DESARROLLO DE APLICACIÓN HÍBRIDA MULTIPLATAFORMA PARA GESTIÓN DE OBJETIVOS

Bea Couchoud Ruiz

Memoria del Proyecto de DAM

IES Abastos
Curso 2020-2021
Grupo 7U
Valencia, 11 de junio de 2021
Tutor individual: Eduardo González Sanz

Tabla de contenido

1. Introducción ... 3
1.1 Contexto y Justificación ... 3
1.2 Objetivos ... 3
1.3 Idea inicial ... 3
2. Diseño ... 5
2.1 Análisis ... 5
2.1.1 Tecnologías y Herramientas ... 5
2.2 Planificación ... 11
2.2.1 Requisitos ... 11
2.2.1.1 Requisitos funcionales ... 12
2.2.1.2 Requisitos no funcionales ... 14
2.2.2 Gestión del proyecto ... 15
2.2.2.1 Ciclo de vida del proyecto ... 15
2.2.2.2 Trello ... 16
2.2.2.3 Diagrama de Gantt ... 17
2.3 Diagramas UML y Base de Datos ... 18
2.4 Guía de estilos ... 22
3. Desarrollo ... 28
3.1 Codificación ... 28
3.1.1 Estructura de carpetas ... 29
3.1.1.1 Front-end ... 30
3.1.1.1.1 Componentes ... 31
3.1.1.1.2 Servicios ... 31
3.1.1.1.3 Guards ... 33
3.1.1.1.4 Páginas y enrutamiento ... 33
3.1.1.1.5 Formularios y seguridad ... 36
3.1.1.2 Back-end ... 38
3.1.1.2.1 Seguridad ... 41
3.1.1.2.2 Middlewares ... 42
3.1.1.2.3 Pruebas ... 42
3.1.1.3 Firebase ... 43
3.2 Implantación y configuración de la aplicación ... 44
3.3 Control de versiones ... 45

4. Conclusiones 46
4.1 Posibles mejoras 46
4.2 Dificultades 47
4.3 Logros 47
4.4 Resultados 47
4.4.1 Muestras del resultado final 47
5. Referencias y Bibliografía 48

1. Introducción

En este trabajo se propone el desarrollo de una aplicación híbrida multiplataforma. En el siguiente documento se pretende presentar la aplicación ideada, comentar el origen y la motivación que llevó a hacer realidad este proyecto, los objetivos iniciales y posteriores del mismo, y detallar todas las características técnicas que permitan entender qué hace la aplicación y cómo lo hace.

1.1 Contexto y Justificación

Casi todo el mundo ha aprovechado el confinamiento vivido durante el 2020 para plantearse nuevos retos, mejores hábitos de vida o retomar objetivos que llevaban tiempo en segundo plano por falta de tiempo. Ahora que cada vez está más cerca la vuelta a la normalidad es posible que gran parte de los logros también vuelvan a su estado inicial. Para evitar perder todo ese esfuerzo realizado se ha decidido desarrollar una aplicación que permita de forma ágil e intuitiva tomar nota de casi cualquier reto u objetivo que queramos iniciar o mantener de la manera mas personalizada posible, para poder llevar un control de las mejoras realizadas y crear mayor adherencia a los nuevos hábitos que se deseen.

Por tanto, este proyecto no busca innovar, ya que hay muchas maneras, físicas y virtuales, de llevar un registro diario. Lo que se desea conseguir con este proyecto es crear una herramienta sencilla y útil que permita llevar un control de objetivos de manera adaptada y personalizable, y sobre todo, que permita ver la evolución de los mismos con un simple vistazo.

1.2 Objetivos

El objetivo principal del proyecto es crear una aplicación multiplataforma, que permita registrar y visualizar los avances realizados desde distintas plataformas (móvil, escritorio y web), a ser posible con sincronización de datos entre las mismas. También sería interesante poder compartir tus retos y logros con amigos y familiares, para motivarse y apoyarse mutuamente.

Como objetivo secundario y personal, se ha buscado realizar esta aplicación híbrida empleando tecnologías y frameworks que no se han visto a lo largo del curso en combinación con otras que si que se han utilizado, aprovechando así la realización de este proyecto para reforzar lo aprendido durante los últimos meses y obtener también nuevas habilidades y conocimientos.

1.3 Idea inicial

En un principio las necesidades mínimas que se buscan cubrir con esta aplicación multiplataforma son:

- Los posibles usuarios deben poder crear su cuenta e iniciar sesión en ella.
- Los usuarios deben poder crear, editar, eliminar y hacer públicos o privados nuevos logros u objetivos, a los que de ahora en adelante nos

referiremos, tanto en el presente documento como durante el desarrollo de la aplicación, como *ítems*.

- Dentro de los ítems se debe poder crear y editar registros diarios.
- Los usuarios deben poder ver de manera rápida sus avances de manera gráfica.
- Los usuarios deben poder ver su perfil y modificar sus datos si fuese necesario.
- La aplicación debe poder lanzar notificaciones.

Si hubiese tiempo suficiente otras características deseables serian:

- Poder añadir o eliminar amigos.
- Poder consultar los ítems públicos de los amigos añadidos y que a su vez los amigos del usuario puedan ver los ítems públicos del mismo.

En caso de que no diese tiempo debido a la limitación de horas se dejaría el código desarrollado y la aplicación preparados para añadir esta nueva funcionalidad de la manera más sencilla posible.

2. Diseño

Este apartado está dedicado a la fase de diseño del proyecto. Basándose en las necesidades de la aplicación y la información de la etapa de análisis, se detallarán los módulos y subsistemas que forman la totalidad del software, con el fin de llevar a cabo su desarrollo.
El objetivo de la etapa de diseño, es la definición de la arquitectura de la aplicación y la especificación detallada de sus distintos componentes, así como del entorno tecnológico necesario para funcionamiento.

2.1 Análisis

Aquí se recoge toda la información relativa a la fase de análisis de la aplicación. Quedan definidos tanto las tecnologías y herramientas necesarias para llevar a cabo este proyecto como los requisitos de software necesarios para usar la aplicación y todo ello sirve de base para la siguiente etapa del proyecto.

2.1.1 Tecnologías y Herramientas

Las tecnologías y herramientas que se han elegido para realizar este proyecto son las siguientes:

Aplicación Híbrida

"Write once, run everywhere". En este paradigma se basa la creación de aplicaciones híbridas, que permiten tener la misma interfaz de usuario entre diferentes plataformas gracias al diseño responsivo y herramientas de detección que permiten adaptar la interfaz de usuario a android o iOS, entre otros.

Las aplicaciones híbridas, a diferencia de las nativas, son aquellas capaces de funcionar en distintos sistemas operativos móviles. Entre ellos: Android, iOS y Windows Phone. De esta manera, una misma app puede utilizarse en cualquier smartphone o tablet, indistintamente de su marca o fabricante.

Para ello, estas aplicaciones tienen componentes que permiten la adaptabilidad de un mismo código a los requerimientos de cada sistema.

Entre las ventajas de las aplicaciones híbridas sobre las nativas destacan:

- El desarrollo es más ágil y sencillo, por lo tanto, más económico.
- Sus actualizaciones son más fáciles de desarrollar e implementar y el mantenimiento es más sencillo.
- Un mismo código se puede utilizar en todos los sistemas y plataformas.
- No se requieren permisos externos para distribuir la app en las tiendas *online*.

- Permite abordar a un mercado de usuarios mucho más amplio.

Así mismo, aunque no aprovechan todo el rendimiento que ofrece cada sistema operativo, pueden tener acceso a los componentes nativos a través de plugins como Capacitor y Apache Cordova.

Ionic

Ionic es un SDK de front-end de código abierto basado en tecnologías web (HTML, CSS y JS) cuyo propósito es el de crear aplicaciones híbridas, por lo que ya de base estamos hablando de un proyecto multiplataforma.

Ionic es compatible y permite el desarrollo de aplicaciones para Android, iOS, Windows, WebApps y Amazon FireOS.

Además, al tratarse de un software multiplataforma, los equipos de desarrollo pueden trabajar también desde diferentes sistemas (Windows, MacOS, Linux..).

Al basar toda la maquetación en HTML y, en este caso concreto, Angular, nos permite el diseño de interfaces interactivas de manera sencilla y totalmente MVC, de forma que la lógica se traslada a los modelos y controladores y las vistas se diseñan por separado.

Además de ser consistente, es portable, es decir, el mismo HTML, JS y CSS puede ser usado por otros frameworks como Electron.

Ionic cuenta con decenas de componentes prediseñados con los que se puede crear de forma rapidísima tarjetas, alertas, botones, menús, formularios…

Todos estos componentes están adaptados a las interfaces nativas de los diferentes sistemas, por lo que, si por ejemplo usamos un componente «Alert», este se mostrará de manera diferente en Android que en dispositivos iOS.

Por último, hay que tener en cuenta que Ionic es un proyecto open source, por lo que es posible su uso libre y gratuito, por lo que, como ya hemos visto, además de ahorrar tiempo ahorraremos dinero en costes de licencias, etc.

Angular / TypeScript

Angular es un framework que permite crear aplicaciones web progresivas y aplicaciones multiplataforma. Es de código abierto, mantenido por Google.

Angular se basa en clases tipo "Componentes", contenidos en “Modulos”, cuyas propiedades son las usadas para hacer el enlace o binding de los datos. En dichas clases hay propiedades (variables) y métodos (funciones a llamar). Algunas de las ventajas que ofrece son:

- Mejora de la productividad: el planteamiento de arquitecturas estándar repercute directamente en la productividad del proyecto. Angular proporciona controladores, servicios y directivas para organizar el proyecto.

- Generación de código: Angular convierte tus plantillas en código altamente optimizado para las máquinas virtuales de JavaScript de hoy en día, ofreciéndote todas las ventajas del código escrito a mano con la productividad de un framework.

- División del código: Las aplicaciones de Angular se cargan rápidamente gracias al nuevo enrutador de componentes. Éste ofrece una división automática de códigos para que los usuarios sólo carguen el código necesario para procesar la vista que solicitan.

- Reutilización de código: El diseño de Angular adopta el estándar de los componentes web. Un componente en Angular es una porción de código que es posible reutilizar en otros proyectos de Angular sin apenas esfuerzo, en otras palabras, te permiten crear nuevas etiquetas HTML personalizadas, reutilizables y auto-contenidas, que luego se pueden utilizar en otras páginas y aplicaciones. Esto facilita el desarrollo de aplicaciones de manera mucho más ágil, pasando de un "costoso" MVC a un juego de puzles con los componentes.

- Angular CLI: Las herramientas de línea de comandos permiten empezar a desarrollar rápidamente, añadir componentes y realizar test, así como previsualizar de forma instantánea la aplicación.

Se ha decidido usar Angular porque, aunque la versión mas reciente de Ionic permite usar cualquier framework de front-end, con Angular tiene más recorrido y por tanto la documentación es más sólida y la comunidad de usuarios más grande.

Aunque Angular no te obliga a usar TypeScript, se ha decidido usarlo para la realización del proyecto por varias razones, una de las primeras es que la base de datos será MongoDB, que trabaja en formato JSON y por tanto TypeScript facilita la organización de los datos. Otra es la consistencia en la documentación. Por ejemplo, ES6 (o sea, ECMAScript 2015) ofrece varias formas diferentes de declarar un objeto, lo cual puede confundir a muchos. Con TypeScript esto no pasa, y toda la sintaxis y la manera de hacer las cosas en el código es la misma, lo que añade coherencia a la información y a la forma de leer el código. Además, la búsqueda de errores de runtime en JavaScript puede ser una tarea imposible. TypeScript proporciona detección temprana de errores (en tiempo de compilación), y tipado fuerte de clases, métodos, así como de objetos y APIs JavaScript ya existentes. TypeScript es un superset de ECMAScript 6, es decir, incluye todas las funcionalidades de ES6, y además incorpora una capa por encima con funcionalidades extra.

Esto, entre otras cosas, significa que puedes mezclar código TypeScript con código ES6 estándar sin problema, y el compilador de TypeScript seguirá pasando el código a ES5 (tanto ES6 como TypeScript se transpilan a ES5).

Capacitor

Capacitor es el puente de Ionic hacia lo nativo. Con capacitor podemos acceder desde las tecnologías de desarrollo web y Javascript a los recursos nativos de los dispositivos, permitiendo una comunicación sencilla y la utilización de todas las características necesarias para realizar aplicaciones asombrosas, consiguiendo un elevado rendimiento.

Gracias a Capacitor es muy sencillo acceder al SDK nativo de cada plataforma, con una interfaz de desarrollo unificada y optimizada para su uso con el framework Ionic. Capacitor es potente, sencillo y extensible vía plugins creados por el propio equipo de Ionic y la enorme comunidad de este framework. Los principales motivos por los que se ha elegido usar Capacitor frente a, por ejemplo, Cordova, son las siguientes:

- Excepcional integración con Frameworks Javascript por medio de Ionic: React, Angular y VueJS.
- Crear APPs PWA o escritorio (gracias a Electron).
- Mantenimiento por parte del equipo de Ionic.
- No existe incompatibilidad con las tiendas oficiales.
- Sin necesidad de utilizar plugins puedes acceder a capacidades como: cámara, geolocalización, notificaciones, acelerómetro, accesibilidad…
- Dispone de sus extensiones propias, y además es compatible con las de Cordova.
- Y tan importante como todo lo anterior: Rápido de instalar y ejecutar sin complicados procesos.

Express / NodeJS

Node (o más correctamente: *Node.js*) es un entorno que trabaja en tiempo de ejecución, de código abierto, multiplataforma, que permite a los desarrolladores crear toda clase de herramientas de lado servidor y aplicaciones en JavaScript. El entorno omite las APIs de JavaScript específicas del explorador web y añade soporte para APIs de sistema operativo más tradicionales que incluyen HTTP y bibliotecas de sistemas de ficheros. Algunas de sus ventajas son:

- El código está escrito en "simple JavaScript", lo que significa que se pierde menos tiempo ocupándose de las "conmutaciones de contexto" entre lenguajes cuando estás escribiendo tanto el código cliente como del servidor.
- JavaScript es un lenguaje de programación relativamente nuevo y se beneficia de los avances en diseño de lenguajes cuando se compara con otros lenguajes de servidor tradicionales (Python, PHP, etc.).
- El gestor de paquetes de *Node* (NPM del inglés: Node Packet Manager) proporciona acceso a cientos o miles de paquetes reutilizables. Tiene además la mejor en su clase resolución de dependencias y puede usarse para automatizar la mayor parte de la cadena de herramientas de compilación.

Express es un framework web, escrito en JavaScript y alojado dentro del entorno de ejecución NodeJS. Es, de hecho, el framework web más popular de Node, y es la librería subyacente para un gran número de otros frameworks web de Node populares. Proporciona mecanismos para:

- Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas).

- Integración con motores de renderización de "vistas" para generar respuestas mediante la introducción de datos en plantillas.
- Establecer ajustes de aplicaciones web como qué puerto usar para conectar, y la localización de las plantillas que se utilizan para renderizar la respuesta.
- Con miles de métodos de programa de utilidad HTTP y middleware a su disposición, la creación de una API sólida es rápida y sencilla.

A pesar de que *Express* es en sí mismo bastante minimalista, los desarrolladores han creado paquetes de middleware compatibles para abordar casi cualquier problema de desarrollo web. Hay librerías para trabajar con cookies, sesiones, inicios de sesión de usuario, parámetros URL, datos POST, cabeceras de seguridad y muchos más.

En contraposición, esta flexibilidad es una espada de doble filo. Hay paquetes de middleware para abordar casi cualquier problema o requerimiento, pero deducir cuáles son los paquetes adecuados a usar algunas veces puede ser muy complicado. Tampoco hay una "forma correcta" de estructurar una aplicación, y muchos ejemplos que puedes encontrar en Internet no son óptimos, o sólo muestran una pequeña parte de lo que necesitas hacer para desarrollar una aplicación web.

Se han elegido estas tecnologías porque, además de lo arriba descrito, han sido empleadas a lo largo del curso en la asignatura Diseño de Interfaces.

MongoDB

MongoDB es una base de datos distribuida, basada en documentos y de uso general que ha sido diseñada para desarrolladores de aplicaciones modernas y para la era de la nube. Es un sistema de base de datos NoSQL, orientado a documentos y de código abierto.

MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida. Las propias consultas son también JSON, por lo que se programan fácilmente.

Además, ofrece un modelo de datos flexible, es decir, los campos pueden variar de un documento a otro; no es necesario declarar la estructura de los documentos al sistema; los documentos se describen por sí mismos. Si se necesita agregar un nuevo campo a un documento, entonces el campo se puede crear sin afectar a todos los demás documentos del sistema, sin actualizar un catálogo central del sistema y sin desconectar el sistema.

Se ha decidido usar MongoDB porque, por un lado, es idóneo para proporcionar a la aplicación que se va a desarrollar la flexibilidad requerida y por otro, ha sido utilizado durante el curso en la asignatura Acceso a Datos.

Material

Material es un sistema de diseño creado por Google para ayudar a los equipos a crear experiencias digitales de alta calidad para Android, iOS, Flutter y la web.

Los componentes de material son bloques de construcción interactivos para crear una interfaz de usuario e incluyen un sistema de estados integrado para comunicar los estados de enfoque, selección, activación, error, desplazamiento, presionar, arrastrar y deshabilitar. El color, la tipografía y la forma de los componentes del material, como los botones, se pueden modificar fácilmente para que coincidan con el branding escogido.

En general, se puede decir que da un aspecto uniforme y un enfoque jerarquizado a una aplicación de manera sencilla. Se ha escogido porque, además de funcionar muy bien en combinación con Angular, se ha visto en el desarrollo del curso durante las clases de Programación Multimedia y Dispositivos Móviles.

Visual Studio Code

Visual Studio Code es un potente editor de código fuente multiplataforma Windows, Mac, Linux con reconocimiento de sintaxis de código y coloreado de una multitud de lenguajes e integración con Git.

VSC es un editor ligero, no un IDE completo, es decir no es un sustituto del Visual Studio, sino que más bien es una herramienta básica para cubrir las necesidades de edición de código simple. Lo más importante de este editor es su gratuidad, que es multiplataforma y que ofrece muchas posibilidades gracias a múltiples extensiones.

Git

Git es el software de control de versiones más utilizado. Entre otras muchas cosas, Git te permite subir y actualizar el código de tu página web a la nube de GitHub. De esta forma siempre puedes disponer de él cuando lo necesites. Pero, además, puedes:

- Conocer quién y cuándo ha realizado una determinada modificación.
- Realizar comparaciones entre versiones de una aplicación.
- Observar la evolución del proyecto con el paso del tiempo.
- Contar con una copia del código para poder retroceder ante cualquier imprevisto.
- Estar al tanto de los cambios en el código fuente.
- Tener una copia de seguridad del proyecto al completo.
- Disponer de un historial en el que se detallen las modificaciones realizadas en el código del sitio web.
- Git es software libre y open-source.

Trello

Trello es un software de administración de proyectos con interfaz web y con cliente para iOS y Android. Emplea el sistema Kanban para el registro de actividades que,

mediante tarjetas virtuales, organiza tareas, permite agregar listas, adjuntar archivos, etiquetar eventos, agregar comentarios y compartir tableros.

Trello es un tablón virtual en el que se pueden colgar ideas, tareas, imágenes o enlaces. Es versátil y fácil de usar pudiendo usarse para cualquier tipo de tarea que requiera organizar información. Es perfecta para la gestión de proyectos ya que se pueden representar distintos estados y compartirlas con diferentes persona que formen el proyecto. Con ella se intenta mejorar las rutinas de trabajo de un equipo generando prioridades, tiempos y avisos entre otras muchas funcionalidades.

Microsoft Office

Microsoft Office es una suite ofimática muy popular, de software privativo. Los programas de Office empleados para la realización del proyecto son:

- Word: Con el procesador de texto se ha realizado la totalidad de la memoria.
- Excel: Con el programa de hoja de calculo se ha realizado el diagrama de Gantt.
- PowerPoint: Se ha empleado para realizar la presentación final.

2.2 Planificación

En esta fase, el objetivo prioritario es identificar cada una de las necesidades que deben ser satisfechas al desarrollar el sistema final. Esto se hará efectivo mediante la obtención de requisitos que deben estar presentes en el sistema de información final, con la intención de dar solución a las necesidades identificadas con anterioridad. Al finalizar esta primera etapa en el desarrollo del producto se consiguen los requisitos, tanto funcionales como no funcionales, y su modelado mediante los casos de uso, desarrollados estos últimos en diagramas. Todo esto facilita la comprensión del sistema haciendo que pueda ser diseñado cumpliendo con todos los objetivos y cubriendo las necesidades de forma satisfactoria.

2.2.1 Requisitos

En esta sección la finalidad es detallar de una manera clara y concisa cada uno de los objetivos que presenta el proyecto, así como todas sus características.

Es de suma importancia para que el desarrollo del proyecto concluya con éxito que antes de empezarlo se tenga una completa y plena compresión de los requisitos del software.

IDENTIFICADOR

NOMBRE:
PRIORIDAD:
DESCRIPCIÓN:

Para organizar de la forma más clara posible todos los requisitos se utilizará esta tabla. El formato del identificador será RF o RNF (requisito funcional o requisito no funcional respectivamente), seguido de un número (un índice de dos dígitos). Estos identificadores podrán ser después usados para organizar tareas con mayor facilidad. Por ejemplo: “RF - 01”.

2.2.1.1 Requisitos funcionales

RF - 01
NOMBRE: Creación de cuenta de usuario
PRIORIDAD: Muy alta
DESCRIPCIÓN: El usuario debe poder crear una cuenta.

RF - 02
NOMBRE: Consulta de datos de la cuenta del usuario
PRIORIDAD: Muy alta
DESCRIPCIÓN: El usuario debe poder consultar los datos de su cuenta.

RF - 03
NOMBRE: Modificación de datos de su cuenta
PRIORIDAD: Muy alta
DESCRIPCIÓN: El usuario debe poder cambiar los datos de su cuenta, a excepción de nombre de usuario que se usará como id. Especialmente importante poder modificar la contraseña.

RF - 04
NOMBRE: Eliminación de la cuenta
PRIORIDAD: Media
DESCRIPCIÓN: El usuario debe poder eliminar su cuenta y todos los datos asociados..

RF - 05

NOMBRE: Inicio de sesión
PRIORIDAD: Muy alta
DESCRIPCIÓN: El usuario debe poder acceder a su cuenta.

RF - 06
NOMBRE: Inicio de sesión con Google
PRIORIDAD: Baja
DESCRIPCIÓN: El usuario debe poder acceder a la aplicación con su cuenta de Google.

RF - 07
NOMBRE: Cierre de sesión
PRIORIDAD: Muy alta
DESCRIPCIÓN: El usuario debe poder cerrar sesión.

RF - 08
NOMBRE: CRUD Item
PRIORIDAD: Muy alta
DESCRIPCIÓN: El usuario debe poder crear, ver, editar y eliminar sus items.

RF - 09
NOMBRE: CRUD Records
PRIORIDAD: Muy alta
DESCRIPCIÓN: El usuario debe poder crear, ver, editar y eliminar los registros de los diferentes Items.

RF - 10
NOMBRE: Visualización de evolución
PRIORIDAD: Muy alta
DESCRIPCIÓN: El usuario debe poder ver su evolución en el tiempo de manera gráfica (Reports).

RF - 11
NOMBRE: Amigos
PRIORIDAD: Media
DESCRIPCIÓN: El usuario debe poder añadir y eliminar amigos.

RF - 12
NOMBRE: Items públicos/privados
PRIORIDAD: Media
DESCRIPCIÓN: El usuario debe poder marcar sus Items como públicos o privados. Los públicos podrán ser vistos por los usuarios agregados como amigos, a su vez, el usuario podrá consultar los Items públicos de los amigos.

2.2.1.2 Requisitos no funcionales

RNF - 01
NOMBRE: Tiempo de desarrollo
PRIORIDAD: Media
DESCRIPCIÓN: El tiempo de desarrollo debe ser de aproximadamente s40 horas.

RNF - 02
NOMBRE: Seguridad
PRIORIDAD: Muy Alta
DESCRIPCIÓN: Control de sesiones, encriptación de contraseñas, seguridad en BBDD, validación y sanitización de datos de entrada en la app.

RNF - 03
NOMBRE: Interfaz de usuario (UI)
PRIORIDAD: Alta
DESCRIPCIÓN: El aspecto de la aplicación debe ser atractivo y visual. La tipografía debe ser clara y el diseño consistente.

RNF - 04
NOMBRE: Experiencia de usuario (UX)
PRIORIDAD: Alta
DESCRIPCIÓN: Satisfacer las necesidades del usuario de forma simple y clara. El resultado debe ser útil e intuitivo. La navegación debe ser simple.

RNF - 04
NOMBRE: Diseño de interacción (IxD)
PRIORIDAD: Media
DESCRIPCIÓN: En la medida de lo posible priorizar gestos y flujos de interacción propios de dispositivos móviles para ofrecer al usuario la sensación de estar usando una aplicación nativa. P. ej.: tocar, deslizar o mantener pulsado en lugar de doble click, click izquierdo…

RNF - 05
NOMBRE: Validación de formularios
PRIORIDAD: Media
DESCRIPCIÓN:

2.2.2 Gestión del proyecto

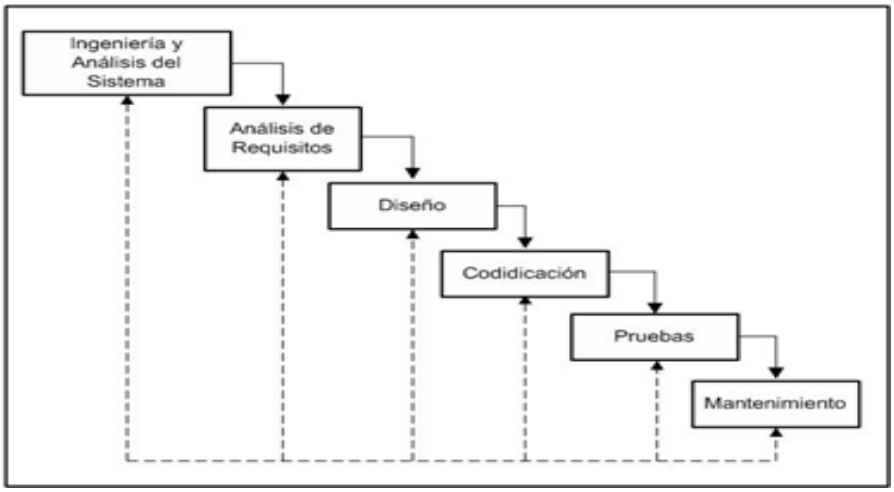
Para gestionar de manera correcta el proyecto se necesita elegir el ciclo de vida más apropiado para el proyecto en función de nuestras necesidades. Otra parte muy importante de la gestión del proyecto es la estimación de tiempo de las diferentes tareas, para lo cual se realiza un diagrama de Gantt.

2.2.2.1 Ciclo de vida del proyecto

El proceso para el desarrollo de software, también denominado ciclo de vida del desarrollo de software, es una estructura aplicada al desarrollo de un producto de software. Hay varios modelos a seguir para el establecimiento de un proceso para el desarrollo de software, cada uno de los cuales describe un enfoque distinto para diferentes actividades que tienen lugar durante el proceso.

El paradigma elegido es el desarrollo en cascada con retroalimentación. Este modelo admite la posibilidad de hacer iteraciones, es decir, durante las modificaciones que se hacen en el mantenimiento se puede ver, por ejemplo, la necesidad de cambiar algo en el diseño, lo cual significa que se harán los cambios necesarios en la codificación y se tendrán que realizar de nuevo las pruebas, es decir, si se tiene que volver a una de las etapas anteriores al mantenimiento hay que recorrer de nuevo el resto de las etapas.

Después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente.



Los motivos principales para su elección han sido que la planificación es muy sencilla, la calidad del producto resultante es alta, permite retroceder en cualquier momento si fuera necesario y es el más sencillo de utilizar cuando el desarrollador tiene poca experiencia.

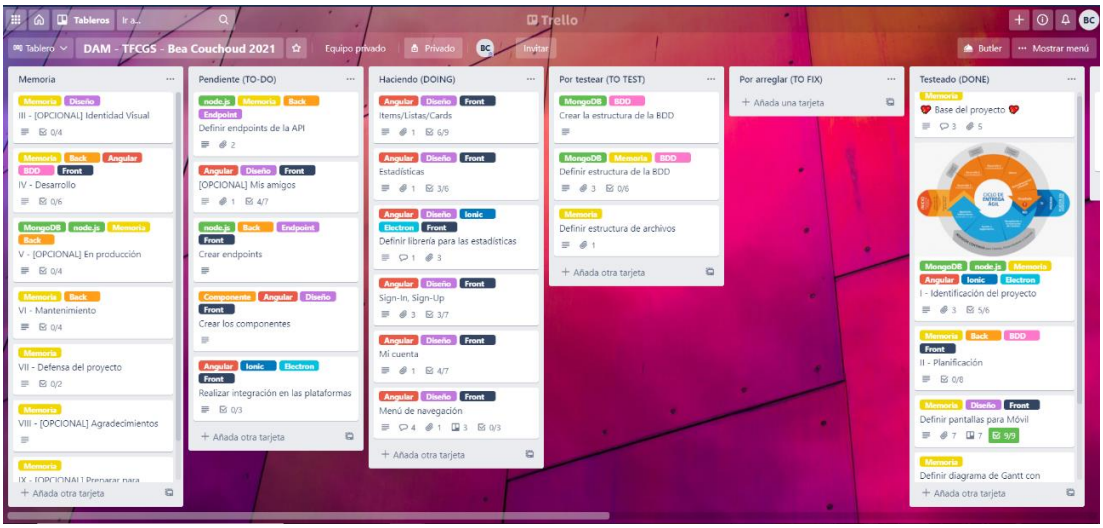
Una vez elegido el modelo de ciclo de vida del proyecto se ha procedido a planificar cada una de sus fases con el objetivo de estimar el tiempo que va a ser utilizado para la realización completa del proyecto.

Por otra parte, dentro del bloque de Codificación, se ha determinado usar la metodología de desarrollo SCRUM. Esta metodología se adapta a los problemas que van surgiendo, y se define y cambia tras cada sprint, lo cual permite mantener una estimación más adecuada de las tareas realizadas y pendientes, realizando pequeños ajustes constantes a los requisitos y estimaciones, resolviendo problemas y aprendiendo lecciones útiles para los siguientes sprints.

Para la codificación de esta aplicación se han determinado sprints semanales (de lunes a viernes) con un periodo de corrección de errores los fines de semana, ya que al realizarse de forma simultaneas a las FCT la gestión del tiempo se vuelve una tarea complicada y solventar problemas requiere mucho tiempo.

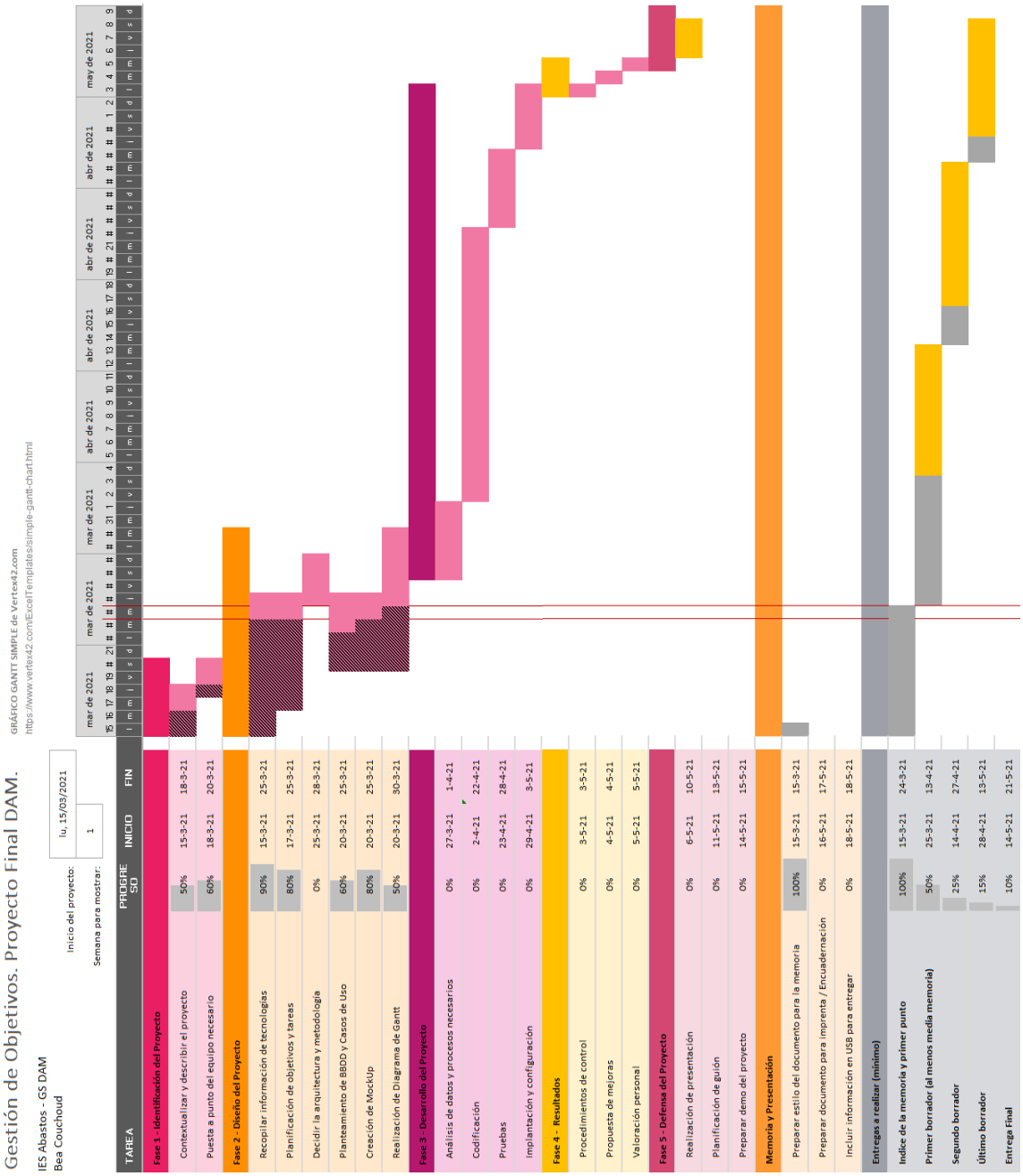
2.2.2.2 Trello

En el tablero de Trello podemos observar las tareas a realizar y el estado actual de cada una de ellas. Permite mejor organización. Muy útil en combinación con metodologías ágiles como SCRUM, ya que permite subdividir, modificar o aplazar tareas y modificar su estado rápidamente.



2.2.2.3 Diagrama de Gantt

En el diagrama de Gantt se puede observar la duración de cada uno de los plazos de realización de tareas, separados por las fases anteriormente descritas. Se puede observar que algunas de las tareas se solapan puesto que pueden ser desarrolladas de forma simultánea, siendo independientes unas con otras, mientras que otras necesitan que las anteriores hayan sido finalizadas para poder iniciarse. La estimación realizada plantea una duración de, aproximadamente, dos meses.



2.3 Diagramas UML y Base de Datos

El Lenguaje Unificado de Modelado o UML (“Unified Modeling Language”) es un lenguaje estandarizado de modelado. Está especialmente desarrollado para ayudar a todos los intervinientes en el desarrollo y modelado de un sistema o un producto software a describir, diseñar, especificar, visualizar, construir y documentar todos los artefactos que lo componen, sirviéndose de varios tipos de diagramas. Estos diagramas de UML son representaciones gráficas que muestran de forma parcial un sistema de información. Los diagramas se clasifican en dos grandes tipos: diagramas de estructura y diagramas de comportamiento. Se ha decidido utilizar para la realización de este proyecto los más representativos de cada uno de los grupos, ya que son los que proporcionan información más útil para la app que se está desarrollando.

Diagrama de Clases

El diagrama de clase es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), la visibilidad y las relaciones entre los objetos. En este caso se han representado todas las entidades que son relevantes para el sistema y sus interrelaciones.

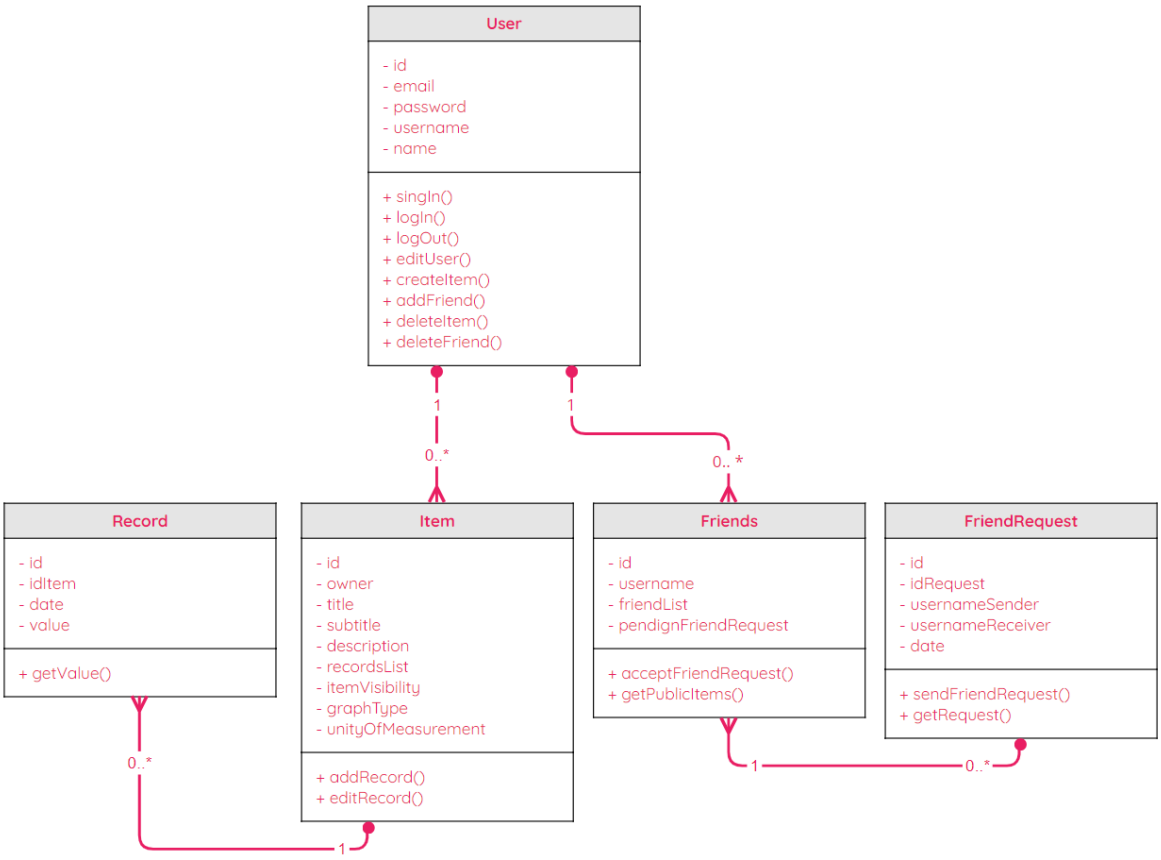
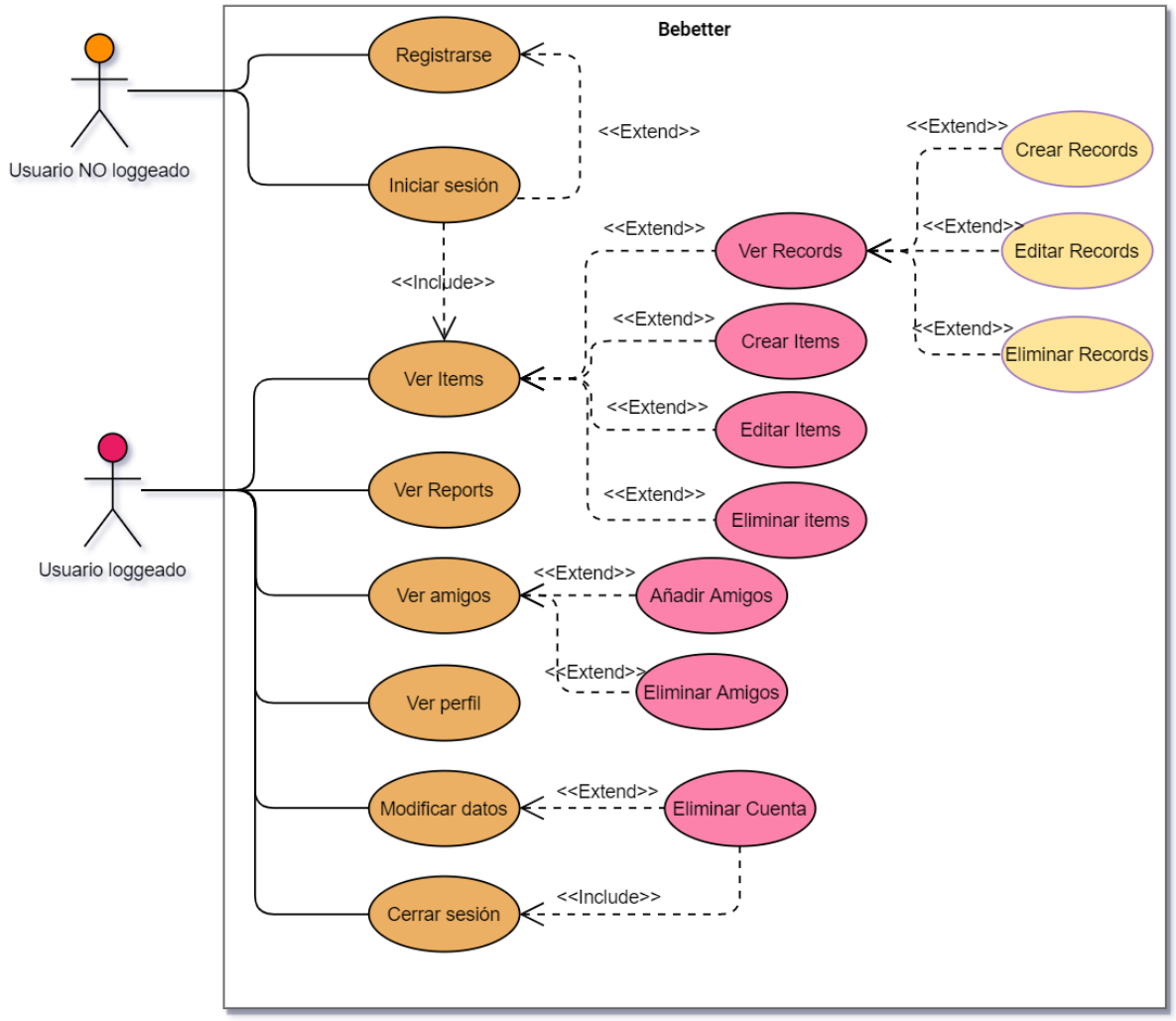


Diagrama de casos de uso

En el diagrama de casos de uso, las funciones del sistema en cuestión se representan desde el punto de vista del usuario (llamado “actor” en UML). De este modo, el diagrama de casos de uso muestra la relación entre un actor y sus requisitos o expectativas del sistema, sin representar las acciones que tienen lugar o ponerlas en un orden lógico.

A continuación, se detallarán los casos de uso, describiendo todas las posibles interacciones entre el usuario del sistema y la aplicación con el fin de alcanzar un objetivo. De esta manera se proporcionan diversos escenarios que indican la sucesión de respuestas surgidas ante las acciones iniciadas por los usuarios sobre el sistema. Esta especificación de casos de uso facilita la comprensión del sistema y expresa la intención con la que el usuario interactuará con la aplicación, profundizando así en dichas necesidades, y que, complementándose con los requisitos anteriormente descritos, establece una firme base a la hora de comenzar con la fase de diseño.

Se ha creado actor para usuarios que no han creado una cuenta o no han iniciado sesión todavía y otro para los usuarios que han iniciado sesión.



Base de Datos

A continuación, se incluye la estructura que formará la base de datos en MongoDB, que como se ha comentado previamente, esta compuesta de Colecciones y Documentos en formato similar a JSON (BSON), ya que se trata de una base de datos no relacional, es decir, no requiere de tablas para el almacenamiento de la información. En estos fragmentos que se muestran a modo de ejemplo se puede observas la estructura interna inicial de cada tipo de documento, aunque pueden variar levemente en un futuro porque las BBDD no relaciones permiten que los datos no estén completamente estructurados. Al mismo tiempo, estos ejemplos también sirven como guía para ver el tipo de dato de cada atributo de los elementos.

- Documento de MongoDB perteneciente a la colección **User**:

```
{
    "_id": ObjectId("5462ee906994db65eeab3075"),
    "username": “bcouchoud”,
    "name": "Bea Couchoud",
    "email": "beacouchoud@gmail.com",
    "password": "DAM2021"
}
```

- Documento de MongoDB perteneciente a la colección **Item**:

```
{
    "_id": ObjectId("5462ee906994db65eeab3076"),
    "owner": “bcouchoud”,
    “title”: “Example Item”,
    “subtitle”: “This is a exmple”,
    “description”: “This is a description of the item.”
    "recordList":
    [
        “idItem1”,
        “idItem2”,
        “idItem3”,
        “idItem4”
    ],
```

"itemVisibility": False,
“graphType”: 2,
"unityOfMeasurement": "vasos de agua"
}

- Documento de MongoDB perteneciente a la colección **Record**:

{
"_id": ObjectId("5462ee906994db65eeab3077"),
"idItem": “idItem1”,
"date": "25/03/2021",
"value": 6
}

- Documento de MongoDB perteneciente a la colección **Friends**:

{
"_id": ObjectId("5462ee906994db65eeab3078"),
"username": “bcouchoud”,
"friendsList”:
[
“usernameFriend1",
“usernameFriend2”,
“usernameFriend3”,
“usernameFriend4”
],
“pendingFriendRequest”:
[
“idRequest1”,
“idRequest2”
]
}

- Documento de MongoDB perteneciente a la colección **FriendRequest**:

{

"_id": ObjectId("5462ee906994db65eeab3077"),

"idRequest": “idRequest1”,

"usernameSender": "usernameUser1",

"usernameReceiver": “bcouchoud”,

“date”: “25/03/2021”

}

2.4 Guía de estilos

El objetivo de esta pequeña guía de estilos es estandarizar la estructura del contenido de la app, aportando coherencia, y simplificar el desarrollo de los diferentes layouts y sus futuras actualizaciones. En las siguientes páginas se describe, por un lado, todo lo referente al *branding* de la aplicación (colores, tipografías..), y por otro, la estructura y navegación, con el objetivo de obtener un diseño limpio y consistente que permita ofrecer la mejor usabilidad y experiencia de usuario posible.

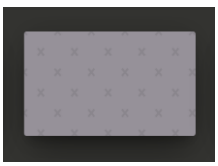
UI – Interfaz de usuario

Esta sección se enfoca en los colores, la tipografia, el logo, y en general, todo lo que defina la marca visualmente. El objetivo es que el usuario final tenga la sensación de estar usando una aplicación moderna y cuidada, a la vez que los distintos colores y elementos de diseño le proporcionan sentimientos de motivación, energía y optimismo para lograr sus propositos.

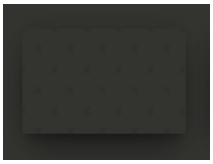
- Colores

Color principal - #E91E63
Color llamativo pero sin ser agresivo. Es el color principal, sello de identidad de la app. Transmite positividad y energía.

Color de acento claro - #FF8F00
Color que llama la atención, complementa perfectamente las sensaciones que transmite el color principal y contrasta con el resto de la paleta de colores.



Color de acento oscuro - #545454
Color que hace resaltar a los tonos principales de la paleta. Se puede emplear, por ejemplo, en botones secundarios o algunos textos e iconos.



Color del fondo - #292929
Color de fondo de la aplicación. Permite descansar la vista, transmite modernidad y hace resaltar aun más los llamativos colores principales.

Color del texto - #FFFFFF
Color utilizado para el texto y algunos iconos.

- Tipografia

Las fuentes escogidas siguen con la linea moderna que se busca para la aplicación, teniendo en cuenta en todo momento que deben ser perfectamente legibles en pantallas pequeñas. Además, ambas son fuentes variables, lo cual permite que el desarrollo de la aplicación sea mas cómodo y se adapten mejor a diferentes pantallas.

Quicksand es una fuente sans serif con acabados redondeados. Utiliza formas geométricas como base. Está diseñada para fines de presentaciones, pero se mantiene lo suficientemente legible como para utilizarla también en tamaños pequeños.

Asap es una fuente sans serif contemporánea con esquinas sutilmente redondeadas. Esta familia, especialmente desarrollada para su uso en pantallas y ordenadores, ofrece una anchura de caracteres estandarizada en todos los estilos, lo que significa que las líneas de texto mantienen la misma longitud. Esta útil característica permite a los desarrolladores cambiar de estilo de letra sobre la marcha sin tener que volver a escribir un cuerpo de texto.

- Logo

  El logo de la aplicación es un imagotipo, es decir, la representación visual de la marca incluye un elemento pictográfico y un texto o elemento que podemos leer.

  El elemento pictográfico consiste en una grafica que supera una barrera, una previsión u objetivo. Emplea los colores principales de la aplicación. En determinados momentos se usará sin acompañarlo del nombre de la app. Por otro lado la parte legible del logo emplea la fuente Asap con el color de acento oscuro anteriormente mencionados.

UX – Experiencia de usuario

Este apartado se centra en mejorar la satisfacción del usuario con la aplicación, aumentando las posibilidades de que siga usandola con el paso del tiempo. Para esto es necesario que la pagina sea intuitiva y útil. Hacer que esto suceda no es una tarea simple, por eso se ha puesto especial atención en puntos clave como el menú, la navegación entre páginas y el feedback que recibe el usuario a diferentes acciones, como, por ejemplo, rellenar un formulario.

- Menú

  La aplicación dispondra de un menú de hamburguesa, tambien accesible deslizando un dedo desde el lado izquierdo de la pantalla hacia el centro. El icono estará disponible en todas las páginas principales, pero en las subpaginas solo se podrá acceder deslizando. Cada elemento del menú tendrá un icono asociado y la pagina en la que se encuentre el usuario estará resaltada para evitar confusiones. Además en el panel del menú se verán los datos básicos del usuario.

- Navegación

  Las secciones principales serán rápidamente accesibles desde cualquier punto de la app gracias al menú principal. A todas las subpáginas se podrá acceder mediante botones que se encontrarán en los layouts principales.

- Formularios

  Los formularios son un elemento muy presente a lo largo de toda la aplicación. Todos los inputs indicarán sin lugar a dudas el dato que se debe introducir. El campo que tenga el foco en cada momento estará claramente resaltado. Todos los formularios serán breves y sencillos de completar para evitar que el usuario se sienta saturado.

IxD – Diseño de interacciones

El diseño de interacciones es un punto de encuentro entre el UI y el UX. En esta sección se detalla, por ejemplo, como reacciona la interfaz a las acciones del usuario. Esto

incluye también las diferentes llamadas a la acción, que ocurre si haces scroll o que sucede si hay un error al tratar de enviar un formulario.

- Botones y llamadas a la acción
  Los botones principales y más importantes ocuparán todo el ancho de la pantalla, contendrán textos muy claros y utilizarán un degradado con los colores principales de la app. Para dirigir la atención del usuario a realizar determinadas acciones en la aplicación se usarán botones flotantes, redondos, con los colores principales, bucando que destaquen lo máximo posible sobre el fondo. Estos botones son los que guiarán al usuario por la aplicación y a las distintas subpáginas. Contendrán iconos lo más descriptivos posibles.

- Gestos
  Para que la aplicación ofrezca una experiencia lo más similar a una app nativa de android se empleará el uso de gestos en lugar de las interacciones y eventos tipicos de una página web. Por ejemplo, el menú será un panel lateral deslizable, los elementos de las listas podrán deslizarse o mantenerse pulsados para ver las acciones disponibles.

Mockups

Un mockup es un modelo o un prototipo que se utiliza para exhibir o probar un diseño. La idea del mockup, en el marco del desarrollo de aplicaciones, es presentar la primera versión del diseño en el entorno apropiado. Esto permite la posibilidad de apreciar cómo será el resultado antes de que se proceda al desarrollo de los distintos layouts. En este caso concreto se ha añadido al mockup como será la navegación entre las pantallas y la forma de interaccionar con distintos elementos de la aplicación.

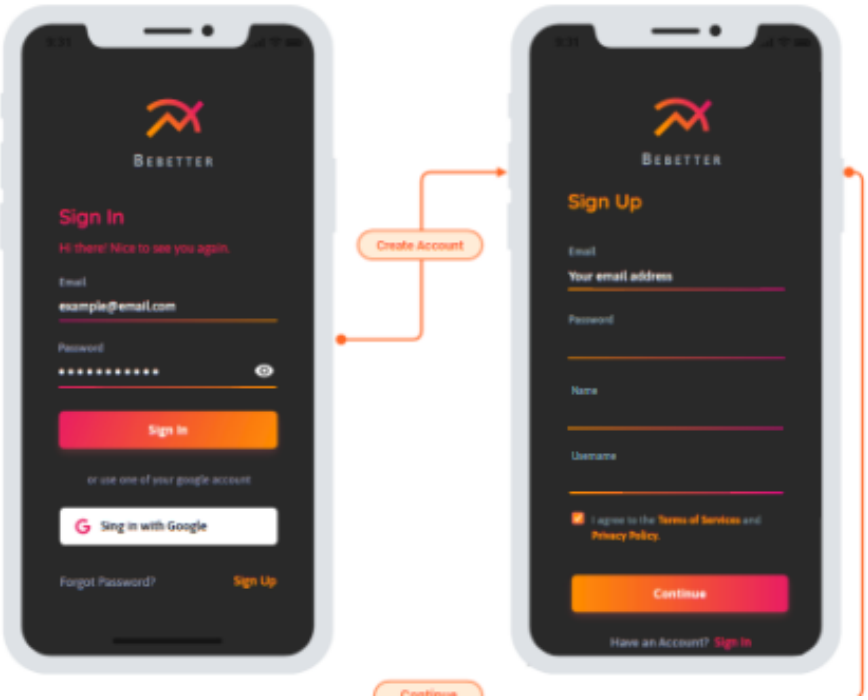


ILUSTRACIÓN 1- FORMULARIOS DE REGISTRO Y DE INICIO DE SESIÓN

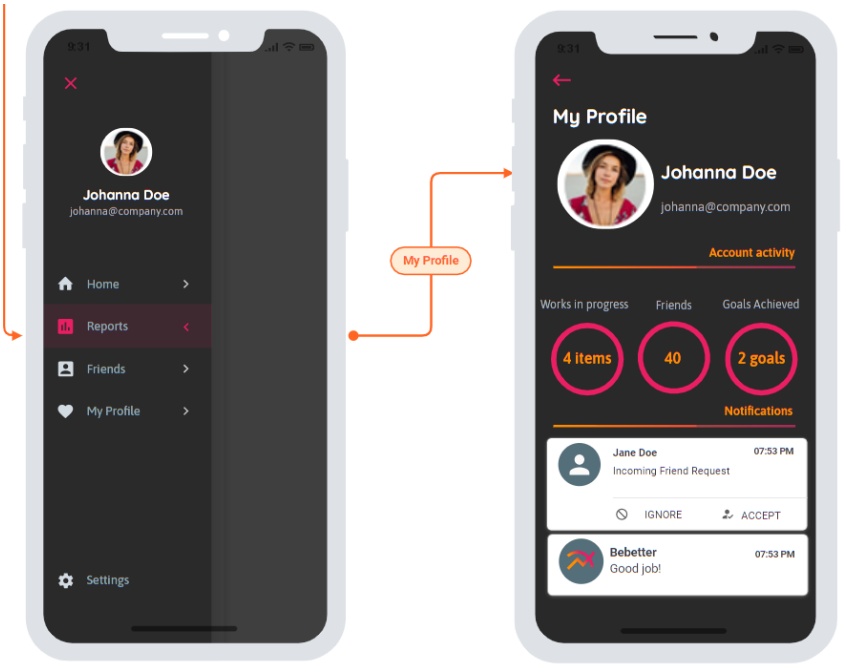


ILUSTRACIÓN 2 - MENÚ LATERAL Y PÁGINA DE PERFIL

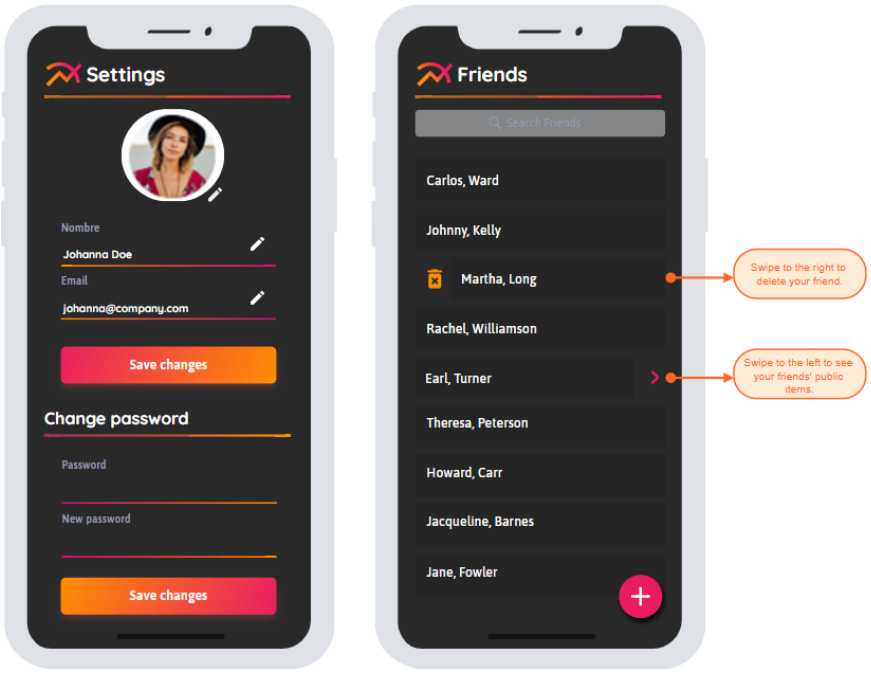


ILUSTRACIÓN 3 - PÁGINAS DE AJUSTES Y AMIGOS

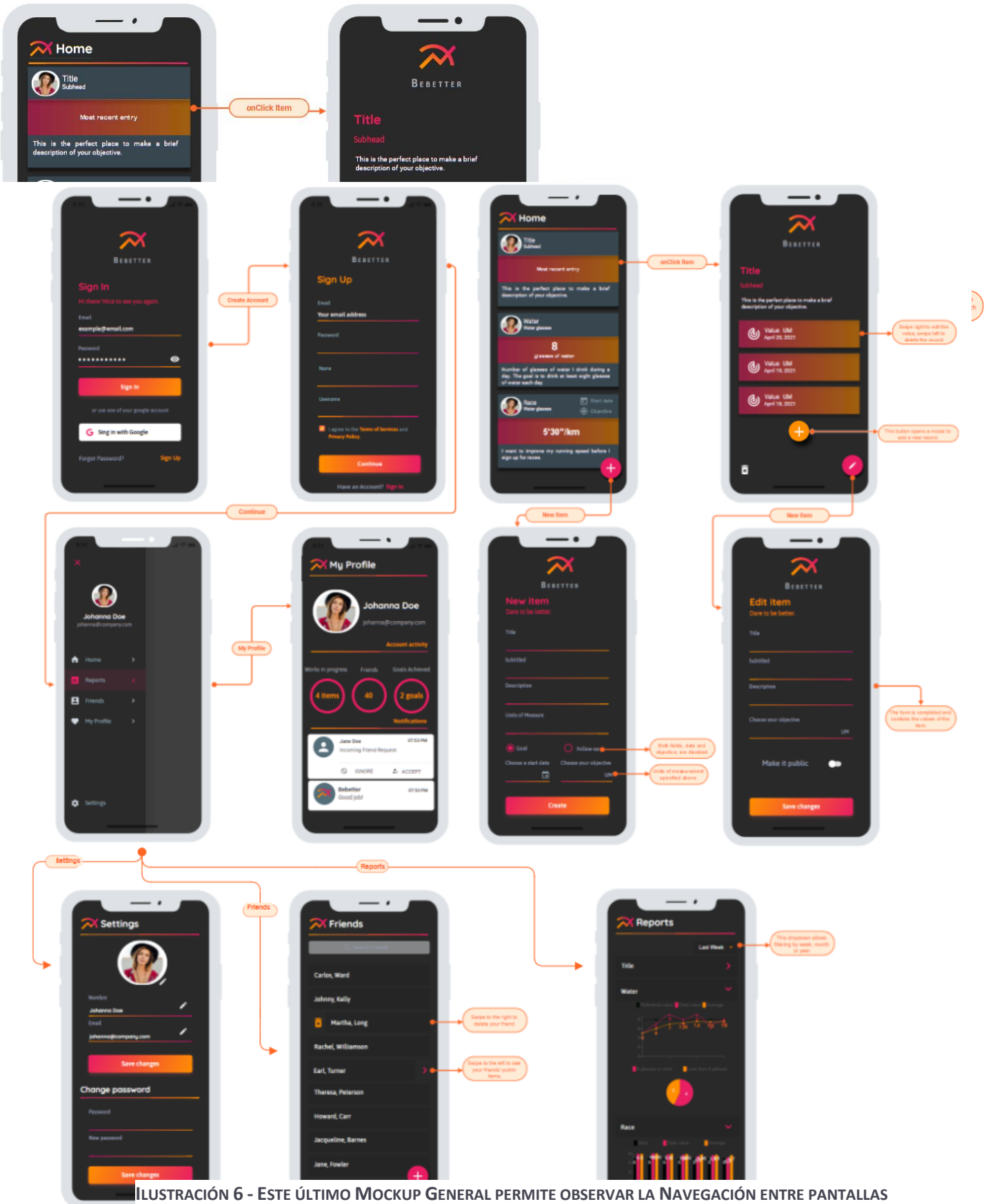


ILUSTRACIÓN 6 - ESTE ÚLTIMO MOCKUP GENERAL PERMITE OBSERVAR LA NAVEGACIÓN ENTRE PANTALLAS

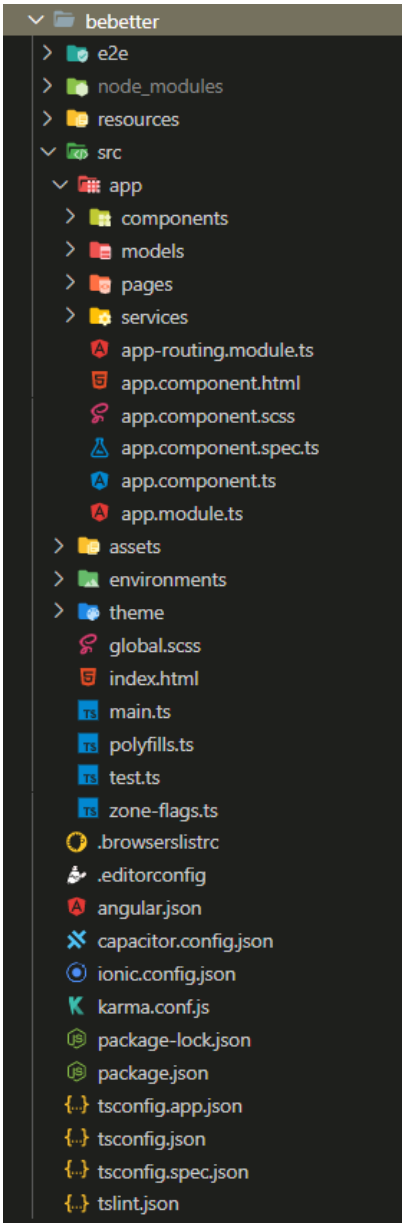
3. Desarrollo

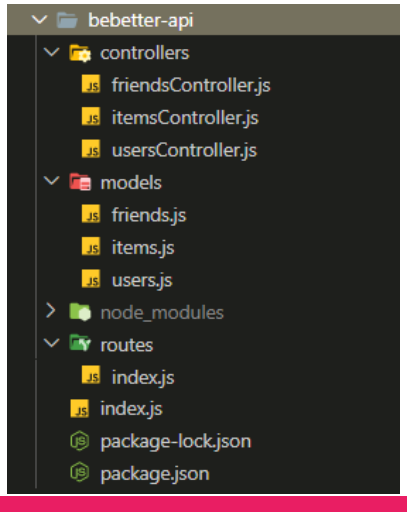
En este apartado se explicará cómo se ha creado todo el código detrás de la aplicación: La estructura escogida, el seguimiento de los archivos, como se desplegaría sobre un servidor…
Esta es la fase en la que se codifica los diferentes módulos y componentes del sistema. Aquí se han creado las funciones y procedimientos para llevar a cabo cada una de las operaciones que deben realizar cada uno. La técnica de desarrollo será la de sprints semanales, tal y como se comenta previamente en este documento, incluyendo refinamientos sucesivos para solventar problemas y errores. La prioridad es crear un prototipo (producto mínimo viable) lo antes posible e ir añadiendo funcionalidades al sistema paulatinamente.

3.1 Codificación

En esta sección se tratará como se han plasmado las ideas y funcionalidades con código. Esta es la fase en la que se realiza el desarrollo del sistema, las pruebas, la subida a producción...

3.1.1 Estructura de carpetas





El archivo **tsconfig.json** contiene configuración sobre la estructura de del proyecto.

La carpeta **e2e**, cuyo nombre es "end to end", contiene una serie de ficheros que se encargaran de realizar test automáticos, como si un usuario real interactuara con nuestra app.

La carpeta **node_modules** contiene todas las dependencias del proyecto.

En el archivo **.editorconfig** se encuentra la configuración del editor de código.

El archivo **tslint.json** es el linter de TypeScript, se usa para mantenibilidad y sostenibilidad del código.

También se puede observar la carpeta **assets**, que contiene los archivos estáticos, y **environments**, las variables de entorno, y muchos otros archivos que forman parte de nuestra aplicación. La carpeta **theme** contiene un archivo con las variables css.

Al crear el proyecto se genera, automáticamente, una carpeta llamada **app** la cual tiene un componente y una vista simple como demostración. Esta carpeta se mantiene en el proyecto para poder orquestar la aplicación como el módulo principal de la aplicación.

También se pueden observar las carpetas **components** (contiene los componentes), **models** (aquí se encuentran los modelos, interfaces, enumeraciones… que se necesiten), **pages** (donde se agrupan las páginas que se incluyen en la app) y services (donde se incluyen todos los servicios que se han ido creando).

El archivo **package.json** contiene todas las librerías y dependencias instaladas en el proyecto. Es el archivo de configuración principal del proyecto y debe encontrarse en la raíz del mismo. En el debe estar reflejado el nombre del proyecto, versión, descripción, scripts, autor, tipo de licencia y algo muy importante las dependencias.

El archivo **index.js** se usa para configurar el servidor. Aquí se realiza la conexión a la base de datos y contiene también la configuración de Express.

La carpeta **models** contiene los esquemas de datos que se usan en los JSON con los datos que se envían o reciben.

La carpeta **controllers** agrupa los distintos controladores. En este caso se ha decidido que cada archivo contenga los métodos correspondientes a uno de los modelos.

Por último, la carpeta **routes** tiene un archivo de rutas, en el cual se relaciona cada ruta con el método del controlador que se va a ejecutar.

3.1.1.1 Front-end

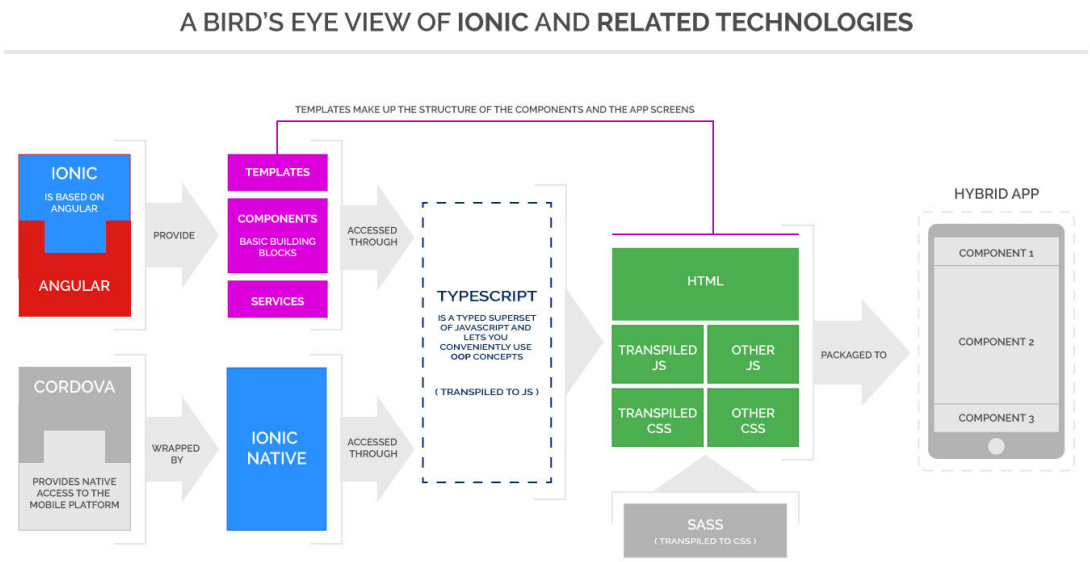


ILUSTRACIÓN 7 - ESQUEMA DEL FRONT-END

En la imagen que se muestra arriba se puede observar un esquema de la relación entre todas las partes que conforman el front de la app. Ionic se encarga sobre todo de lo relativo a los templates, aportando etiquetas propias al html que facilitan el desarrollo de layouts, mientras que Angular se encarga del comportamiento y renderizado de la app, controlando los eventos, las propiedades, el uso de directivas y filtros, realizando peticiones a la api y transformando datos en los servicios…
A continuación se incluye un esquema de la arquitectura utilizada por Angular, que no tiene un modelo-vista-controlador (MVC) clásico, sino que el modelo tiene mucha relación con la vista.

Esto es así por el concepto base de Angular de two-way data binding, ya que la forma de sincronizar los datos entre la vista y el modelo-vista es totalmente dependiente, es decir, en la vista podemos modificar el modelo y en el modelo podemos modificar la vista.

Esto hace que la independencia que se produce en un modelo-vista-controlador clásico aquí no se produzca, y por lo tanto tiende a llamarse modelo-vista vista-modelo (MVVM) o bien modelo-vista-whatever (MVW), porque no se sabe muy bien cómo identificarlo.

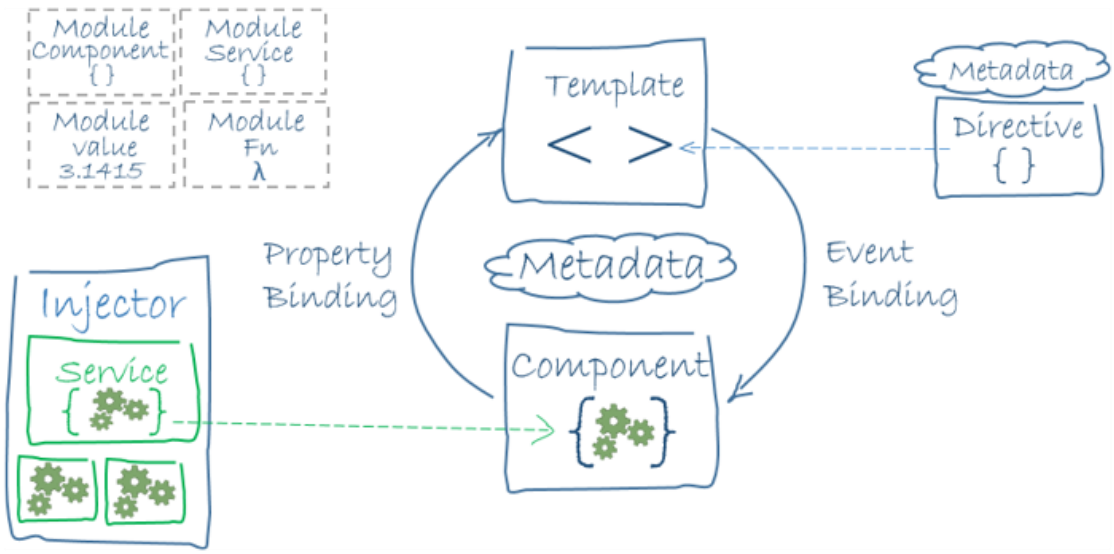


ILUSTRACIÓN 8 - ARQUITECTURA DE ANGULAR

A continuación, se hace una breve explicación de cada apartado que se observa en los esquemas mostrados, incluyendo fragmentos de código pertenecientes a la aplicación que se está desarrollando. Además se incluyen otros puntos relevantes en cuanto al desarrollo de front.

3.1.1.1.1 Componentes

Un componente controla un espacio de la pantalla, que se denomina vista. Es, en realidad, una clase de JavaScript (ES6) con el decorador **@Component** que incluye las propiedades y métodos disponibles para su template. Se debe incluir en esta clase todo lo referente al controlador de la vista y abstraer todos los demás métodos en servicios que serán inyectados posteriormente. Cada componente contiene una combinación de un archivo .html con un .ts y algunas veces un .css para crear un elemento con características propias tanto de comportamiento como de apariencia que se puede mostrar. Esto resulta de gran utilidad, por ejemplo, para evitar repetir código. También permite que el código sea más mantenible, y facilita la resolución de bugs y errores ya que los archivos son mas cortos y sencillos de leer.

El **template** es el encargado de definir la vista del componente. En el caso de Angular se trata de un HTML tradicional pero edulcorado con una serie de expresiones y directivas que mejoran el comportamiento de este y facilita el trabajo. En el template de un componente podemos encontrar, además de los tags normales de HTML, otros elementos distintos que utiliza Angular como ***ngIf, *ngFor, (event), [property], <component></component>** o la **interpolación** y los **pipes**.

3.1.1.1.2 Servicios

En las buenas prácticas de Angular se recomienda liberar a los componentes de cualquier lógica no relacionada con la vista. Se debe mover toda esa lógica, tal y como se ha mencionado anteriormente, a un servicio.

Básicamente un servicio es un proveedor de datos, que mantiene la lógica de acceso a los mismos y la operativa relacionada con el negocio y las cosas que se hacen con los datos dentro de una aplicación. Los servicios serán consumidos por los componentes,

que delegarán en ellos la responsabilidad de acceder a la información y la realización de operaciones con los datos. Para poder usar un servicio es necesario agregarlo a un módulo o componente. Algunos de sus propósitos son, entre otros, contener la lógica de negocio, clases para acceso a datos, encerrar funciones útiles. En este proyecto se han utilizado principalmente para invocar a un servidor HTTP para consumir una API. También es el mecanismo para compartir funcionalidad entre componentes ya que estas clases son perfectamente instanciables desde cualquier otro fichero que las importe

La librería **@angular/common/http** contiene el módulo **HttpClientModule** con el servicio inyectable **HttpClient**. Lo primero es importar dicho módulo. A partir de este momento sólo queda invocar los métodos REST en la propiedad this.http. Para cada verbo http tenemos su método en el servicio HttpClient. Su primer parámetro será la url a la que invocar. Estos métodos retornan un objeto **observable**. Los observables http han de consumirse mediante el método subscribe para que realmente se lancen. Dicho método subscribe admite hasta tres callbacks para responder a tres sucesos posibles: retorno de datos correcto, retorno de un error y señal de finalización.

El equipo de Angular decidió utilizarla para el desarrollo de las comunicaciones asíncronas. Toda operación asíncrona retornará una instancia observable a la cual habrá que subscribirse para recibir los datos o los errores cuando termine.

Los observables representan también una de las mejores formas de optimizar una aplicación, aumentando su rendimiento. El patrón observable no es más que un modo de implementación de la programación reactiva, que básicamente pone en funcionamiento diversos actores para producir los efectos deseados, que es reaccionar ante el flujo de los distintos eventos producidos. La programación reactiva es la programación con flujos de datos asíncronos. En definitiva, un observable es aquello que se quiere observar, que será implementado mediante una colección de eventos o valores futuros. Un observable puede ser creado a partir de eventos de usuario derivados del uso de un formulario, una llamada HTTP, un almacén de datos, etc. Mediante el observable es posible suscribirse a eventos que nos permiten hacer cosas cuando cambia lo que se esté observando.

Los métodos subscribe(), aunque vayan vacíos, son imprescindibles para que se ejecute la llamada. Esto puede resultar poco intuitivo, pero la realidad es que los observables Http de Angular sólo trabajan si hay alguien mirando.

Para este proyecto se ha separado la lógica de negocio en 3 servicios independientes en función de con qué elemento estén relacionados (usuarios, ítems o amigos), un servicio de utilidades para tener fácilmente accesibles métodos o datos que se requieren a lo largo de toda la aplicación y, por último, un servicio dedicado exclusivamente a la autenticación de los usuarios mediante un token que es llamado antes de realizar cualquier otro método de los demás servicios. Estos servicios después se instancian desde los demás componentes del proyecto.

3.1.1.1.3 Guards

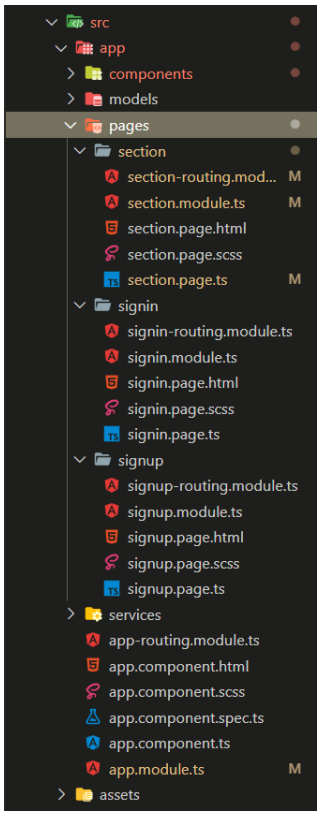
Los Guards en Angular son, de alguna manera, middlewares que se ejecutan antes de cargar una ruta y determinan si se puede cargar dicha ruta o no.

Como middleware, estos componentes se ejecutan de manera intermedia antes de determinadas acciones y si se obtiene true la ruta sigue su carga normal, en caso negativo, el Guard devuelve un false y la ruta no se carga. Generalmente en caso de que no se cumpla la condición del Guard, se suele hacer una redirección a la ruta anterior o a una ruta definida como la interfaz de autenticación.

Una de las ventajas de utilizarlos, es que al no cargar la ruta evitamos que los usuarios vean una interfaz a la que no tienen acceso (aunque sea por unos pocos milisegundos). Por otra parte, con estos componentes es posible estructurar el código de la aplicación de una manera más organizada y donde la lógica de la ruta está en la ruta en sí y la lógica de permisos y acceso a recursos se encuentra aislada en estos componentes.

Existen 4 tipos diferentes de Guards, pero para el desarrollo de esta aplicación solo ha sido necesario utilizar (CanActivate), que se ejecuta siempre antes de cargar los componentes de la ruta. Se han utilizado, como se puede apreciar en las imágenes que se muestran a continuación, para evitar que un usuario que no esta loggeado pueda acceder a determinadas rutas, y por otro lado, para que una vez está el usuario loggeado sea redirigido a Home y no al formulario de inicio de sesión si cierra la aplicación temporalmente o la deja en segundo plano sin cerrar la sesión.

3.1.1.1.4 Páginas y enrutamiento



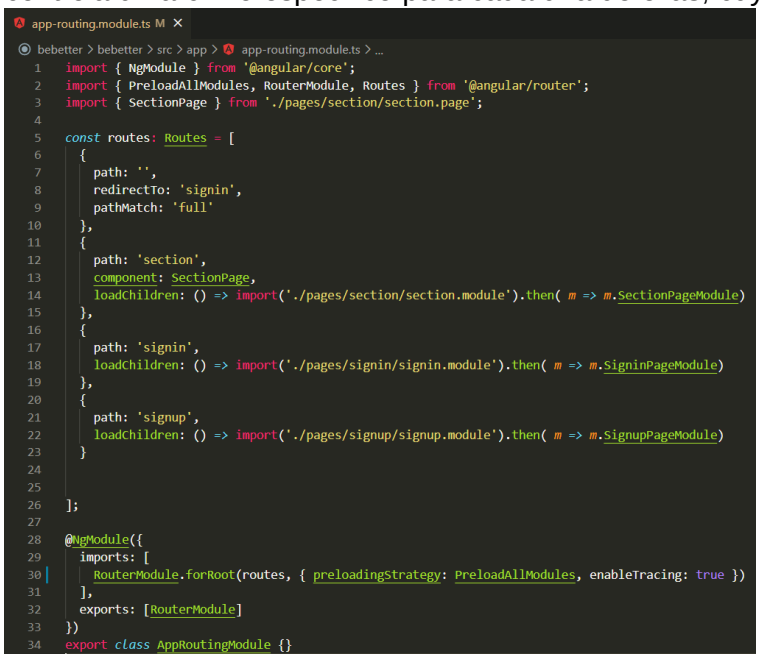
Una de las responsabilidades de las que se hace cargo el front es la de procesar las rutas y determinar cuál será la vista que se deba mostrar en cada dirección

En este caso concreto la aplicación tiene tres páginas, registro, inicio de sesión y section, que sería el contenedor en el que se vuelca el resto de secciones de la app y se cargan el resto de los componentes. Tal como se ve en el código más adelante, para poder asignar un componente a una ruta antes se ha de haber importado. Y eso significa que pasará a formar parte del código que se transpile, empaquete y envíe. Por esa razón, para evitar que el usuario se descargue la definición de los componentes antes de visitarlos e incluso componentes de rutas que quizá nunca visite, se ha decidido implementar Lazy Loading, que permite mejorar el rendimiento de la aplicación por una cuestión de peso y velocidad.

Por otro lado, el orden de las rutas es importante ya que Router accede a la primera ruta que coincide, por lo que las rutas más específicas deben colocarse por encima de las rutas menos específicas. La ruta comodín aparece en último lugar porque coincide con todas las URL y la Router selecciona solo si ninguna otra ruta coincide primero.

A continuación se muestran imágenes que contienen fragmentos de código de cada uno de los archivos y se realiza una breve explicación de su contenido y como se aplica el Lazy Loading.

La navegación entre estas tres paginas se controla mediante archivo de rutas general de la aplicación (app-routing-module.ts) y la navegacion dentro de cada una de las páginas la controla un archivo especifico para cada una de ellas, cuyo objetivo es diferir la descarga de las rutas no visitadas y para ello se empaqueta cada ruta en un bundle.



Es decir, cuando se active la ruta section, por ejemplo, entonces se le transfiera el control al módulo SectionPageModule mediante una instrucción asíncrona. De esta forma se consiguen dos cosas: por un lado al no usarse ningún componente explícito no hay que importarlo; por otro lado la descarga del módulo que resuelva la ruta se ejecutará en segundo plano y sólo si el usuario decide visitar la ruta.

```
section-routing.module.ts M
bebetter > bebetter > src > app > pages > section > section-routing.module.ts > route
import { UserSettingsComponent } from 'src/app/components/user-sett
import { ViewItemComponent } from 'src/app/components/view-item/vie
import { HomeComponent } from '../../components/home/home.componen

const routes: Routes = [
  {
    path: '',
    redirectTo: 'Home',
    pathMatch: 'full',
  },
  {
    path: 'Home',
    children: [
      {
        path: '',
        data: { title: 'Home' },
        component: HomeComponent,
      },
      {
        path: 'NewItem',
        data: { title: 'New Item' },
        component: NewItemFormComponent,
      },
      {
        path: 'EditItem/:id',
        data: { title: 'Edit Item' },
        component: EditItemFormComponent,
      },
      {
        path: 'ViewItem/:id',
        data: { title: 'View Item' },
        component: ViewItemComponent,
      },
    ],
  },
  {
    path: 'Reports',
    data: { title: 'Reports' },
    component: ItemsReportsComponent,
  },
  {
    path: 'Friends',
    data: { title: 'Friends' },
    component: FriendsComponent,
  },
  {
```

La instrucción loadChildren delega el enrutado en otro módulo; el SectionPageModule. Dicho módulo depende a su vez de su propio archivo de enrutamiento, el SectionPageRoutingModule. Las rutas que aparecen en este archivo se concatenan a la ruta padre del archivo anterior. Además aquí, por una cuestion de usabilidad, se ha decidido emplear rutas anidadas. También se puede observar algun ejemplo de ruta con parametros y redirección.

Para cargar los componentes que corresponden con cada página se emplea un componente llamado <router-outlet> y actúa como un contenedor dinámico, incrustando el componente adecuado apara cada ruta.

Además, para que los hijos aparezcan dónde deben, hay que usar un segundo <router-outlet> dentro de la vista del componente padre (SectionPageModule).

```
      </ion-content>
      <ion-footer>
        <ion-item routerDirection="root" [routerLink]="'/signin'" lines="none" detail="false" >
          <ion-icon name="log-out" class="ion-margin-end"></ion-icon>
          <ion-label class="ion-margin-start">Log Out</ion-label>
        </ion-item>
      </ion-footer>
    </ion-menu>
  </ng-container>
    <ion-router-outlet id="main-content"></ion-router-outlet>
  </ion-split-pane>
</ion-app>
```

```
</ion-header>
<ion-content [fullscreen]="false">
  <div id="container">
    <ion-router-outlet></ion-router-outlet>
  </div>
</ion-content>
<ion-footer>
</ion-footer>
```

Otra directiva que se ha empleado ha sido routerLink. En concreto esta directiva, que también viene en el módulo routerModule, se usa en sustitución del atributo estándar href. Instruye al navegador para que no solicite la ruta al servidor, sino que el propio código local de JavaScript se encargue de procesarla. Por último, otra forma que se ha empleado de navegar entre componentes es, sencillamente, desde el propio componente.

```
public signup(): void {
  if (this.form.valid) {
    this.userService.createUser(this.form.getRawValue())
    .subscribe((usu: IUser) => {
                                this.user = usu;
                                this.router.navigate(['/signin']);},
                                (error) => this.handleError(error));
```

```
<ion-footer>
  <ion-row>
    <ion-col>
      <p>Have an account? <ion-text [routerLink]="'/signin'">Sign In</ion-text></p>
    </ion-col>
  </ion-row>
</ion-footer>
```

3.1.1.1.5 Formularios y seguridad

Formularios reactivos

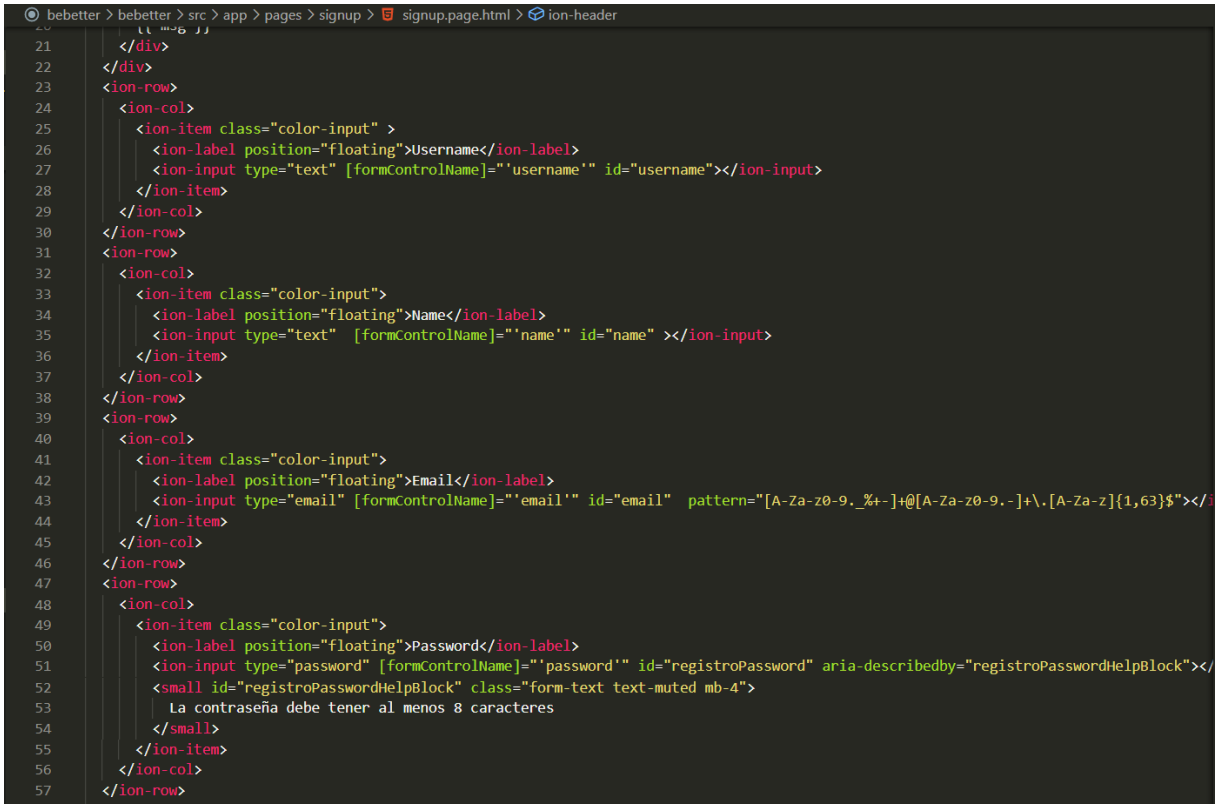
FormBuilder es un servicio que permite desacoplar el modelo del fomulario de la vista. Permite construir un formulario creando un FormGroup, (un grupo de controles) que realiza un seguimiento del valor, el estado de cambio y la validez de los datos. El formulario se define como un grupo de controles. Cada control tiene un nombre y una configuración. Esa definición permite establecer un valor inicial al control, asignando valores por defecto. Incluso permite modificar o transformar datos previos para ajustarlos a cómo los verá el usuario; sin necesidad de cambiar los datos originales de la base de datos.

```
23   public formUserData: FormGroup;
24   public formPassword: FormGroup;
25
26   constructor(private fb: FormBuilder) { }
27
28   ngOnInit() {
29     this.initFormUserData();
30     this.initFormPassword();
31   }
32
33   private initFormUserData(): void {
34     this.formUserData = this.fb.group({
35       name: [this.user.name, [Validators.required]],
36       email: [this.user.email, [Validators.email, Validators.required]]
37     });
38   }
39
40   private initFormPassword(): void {
41     this.formPassword = this.fb.group({
42       oldPassword: [null, [Validators.minLength(8), Validators.required]],
43       newPassword: [null, [Validators.minLength(8), Validators.required]]
44     });
45   }
46
```

En esta imagen se puede ver un componente que tiene dos formularios distintos, el segundo es normal (se espera que el usuario introduzca los datos, por defecto el valor de los campos de entrada es null), en cambio en el primero de ellos precargamos los datos del usuario que tiene la sesion iniciada (este formulario le permite modificar sus datos personales) para que se muestren directamente al cargar la página.

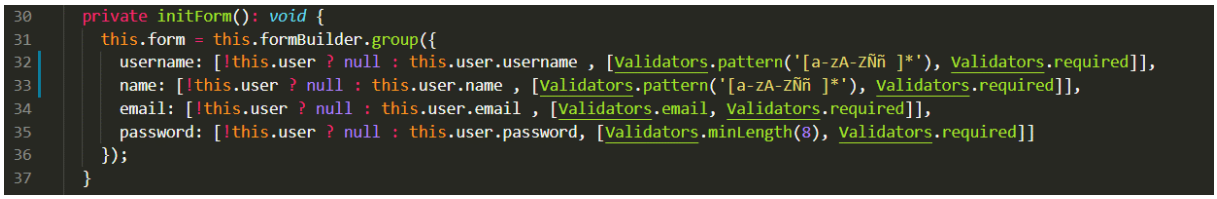
Validación y estado de cambio de formularios

La validación es una pieza clave de la entrada de datos en cualquier aplicación. Es el primer frente de defensa ante errores de usuarios; ya sean involuntarios o deliberados. Validar correctamente los campos protege a la aplicación de entradas maliciosas y provee también una mejor experiencia de usuario. La validación de campos reactivos es simple. Por defecto, Angular contiene validaciones para la gran mayoría de casos, pero también permite crear nuevas.



Ionic permite aplicar a sus componentes seguridad básica al igual que en html, indicando el tipo de input, haciéndolos obligatorios, incluyendo expresiones regulares… Los atributos que aparecen entre corchetes son directivas de Angular, necesarias para los formularios reactivos y acceder a los diferentes inputs desde el componente.

Además, los formularios también se validan en el propio componente antes de enviar los datos al back-end, para ello Angular proporciona un conjunto de validadores integrados que pueden usar los controles de formulario (también permite crear validadores personalizados mediante la clase Validators). Un validador es una función que procesa una colección de controles y devuelve un map de errores o null. Un map null significa que la validación ha pasado sin problemas.



Al establecerse reglas para los controles se activa el sistema de validación y control del estado de cada uno de ellos, así como del formulario en su conjunto. Los estados de validación mas usados son valid e invalid y los de modificación son dirty, touched y untouched. Aun que hay algunos más no se han empleado en el desarrollo de la app.

Esto permite, por ejemplo, controlar fácilmente si se debe habilitar o deshabilitar algún campo o botón, además de validar los datos fácilmente antes de sanitizarlos por completo en la API, lo cual aporta una capa de seguridad extra al sistema. Por otro lado, también facilita el feedback al usuario final si hay algún error en el formulario,

ya que la validación particular para cada control permite informar al usuario del fallo concreto.

Gestión de credenciales

Arpovechando el conocimiento adquirido sobre Interceptores y Observables se ha montado un pequeño sistema de gestión de credenciales. La idea es detectar respuestas a llamadas no autenticadas y redirigir al usuario a nuestra pantalla de registro. Si el usuario se registra correctamente recibiremos un token que lo identifica. Lo que haremos será guardarlo en una cookie de sesión y usarlo en el resto de las llamadas. De la misma manera, cuando el usuario cierre sesión se eliminará el token guardado.

Tal y como se ha comentado anteriormente se ha creado un servicio para interceptar los fallos de seguridad. En este caso interesan las respuestas con error 401 (Not Authorized). Se ha empleado el Router de Angular para obligar al usuario a visitar la página de inicio de sesión cuando esto ocurra. Por último, se proveerá el servicio creado en el archivo app.module.ts, para que se accesible por todos los componentes y servicios de la app.

De esta manera, antes de realizar cualquier petición a la API se añadirá el id de sesión a modo de token en la cabecera de todas las llamadas.

En definitiva, los interceptores son una manera sencilla de manipular las peticiones y respuestas, evitando así realizar dicha modificación en cada servicio que realizamos manualmente. Esto nos ayuda a tener un código limpio y mantenible.

3.1.1.2 Back-end

El front-end muestra la información, pero es el back-end quien realmente trabaja con esta y filtra todas las acciones, enruta los accesos y vigila que todo fluya como tenga que fluir evitando accesos no permitidos según el usuario.
En este apartado se tratarán todas las funciones relacionadas con el servidor. Tanto la conexión con la base de datos como la lógica de negocio.
En el back se ha construido un API REST usando para ello la tecnología de NodeJS y MongoDB como base de datos. Se ha optado por esta solución a causa de la restricción de horas para la realización de este proyecto, ya que brinda la facilidad de escalar y hacer crecer el sistema de una manera muy sencilla. La arquitectura REST expone a los clientes una interfaz uniforme desde el servidor ya que todos los recursos del servidor tienen un nombre en forma de URL o hipervínculo y toda la información se intercambia a través del protocolo HTTP.

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');

const routes = require('./routes');
const app = express();

mongoose.Promise = global.Promise;
mongoose.connect(
    'mongodb://localhost/bebetter',
    {
        useNewUrlParser: true
    }
);

app.use(express.json());
app.use(express.urlencoded({extended: true}));
app.use(cors());

app.use('/', routes());

app.listen(5000, function(){
    console.log('servidor express en ejecución')
})
```

En este archivo se pueden observar algunas de las dependencias requeridas, la conexión con la base de datos, el puerto que se va a usar, la configuración de la app… El objeto **app** hace referencia por convención a la aplicación Express.

ILUSTRACIÓN 9 - ARCHIVO DE CONFIGURACIÓN DEL SERVIDOR

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const userSchema = new Schema({
    username: {
        type: String,
        trim: true,
        unique: true
    },
    name: {
        type: String,
        trim: true
    },
    email: {
        type: String,
        trim: true,
        unique: true,
        lowercase: true
    },
    password: {
        type: String,
        trim: true
    }
});

module.exports = mongoose.model('users', userSchema);
```

Ejemplo de uno de los modelos incluidos en la API, concretamente es el esquema creado para la información de los usuarios. Se pueden observar distintas propiedades que permiten definir con mayor precisión como será la información contenida en cada uno de los datos, por ejemplo, **trim** limpia los datos, **type** indica el tipo, **unique** verifica que el valor de la propiedad no se repita en los distintos documentos de la colección…

ILUSTRACIÓN 10 - MODELO DE USUARIO

A la derecha se ha incluido un fragmento de código perteneciente al controlador de ítems. Aquí se pueden ver algunos de los métodos creados y las consultas a la base de datos que se realizan.

```
const items = require('../models/items');

//crear nuevo item
exports.addItem = async(req, res) => {
    const item = new items(req.body);
    try {
        await item.save();
        res.json({msg: 'Nuevo item creado'});
    } catch(error) {
        console.log(error);
        res.send(error);
        next();
    }
}

//devuelve todos los items privados de un usuario
exports.listPrivateItems = async(req, res) => {
    try {
        const allItems = await items.find({"owner": req.params.owner, "private": true});
        res.json(allItems);
    } catch (error) {
        console.log(error);
        res.send(error);
        next();
    }
};

//devuelve todos los items publicos de un usuario
exports.listPublicItems = async(req, res) => {
    try {
        const allItems = await items.find({"owner": req.params.owner, "private": false});
        res.json(allItems);
```

ILUSTRACIÓN 11 - FRAGMENTO DE UN CONTROLADOR

```
const express = require('express')
const router = express.Router();
const usersController = require('../controllers/usersController');
const itemsController = require('../controllers/itemsController');
const friendsController = require('../controllers/friendsController');

//return router
module.exports = function() {
    //devuelve todos los usuarios
    router.get('/users', usersController.listUsers);

    //crea un nuevo usuario
    router.post('/users', usersController.addUser);

    //devuelve un usuario
    router.get('/users/:id', usersController.getUser);

    //actualiza datos de un usuario
    router.put('/users/:id', usersController.updateUser);

    //elimina todos los datos de un usuario
    router.delete('/users/:id', usersController.deleteUser);



    //crear nuevo item
    router.post('/item', itemsController.addItem);

    //devuelve todos los items privados de un usuario
    router.get('/itemsPrivados/:owner', itemsController.listPrivateItems);

    //devuelve todos los items publicos de un usuario
    router.get('/itemsPublicos/:owner', itemsController.listPublicItems);

    //obtener un item
```

ILUSTRACIÓN 12 - FRAGMENTO DEL ARCHIVO DE RUTAS

Por último, en este archivo relacionamos cada una de las URI a las que se enviaran peticiones desde el cliente con el método que se necesite usar.

Los métodos HTTP que se usan en la API son:

GET: Para solicitar información al servidor.

POST: Para enviar datos al servidor.

PUT: Para enviar datos al servidor respecto a un id conocido. Es decir, para editar datos de un documento ya existente en la base de datos.

DELETE: Para eliminar un recurso concreto.

3.1.1.2.1 Seguridad

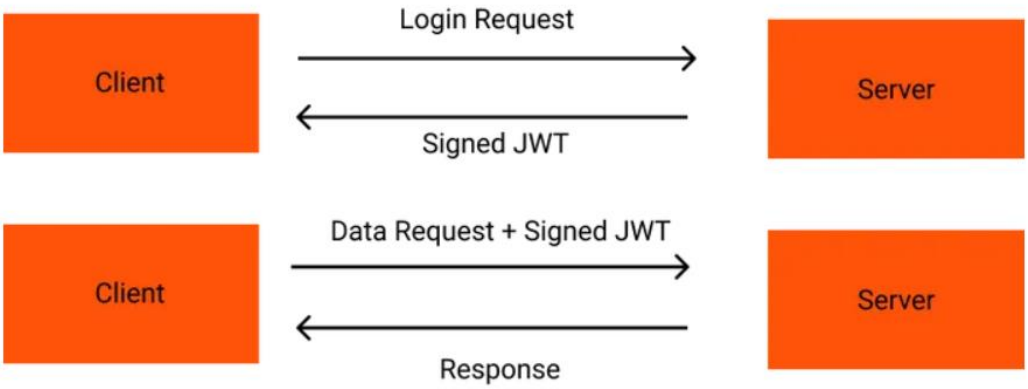
La seguridad es un tema de suma importancia en la actualidad. En nuestro caso, al no permitir pagos online ni exigir identificación mediante algún documento oficial, podemos afirmar que no se guardan datos extremadamente sensibles de los usuarios y, por tanto, es poco probable que haya vulnerabilidades de seguridad graves. Aun así, se han implementado unas cuantas prácticas para garantizar un mínimo de confianza:

- Se ha filtrado y saneado la entrada de usuario, usando mongo-sanitize, un modulo que evita que cualquier palabra que comience con '$' sea introducida en una query de MongoDB, para proteger ataques de scripts entre sitios (XSS) e inyección de mandatos. Por otro lado, se ha evitado el uso de $where ya que es uno de los operadores menos seguros de usar, ya que, aunque se filtren datos es difícil evitar que se introduzca un 0 o un true como valor en determinadas querys. Además, al haber creado un esquema o modelo para cada tipo de objeto en la api queda todavía mas delimitado el tipo de dato que se puede introducir en cada elemento y por tanto en las querys y la base de datos.

- Se ha limitado el tamaño del cuerpo de las peticiones. Al igual que se controla el tipo de los datos de entrada, también se ha limitado su tamaño máximo para evitar ataques DoS. Para eso se ha utilizado body-parser, otro de los middlewares más usados (y ahora integrados en Express), cuya función es llevar a cabo el parseo del cuerpo de las peticiones. Por defecto tiene un tamaño máximo del body de 100Kb.

- Se encriptan las contraseñas para mayor seguridad con la ayuda de bcrypt,una librería de encriptación que, básicamente, es una función de hashing de contraseñas, basado en cifrado Blowfish. Lleva incorporado un valor llamado salt, que es un fragmento aleatorio que se usará para generar el hash asociado a la password, y se guardará junto con ella en la base de datos. Así se evita que dos passwords iguales generen el mismo hash y los problemas que ello conlleva, por ejemplo, ataque por fuerza bruta a todas las passwords del sistema a la vez. Otro ataque relacionado es el de Rainbow table (tabla arcoíris), que son tablas de asociaciones entre textos y su hash asociado, para evitar su cálculo y acelerar la búsqueda de la password. Con el salt, se añade un grado de complejidad que evita que el hash asociado a una password sea único.

- Se ha implementado una autenticación mediante token. JWT (JSON Web Token) es un estándar qué está dentro del documento RFC 7519. En este se define como un mecanismo para poder propagar entre dos partes, y de forma segura, la identidad de un determinado usuario. La firma de un token JWT se construye de tal forma que vamos a poder verificar que el remitente es quien dice ser, y que el mensaje no se ha modificado por el camino, es decir, si alguien modifica el token por el camino, por ejemplo, inyectando alguna credencial o algún dato malicioso, entonces podríamos verificar que la comprobación de la firma no es correcta, por lo que no podemos confiar en el token recibido y deberíamos denegar la solicitud de recursos que nos haya realizado, ya sea para obtener datos o modificarlos.



Evidentemente quedarían decenas de medidas de seguridad que aún deberíamos tener en cuenta, pero es un buen comienzo como primer acercamiento de fácil aplicación.

3.1.1.2.2 Middlewares

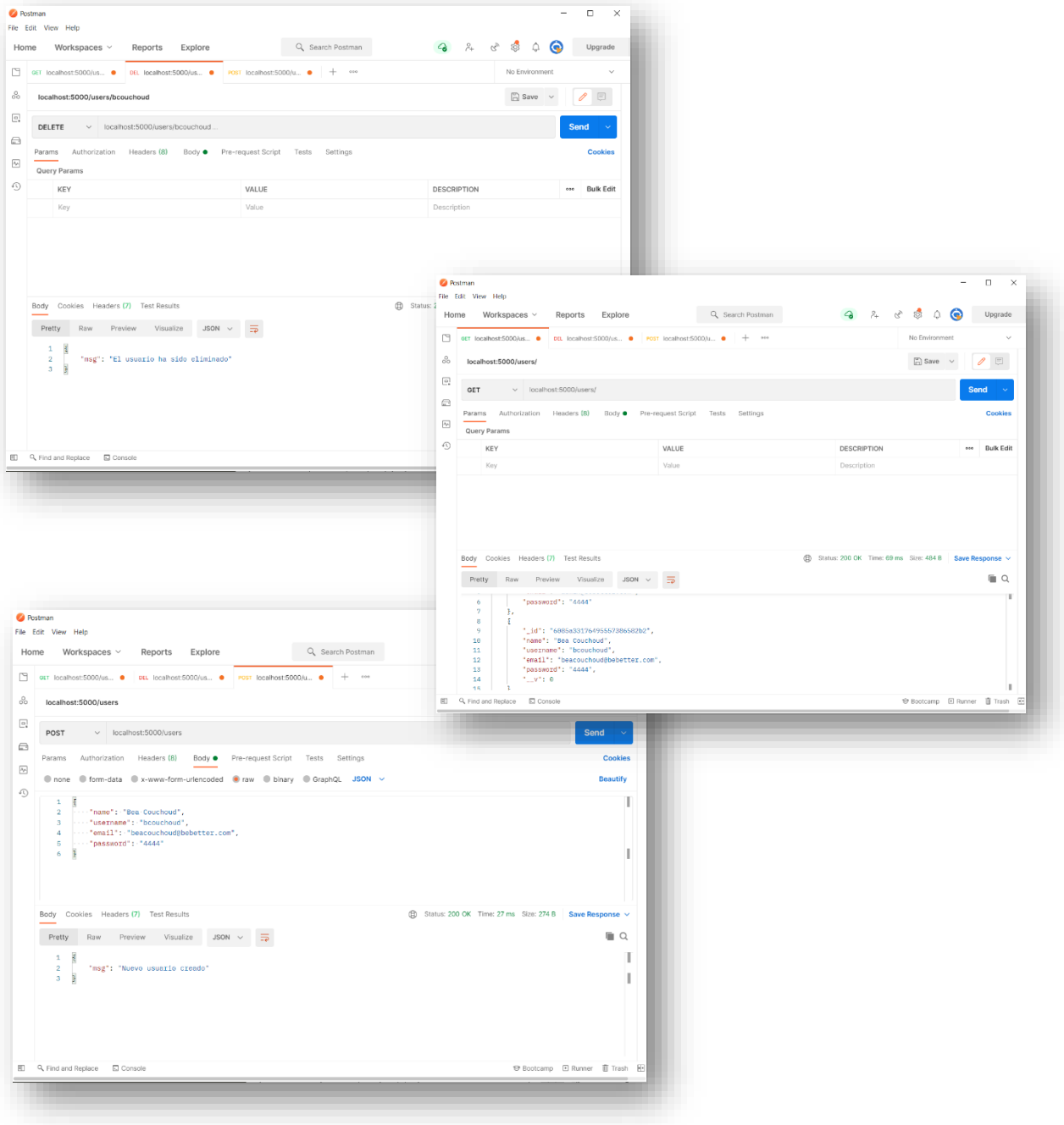
Express es una infraestructura web de direccionamiento y middleware que tiene una funcionalidad mínima propia: una aplicación Express es fundamentalmente una serie de llamadas a funciones de middleware.

Las funciones de *middleware* son funciones que tienen acceso al objeto de solicitud (req), al objeto de respuesta (res) y a la siguiente función de middleware en el ciclo de solicitud/respuestas de la aplicación. La siguiente función de middleware se denota normalmente con una variable denominada next. Pueden utilizarse a nivel de aplicación, para redireccionar, para el control y manejo de errores, etc. Para el funcionamiento de la app se han desarrollado dos middlewares uno para verificar que los tokens de las distintas peticiones eran válidos y otra para verificar que el usuario y el email no estaban en uso antes de insertar un nuevo usuario en la base de datos; por otro lado se han usado varios middlewares de terceros, ya mencionados previamente (mongo-sanitize, bcrypt, jsonwebtoken, bodyparser, mongoose).

3.1.1.2.3 Pruebas

Para elaborar test que permitan validar el comportamiento de la API se ha utilizado Postman. Postman es una herramienta que permite realizar peticiones a APIs y generar colecciones de peticiones que nos permitan probarlas de manera rápida y sencilla. Se puede definir en ella el tipo de petición (GET, POST…), tokens de autenticaciones, cabeceras y parámetros de la petición… Ha sido indispensable para el desarrollo de esta aplicación ya que, al desarrollar el back antes que el front era la forma más simple de probar que los métodos estuviesen bien desarrollados y el servidor bien configurado.

A continuación se incluyen algunos ejemplos de las pruebas realizadas:



3.1.1.3 Firebase

Firebase es una plataforma digital empleada para facilitar el desarrollo de aplicaciones web y aplicaciones móviles de una forma efectiva, rápida y sencilla, lanzada en 2011 y adquirida por Google en 2014.1

Es una plataforma ubicada en la nube, integrada con Google Cloud Platform, que usa un conjunto de herramientas para la creación y sincronización de proyectos que serán dotados de alta calidad, haciendo posible el crecimiento del número de usuarios y dando resultado también a la obtención de una mayor monetización. En este caso la aplicación se ha

decidido no desarrollar desde el principio la aplicación con Firebase, sin embargo, con la aplicación prácticamente terminada se ha decidido integrar sus servicios para el envío y recepción de notificaciones push, tan características de las aplicaciones móviles.

A continuación se incluye una breve explicación de como funciona este servicio y unas capturas.

*se incluirán más adelante

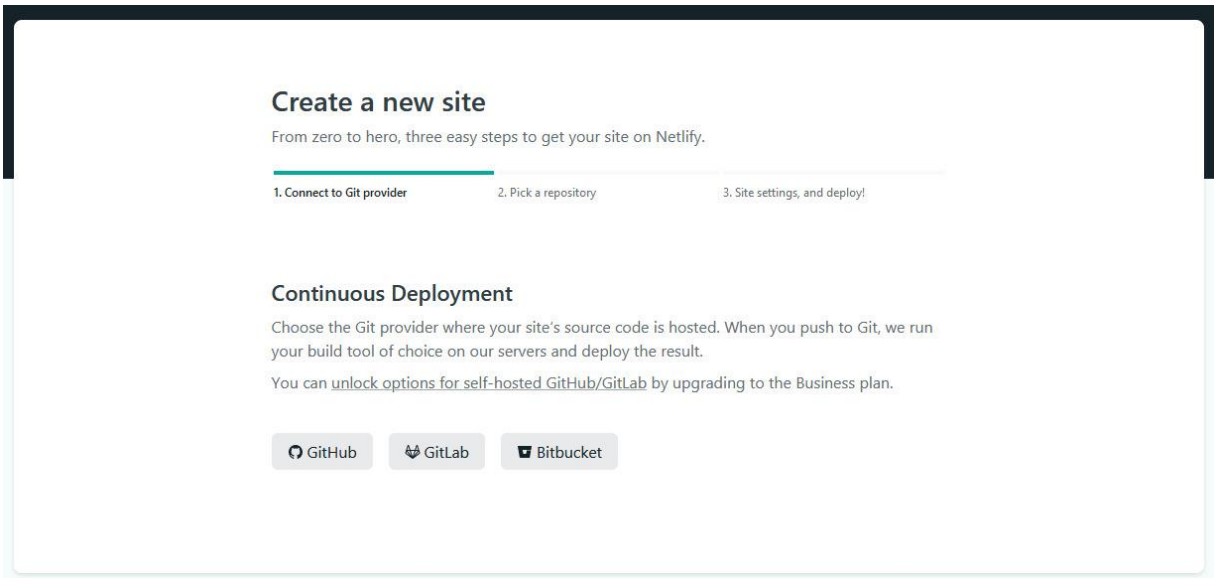
3.2 Implantación y configuración de la aplicación

Para poder levantar la API en local, el primer paso sería clonar el repositorio de GitHub. Una vez tenemos una copia del proyecto en local deberíamos instalar los servidores y levantarlo. Para ello se tiene que descargar Node.js desde su página oficial. Una vez descargado el archivo hay que seguir con la instalación con la instalación, que no es más que seguir el wizard y dar Next. Mencionar que conjuntamente con Node.js se va instalar el gestor de paquetes NPM. A continuación, hay que acceder a la carpeta correspondiente desde la consola y ejecutar **npm install**.

En lo referente al front, se necesita **instalar angular CLI**, para ello hay que ejecutar el comando **npm install -g @angular/cli**. vez a terminado la instalación se debe ir a la carpeta del proyecto (la que corresponde al front) y ejecutar **ng serve** para desplegar la aplicación.

Para la base de datos será necesario instalar un cliente de MongoDB e impotar los documentos .json que incluyen todos los datos.

Para subir la aplicación se ha escogido un hosting gratuito de Netlify.



Es bastante sencillo, simplemente se han ido siguiendo los pasos que se indicaban en el sitio web. Además se ha sincronizado con GitHub para poder subir facilmente os cambios durante el proceso de desarrollo. Aunque las capacidades de la versión gratuita son algo limitadas se ha considerado que es suficiente para realizar la demo de esta aplicación. Aclarar que unicamente se suben a producción la API y la BBDD, ya que laparte de cliente se instala en el movil mediante una apk que generamos gracias a Capacitor.

3.3 Control de versiones

Para el control de versiones y desarrollo de la aplicación se ha creado un repositorio privado en GitHub. Se han incluido, tal y como se ve en la imagen, cuatro ramas; master para los cambios definitivos que se subirán a producción, develop para desarrollo, test para realizar pruebas y funcionalidades que pueden modificarse o eliminarse mas adelante si el resultado no es el esperado.

Para manejar la información que se sube al repositorio se ha utilizado la funcionalidad integrada en Visual Studio Code por su rapidez y sencillez, además de la facilidad para ver y descartar los cambios que se realizan en cada archivo. Para operaciones más complejas como hacer rebase o volver a puntos anteriores de desarrollo se ha utilizado la consola de git.

*Incluir captura

4. Conclusiones

El desarrollo de esta aplicación ha dado lugar a muchos aprendizajes, nuevas formas de enfrentarse a problemas hasta ahora desconocidos, y a reaccionar y encontrar soluciones creativas a problemas que no se contemplaban originalmente. También a capacidades mas generales, como la gestión de tiempo, la paciencia y la organización de tareas. Algunas de las conclusiones obtenidas son las siguientes:

- Angular junto a Ionic+Capacitor permiten crear aplicaciones híbridas que pueden ser compiladas y desplegadas en múltiples plataformas.
- Los detalles en la UX son muy importantes, no todo es el rendimiento, el feedback al usuario es fundamental, los colores y las fuentes juegan un papel muy importante.
- El desarrollo orientado a dispositivos móviles es considerablemente más complejo que para la web. La interacción con pantallas táctiles o pequeñas a veces no es compatible con equipos de escritorio tradicionales y pantallas grandes.
- El desarrollo híbrido multiplataforma brinda grandes ventajas para equipos pequeños, o en este caso, de un solo desarrollador. Además, me ha permitido reforzar conocimientos adquiridos durante este curso y refrescar los adquiridos en otros cursos.
- Es tan importante el planteamiento de la aplicación propia como el tener en cuenta las librerías y software de terceros que se va a emplear, a veces ciertas versiones tienen bugs e incompatibilidades que conviene tener en cuenta para solventarlo antes de tener el desarrollo encaminado hacia una dirección clara.

4.1 Posibles mejoras

Por supuesto, esta aplicación podría considerarse el producto mínimo viable en vistas a hacer una aplicación más grande, una versión beta de lo que podría ser el resultado final, ya que 40 horas no dan para hacer mucho más. Las posibles mejoras y nuevas funcionalidades a añadir son múltiples, sin embargo, en caso de disponer de más tiempo las cosas que me gustaría mejorar en esta fase de desarrollo serian las siguientes:

- Mejorar el responsive. En el entorno móvil cada dispositivo es un mundo en cuanto a resoluciones, tamaños, opciones personalizadas (modo claro/oscuro, uso de botones o gestios, fuentes, etc) y es muy difícil garantizar que en todos ellos se vaya a ver la interfaz de forma adecuada. Además, implementaría el modo claro (la aplicación se ha desarrollado en modo oscuro).
- Subida de aplicación a tiendas online (Play Store, Microsoft Store).
- Funcionalidades que no ha dado tiempo a implementar o que podrían ser mejores. Por ejemplo, se ha descartado la inclusión de gráficas mediante alguna librería por problemas de incompatibilidad y se ha sustituido por una pequeña pantalla de estadísticas, sería interesante poder combinar ambas soluciones, compartir los resultados en redes sociales, etc.
- Traducir a más idiomas. Se ha decidido desarrollar la aplicación en inglés, tanto el código como la interfaz, sin embargo, estaría bien que el usuario pudiera escoger en que idioma ve la app.

4.2 Dificultades

- La principal dificultad ha sido la gestión de tiempo. Tal vez por la falta de experiencia la detección y resolución de problemas ha consumido mucho más tiempo del esperado.
- Mucha variedad de pantallas, ha resultado muy complicado que la interfaz sea lo mas parecida posible en distintos dispositivos.
- Graves problemas con la compatibilidad entre dependencias, cambios grandes entre distintas versiones, bugs y errores en librerías de terceros, etc.
- Uso de tecnologías no vistas a lo largo del curso. Si bien en internet hay mucha documentación y se ha tratado de no crear funcionalidades excesivamente complejas requiere más tiempo y dedicación aprender un lenguaje, framework o arquitectura al mismo tiempo que se está utilizando, ya que puede derivar en un mal uso de ciertos componentes, métodos… que retrasan el desarrollo.

4.3 Logros

- Se ha desarrollado una aplicación funcional y con un diseño muy parecido al mockup inicialmente planteado. En general se han seguido las pautas establecidas y considero que se ha logrado el objetivo, aunque queden cosas por mejorar o añadir.
- La aplicación se ha desarrollado íntegramente en inglés. Mi nivel de inglés no es muy alto y considero que me ha resultado bastante útil para mejorar, tanto el desarrollo de la aplicación como la lectura de documentación o visualización de videos explicativos en inglés.
- Puesta en práctica de conocimientos adquiridos a lo largo de todo el grado superior, tratando prácticamente todas las asignaturas tanto de primero como de segundo curso.
- Aprendizaje de nuevas tecnologías de forma autodidacta.

4.4 Resultados

- App hibrida, multiplataforma, funcional, segura, útil, portable, en inglés, diseño amigable, etc.

4.4.1 Muestras del resultado final

**Añadir imágenes

5. Referencias y Bibliografía