit. 41

Glosario

API Una API (Application Programming Interface) es una interfaz que define un contrato para acceder a unas funcionalidades provistas por un servicio.. 4

backend El backend de una aplicación es el que contiene la lógica de negocio del servicio provisto. Habitualmente éste está contenido en un servidor accedido mediante peticiones que envían ciertos datos para ser procesados.. 4

backlog El backlog es el conjunto de tareas o historias de usuario a llevar a cabo para el desarrollo del producto.. 23, 25

blocker Un blocker es cualquier ocurrencia, situación o característica que impida el avance de una o varias tareas.. 23

DAO Los DAO o Data Access Object son un tipo de clase utilizada para acceder a datos, generalmente de una base de datos u otro tipo de repositorio.. 54

DTO Los DTO o Data Transfer Object son un tipo de clase utilizada para almacenar datos y que sean transferido a través de ella de un proceso a otro. Normalmente son el objetivo de una serialización o deserialización. Los DTO pueden agregar datos de otros objetos para almacenar en éstos todos los datos pertinentes, por ejemplo, a una petición HTTP, en el caso de un servicio REST.. 6

fast lanes Las fast lanes o calles rápidas tratan de un proceso alternativo al flujo de trabajo habitual de un tablero Kanban, propuestas para tipos de tareas especiales, habitualmente con un carácter urgente como, por ejemplo, el arreglo de un fallo en producción.. 26

sprint En Scrum, duración de tiempo determinada para la resolución de un conjunto de tareas convenidas para el mismo. Las tareas deben ser estimadas y no se suele permitir el introducir o quitar tareas dentro del sprint.. 23

stream Flujo de datos que dispara un evento cada vez que se añaden datos al mismo.. 54

Bibliografía

[1] Google, “Cloud firestore.” [En línea]. Disponible en: https://firebase.google.com/docs/firestore/

[2] “Node.” [En línea]. Disponible en: https://nodejs.org/

[3] “Express.” [En línea]. Disponible en: https://expressjs.com/

[4] Google, “Cloud functions for firebase.” [En línea]. Disponible en: https://firebase.google.com/docs/functions

[5] “Javascript.” [En línea]. Disponible en: https://www.javascript.com/

[6] “Dart.” [En línea]. Disponible en: https://dart.dev/

[7] “Flutter.” [En línea]. Disponible en: https://flutter.dev/

[8] “git.” [En línea]. Disponible en: https://git-scm.com/

[9] “Gitlab.” [En línea]. Disponible en: https://about.gitlab.com/

[10] “Vscode.” [En línea]. Disponible en: https://code.visualstudio.com/

[11] “What is scrum?” [En línea]. Disponible en: https://www.scrum.org/resources/what-is-scrum

[12] “What is kanban?” [En línea]. Disponible en: https://www.agilealliance.org/glossary/kanban/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'kanban))~searchTerm~'~sort~false~sortDirection~'asc~page~1)

[13] “What is a rest api?” [En línea]. Disponible en: https://www.redhat.com/en/topics/api/what-is-a-rest-api

[14] “What is nosql?” [En línea]. Disponible en: https://www.mongodb.com/nosql-explained

[15] "Trello." [En línea]. Disponible en: https://trello.com/

[16] "Gitkraken." [En línea]. Disponible en: https://www.gitkraken.com/boards

[17] "Jira." [En línea]. Disponible en: https://www.atlassian.com/software/jira

[18] "Markdown." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Markdown

[19] "Scrum roles." [En línea]. Disponible en: https://www.atlassian.com/agile/scrum/roles

[20] "Scrum meetings." [En línea]. Disponible en: https://www.compuware.com/scrum-teams-whats-scrum-meetings/

[21] "Compilación dart." [En línea]. Disponible en: https://dart.dev/platforms

[22] "Dart asíncrono." [En línea]. Disponible en: https://dart.dev/codelabs/async-await

[23] "Promesas en javascript." [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[24] I. Lasn, "Componentorientedprogramming." [En línea]. Disponible en: https://medium.com/better-programming/what-is-component-oriented-programming-cop-10b32ae1fa1c

[25] "Composite pattern." [En línea]. Disponible en: https://refactoring.guru/design-patterns/composite

[26] "Generics." [En línea]. Disponible en: https://dart.academy/generics-in-dart-and-flutter/

[27] "Json." [En línea]. Disponible en: https://www.json.org/json-en.html

[28] "Template method pattern." [En línea]. Disponible en: https://refactoring.guru/design-patterns/template-method

[29] D. Boelens, "Reactive programming-streams-bloc," 2018. [En línea]. Disponible en: https://www.didierboelens.com/2018/08/reactive-programming-streams-bloc/

[30] Google, "Asynchronous programming: streams." [En línea]. Disponible en: https://dart.dev/tutorials/language/streams

[31] "Escrituras en batch." [En línea]. Disponible en: https://firebase.google.com/docs/firestore/manage-data/transactions#batched-writes

[32] "Oauth." [En línea]. Disponible en: https://oauth.net/2/