
Implementación de una infraestructura REST con privacidad para GTD basada en Express y React Native



TRABAJO FIN DE GRADO

Pablo Gamo González, Carlos Gómez López, Javier Gil Caballero,
Alejandro Del Río Caballero

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería del Software y Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid
Curso 2023-2024

Implementación de una infraestructura REST con privacidad para GTD basada en Express y React Native

Memoria de Trabajo Fin de Grado

Pablo Gamo González, Carlos Gómez López, Javier Gil Caballero,
Alejandro Del Río Caballero

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería del Software y Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid
Curso 2023-2024

Resumen

La metodología de productividad personal conocida como “*Getting Things Done*” (*GTD*), creada por David Allen, es uno de los métodos más efectivos para la organización de tareas en la actualidad. Su objetivo es maximizar la productividad a través de la consolidación de todas las tareas, proyectos y actividades en un solo lugar. Aunque existen muchas aplicaciones disponibles para ayudar a poner en práctica la filosofía *GTD*, la mayoría son propiedad de empresas que pueden tener acceso a la información personal de los usuarios, lo que puede violar su privacidad.

Para solventar este problema en este proyecto se ha desarrollado una infraestructura formada por una base de datos, un servicio *REST*, y un conjunto de clientes para entornos de escritorio y móvil. En el proyecto se ha empleado la tecnología *React Native* para el desarrollo de clientes multiplataforma. Algunos aspectos clave de la infraestructura desarrollada son su *modo offline*, para permitir el funcionamiento de los clientes incluso sin conexión a *Internet*, y el empleo de estándares de autorización como *OAuth* para dotar de seguridad a la *API REST*. Para la implementación de esta *API* se ha hecho uso del framework *Express.js* que ha simplificado sustancialmente el desarrollo. Asimismo, también se han explorado métodos para interactuar con el servicio *REST GTD* desde asistentes conversacionales.

palabras clave: GTD, React Native, Express.js, REST API, OAuth 2.0, Multiplataforma

Abstract

Blah Blah ...

Autorización de difusión y utilización

Los abajo firmantes, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado “Implementación de una infraestructura REST con privacidad para GTD basada en Express y React Native”, realizado durante el curso académico 2023-2024 bajo la dirección de Juan Carlos Sáez Alcaide en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Pablo Gamo González, Carlos Gómez López, Javier Gil Caballero, Alejandro Del Río Caballero

Juan Carlos Sáez Alcaide

Índice general

Resumen	iii
Abstract	v
Autorización de difusión y utilización	i
1 Introducción	1
1.1 Motivación	1
1.2 Objetivos	2
1.2.1 Arquitectura del sistema	5
1.3 Plan de Trabajo	5
1.3.1 Tareas	5
1.3.2 Planificación temporal	6
1.3.3 Estructura de la memoria	7
2 Metodología GTD y aplicaciones	9
2.1 ¿Qué es GTD?	9
2.2 Método y técnica	10
Collect	12
Process	12
Organize	13
Review	13
Do	14
2.3 Aplicaciones	14
2.4 Desafíos que encontramos	15
2.5 Conclusiones	17
3 Planificación del proyecto	19
3.1 Gestión de configuración del Software	20
3.1.1 Sistema de control de versiones	20
3.1.2 Entornos de desarrollo e Integración	21
4 Modelo de datos e implementación de la Base de datos	25

4.1	Descripción de entidades	25
4.2	Relaciones entre entidades	27
4.3	Modelo físico de la BD	28
4.4	Rendimiento y Optimización de la Base de Datos	32
4.5	Seguridad de la Base de Datos	32
5	Diseño e implementación del Backend	35
5.1	Uso de REST	35
5.2	Descripción tecnologías: Node JS + Express	36
5.3	Express.js	36
5.4	Diseño de la API	37
5.5	Aspectos de seguridad de la API	37
5.5.1	Implementación de OAuth 2.0	38
	Autorización, Autenticación y posibilidades de <i>OAuth</i>	38
	Roles y Flujo del proceso	39
	Aplicación del estándar <i>OAuth2.0</i>	39
	Implementación del estándar <i>OAuth2.0</i> en <i>Node.js</i>	43
	Configuración de clientes con OAuth	45
6	Diseño e Implementación del Frontend	47
6.1	Percepción del funcionamiento del sistema	47
6.2	Metáforas, Expresiones y Conceptos de diseño	48
6.2.1	Proximidad y Consistencia	49
6.2.2	Visibilidad	50
6.3	Prototipos e Interfaces	50
6.4	Implementación: ¿Qué es React Native?	61
7	Integración con agentes conversacionales	63
7.1	Configuración de la skill de Alexa	63
7.2	Vinculación de la cuenta del usuario	63
7.3	Implementación del flujo de autorización Oauth	64
7.4	Guía para vincular Alexa con SwiftDo	65
8	Manual de usuario	69
8.1	Configuración del servidor	69
8.2	Inicio de sesión y registro	69
8.3	Inbox	69
8.4	Filtros	70
8.5	Sidebar	70
8.6	Ajustes	73
8.7	Crear tarea y crear proyecto	73
8.8	Hoy	76
8.9	Cuanto antes	76

8.10 Programadas	76
8.11 Archivadas	76
8.12 Proyectos	81
8.13 Tareas	81
9 Conclusiones y Trabajo futuro	83
9.1 Conclusiones	83
9.2 Trabajo futuro	84
Guía de despliegue	87
9.3 Estructura de carpetas del proyecto	87
9.4 Despliegue del <i>backend</i>	88
9.5 Compilación de la aplicación web	89
9.6 Compilación de la aplicación de escritorio	90
Endpoints	91
9.6.1 Tarea	91
9.6.2 Usuario	94
9.6.3 Auth	95
9.6.4 Proyecto	96
9.6.5 Contexto	98
9.6.6 Etiqueta	100
Bibliografía	101

Índice de cuadros

Índice de figuras

1.1	Diagrama de bloques del sistema	4
1.2	Diagrama de Gantt backend	6
1.3	Diagrama de Gantt frontend	7
2.1	Workflow GTD - Getting Things Done de David Allen. Fuente: [2]	11
3.1	Tablero de las tareas en Notion	20
3.2	Diagrama del sistema de despliegue	22
4.1	Diagrama Entidad-Relación de la aplicación	26
4.2	Diagrama Entidad-Relación en la base de datos	29
5.1	Proceso de autorización OAuth	40
5.2	Flujo con código de autorización	42
5.3	Flujo de refresco del token de acceso	42
5.4	Flujo de autorización implementado	44
6.1	Menu lateral	48
6.2	barra lateral	49
6.3	Botón para añadir tarea/proyecto	49
6.4	Prototipo de Bandeja de entrada, Cuanto antes, Programadas y Archivadas	51
6.5	Nombre de usuario y fecha	52
6.6	Sección para ver y editar aspectos más detallados de la tarea (incluso observaciones en Markdown)	53
6.7	Inicio de sesión y Registro	54
6.8	Escritorio	54
6.9	Pantalla de sección “hoy”	55
6.10	Ajustes - Móvil	56
6.11	Ajustes - Escritorio	56
6.12	Opciones de ajustes	57
6.13	Editor de tareas	57
6.14	Añadir tareas o proyectos	58
6.15	Menu para añadir tareas	59
6.16	Visualización de tarea	60

7.1	Diagrama de secuencia del proceso de vinculación de la cuenta. Fuente: [19]	64
7.2	Página principal aplicacion “Amazon Alexa”	65
7.3	Página de “Skills y juegos” de la aplicación	66
7.4	Página de la Skill SwiftDo ya habilitada	66
7.5	Página redirigida para vinculación de Alexa con SwiftDo	67
8.1	Configuración del servidor	70
8.2	Inicio de sesión y registro	71
8.3	Inbox	72
8.4	Filtros	73
8.5	Sidebar	74
8.6	Ajustes	75
8.7	Crear tarea y crear proyecto	76
8.8	Hoy	77
8.9	Cuanto antes	78
8.10	Programadas	79
8.11	Archivadas	80
8.12	Proyectos	81
8.13	Funcionalidad tareas	82

Capítulo 1

Introducción

1.1 Motivación

Inmersos en plena era digital, caracterizada por la cultura de la inmediatez, no resulta tarea fácil mantener el enfoque y evitar distracciones en medio de una vorágine de información y estímulos. La metodología *GTD* (*Getting Things Done*) tiene como objetivo ayudar a las personas a realizar sus tareas del día a día de manera que no dependan de su memoria y se centren en el ahora, sin estar pendiente de futuras tareas. Este método fue creado por David Allen, quien lo recogió en su libro titulado *Getting Things Done* [1] y fue traducido al español como “Organízate con eficacia”.

Asimismo, el sistema *GTD*, es conocido por su eficacia tanto a nivel personal como profesional en la planificación, organización y administración de tareas. Sin embargo, algunas de las aplicaciones que lo implementan tienen varios inconvenientes. En primer lugar, muchas de estas aplicaciones no informan claramente del proceso de tratamiento de los datos de sus usuarios. (p.ej.: los datos son alojados en servidores de terceros y se desconocen los protocolos de seguridad que tienen estos).

Por otro lado, muchas de estas aplicaciones tienen una alta curva de aprendizaje o son compatibles con un número reducido de sistemas operativos. Un claro ejemplo de esto es *Things*¹ que a pesar de ser una muy buena aplicación, solo está disponible para dispositivos Apple. Esto condiciona el acceso de esta metodología a un público más general.

Por ello, motivados por poner solución a esta problemática, buscamos desarrollar una aplicación multiplataforma que además de implementar la metodología *GTD*, destaque por su interfaz intuitiva y amigable, guiada por una arquitectura *REST*, con el objetivo de permitir al usuario tener el control de sus datos, garantizando la privacidad y la transparencia de los mismos. Esta aplicación, llamada *SwiftDo*, pretende ofrecer una

¹Things

alternativa accesible y versátil para la gestión eficiente de tareas en el día a día.

1.2 Objetivos

El objetivo central de este proyecto es desarrollar una aplicación multiplataforma *GTD*. Para ello hemos utilizado el framework *React Native* que permite crear una interfaz de usuario coherente que funcione en todos los sistemas operativos principales, como *Android*, *iOS*, *MacOS*, *GNU/Linux* y *Windows*, y garantiza que los usuarios puedan gestionar sus tareas y proyectos de manera eficiente desde cualquier dispositivo, sin importar la plataforma que utilicen.

La utilización de *React Native* simplifica el desarrollo, mantenimiento y escalabilidad del proyecto, lo que resulta en una aplicación ágil y adaptable a futuras actualizaciones y cambios en las plataformas de destino.

Nuestra aplicación *SwiftDo* destaca por su modo offline, que permite a los usuarios gestionar tareas incluso sin conexión a Internet (como puede darse el caso en dispositivos móviles). A diferencia de otras alternativas, nuestra *app* garantiza una experiencia ininterrumpida al almacenar datos localmente y sincronizar automáticamente con el servidor cuando se recupera la conexión, asegurando la disponibilidad constante de la información en todos los dispositivos del usuario.

En el desarrollo de nuestra aplicación, la *API REST* desempeña un papel crucial al proporcionar *endpoints* para la comunicación cliente-servidor, permitiendo operaciones *CRUD* en los datos, como por ejemplo, tener un *endpoint* '/tareas' para la creación y lectura de tareas. De esta manera, la *API REST* proporciona una interfaz estandarizada y eficiente para la manipulación de datos en nuestra plataforma *GTD*. Además, la implementación de sólidas prácticas de seguridad, como autenticación y cifrado de datos, asegura la integridad y confidencialidad de la información, garantizando una experiencia segura para nuestros usuarios.

Al desarrollar nuestra aplicación perseguimos una serie de objetivos específicos que quedaron representados mediante los siguientes requisitos de alto nivel:

- **Gestión de tareas centralizada:**

La aplicación debe permitir a los usuarios crear nuevas tareas, las cuales por defecto se agregarán al *Inbox* (almacén de tareas sin organizar dentro de la metodología *GTD*). Desde allí, los usuarios podrán asignarles etiquetas, vincularlas a proyectos o áreas específicas, y moverlas entre diferentes secciones.

- **Flexibilidad en la Organización:**

Los usuarios deben tener la capacidad de organizar sus tareas de manera flexible, añadiéndolas a proyectos o áreas relevantes, y asignándoles etiquetas para una

clasificación más detallada. Además, la aplicación debe permitir la edición y eliminación de tareas según sea necesario

- **Seguimiento y Priorización:**

La aplicación debe proporcionar una serie de funcionalidades para el seguimiento y la priorización de las tareas de manera eficiente y efectiva. Los usuarios deben tener la capacidad de completar tareas y asignarles una importancia relativa. Además, la aplicación debe facilitar el acceso a información detallada sobre cada tarea, incluyendo su estado actual, fecha de vencimiento y cualquier nota asociada. Es crucial que las tareas se prioricen automáticamente según su relevancia para el usuario.

- **Seguridad:**

Nuestra aplicación *SwiftDo* debe implementar un robusto sistema de autenticación y autorización para garantizar que solo usuarios autorizados puedan acceder y manipular los datos a través de la *API REST*. Además, se requiere cifrado de extremo a extremo para proteger la confidencialidad de la información durante su transmisión y almacenamiento, asegurando así una experiencia segura para todos los usuarios.

- **Funcionalidades de Búsqueda y Filtrado:**

Los usuarios deben poder buscar y filtrar sus tareas según etiquetas, áreas, proyectos u otros criterios relevantes, lo que facilita la gestión y visualización de las tareas.

- **Usabilidad:**

La aplicación proporciona un modo *offline* que permite gestionar tareas sin conexión almacenando los datos localmente en el dispositivo del usuario y asegurando la continuidad del trabajo sin importar la disponibilidad de conexión.

- **Integración con agentes conversacionales:**

La aplicación se integra con agentes conversacionales como *Alexa* ofreciendo una experiencia más versátil, permitiendo así a los usuarios interactuar con la aplicación mediante comandos de voz, simplificando aún más la entrada y gestión de tareas de forma intuitiva y sin esfuerzo.

Estos requisitos proporcionan una base sólida para el desarrollo de la aplicación *SwiftDo*, garantizando una experiencia integral que cumpla con los principios fundamentales de la metodología *GTD* y satisfaga las necesidades de los usuarios en la gestión efectiva de sus tareas y proyectos.

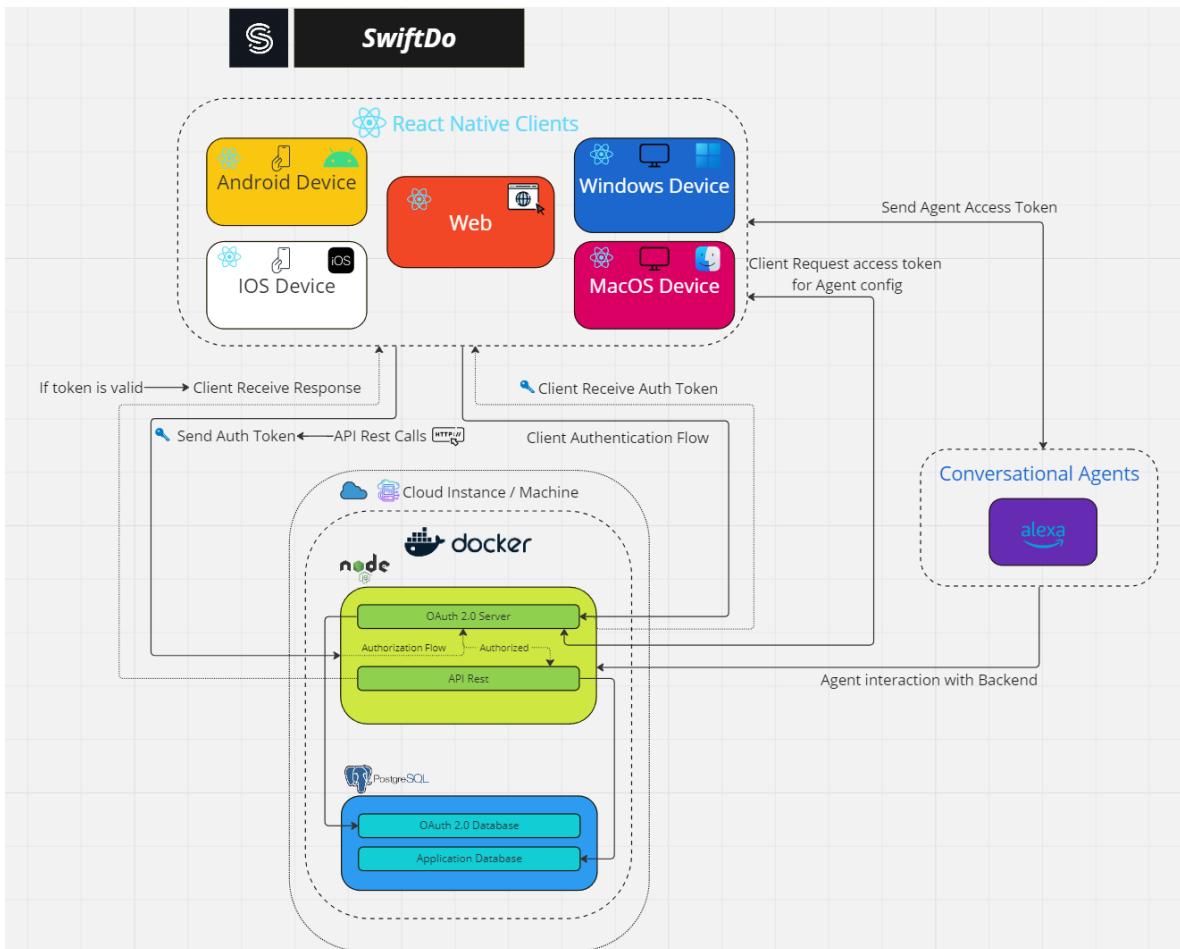


Figura 1.1: Diagrama de bloques del sistema

1.2.1 Arquitectura del sistema

La figura 1.1 muestra el diagrama de bloques del sistema, que está compuesto por 2 componentes principales: una aplicación cliente, formado esencialmente para diversos dispositivos y un *backend* el cual cuenta con una *API REST* con un sistema de autorización seguro.

La aplicación cliente está implementada con el framework multiplataforma *React Native*, pudiendo ser ejecutada en diversos dispositivos. Los clientes interactúan mediante *REST* sobre *HTTPS* con el *backend*.

El *backend* implementa 2 componentes principales e incluye una base de datos. Por una parte está el módulo *OAuth 2.0* el cual se encarga de gestionar el flujo de autenticación y autorización, es decir gestiona el acceso de los usuarios de las aplicaciones cliente a la información y a los servicios que proporciona el *backend*. Por otra parte, el *backend* también está compuesto por una *API* que sigue la arquitectura *REST* e implementa y expone mediante diversos *endpoints* los diferentes servicios de la aplicación. Por último el *backend* contiene también la base de datos de la aplicación, la cual contiene tanto las tablas que utiliza el módulo *API Rest* como el módulo *OAuth 2.0*. Todos los servicios que contiene el *backend* están gestionados mediante contenedores *Docker*, de esta manera es posible arrancar, conectar y configurar los diversos módulos de manera sencilla y ágil, con el fin de poder desplegar dichos servicios en cualquier máquina sin necesidad de más configuración particular a cada entorno.

Por último, el sistema contiene un tercer componente el cual permite conectar algunos agentes conversacionales con la aplicación. Desde los clientes es posible realizar dicha configuración de agentes mediante la generación de una clave especial para estos, de manera que los agentes una vez configurados puedan acceder a los servicios del *backend* mediante comandos de voz.

1.3 Plan de Trabajo

1.3.1 Tareas

Para llevar a cabo el desarrollo del proyecto hemos dividido las tareas a realizar en varias fases que se comentan a continuación.

En primer lugar y de manera individual, realizamos una labor de búsqueda comparando nuestro modelo de proyecto con otras aplicaciones existentes en el mercado para posteriormente poner en común las distintas ideas. Cada miembro del equipo instaló y analizó una de estas aplicaciones y fue apuntando las posibles mejoras que podríamos implementar para aportar más valor a nuestro producto. Gracias a este ejercicio, llegamos a muchas de las conclusiones explicadas en la [sección 1.1](#).

Posteriormente, hubo una fase de diseño en la cual realizamos un *mockup* de cómo nos

gustaría que fuese nuestra aplicación.

A continuación, tuvimos una fase de aprendizaje en la cual buscamos información y experimentamos con las distintas herramientas y componentes necesarios para montar nuestra aplicación. Entre estos componentes buscamos familiarizarnos con *Express*, *Docker*, *AWS*, *React Native*, *firebase* entre otros para realizar una comparativa y escoger las tecnologías que mejor se adaptaban a nuestras necesidades para el futuro desarrollo de nuestra aplicación.

Una vez terminadas las distintas tareas previas al desarrollo comentadas anteriormente, comenzamos con la programación del *backend* que nos llevó aproximadamente cuatro meses. Durante este tiempo estuvimos implementando los servicios principales de la aplicación además del sistema de autenticación y autorización. Una vez terminada la parte esencial del *backend*, comenzamos con el desarrollo del *Frontend*, que ha sido la fase más costosa en tiempo y esfuerzo, ya que hemos dedicado un gran trabajo a crear una interfaz amigable y usable además de funcionalidad extensa (filtrado, búsqueda, edición...). Es por ello que hemos ido realizando actualizaciones necesarias en *backend* para el correcto funcionamiento de la aplicación.

1.3.2 Planificación temporal

Las figuras 1.2 y 1.3 muestran las tareas asociadas al desarrollo del *backend* y del *Frontend* respectivamente, así como la fecha de inicio, la fecha de fin y la duración de cada tarea:

Backend

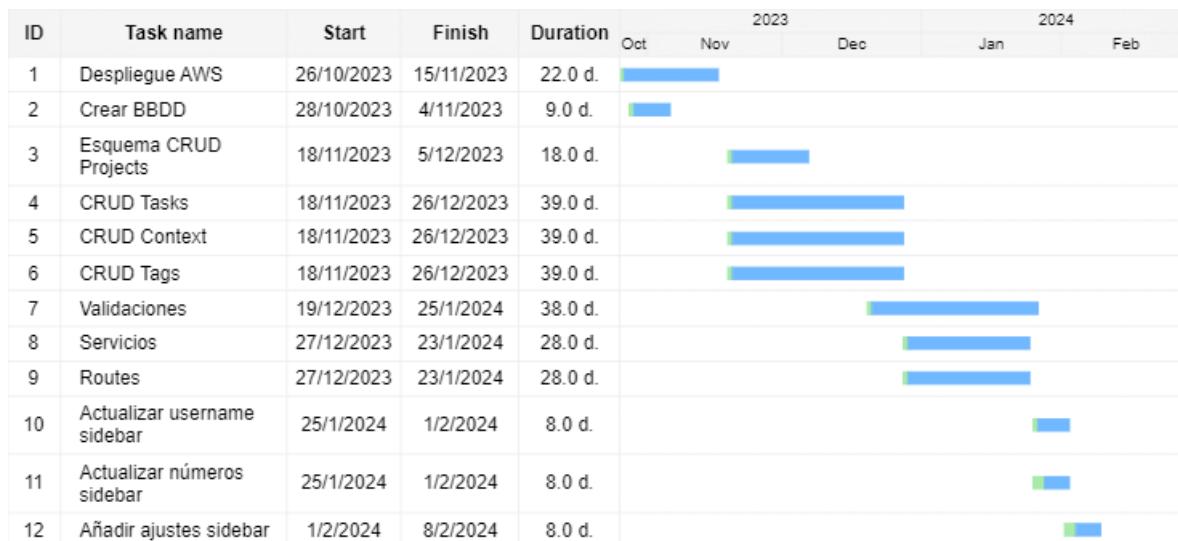


Figura 1.2: Diagrama de Gantt backend

Frontend

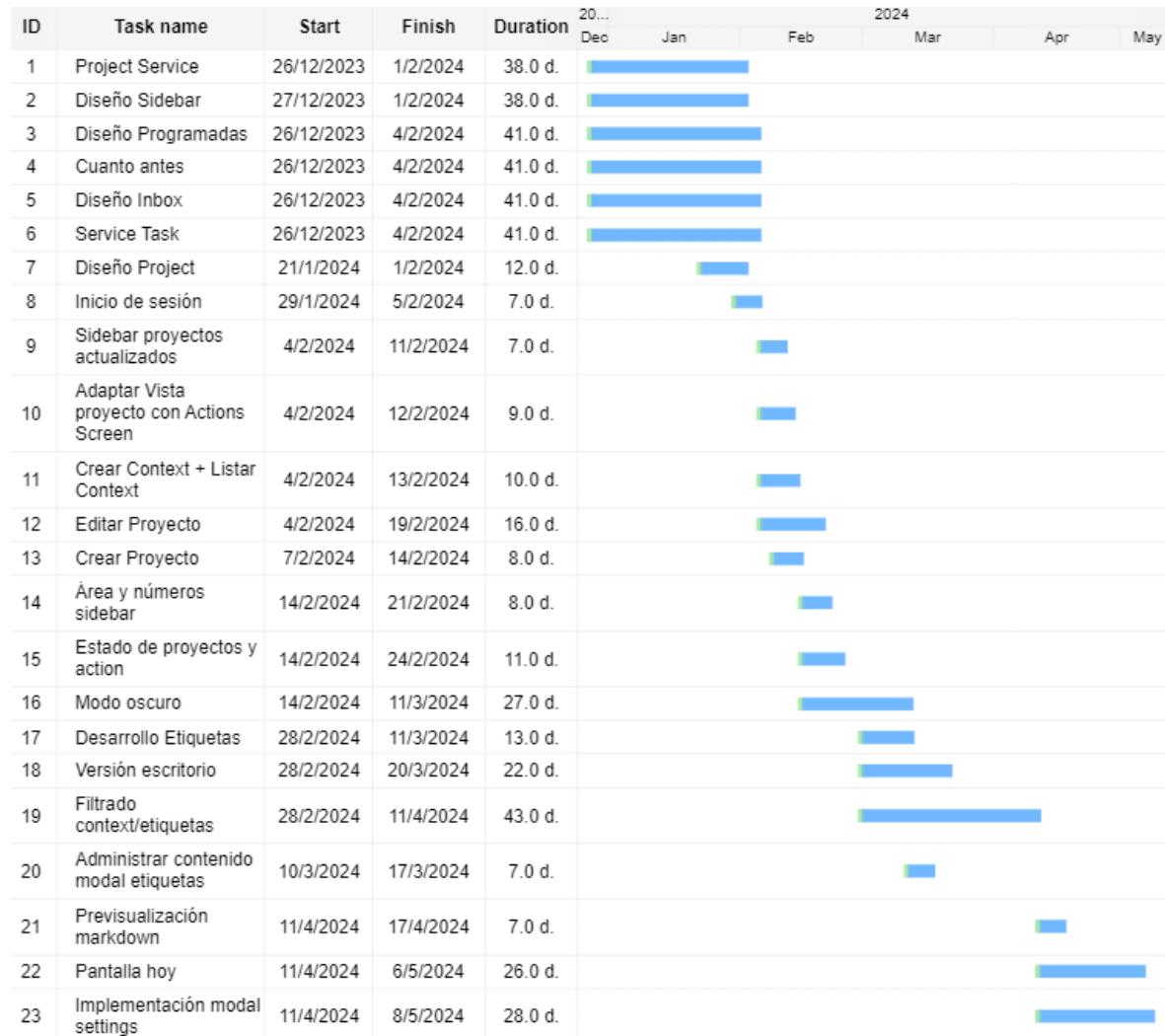


Figura 1.3: Diagrama de Gantt frontend

1.3.3 Estructura de la memoria

El resto de memoria se organiza como sigue:

- En el **capítulo 2** se realiza un análisis sobre qué es *GTD*, cuál es su método y técnica. También se describen las distintas aplicaciones en el mercado que implementan *GTD*.
- En el **capítulo 3** se discute la planificación del proyecto, tratando puntos como el sistema de control de versiones, los entornos de desarrollo e integración utilizados y el sistema de despliegue.
- En el **capítulo 4** presentamos el modelo de datos y la implementación de la base

de datos. Aquí se describen las entidades, el modelo físico de la base de datos, su rendimiento y escalabilidad y la seguridad de la misma.

- En el **capítulo 5** se introduce el diseño e implementación del *backend*, explicando el uso que hacemos de *REST*, el diseño de la *API*, los *endpoints* de la aplicación, los aspectos de seguridad de la *API* y la implementación de *OAuth 2.0*.
- En el **capítulo 6** se detallan los principios de diseño que han guiado la creación de la *app*, junto a las distintas tecnologías empleadas en el desarrollo de la misma, desde la etapa inicial del prototipado hasta su implementación, explorando cómo los fundamentos del método GTD se han ido plasmando en *SwiftDo*.
- En el **capítulo 7** se describe la integración de nuestra aplicación GTD con agentes conversacionales, en nuestro caso con *Alexa*. Explicaremos cómo hemos configurado la *skill* de *Alexa*, cómo hemos vinculado nuestra cuenta de usuario, la implementación del flujo de autorización *OAuth 2.0* y los problemas que nos han surgido en este proceso.
- En el **capítulo 8** se encuentra el manual de usuario, mostrando la funcionalidad al completo de nuestra aplicación de manera fácil e intuitiva.
- En el **capítulo 9** se recogen las principales conclusiones a las que hemos llegado y se discuten líneas de trabajo futuras.
- Esta memoria consta también de cuatro apéndices. Los dos primeros (apéndices A y B) corresponden a la traducción al inglés de la introducción y las conclusiones, el tercero (apéndice C) enumera las contribuciones que ha hecho cada miembro del equipo al proyecto, y el último, (apéndice D) proporciona una guía de despliegue para que cualquiera que quiera utilizar nuestra aplicación pueda hacerlo.

[]:

Capítulo 2

Metodología GTD y aplicaciones

En este capítulo nos preocuparemos en saber como funciona detalladamente el método *GTD* y cuales son las fases que debemos seguir para poder implementarlo cotidianamente. Por otro lado, analizaremos la competencia a la que se enfrenta *SwiftDo*, destacando los puntos positivos y negativos de las aplicaciones populares que implementan o permiten implementar *Getting Things Done*

2.1 ¿Qué es GTD?

El desarrollo tecnológico, sumado a la creciente complejidad del mundo moderno, ha desembocado en un notable incremento de responsabilidades tanto personales como profesionales. Es por ello que mantener la atención al detalle y la gestión eficaz del tiempo se ha vuelto un gran desafío para todo tipo de personas.

En la actualidad, existen varios métodos que afirman ser de gran utilidad para aumentar la productividad diaria y el desarrollo personal, tales como la técnica pomodoro,¹ Kanban² o la matriz de Eisenhower.³ Sin embargo, a pesar de que algunos de estos métodos mejoren la visualización de prioridades o gestionen correctamente el flujo de trabajo, carecen de sentido para varios aspectos con los que nos topamos a diario. O bien son muy específicos, y carecen de una estructura bien definida, necesitando implementarlos en áreas muy concretas y tareas a corto plazo, o bien, requieren de una alta curva de aprendizaje y dedicación del usuario para mantener dicha productividad que prometen.

En este contexto, David Allen, en la década de los ochenta, tras años investigando para dar solución a la problemática anteriormente mencionada, creó un sistema que denominó como el método *Getting Things Done* (al que nos referiremos más adelante como *GTD*),

¹Técnica pomodoro

²Tablero Kanban

³Matriz de Eisenhower

título que le daría a su libro publicado en 2001, [1]. En su obra, detalla un conjunto de técnicas que ayudarían a mejorar la productividad del individuo implementándolas a su rutina diaria.

A pesar del desarrollo tecnológico y el paso de los años, *GTD* sigue siendo uno de los métodos de referencia para las personas que buscan alejarse “del ruido” del día a día, permitiéndoles enfocarse en lo que realmente importa; tanto en los aspectos personales como profesionales. De esta forma, permite al individuo gestionar de forma óptima el tiempo del que dispone a diario. Y para ello, hace uso de un lenguaje sencillo, basándose en dos objetivos principales:

- Recopilar en un “almacén” todas las actividades que necesitan estar hechas, sin importar el momento de finalización o la importancia de las mismas.
- Obtener una disciplina que permita controlar y organizar lo anteriormente reunido en diferentes acciones.

El fijar objetivos tan abstractos y concisos, hace que *GTD* obtenga un carácter de mayor relevancia para tareas de ámbito general, sin encasillarlo en áreas muy específicas de la vida, como puede ser el trabajo, un proyecto o simples labores domésticas. Esto posibilita la integración de todo tipo de tareas, reduciendo la carga de trabajo mental que supone acordarse de todo constantemente, evitando posibles distracciones mientras nos mantenemos enfocados en lo que de verdad importa.

Por último, para enfocarnos en un estado de máxima concentración, debemos seguir una serie de principios fundamentales que resumiremos en las siguientes secciones.

2.2 Método y técnica

Como comentamos previamente, para seguir los principios en los que se basa, *GTD* se apoya en cinco fases “collect”, “process”, “organize”, “review” y “do” que detallaremos en este apartado. El buen uso de estas, facilitará las gestiones cotidianas, a la par que nos libera del estrés puntual que supone la multitarea a la que el mundo moderno nos tiene acostumbrados. Asimismo, para implementar correctamente la metodología *Getting Things Done*, representada en la figura 2.1, es imprescindible seguir todos estos pasos de forma secuencial sin evitar ninguna fase, ya que de lo contrario podríamos vernos envueltos en un bucle de descontrol, desviándonos así del objetivo de la metodología explicada en el libro [1].

En su libro, Allen nos relata este apartado teniendo por analogía la *RAM* de un ordenador. Al igual que este, somos seres secuenciales, sin embargo, podemos realizar cierta multitarea y trabajar en muchos aspectos en paralelo. No obstante, para poder lograrlo, no sólo debemos tener bastante capacidad de almacenaje, sino también saber cómo administrarla de forma eficiente. Para ello, se deben seguir las cinco fases del método *GTD*:

Getting Things Done

Quick Reference Card

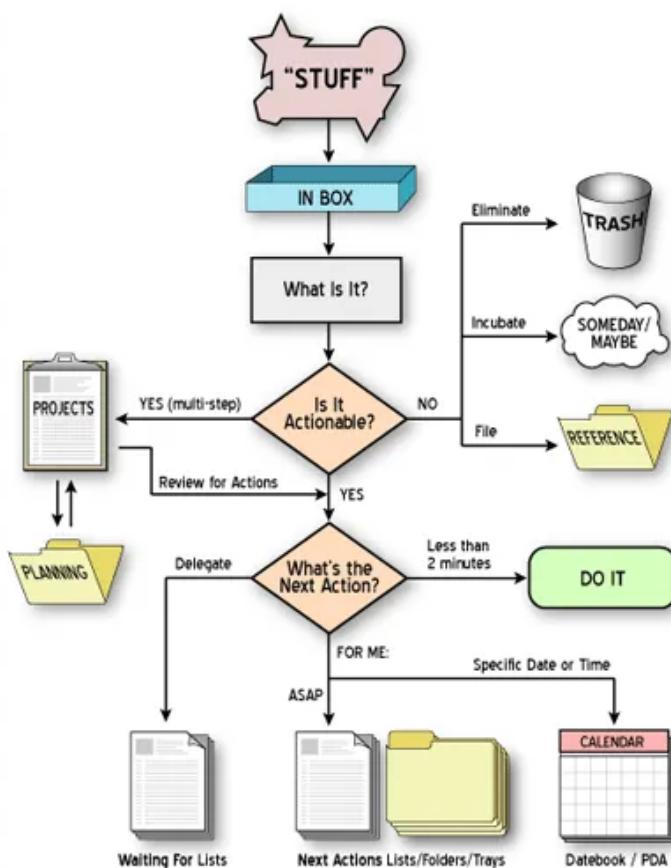


Figura 2.1: Workflow GTD - Getting Things Done de David Allen. Fuente: [2]

Collect

Se trata de una de las fases fundamentales en las que se basa la filosofía *GTD*, en ella debemos recopilar todas las tareas que durante el día nos han ido surgiendo a la mente; independientemente de la importancia de las mismas, pudiendo ir desde tareas tan triviales como “limpiar los platos” a “entregar un proyecto antes de cierto plazo”.

La idea es que, como indica Allen en su libro, “*saquemos todo lo que nos preocupa de la cabeza*” de forma que cada idea, tarea o asunto que nos vaya surgiendo a lo largo del día, la podamos almacenar para una organización posterior.

Sin embargo, de nada sirve que cumplamos lo anterior si periódicamente no hacemos una limpieza a nuestra *RAM*. De lo contrario, tendríamos un cajón de sastre abocado a la procrastinación, motivo que queremos evitar con *GTD*.

Process

Una vez recopiladas, debemos encontrar un hueco para poder catalogarlas, es decir, clasificarlas y procesarlas. Para ello, llega el momento donde nos haremos ciertas preguntas como ¿qué es esto que he apuntado?, ¿requiere más de una acción? ¿necesito clarificarla?

1. “¿Qué es esto que he apuntado?”

Al haber reunido todo lo que se nos ha venido a la cabeza, es muy común que tengamos ciertos asuntos que han perdido importancia o simplemente han perdido el sentido de ser que tenían en su momento. Además, debemos esclarecer si las tareas que tenemos por procesar requieren más de una acción o no. Si la respuesta es negativa, debemos clasificarlas en tres categorías, en caso contrario deberíamos ir directamente al *punto 2*:

- **Eliminarla:** Si la tarea ha dejado de tener importancia, la eliminamos o la marcamos como completada.
- **Incubarla:** Se necesita de la finalización de una tarea previa para poder empezar a trabajar con la otra.
- **Referenciarla:** A diferencia de la incubación, en esta categoría, se almacenan las tareas que no requieren de una acción previa para poder ser finalizadas. Es decir, se almacenan tareas que contienen información relevante a la que queremos acceder en algún momento futuro.

2. “¿Necesito clarificarla?”

Si la respuesta a esta pregunta es afirmativa, volvemos a tener 3 posibles acciones:

- **Hacerla:** Siempre y cuando completarla nos lleve menos de 2 minutos.
- **Aplazarla:** Si la tarea nos ocupa más de 2 minutos en completarla.

-
- **Diferir su naturaleza:** Se tarda más de 2 minutos en completarla y necesitamos que se finalice una acción previa para clasificarla en los puntos 1 o 2.

Organize

En esta fase tratamos con mayor énfasis las tareas anteriormente procesadas. De esta forma, facilitamos su seguimiento y ejecución, listándolas en categorías más concretas como por ejemplo “Siguientes” o “Calendario” para tareas que requieren menos de 2 minutos y que debíamos “Diferir su naturaleza” y no tienen múltiples pasos o por el contrario, proyectos. Por otro lado, tenemos la posibilidad de listarlas en “Esperando”. Esta en sí tiene una pequeña peculiaridad ya que se trata de tareas que necesitamos *aplazarlas*, porque estamos pendientes de que finalice un evento anterior que nos permita avanzar. Sin embargo, estas pueden estar en “Siguientes” teniendo en cuenta lo mencionado anteriormente.

Por último Allen, deja claro en su libro qué tipo de material puede ir o no en cada listado:

- **Proyectos:** tareas que requieren una serie de pasos para ser completadas. Los proyectos a diferencia de las tareas, solo se usan en forma de índice y los artefactos generados se deben ir listando en otras categorías.

Por ejemplo, si necesitamos grabar un vídeo para YouTube, en el proyecto iremos listando un índice de todo lo que debemos hacer para dar esta tarea por finalizada. Si uno de los ítems era “grabar en x localización” podremos tener otra tarea o proyecto relacionado con las labores que tengamos que hacer en dicha localización.

- **Calendario:** Tareas con acciones de tiempo específicas e información relacionada con horarios, día o semana concreta. Sin embargo, debemos tener en cuenta que no tenemos que caer en el error de utilizarlo como un listado para tareas diarias.
- **Siguientes:** Categoría encargada de listar las tareas que requieren hacerse “cuanto antes”. Es decir, tareas que sin tener fecha específica de finalización, conviene acabarlas en el menor tiempo posible.

Review

Llegados a este momento, debemos realizar un seguimiento, aproximadamente semanal, en el que volvemos a ejecutar las fases anteriores observando si ha cambiado la naturaleza de alguna tarea o por el contrario existan algunas “rezagadas”, evitando que estas queden en el olvido.

En otras palabras, si queremos que un ordenador funcione como el primer día, debemos ejecutar periódicamente un mantenimiento, una *fase de review*, eliminando todos los archivos que no se estén utilizando, depurando el sistema y aligerando la carga de la memoria *RAM*.

Do

Por último, tenemos la fase *Do*. En esta fase ya hemos terminado de *capturar*, *procesar*, *organizar* y *revisar* por tanto deberíamos tener un sistema coherente y preciso. No obstante, Allen en su libro nos explica que aún podemos mejorar el sistema, ya que si sobrecargamos la categoría “Siguientes”, podríamos llegar a cuestionarnos, ¿Por qué tarea deberíamos empezar? La respuesta corta es seguir nuestra intuición, sin embargo, esto puede que no sea correcto en algunos casos. Por ello, a pesar de que en el libro nos explican tres modelos, creemos que para mantener una visión simple e intuitiva deberíamos seguir uno de los tres, en concreto, el modelo de los *cuatro criterios*:

1. **Contexto:** conjunto de acciones que se pueden realizar según la localización en la que nos encontremos.
2. **Tiempo disponible:** ¿Disponemos de suficiente tiempo para ejecutar esta tarea?
3. **Energía disponible:** ¿Sabiendo el gasto de energía que supone realizar esta tarea, es factible ejecutarla ahora mismo?
4. **Prioridad:** ¿Existe alguna otra tarea que podamos realizar y tenga mayor importancia sobre otras?

2.3 Aplicaciones

Por lo general, David Allen, nos da ciertas herramientas para poder implementar la filosofía *GTD* como una extensión más allá de nuestra rutina cotidiana. Sin embargo, debemos tener en cuenta que el libro fue escrito a comienzos de los años dos mil. Por ello, en lugar de usar términos como “papeleras”, “notas de voz” o “apuntes en papel”, usaremos herramientas actuales que vemos todos los días, tales como los ordenadores, móviles o tablets, de forma que podamos integrarlo con mayor facilidad en nuestro día a día.

Basándonos en esta idea, hemos explorado el mercado de aplicaciones que existen tanto en *App Store* y *Google Play*, encontrándonos con las aplicaciones más destacadas para poder implementar la metodología, siendo estas:

- **NirvanaHQ:** Aplicación web y móvil diseñada específicamente para *GTD*. Ofrece características como listas de proyectos, listas de acciones y enfoque en el flujo de trabajo *GTD*.
- **TodoIst:** Aplicación de gestión de tareas. Permite crear listas de tareas, recordatorios, establecer etiquetas y organizar tus tareas según los principios básicos de *GTD*.
- **Notion:** *Notion* es una herramienta de productividad versátil que puede personalizarse para adaptarse a *GTD*. Permite crear bases de datos, tablas y tableros para organizar tus tareas y proyectos.

-
- **Things**: Se trata de una aplicación de gestión de tareas disponible para dispositivos de *Apple*. Tiene una interfaz elegante y herramientas que se alinean con *GTD*, como áreas, proyectos y tareas.
 - **Omnifocus**: Ofrece una amplia gama de características de organización y permite crear proyectos, tareas y contextos para seguir la metodología *GTD*.
 - **TickTick**: *TickTick* es una aplicación de gestión de tareas con múltiples características que pueden utilizarse para implementar *GTD*. Ofrece listas de tareas, calendario y recordatorios.

2.4 Desafíos que encontramos

Cuando planteamos este trabajo de fin de grado teníamos como premisa, fusionar lo mejor de cada aplicación que nos encontramos en el mercado en una sola, dando así una solución completa para todo tipo de usuarios.

A pesar de que las apps mencionadas en el anterior apartado son excelentes, observamos ciertas deficiencias en cada una de ellas que hacen que dificulten el acceso y la implementación del método de David Allen, tal y como detallamos en la siguiente tabla:

Aplicación	¿Multiplataforma?	Aspectos positivos	Aspectos negativos
NirvanaHQ	Si	Diseñada especialmente para GTD Interfaz intuitiva y fácil de utilizar Soporte multiplataforma Funciones avanzadas de recordatorio, etiquetas y filtrado	Interfaz demasiado simple pudiendo confundirse
To-dolist	Si	Altamente personalizable y con grandes capacidades (Uso de base de datos, tablas..) Se puede integrar gran cantidad de métodos no solo <i>GTD</i>	Versión gratuita con muchas restricciones Difícil integración de <i>GTD</i> ya que carece de la función de creación de proyectos
No-tion	Si	Diseño elegante, minimalista y centrado en <i>GTD</i>	Alta curva de aprendizaje combinado con un sentimiento abrumador dada la cantidad de características y flexibilidad que ofrece
Things Solo	Solo dispositivos		Uso reducido a dispositivos <i>Apple</i> Carece de características avanzadas
<i>Apple</i>	Solo dispositivos	Alta capacidad de organización para proyectos, tareas y manejos de contextos	Uso reducido a dispositivos <i>Apple</i> Alta curva de aprendizaje
Tick-Tick	Si	Interfaz intuitiva y fácil de usar	La versión gratuita difiere bastante de la versión premium

Teniendo esto en cuenta, debemos tener en consideración que no solo hay falta una aplicación con una buena interfaz y que integre todas las características del método, sino que además debe ser fácil e intuitiva de utilizar para que se pueda poner en práctica diariamente. Y sobre todo, que aunando estas características, proporcione e informe de forma transparente el tratamiento de datos de sus usuarios.

2.5 Conclusiones

En resumen, la filosofía *Getting Things Done* pone por escrito muchas de las funciones que usualmente tenemos en mente, pero que no conseguimos ejecutar correctamente por motivos diversos, estando estos relacionados con la organización. Es por esto por lo que tener fases bien definidas, permite gestionar mejor el día a día, consiguiendo enfocarnos en lo que realmente importa.

A pesar de que, como todo método, puede presentar ciertas dificultades, merece la pena implementarlo en la rutina diaria pues tiene una corta curva de aprendizaje siendo fácilmente adaptable a múltiples facetas de la vida, a diferencia de otros métodos y/o técnicas (p.ej.: técnica pomodoro).

Teniendo esto en cuenta, detallaremos en el Capítulo 6 la implementación del *Frontend* de la app, así como qué características destacamos para hacer llegar este método a los dispositivos personales de todo individuo.

Capítulo 3

Planificación del proyecto

A la hora de gestionar y planificar un proyecto existen dos tipos de enfoque, el predictivo y el adaptativo. El enfoque predictivo se utiliza cuando se tiene claro cómo va a ser el proyecto y se conocen las variables y resultados del mismo, mientras que el enfoque adaptativo es más flexible y permite modificar el alcance del proyecto conforme a las necesidades que van surgiendo a lo largo del desarrollo del propio proyecto. Este segundo enfoque conduce a una mayor calidad y productividad aumentando el compromiso del equipo [3].

En nuestro caso, hemos seguido un enfoque adaptativo ya que nuestro proyecto requiere flexibilidad y velocidad. Para ello, hemos seguido una metodología *agile* para desarrollar nuestro proyecto. Esta metodología consiste en trocear el proyecto en pequeñas partes que se tienen que ir completando en cortos períodos de tiempo. Siguiendo esta idea, hemos dividido las tareas semanalmente, de tal manera que en cada reunión definimos las tareas a realizar la próxima semana.

Adicionalmente, hemos empleado herramientas ágiles como el tablero *Kanban*, el cual nos ayuda a gestionar el proyecto de una manera más visual, viendo en qué estado se encuentra cada tarea que hemos definido. Para ello hemos utilizado la herramienta *Notion*¹ como podemos ver en la figura 3.1, la cual muestra un ejemplo de cómo hemos ido organizando las distintas tareas a lo largo de este proyecto. *Notion* también nos permite crear un tablero distinguiendo el estado actual de las tareas en “sin empezar”, “en curso” o “terminadas” y también ofrece la posibilidad de asignar cada tarea a uno o varios miembros del equipo que serán los encargados de completarlas.

Nuestro proyecto ha seguido un ciclo de vida iterativo e incremental, de tal manera que en cada iteración se ha revisado y mejorado el producto, el cual ha sido desarrollado por partes que se han ido integrando para construir el producto final.

A continuación, describiremos los puntos más importantes que hemos seguido en cuanto a

¹Notion

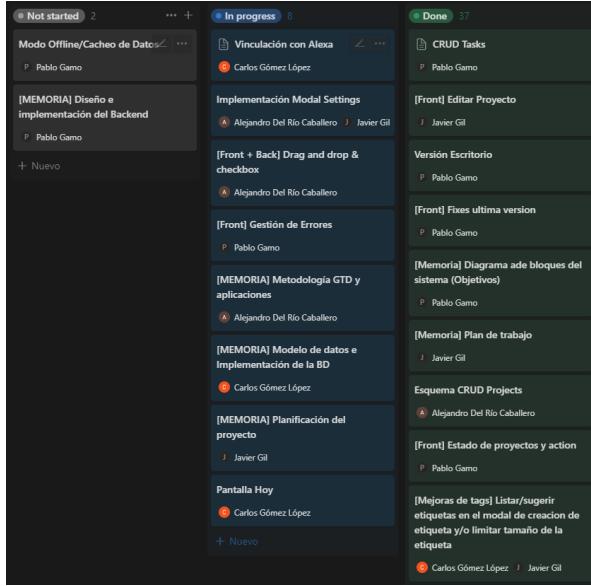


Figura 3.1: Tablero de las tareas en Notion

la planificación del proyecto, en aspectos como el sistema de control de versiones utilizado, la gestión de las pruebas y los sistemas de despliegue de versiones implementados.

3.1 Gestión de configuración del Software

3.1.1 Sistema de control de versiones

Para una correcta gestión y desarrollo de cualquier proyecto *software* un sistema de control de versiones es esencial e imprescindible para un equipo de desarrollo. Para nuestro trabajo hemos utilizado el ampliamente extendido *Git*. Este sistema nos permite sincronizar las distintas versiones garantizando la gestión de conflictos entre los cambios realizados por el equipo de desarrollo. También guarda los datos de las distintas versiones para que sean recuperables en cualquier momento si fuese necesario. Para el alojamiento del repositorio del código hemos utilizado la plataforma *Github*.

Además hemos establecido un procedimiento de gestión de ramas en nuestro repositorio, con el fin de tener controladas las distintas funcionalidades desarrolladas. El repositorio contiene 2 ramas principales, la rama “dev” que es la que utilizamos diariamente para desarrollar nuestro producto, y que es donde se encuentra todo el código que está siendo desarrollado en el momento y que por lo tanto no está acabado. Por otro lado rama “main”, que es la principal y la que contiene las funcionalidades terminadas y probadas. Esta rama “main” contiene la versión más actualizada de la aplicación. Cuando se ha terminado de desarrollar una versión y esta está probada en la rama “dev”, se procede a hacer *merge* hacia la rama “main”.

Por último, tenemos ramas auxiliares que utilizamos para desarrollar funcionalidades más específicas, de tal manera que si los cambios realizados hacen que otras partes de la aplicación puedan no funcionar, no afecte al resto y se pueda realizar un *backup* de manera sencilla. Un ejemplo de estas últimas pueden ser las ramas llamadas *OAuth2.0* o *modoOffline* cuyo nombre hace referencia al desarrollo de dicha funcionalidad que se decidió hacer por separado a las ramas principales. Una vez implementadas dichas funcionalidades, estos cambios pasan a la rama dev y posteriormente a main.

3.1.2 Entornos de desarrollo e Integración

Amazon Web Services (AWS) ha sido una parte clave para nuestro proyecto. Este famoso proveedor de servicios en la nube dispone de gran cantidad de recursos de computación. Para nuestro proyecto hemos utilizado el servicio *EC2* bajo la prueba gratuita de este mismo. Este servicio permite crear instancias de máquinas en los servidores de *AWS* donde se puede ejecutar cualquier tipo de aplicación o proceso. La prueba gratuita ofrece la posibilidad de crear y tener encendida una de estas instancias, sin ningún coste durante un año siempre y cuando no se sobrepasen ciertos límites de uso, que de darse el caso se cobra las tasas estándar. Sin embargo, para el desarrollo del trabajo no hemos llegado ni por asomo a superarlos. Estos límites son 750 horas de uso de instancias mensuales, 35GB de uso de almacenamiento y 100GB de tráfico de red mensuales.

Para el desarrollo del trabajo creamos una instancia con un sistema operativo *Linux* que en este caso era una distribución de *Amazon* llamada *Amazon Linux*, esta distribución está diseñada para ser fácilmente configurada para los diversos servicios de *AWS* aunque también es posible usar cualquier otro tipo de distribuciones u otros sistemas operativos. El principal uso que hemos hecho de esta instancia ha sido la ejecución de nuestro *backend* en la nube, lo cual nos ha facilitado bastante el proceso de desarrollo ya que hemos podido ejecutar nuestra aplicación en un entorno de servidor y realizar las correspondientes integraciones con el *frontend*. Además tener el proyecto siendo ejecutado en la nube nos ha permitido que todos los miembros del equipo pudieran probar con facilidad los cambios realizados por los demás así como compartir el contexto de datos de la aplicación, lo que ha facilitado las pruebas.

En la instancia previamente mencionada hemos instalado el sistema de gestión de contenedores *Docker*, que es el sistema con el que se ejecuta completamente el *backend*. Mediante la herramienta de dicho sistema llamada *Docker-compose* podemos ejecutar varios contenedores y que éstos sean gestionados de manera conjunta por *Docker*. Para ello definimos en un fichero *.yaml* el conjunto de contenedores que serán ejecutados dentro del mismo entorno así como la configuración de cada uno. Esta configuración coincide con la modelada en la sección 1.2.1. En nuestro caso tenemos 2 contenedores, uno formado a partir de una imagen de *Node.js* donde se ejecuta la *API Rest* y otro donde se ejecuta la base de datos, el cual está generado a partir de una imagen de *Postgresql*. La principal ventaja de esta herramienta es la posibilidad de definir todos los servicios a ejecutar dentro de un solo fichero con el fin de luego desplegar o ejecutar en

la máquina todos ellos con un solo comando en vez de tener que gestionar por separado todos estos servicios. Además *Docker* crea un red interna para los contenedores de manera que puedan conectarse entre sí, si la situación lo requiere, como por ejemplo en nuestro caso, donde tenemos por separado un contenedor con la base de datos y otro con la *API REST*.

3.1.2.1 Sistema de despliegue

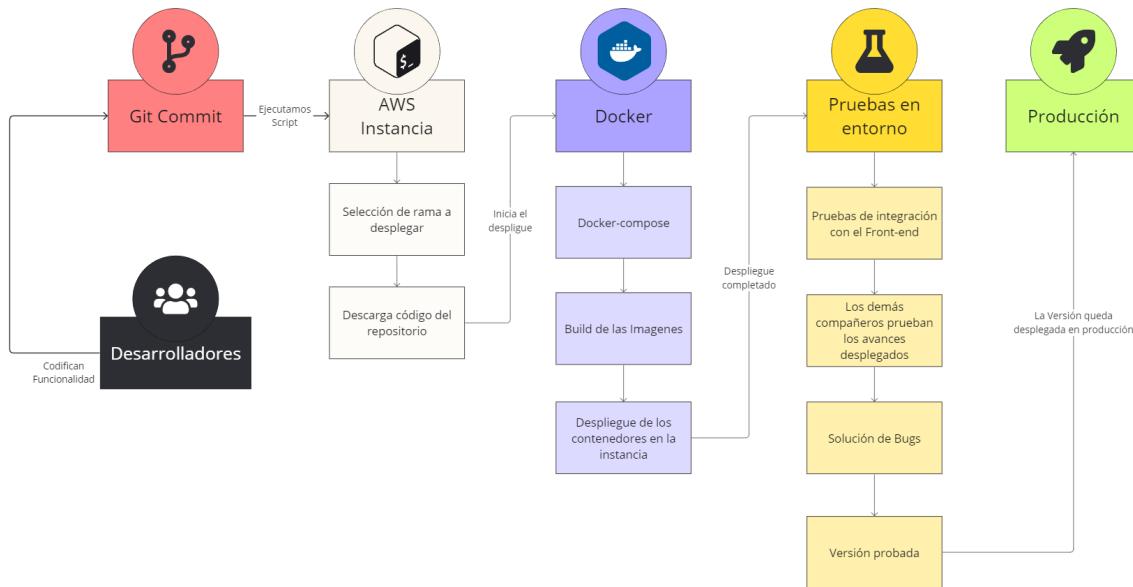


Figura 3.2: Diagrama del sistema de despliegue

Uno de los aspectos que más nos han ayudado a la hora de desarrollar nueva funcionalidad y gestionar la calidad de esta ha sido la puesta en marcha de un sistema de despliegue de versiones en el entorno de la nube mencionado anteriormente. Este sistema consiste en un *script* de *bash* que hemos escrito para poder realizar con agilidad despliegues en el entorno y facilitar las pruebas de integración con el *frontend*, así como la puesta en producción de funcionalidad.

El *script* es lanzado cuando queremos desplegar una nueva versión para probar los cambios desarrollados en local en el entorno del servidor. El proceso que sigue el *script* puede verse modelado en la figura 3.2 donde se observa el flujo que sigue nuestro código cuando realizamos el despliegue. En primer lugar lanzamos el *script* precedido por un *commit* que hayamos realizado y que queramos probar. Después se procede a clonar el código de nuestro repositorio en la máquina. A continuación nos pregunta qué rama queremos desplegar; casi siempre suele ser la rama “dev” ya que es donde se desarrolla la funcionalidad principal, aunque en ocasiones utilizamos las ramas auxiliares que hemos comentado en el apartado anterior. Tras seleccionar la rama, el *script* inicia el proceso de despliegue de los contenedores, ejecutando la herramienta *Docker-compose*. El *script*

primero detiene los contenedores en ejecución y a continuación inicia el *build* y el *run* de los nuevos, es decir prepara las imágenes de los contenedores con los cambios descargados previamente del repositorio para luego iniciar su ejecución. Tras este último paso los cambios realizados sobre el repositorio se encuentran ya en el entorno.

Cada vez que se realiza un despliegue comenzamos con las pruebas en entorno donde volvemos a comprobar que los cambios también funcionan en este. Para ello seguimos el mismo procedimiento que en local pero apuntando a la *IP* del servidor. Para realizar esta acción utilizamos la herramienta *Postman* que permite lanzar peticiones a nuestra *API* y ejecutar sus operaciones. Si observamos que el resultado de la ejecución de las llamadas al servidor no es correcto y detectamos algún fallo procedemos a mirar los logs del contenedor que ejecuta la *API*. Para este aspecto hemos incluido en el código métodos de *logging* y trazado para facilitar la detección de errores así como para depurar el funcionamiento del código. Entre estos métodos disponemos de un sistema que imprime todas las *queries* ejecutadas y el resultado de éstas en cuanto a datos devueltos o tiempo de respuesta entre otros. También disponemos de la librería de *Node.js Morgan*² la cual imprime en el *log* todas las *request* que recibe la *API* así como la información sobre éstas como método *HTTP*, código de respuesta o tiempo de respuesta entre otros. Si detectamos algún error procedemos a replicarlo en local, solucionarlo e iniciar de nuevo el proceso de despliegue.

²Morgan

Capítulo 4

Modelo de datos e implementación de la Base de datos

Este capítulo tiene como objetivo describir en detalle el modelo de datos utilizado en nuestra aplicación *SwiftDo*, proporcionando una visión exhaustiva de cómo se organizan y relacionan los datos esenciales para su funcionamiento.

En primer lugar, presentaremos las principales entidades que componen nuestro modelo, incluyendo tareas, usuarios, proyectos, áreas, etiquetas y las relaciones asociadas a la autorización *OAuth*. Cada entidad es examinada en profundidad, detallando sus atributos y el propósito que cumplen dentro del contexto de la aplicación.

Posteriormente, analizaremos las relaciones entre estas entidades, destacando cómo se conectan y cómo estas conexiones facilitan el flujo de información y la interacción dentro de la aplicación.

Además, discutiremos la implementación física del modelo de datos en la base de datos. Nuestra base de datos está alojada en un entorno *Docker* en *AWS (Amazon Web Services)*, utilizando *PostgreSQL* como sistema de gestión de bases de datos. Describiremos la estructura de tablas, índices y restricciones de integridad referencial, resaltando cómo estas decisiones de diseño se traducen en la configuración final de la base de datos.

Por último, analizaremos aspectos críticos como la seguridad de la base de datos, las consideraciones de rendimiento y escalabilidad. Estos temas son esenciales para garantizar la integridad, confidencialidad y disponibilidad de los datos.

4.1 Descripción de entidades

En la figura 4.1 se muestra la relación entre las distintas entidades. A continuación, desarrollaremos las principales entidades de nuestra aplicación, junto con sus atributos y funciones dentro del sistema:

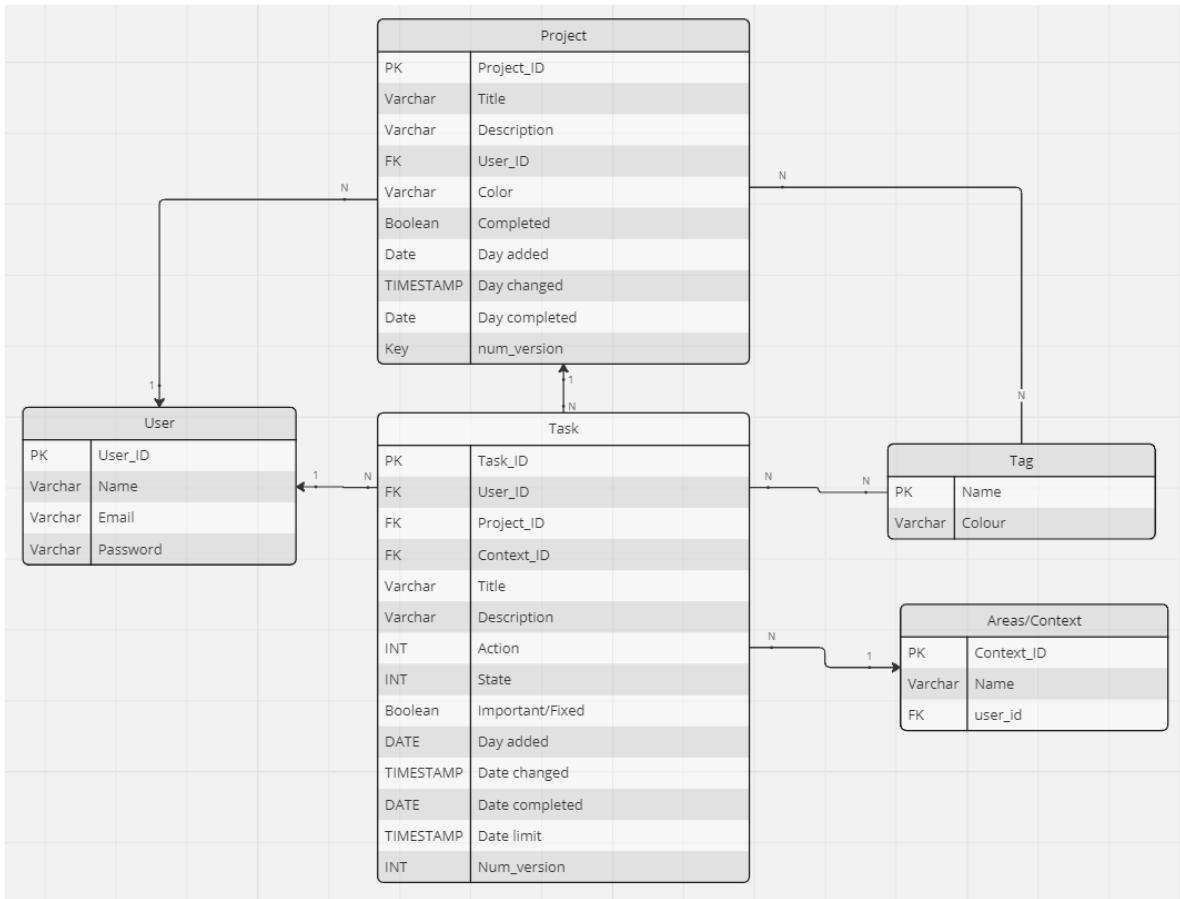


Figura 4.1: Diagrama Entidad-Relación de la aplicación

-
- **Tarea:** Representa las actividades a realizar dentro de la aplicación. Cada tarea es creada por un usuario y puede contener atributos como título, descripción, fecha límite y prioridad. Además, las tareas pueden ser modificadas en cualquier momento por el usuario propietario, completadas cuando se finalicen, ser asignadas a proyectos específicos o etiquetadas con *tags* relevantes para una mejor organización.
 - **Usuario:** Los usuarios tienen un rol central en la organización y gestión de la aplicación, ya que tienen la capacidad de crear, modificar y eliminar tanto tareas como proyectos. Pueden también crear y gestionar áreas para una organización más eficiente de su espacio de trabajo. Además, tienen el control sobre aspectos de personalización de la aplicación, junto con la capacidad de modificar su propio perfil.
 - **Proyecto:** Permite a los usuarios organizar sus tareas en conjuntos más amplios relacionados con un objetivo común. Los proyectos pueden ser creados, editados y eliminados por los usuarios, y las tareas pueden ser asignadas a proyectos específicos para una gestión más eficiente.
 - **Área:** También conocida como “Contextos”, ofrece una forma adicional de categorizar las tareas. Los usuarios pueden crear, modificar y eliminar áreas según sus necesidades, y asignar tareas a áreas específicas para una mejor organización y seguimiento.
 - **Etiqueta:** La entidad etiquetas permite etiquetar las tareas con palabras clave relevantes para una clasificación más detallada. Los usuarios pueden crear, editar y eliminar etiquetas, y asignarlas a tareas individuales para una organización más flexible y personalizada.
 - **Relaciones de OAuth:** Las entidades *oauth_authcode*, *oauth_clients* y *oauth_tokens* están relacionadas con el proceso de autorización *OAuth* para la autenticación de usuarios en la aplicación, facilitando la seguridad y la gestión de accesos.

4.2 Relaciones entre entidades

En esta sección, exploraremos las relaciones entre las diferentes entidades dentro del Modelo de Datos de nuestra aplicación. Estas relaciones son fundamentales para comprender cómo interactúan los distintos componentes del sistema y cómo se organiza la información.

- **Tarea y Usuario:** Cada tarea de la aplicación está asociada a un usuario que la crea y gestiona. Esta relación permite a los usuarios tener control total sobre sus propias tareas, incluyendo la creación, modificación y eliminación.
- **Tarea y Proyecto:** Las tareas pueden estar vinculadas a proyectos específicos, lo que facilita la organización y seguimiento de las actividades dentro de entornos

más amplios. Esta relación permite a los usuarios agrupar las tareas relacionadas bajo un objetivo común y gestionarlas de manera eficiente.

- **Tarea y Área:** Las tareas también pueden estar asociadas a áreas o contextos específicos, lo que proporciona una categorización adicional para una mejor organización. Los usuarios pueden asignar tareas a áreas relevantes según el entorno en el que deben realizarse, lo que facilita la priorización y gestión de estas.
- **Tarea y Etiqueta:** Las etiquetas se utilizan para clasificar y categorizar las tareas según temas o características comunes. Las tareas pueden estar etiquetadas con una o más etiquetas, lo que permite una organización más detallada y organizada. Esta relación permite a los usuarios filtrar y buscar tareas según etiquetas específicas para una gestión más eficiente.
- **Usuario y Proyecto/Área:** Los usuarios tienen la capacidad de crear, modificar y eliminar tanto proyectos como áreas dentro de la aplicación. Esta relación permite a los usuarios organizar y personalizar su espacio de trabajo de acuerdo con sus necesidades y preferencias.

4.3 Modelo físico de la BD

En la figura 4.2 se mostrará un diagrama de la estructuración de las tablas en la base de datos. A continuación, vamos a detallar la implementación concreta del modelo de datos en la base de datos real que respalda nuestra aplicación. Describimos la estructura de tablas, los tipos de datos utilizados, así como las relaciones y restricciones de integridad referencial.

Empezamos con las descripciones detalladas de las tablas que componen la base de datos:

- **tasks:**
 - “task_id”: Identificador único de la tarea
 - “user_id”: Identificador del usuario al que pertenece la tarea (obligatorio).
 - “context_id”: Identificador del área/contexto en el que puede estar una tarea (opcional).
 - “project_id”: Identificador del proyecto al que pertenece la tarea (opcional).
 - “title”: Título de la tarea (obligatorio).
 - “description”: Descripción de la tarea (opcional).
 - “state”: Sección en la que se encuentra la tarea (Inbox, Cuanto Antes, Programadas, etc.).
 - “completed”: Booleano que indica si la tarea está completa o no.
 - “important_fixed”: Booleano que indica si la tarea es importante (prioridad a la hora de hacer tareas).
 - “date_added”: Fecha en la que se añade la tarea.

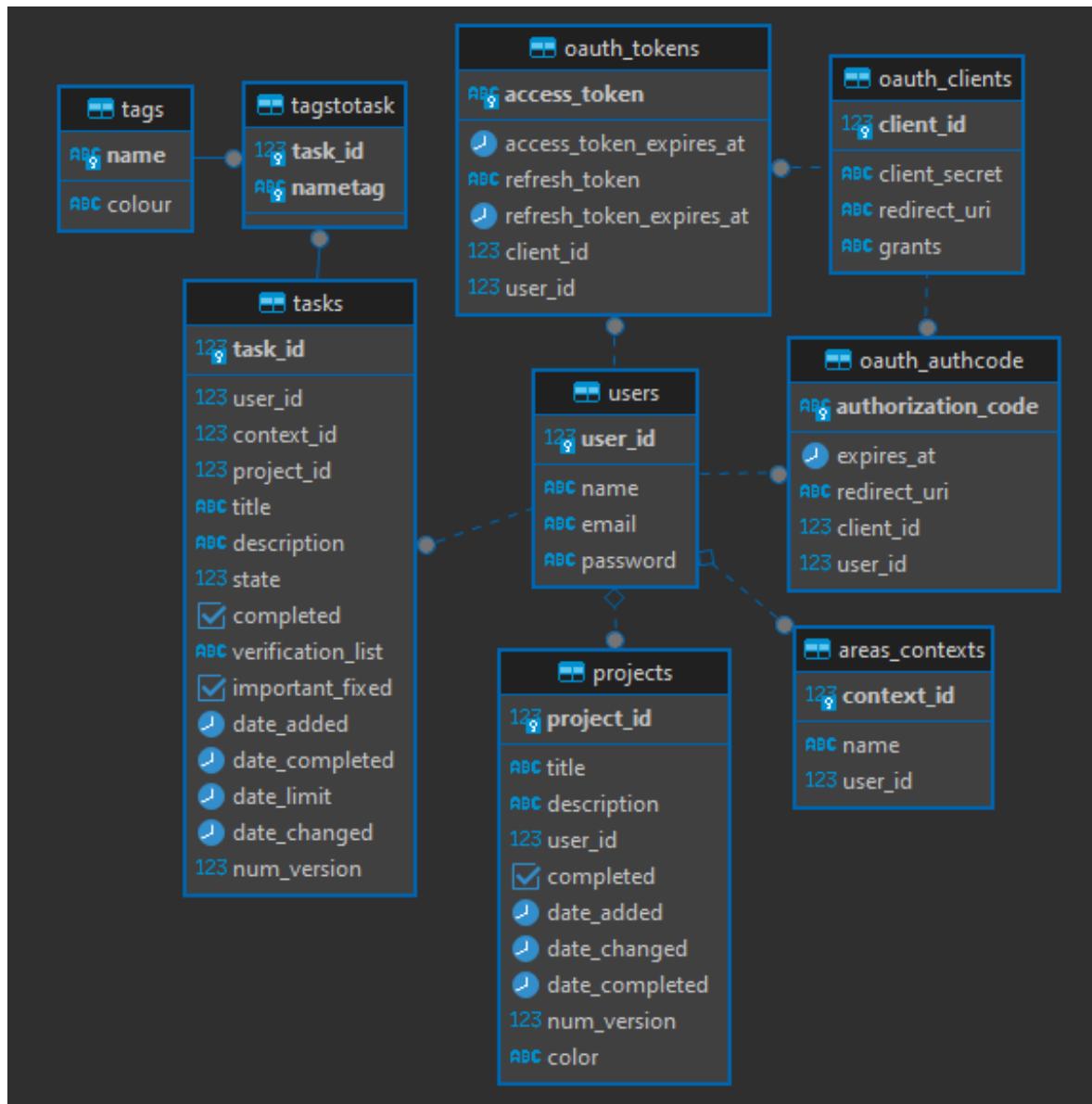


Figura 4.2: Diagrama Entidad-Relación en la base de datos

-
- “date_completed”: Fecha en la que se completa la tarea.
 - “date_limit”: Fecha límite para realizar la tarea (opcional).
 - “date_changed”: Fecha en la que se realiza algún cambio en la tarea.
 - “num_version”: Número de versión por la que se va en la tarea (por cada cambio).
 - **users:**
 - “user_id”: Identificador del usuario
 - “name”: Nombre del usuario
 - “email”: Correo del usuario
 - “password”: Contraseña del usuario
 - **projects:**
 - “project_id”: Identificador del proyecto
 - “title”: Título del proyecto
 - “description”: Descripción del proyecto
 - “user_id”: Identificador del usuario al que pertenece el proyecto
 - “completed”: Boolean que indica si está completo el proyecto o no
 - “date_added”: Fecha en la que se añade el proyecto
 - “date_changed”: Fecha en la que se modifica el proyecto
 - “date_completed”: Fecha en la que el proyecto ha sido completado
 - “num_version”: Número de version en la que se encuentra el proyecto
 - “color”: Color que corresponde al proyecto
 - **areas_contexts:**
 - “context_id”: Identificador del área/contexto
 - “name”: Nombre del área
 - “user_id”: Identificador del usuario al que corresponde el área
 - **tags:**
 - “name”: Nombre de la tag
 - “color”: Color de la tag
 - **tagstotask:**
 - “task_id”: Identificador de la tarea a la que pertenece la tag
 - “nametag”: Nombre de la tag
 - **oauth_authcode:**
 - “authorization_code”: Código de autorización
 - “expires_at”: Fecha de expiración del código
 - “redirect_uri”: Dirección a la que te redirige si la obtención del código de autenticación es correcto.
 - “client_id”: Identificador del cliente

-
- “user_id”: Identificador del usuario que recibe el código

- **oauth_clients:**

- “client_id”: Identificador del cliente
- “client_secret”: Número secreto del cliente
- “redirect_uri”: Dirección a la que te redirige si la obtención del código de autenticación es correcto.
- “grants”: Indica los flujos de Oauth configurados para el cliente.

- **oauth_tokens:**

- “access_token”: Token de acceso
- “access_token_expires_at”: Fecha de expiración del token
- “refresh_token”: Token refrescado
- “refresh_token_expires_at”: Fecha de expiración del token refrescado
- “client_id”: Identificador del cliente
- “user_id”: Identificador del usuario

A continuación, comentamos cómo se establecen las relaciones entre las tablas para mantener la coherencia de los datos y garantizar su integridad:

- **Usuarios con Tareas/Proyectos/Áreas:** La tabla de “users” con las tablas “tasks”, “projects”, “areas_contexts” tiene una relación uno a muchos. (Creo que se podría quitar) Un usuario puede tener muchas tareas, proyectos y áreas, pero cada una de ellas pertenece a un solo usuario. Esta relación uno a muchos se establece mediante la clave foránea “user_id” en las tablas “tasks”, “projects” y “areas_contexts”, que referencia al identificador único del usuario en la tabla “users”.
- **Proyectos y Tareas:** La tabla “projects” tiene una relación uno a muchos con la tabla “tasks”. Un proyecto puede tener muchas tareas, pero cada tarea pertenece solo a un proyecto. Esta relación uno a muchos se establece mediante la clave foránea “project_id” en la tabla “tasks”, que referencia al identificador único del proyecto en la tabla “projects”.
- **Áreas y Tareas:** La tabla “areas_contexts” tiene una relación uno a muchos con la tabla “tasks”. Un área puede tener muchas tareas, pero cada tarea pertenece solo a un área. Esta relación uno a muchos se establece mediante la clave foránea “context_id” en la tabla “tasks”, que referencia al identificador único del área en la tabla “areas_contexts”.
- **Tareas y Tags:** Una tarea puede tener muchas etiquetas, y una etiqueta puede estar asociada a muchas tareas. Esta relación muchos a muchos se implementa mediante una tabla intermedia “tagstotask”, que contiene las claves foráneas “task_id” y “nametag” que relacionan las “tasks” con las “tags”.

4.4 Rendimiento y Optimización de la Base de Datos

En esta sección, detallamos la estructura y el rendimiento de la base de datos implementada. Describimos las acciones concretas llevadas a cabo para mejorar la eficiencia y capacidad de respuesta del sistema ante un crecimiento progresivo de carga de trabajo.

Para mejorar el rendimiento de la base de datos hemos realizado una optimización de consultas, definiendo claves primarias en las tablas pertinentes para garantizar la unicidad de las filas, que además de proporcionar integridad, nos ha ayudado a mejorar el rendimiento de las consultas. Además hemos diseñado consultas eficientes para minimizar la carga en el servidor de la base de datos.

Por otro lado, la base de datos ha sido diseñada siguiendo reglas de normalización, con el fin de mantener la integridad de los datos, aunque es posible que si detectamos con el tiempo que alguna consulta puede ser optimizada, realizaremos un proceso de desnormalización de las tablas para mejorar el rendimiento. De la misma manera se estudiará la implementación de índices de búsqueda en aquellas claves más utilizadas en las consultas.

4.5 Seguridad de la Base de Datos

La seguridad de la base de datos es un componente fundamental para proteger la integridad, confidencialidad y disponibilidad de los datos almacenados. En esta implementación, hemos adoptado diversas medidas para garantizar un entorno seguro:

- **Autenticación y Autorización:** Hemos implementado un sistema de autenticación robusto que requiere credenciales válidas para acceder a la base de datos. Se hablará de este sistema en la sección 5.5.1.
- **Cifrado de datos:** Implementamos técnicas de cifrado utilizando la biblioteca *bcrypt* [4] para proteger la información sensible almacenada en la base de datos que pueda ser vulnerable a accesos no autorizados (contraseñas). *Bcrypt* es un algoritmo de hashing adaptativo diseñado específicamente para almacenar contraseñas de manera segura. Este enfoque garantiza que las contraseñas estén protegidas contra ataques de fuerza bruta y de diccionario, proporcionando una capa adicional de seguridad para mantener la información confidencial protegida en todo momento.
- **Registro de actividades:** Empleamos las capacidades integradas de registro y auditoría proporcionadas por AWS para supervisar todas las actividades realizadas en nuestra base de datos. Estas funciones nos permiten rastrear quién accede a la base de datos, cuándo lo hace y qué operaciones realiza, garantizando la integridad y seguridad de los datos almacenados en la nube.

Para finalizar este capítulo, es crucial destacar la importancia del diseño y la implementación eficientes del modelo de datos en nuestra aplicación de gestión de tareas. A través

de un análisis exhaustivo de las entidades, relaciones y consideraciones técnicas, hemos establecido una base sólida para el funcionamiento de nuestra base de datos. Al comprender la estructura subyacente y las decisiones de diseño, estamos mejor preparados para abordar los desafíos futuros y garantizar la integridad, seguridad y escalabilidad continuas de nuestra aplicación *SwiftDo*.

Capítulo 5

Diseño e implementación del Backend

La arquitectura cliente-servidor es una de las más usadas hoy en día y por lo tanto tener un *backend* robusto es de vital importancia en cualquier aplicación. Por ello hemos invertido gran cantidad de esfuerzo en el diseño y la implementación del *backend* para asegurar un correcto funcionamiento de la aplicación a nivel de lógica de negocio además de garantizar seguridad en la comunicación con los clientes a la hora de acceder a datos y ejecutar operaciones sobre los mismos.

En este capítulo vamos a comenzar analizando el proceso que hemos seguido para el diseño de la *API*, desde aspectos de diseño como el empleo de la Arquitectura *REST* a la elección de las tecnologías empleadas. A continuación describiremos el proceso de implementación del *backend*, donde explicaremos el funcionamiento de las tecnologías aplicadas y otras características importantes como los aspectos de seguridad abordados.

5.1 Uso de REST

Una de las características a destacar es el uso de la arquitectura *REST* [5] para nuestra *API*. La clave de esta arquitectura es el uso del protocolo *HTTP* para el acceso a una interfaz de operaciones bien definidas. Además la arquitectura *REST* busca definir una sintaxis para identificar los recursos y sus diferentes operaciones consiguiendo así una organización clara y mantenible de los diferentes elementos de nuestra aplicación para facilitar el acceso desde los clientes. El uso de esta arquitectura en nuestro proyecto viene dada por la necesidad de organizar los distintos recursos a los que puede acceder la aplicación cliente con el objetivo de estandarizar el servicio y facilitar la comunicación y evolución de este.

Por otro lado, aunque *REST* permite la comunicación entre el cliente y el servidor mediante cualquier formato, usamos el formato *JSON* ya que es un formato muy sencillo de comprender y manipular. Además, en comparación a otros formatos como por ejemplo *XML*, *JSON* es mucho más ligero en cuanto a demanda de ancho de banda.

5.2 Descripción tecnologías: Node JS + Express

Para el desarrollo del *backend* nos hemos decantado por el uso del entorno *Node.js* en conjunto con el *framework* llamado *Express.js*, el cual proporciona una manera muy sencilla de implementar gran cantidad de servicios de aplicaciones web.

Node.js es un entorno de ejecución multiplataforma basado en el lenguaje de programación *JavaScript* para la capa del servidor. Este entorno utiliza el motor de *JavaScript V8* utilizado en el navegador *Google Chrome*, pero en este caso se utiliza fuera de un navegador lo cual incrementa el rendimiento considerablemente. Una aplicación con *Node.js* se ejecuta en un mismo proceso, es decir no crea un proceso para cada *request* y es por ello que este entorno provee de un conjunto de estándares asíncronos para prevenir que el código *JavaScript* tenga bloqueos. Cuando *Node.js* realiza operaciones de entrada/salida como acceder a una base de datos o leer del sistema de ficheros, en vez de bloquear el hilo y perder ciclos de *CPU* lo que hará será reanudar las operaciones cuando vuelva la respuesta a la lectura/escritura. En resumen *Node.js* permite realizar gran cantidad de conexiones en el servidor sin emplear concurrencia entre hilos. Otra de las principales características de *Node.js* es el lenguaje *JavaScript* el cual es ampliamente utilizado por desarrolladores de *front-end* en la web y que con *Node.js* tienen la posibilidad de escribir código de parte del servidor sin necesidad de aprender otros lenguajes.

Desarrollar en el entorno *Node.js* permite además utilizar gran cantidad de paquetes que se gestionan a través de la herramienta *npm*. Con esta herramienta, que se incluye con la instalación del entorno, es posible descargar e instalar paquetes de los repositorios de *npm* con gran facilidad ya que este gestiona de manera automática todas las dependencias. Una vez se instalan paquetes, *npm* genera el fichero *package.json* y *package-lock.json* los cuales contienen información sobre todos los paquetes instalados y sus dependencias. De esta manera, el proyecto puede ser instalado y ejecutado en cualquier máquina resolviendo *npm* todo el proceso de instalación. Esta característica de *Node.js* nos ha facilitado bastante la colaboración en el proceso de desarrollo de la *API* ya que en nuestros repositorios hemos incluido los ficheros mencionados previamente y *npm* ha gestionado completamente la instalación del entorno.

5.3 Express.js

Para la implementación de la *API* hemos utilizado el *framework* *Express.js* [6] el cual proporciona herramientas para implementar gran variedad de servicios de *backend* de una manera minimalista y sencilla. Express proporciona principalmente herramientas para gestionar el enrutamiento en una aplicación web, facilita el manejo de *requests* y *responses* de *HTTP* y la implementación de *middlewares* entre otras características. La razón por la que hemos escogido este *framework* es su sencillez para definir manejadores de ruta con los cuales podíamos implementar los diferentes *endpoints* de nuestra *API* además de las facilidades que proporciona para gestionar tanto *requests* como *responses*.

lo que nos ha servido para gestionar la de entrada de datos y los errores.

Como hemos mencionado anteriormente, gracias a la facilidad con la que puede gestionarse la instalación de paquetes en un entorno *Node.js*, hemos hecho uso de varios paquetes que complementan a *Express.js* para el desarrollo de la *API*. Entre ellos destacan el paquete *express-validator* para facilitar la validación sintáctica de los valores de entrada a los *endpoints*. También se han empleado otros paquetes como *pg*, que es el módulo para conectar un sistema de *Node* a una base de datos *PostgreSQL*, el paquete *bcrypt* que hemos utilizado para cifrar las contraseñas de los usuarios cuando se registran en la aplicación y el módulo de *OAuth 2.0* para *Node.js*, del cual hablaremos más en profundidad en la sección 5.5.1.

5.4 Diseño de la API

La arquitectura *REST* tiene como principal característica el empleo de una sintaxis para identificar los diferentes *endpoints*. Como se ha mencionado anteriormente, esta sintaxis busca relacionar de manera clara los *endpoints* con los recursos a los que accede y sus operaciones, definiendo una serie de reglas a la hora de diseñar la *API*.

Entre estas prácticas que hemos seguido para nombrar los *endpoints*, están utilizar el nombre del recurso al que se accede sin utilizar verbos, es decir, sin utilizar por ejemplo el nombre de la operación (crear, borrar, modificar...) ya que esta información ya la proporciona el método de *HTTP* utilizado. Precisamente esta última apreciación también la hemos tenido en cuenta, ya que hemos sacado partido de los diferentes métodos de *HTTP* para definir de manera clara y organizada las operaciones. También hemos hecho uso de rutas parametrizadas, es decir el uso de parámetros directamente en la *uri*, por ejemplo `/example/:example_id`. Otros de los aspectos de gran importancia en cualquier *API* y en el que nos hemos centrado para el diseño de la nuestra es la gestión de errores. Hemos especificado para cada endpoint qué códigos de status *HTTP* devolverán en función del resultado para poder gestionar cuándo una operación ha sido ejecutada con éxito o ha ocurrido algún error.

5.5 Aspectos de seguridad de la API

En cualquier servicio web la seguridad es uno de los aspectos más importantes a tener en cuenta, siendo más importante hoy en día debido al incremento de ataques y delitos informáticos. Además en nuestra aplicación disponer de mecanismos complejos y robustos de seguridad es principal debido a que los datos que comparten los usuarios pueden ser y principalmente serán datos personales. Esto es así porque se trata de una aplicación de organización personal y por lo tanto las tareas u otras partes de la aplicación contendrán información sensible que es importante proteger.

Debido a las razones explicadas previamente hemos puesto gran énfasis en implementar

mecanismos que dotan de seguridad a nuestro servicio. Entre estos mecanismos destacan la implementación del estándar de autorización *OAuth 2.0*, el cifrado de contraseñas o el tratado de datos de entrada para prevenir la *inyección SQL* entre otros.

5.5.1 Implementación de OAuth 2.0

Uno de los aspectos más importantes que hemos podido implementar en nuestra *API* es el estándar *OAuth 2.0* [7]. Este estándar introduce una nueva capa de autorización separando este rol del servidor de recursos y proporcionando así varias ventajas principalmente en términos de seguridad. A continuación se describe en profundidad cómo funciona este estándar y cuáles son sus principales ventajas.

Autorización, Autenticación y posibilidades de OAuth

Es común que con frecuencia se confundan o se mezclen los conceptos de autorización y autenticación, ya que en el modelo tradicional de autenticación entre cliente-servidor no se utilizaban y ahora es frecuente verlos siempre que se habla de acceso a *APIs* o recursos web.

En el modelo cliente-servidor tradicional el cliente solicitaba acceso a un recurso con acceso protegido mediante el uso de credenciales del servidor propietario de los recursos, es decir el cliente realizaba un proceso de autenticación en el que necesitaba las credenciales del servidor para acceder a los recursos. De esta manera cualquier cliente o aplicación de terceros necesita las credenciales del servidor para acceder. Esto tiene varios inconvenientes y limitaciones:

- Las aplicaciones de terceros necesitan almacenar las credenciales para su futuro uso, ya que es necesario enviarlas con cada *request* al servidor. De esta manera existe un gran riesgo de brecha de seguridad en el caso de que las credenciales sean robadas, ya que con ellas se tiene acceso prácticamente ilimitado a los recursos.
- Las aplicaciones de terceros tienen prácticamente acceso ilimitado al servidor de recursos ya que disponen de las credenciales con las que pueden acceder en cualquier momento mientras que el propietario de los recursos no puede limitar la duración del acceso o el conjunto de datos al que los clientes pueden acceder.
- El propietario de los recursos no puede revocar el acceso a estos a un cliente concreto sin revocarlo a todos ya que debe hacerlo cambiando la contraseña.
- Compromete al propietario de los recursos al uso que hagan las aplicaciones de terceros con la contraseña del usuario del servidor y toda su información protegida.

OAuth aborda estos problemas mediante la introducción de una capa de autorización separando así esta responsabilidad del servidor de recursos. En este estándar el cliente solicita acceso a los recursos y desde el servidor se le conceden unas credenciales completamente desacopladas de las del servidor de recursos.

Por lo tanto, en vez de utilizar las credenciales del servidor de recursos, el cliente obtiene un *token* de acceso, que es una cadena de caracteres que además de garantizar acceso, denota otros atributos como el alcance o la vida de este.

Roles y Flujo del proceso

Existen cuatro roles en *OAuth*:

- Propietario de los recursos: Es la entidad capaz de conceder acceso a un recurso protegido.
- Servidor de recursos: Es el servidor que aloja los recursos protegidos y es capaz de responder a las peticiones que llegan para acceder a los recursos mediante *tokens* de acceso.
- Cliente: Es una aplicación que realiza peticiones al servidor de recursos bajo la autorización del propietario.
- Servidor de autorización: Es el servidor encargado de gestionar los *tokens* de acceso y dar autorización a los clientes.

En la figura 5.1 se puede observar cómo interactúan entre sí los distintos roles en el proceso de autorización. En primer lugar el cliente pide acceso a los recursos al propietario y este concede o rechaza el permiso. Esta petición puede hacerse directamente al servidor de recursos o preferiblemente indirectamente a través del servidor de autorización como intermediario. A continuación al cliente ya se le ha concedido el permiso de acceso por lo que puede proceder a solicitar un *token*. De esta manera el cliente solicita un *token* y el servidor de autorización valida si el permiso es correcto devolviéndole uno en tal caso. Por último, una vez el cliente dispone del *token* de acceso, este puede solicitar al servidor de autorización el acceso a un recurso y acceder con éxito si este valida correctamente su *token*.

Aplicación del estándar *OAuth2.0*

Tras explicar los principales puntos de este estándar de autorización, a continuación se va a explicar cómo hemos implementado este proceso para nuestra aplicación.

En primer lugar se van a identificar cuales son las diferentes entidades o roles de *OAuth* en el caso de nuestra aplicación:

- **Propietario de los recursos:** En nuestro caso esta entidad será la persona o personas que gestionen el servicio de la aplicación, ya que esta podrá ser instalada y gestionada libremente en cualquier servidor.
- **Servidor de recursos:** Para nuestra aplicación el servidor de recursos es el servidor donde se encuentra el servicio de *API REST* con todos sus *endpoints* que tienen

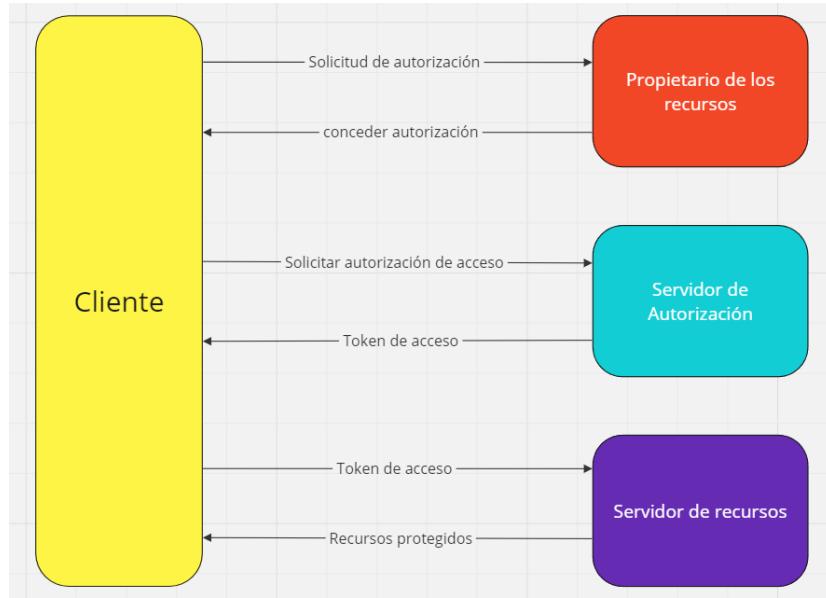


Figura 5.1: Proceso de autorización OAuth

la capacidad de interactuar con recursos de la aplicación, es decir con tareas, proyectos, usuarios, etc.

- **Cliente/s:** Los clientes son las aplicaciones de escritorio y la de móvil que realizan peticiones al servidor de recursos y que por lo tanto necesitan autorización del propietario.
- **Servidor de autorización:** Este servidor ha sido implementado dentro del mismo entorno que la *API REST* y es el encargado de gestionar la autorización del servicio.

En *OAuth* hay varios tipos de *grants* o formas de concesión de acceso a los recursos por parte del propietario. En otras palabras, hay varias formas de conseguir un *token* de acceso y cada una de estas tiene un procedimiento distinto para conseguirlo. Existen los siguientes cuatro tipos:

- **Código de autorización:** El código de autorización se obtiene utilizando el servidor de autorización como intermediario entre el cliente y el propietario de los recursos. De esta forma en vez de pedir directamente autorización al propietario, esta petición se dirige al servidor de autorización, que devuelve al cliente un código de autorización con el que luego podrá solicitar un *token* de acceso. Este tipo de *grant* es el que hemos implementado para nuestra aplicación y es por ello que lo explicaremos con mayor extensión más adelante.
- **Implícito:** En esta forma de conceder autorización, se simplifica el flujo del código de autorización ya que este método está optimizado para clientes implementados en navegador en lenguajes de *scripting* tipo *JavaScript*. En el flujo implícito en

vez de devolver al cliente un código de autorización se le devuelve directamente un *token* de acceso. El tipo de concesión de acceso es implícito puesto que el servidor de autorización no autentica al cliente, pero en algunos casos la identidad del cliente se puede realizar a través de la URI de redirección usada.

- **Credenciales del propietario de los recursos:** Las credenciales del propietario de los recursos pueden usarse también como forma de concesión de un *token* de acceso. Estas credenciales deben usarse sólo cuando exista gran confianza entre el cliente y el propietario y cuando otros métodos de concesión no están disponibles.
- **Credenciales del cliente:** Este tipo de concesión con credenciales del cliente se usa normalmente cuando el alcance de la autorización se limita solamente a los recursos bajo el control del mismo cliente, en otras palabras cuando el cliente también es el propietario de los recursos.

Para nuestro caso de uso nos hemos decantado por el tipo de concesión de código de autorización ya que es el que dispone de más beneficios en términos de seguridad, como la transmisión del *token* de acceso directamente al cliente sin pasar por el propietario.

En la figura 5.2 se ve reflejado el proceso que sigue un cliente a la hora de solicitar autorización para acceder a recursos protegidos de la *API REST*. En primer lugar el usuario realiza un proceso de *login* clásico con este, el servidor de autorización autentica al usuario y si es correcto le devuelve un código de autorización. Después el cliente utiliza este código para solicitar un *token* de acceso que siéndole concedido puede utilizar para realizar peticiones al servidor de recursos, en nuestro caso la *API REST*.

Como hemos comentado a lo largo de esta sección, el fin del proceso de autorización es la obtención por parte del cliente de un *token* de acceso con el que puede acceder a los recursos. Este *token* dispone de algunos atributos que lo complementan con el fin de añadir mayor seguridad al proceso de acceso a recursos. Estos atributos son la vida del *token* y el *refresh_token*. La vida útil representa la duración del *token*, es decir durante cuánto tiempo será válido, y por lo tanto se permitirá acceder a recursos con este. La finalidad de este atributo es que si por cualquier razón el *token* acaba en las manos equivocadas no disponga de acceso durante mucho tiempo a recursos protegidos. Es por ello que el valor en tiempo de vida de los *tokens* debe ser corto en la medida de lo posible para mitigar este posible inconveniente. Sin embargo esto propicia que haya que renovar los *tokens* con más frecuencia. En cuanto al *refresh_token* es otro *token* que sirve para renovar el *token* de acceso actual. Este también dispone de vida útil, la cual suele ser bastante más amplia que la del *token* de acceso con el fin de dar un mayor margen de renovación del mismo que caduca frecuentemente y no tener que realizar el proceso de autorización previo en caso de caducidad de ambos *tokens*. En la figura 5.3 se puede observar en forma de diagrama de flujo el proceso de refresco del *token* de acceso.

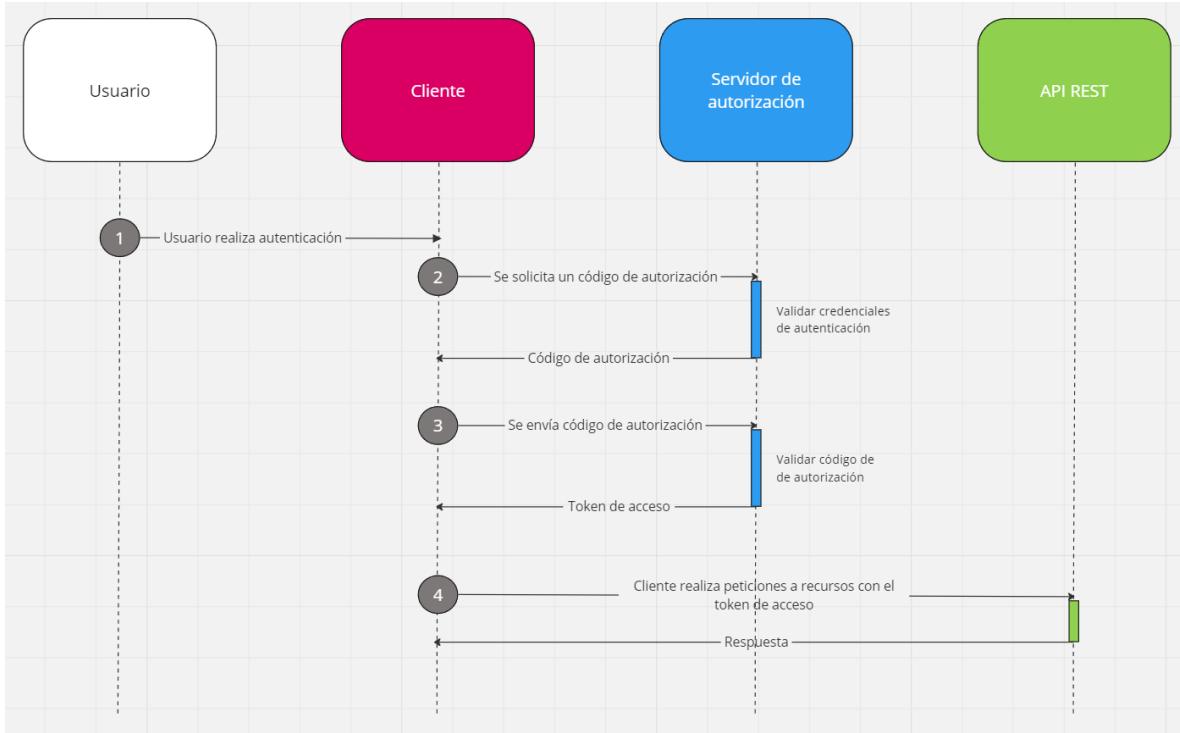


Figura 5.2: Flujo con código de autorización

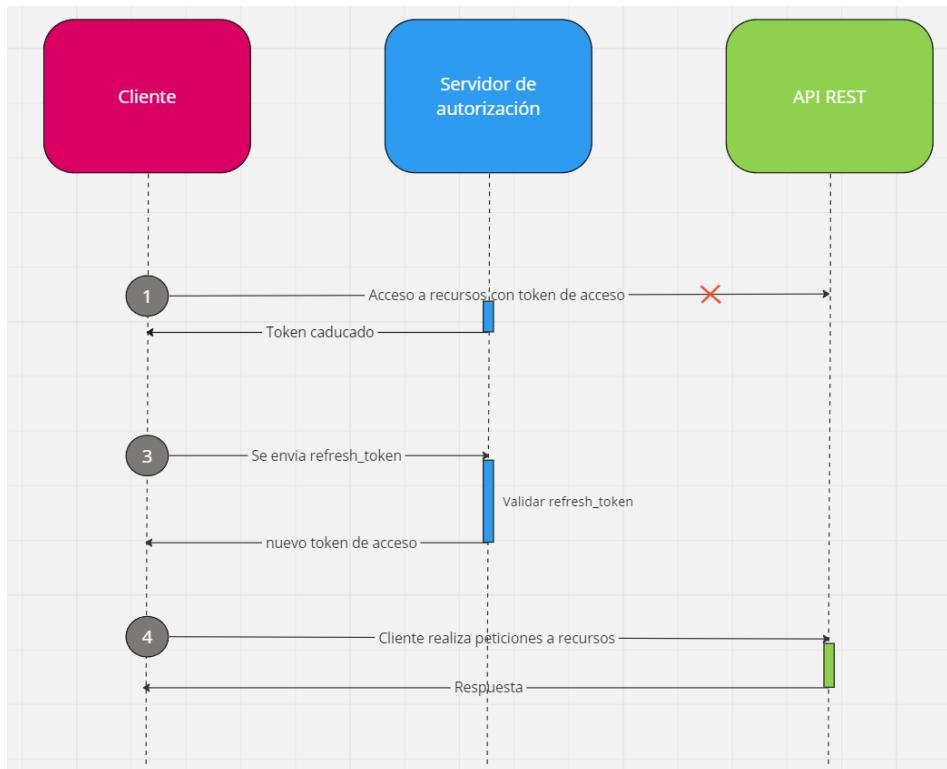


Figura 5.3: Flujo de refresco del token de acceso

Implementación del estándar *OAuth2.0* en *Node.js*

Para implementar el estándar *OAuth* en un servicio existen diversas aproximaciones. Una de las más comunes es delegar este proceso en un proveedor externo al servidor de recursos, siendo estos entidades reconocidas y fiables en cuanto a seguridad. De esta manera son estos proveedores los que autorizan el acceso a tus recursos. Un ejemplo de estos proveedores es *Google* con su famosa opción que se puede encontrar en gran cantidad de servicios “*Iniciar sesión con Google*”. Con servicios como estos podemos configurar el acceso a los recursos de la aplicación mediante el estándar *OAuth* gestionado por dichos proveedores. Sin embargo nosotros no hemos optado por esta manera ya que queríamos desarrollar una aplicación sobre la que el usuario gestor de sistema pudiera tener control completo sobre este, incluido el proceso de autorización. Es por esta razón que hemos realizado una implementación personalizada de este estándar y sobre el que tenemos control absoluto de su gestión.

La implementación del servidor de autorización ha sido realizada en el mismo entorno que la *API REST* y por ello con las mismas tecnologías, es decir *Node.js* en cuanto a entorno y *PostgreSQL* para el almacenamiento de los datos necesarios para el estándar, como los *tokens* entre otros. Esto ha sido posible gracias al paquete *node-oauth2-server* y su *wrapper* para el framework *express*. Este paquete facilita la implementación de un servidor de autorización *OAuth2.0* en *Node.js*. Para ello proporciona una interfaz con las distintas operaciones del estándar donde se puede implementar la funcionalidad de éstas en base a las necesidades propias o arquitectura. Este paquete permite implementar los distintos tipos de concesión de autorización explicados en el previo apartado, habiendo implementado nosotros el ya mencionado proceso de código de autorización.

Para la implementación del flujo ha sido necesario crear varias tablas en la base de datos para almacenar la información necesaria, por ejemplo códigos de autorización o *tokens* de acceso entre otros. En concreto son tres tablas, una para almacenar la información de los clientes de *OAuth*, otra para almacenar los códigos de autorización y por último una para almacenar los *tokens*. Estas son respectivamente *oauth_clients*, *oauth_authcode* y *oauth_tokens*. En la sección 4.3 se definen con detalle dichas tablas.

Como hemos mencionado, el paquete *oauth2-server* proporciona la interfaz de operaciones de *OAuth* que hemos utilizado para definir nuestra propia implementación, que consiste principalmente en la introducción de los datos generados por el paquete en nuestra base de datos, ya que todas las demás gestiones las realiza el propio paquete como por ejemplo la generación y validación de los *tokens* o el tratamiento de las *requests* y la *responses*. Para la implementación del tipo de concesión de código de autorización es necesario incluir dos *endpoints*, uno para realizar el proceso de obtener un código de autorización y otro para el proceso de obtener un *token*, que además también sirve para el flujo de renovación. En la figura 5.4 podemos observar el flujo explicado previamente y representado en la figura 5.2 pero mostrando los distintos *endpoints* implementados.

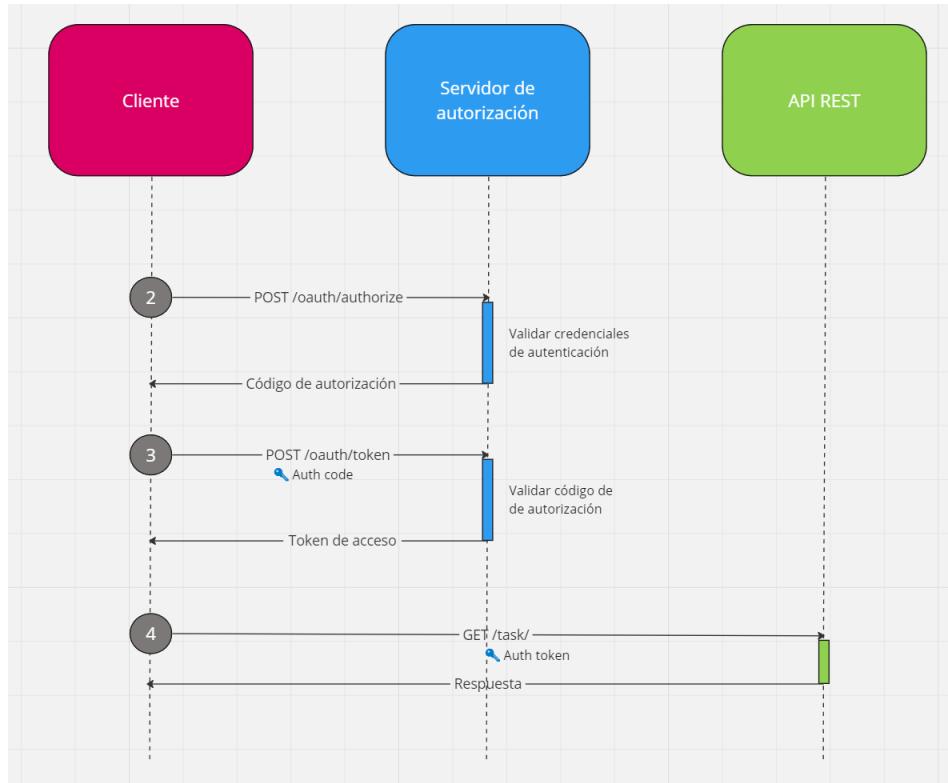


Figura 5.4: Flujo de autorización implementado

Configuración de clientes con OAuth

Para que pueda realizarse el proceso de autorización de clientes dentro de la aplicación es necesario realizar un paso previo de configuración, ya que el servidor de autorización debe tener identificados a los clientes para poder realizar el proceso. Cada cliente tiene un identificador llamado *client_id* y además para poder validar su autenticidad a la hora de solicitar autorización el cliente también dispone de un secreto (*client_secret*), es decir una clave que debe de proporcionar al servidor para el proceso de solicitud de *tokens*. De esta manera los *tokens* y los códigos de autorización están asociados a un cliente.

Por esta razón, para que un cliente pueda proceder a solicitar al servidor de autorización acceso a los recursos de la *API*, es necesario previamente que éste sea registrado en la base de datos de clientes de *OAuth* para próximamente poder asociarle códigos de autorización y *tokens* de acceso. Este proceso se realiza cuando desde la aplicación cliente se procede a configurar un servidor, paso que debe hacerse al menos una vez si se desea utilizar la aplicación ya que de inicio no hay ningún servidor configurado. Desde el menú de configuración de servidores de la aplicación que se describe en el capítulo 8, una vez introducimos la *URL* donde se encuentra el servidor con la parte *backend* instalada y procedemos a realizar la conexión, éste la guardará en el cliente para próximamente poder iniciar sesión y utilizar la aplicación con normalidad. Es en este proceso cuando se realiza la identificación del cliente por parte del servidor de autorización, que genera un secreto que le devolverá al cliente y que este guardará en el dispositivo para poder realizar el proceso de autorización. De la misma manera el servidor registra en la base de datos la información del cliente para poder validarla en futuros procesos de autorización.

Capítulo 6

Diseño e Implementación del Frontend

En este capítulo ahondaremos en varias de las técnicas de diseño de software que hemos usado para bocetar el conjunto de interfaces de usuario que conforman *SwiftDo*. Además, explicaremos las herramientas que hemos utilizado para prototipar e implementar el *frontend* de la app. Para que una aplicación llegue a ser utilizada de manera cotidiana, deberá cumplir con los criterios y objetivos para los que fue concebida, así como disponer de un buen diseño que lo complemente.

— ¿Qué hace que una aplicación esté bien diseñada? — Para responder a esta pregunta hemos explorado algunos libros y/o asignaturas cursadas en la titulación y, después de una exhaustiva búsqueda, hemos encontrado [8] de *D. Norman*, uno de los libros más importantes que relaciona el diseño de producto y la cognición humana. A continuación proporcionaremos de manera desglosada las técnicas en las que nos apoyamos.

6.1 Percepción del funcionamiento del sistema

Para empezar a diseñar la app, investigamos, no sólo cómo funciona el método *GTD*, explicado en el capítulo 2, sino también cómo trasladar esa experiencia fielmente al software. La percepción de la filosofía *GTD* puede variar según cada usuario, sin embargo, la esencia principal de esta debe ser la misma. Para ello debemos meditar qué tipo de elementos deben estar o no presentes en la app. De esta forma conseguimos que coincidan el **modelo mental**, es decir, cómo el usuario piensa que funciona la aplicación, y el **modelo tecnológico**, que representa el funcionamiento interno.

Al haber reunido los conceptos fundamentales en el mapa mental de la figura ?? conseguimos reducir la fricción cognitiva, es decir, reducir la fatiga visual mediante la simplificación de conceptos y hacer que su uso sea intuitivo y eficiente.

6.2 Metáforas, Expresiones y Conceptos de diseño

Con la finalidad de hacer que el usuario se vea familiarizado con los conceptos empleados en la app, nos hemos visto envueltos en un conjunto de metáforas y expresiones.

En primer lugar, usamos metáforas para asociar cada “acción” del método *GTD* con un ícono que exprese fielmente lo que realiza esa categoría. Ejemplo de ello, pueden ser la “entrada”, representada por una bandeja de entrada de mensajes, la bandera roja indicando la urgencia o el símbolo de progreso de cada proyecto, tal y como se puede observar en las figura 6.1:

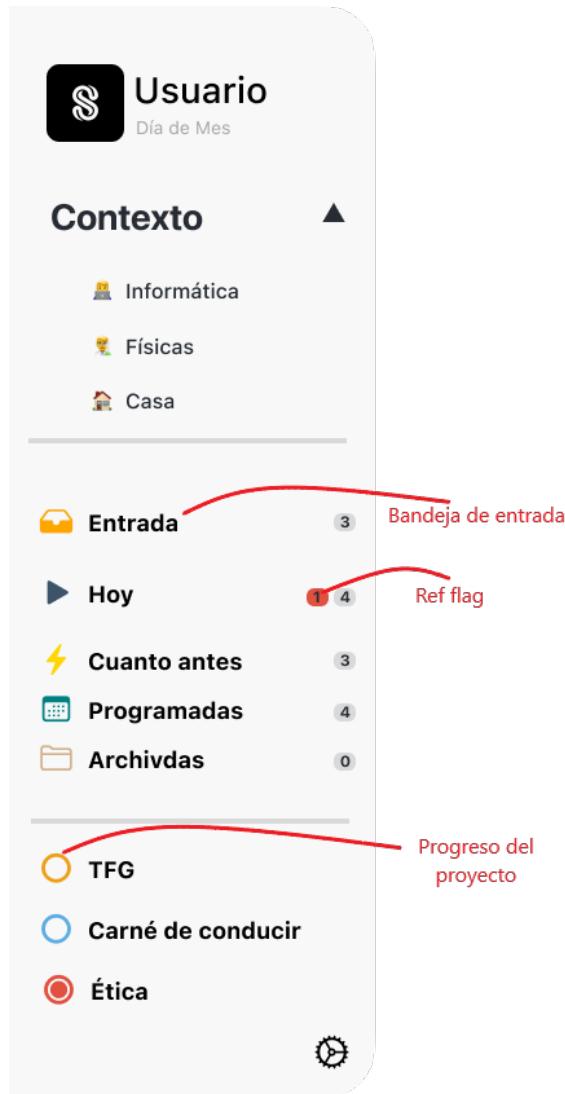


Figura 6.1: Menú lateral

Por otro lado, tenemos las expresiones. Estas son un concepto fundamental, ya que ayudan a definir cómo interactúa el usuario con la aplicación, haciéndola más eficiente y

consistente. Estas las podemos ver en el menú lateral, con las secciones 6.2 de “Entrada”, “Hoy”, “Cuanto antes”, “Programadas”, etc.

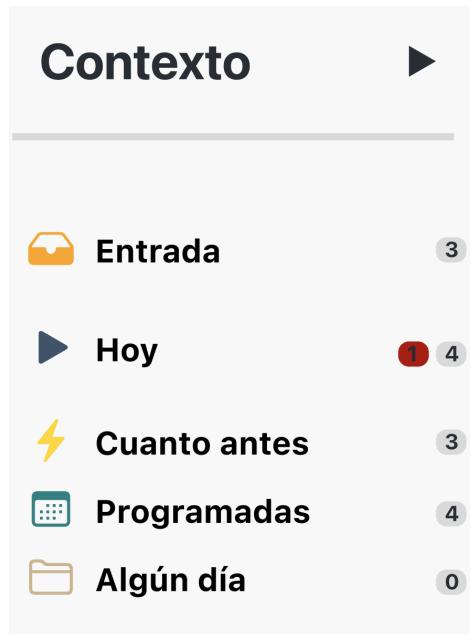


Figura 6.2: barra lateral

Finalmente, hemos optado por un diseño plano y elegante, con pocas trazas de esqueuomorfismo, evitando sobrecargar con detalles superfluos la interfaz. Para ello nos hemos basado en el concepto de *affordance*, haciendo que cada componente que se encuentre en la app sea autoexplicativo, como por ejemplo el botón de añadir tarea y/o proyecto que se muestra en la figura 6.3. Para explorar con mayor detalle estos aspectos, nos hemos basado en varios principios de diseño que describiremos a continuación.

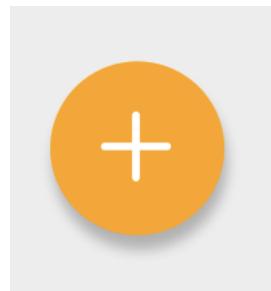


Figura 6.3: Botón para añadir tarea/proyecto

6.2.1 Proximidad y Consistencia

Estos principios se encuentran relacionados con la coherencia con la que diseñamos la interfaz. El primero de estos principios, **proximidad**, indica que todos los elementos

que se encuentren relacionados entre sí deben agruparse visualmente, familiarizándolo y simplificando el proceso de aprendizaje del usuario con los conceptos tratados, como se observa en la figura 6.2

En segundo lugar, detallamos el principio de **consistencia**, que ha sido fundamental tanto para el diseño como para la implementación. Esto ha sido de gran utilidad para la optimización del código, diseño, que se muestra en la figura 6.4, y aprendizaje de la interfaz por parte del usuario final.

6.2.2 Visibilidad

D. Norman en su libro [8] explica que todo componente debe proporcionar una representación clara, tanto de su estado como de las funciones que estas desempeñan. Tanto es así, que explica que cuanto más visible sea un objeto, mayor será la interacción que tendrá el usuario con él.

Por esto, hemos usado un diseño plano y minimalista, donde los detalles que tienen alta relevancia y que se relacionan con el método GTD, pasan a un primer plano, como son las “acciones”, que se representa en la figura 6.4, mientras los demás ocupan un segundo plano.

Por último, es conveniente gestionar el estado visible de los componentes de la app, es decir, que el usuario pueda observar claramente el estado actual del sistema. Un ejemplo de ello son los detalles en el menú lateral, como el nombre de usuario, que le informa que ha iniciado sesión en la app, así como la fecha actual, que puede ser de especial relevancia para la creación de tareas (véase la figura 6.5). Por otro lado, las etiquetas y el contexto asociado a cada tarea en la pantalla de “detalles”, que se muestra en la figura 6.6, dirigen la atención a lo que realmente importa.

6.3 Prototipos e Interfaces

Durante el diseño inicial de la interfaz de la aplicación, exploramos diferentes herramientas de diseño para poder elaborar varios de los *mockups* que componen la app. Entre ellas destacamos **Balsamiq**¹ y **Figma**². Sin embargo, nos decantamos por Figma, ya que es una herramienta más versátil y con mayores opciones de personalización, abarcando desde modelos de baja a alta fidelidad.

Figma es una plataforma de diseño colaborativo que permite desde la creación de prototipos interactivos hasta interfaces de usuario con alto nivel de detalle. Además de permitir diseñar fácilmente las interfaces, habilita a otros miembros del grupo a colaborar y compartir el contenido que se está diseñando, lo que ha ayudado, en gran parte, a la pronta implementación del *frontend* del proyecto.

¹Balsamiq

²Figma

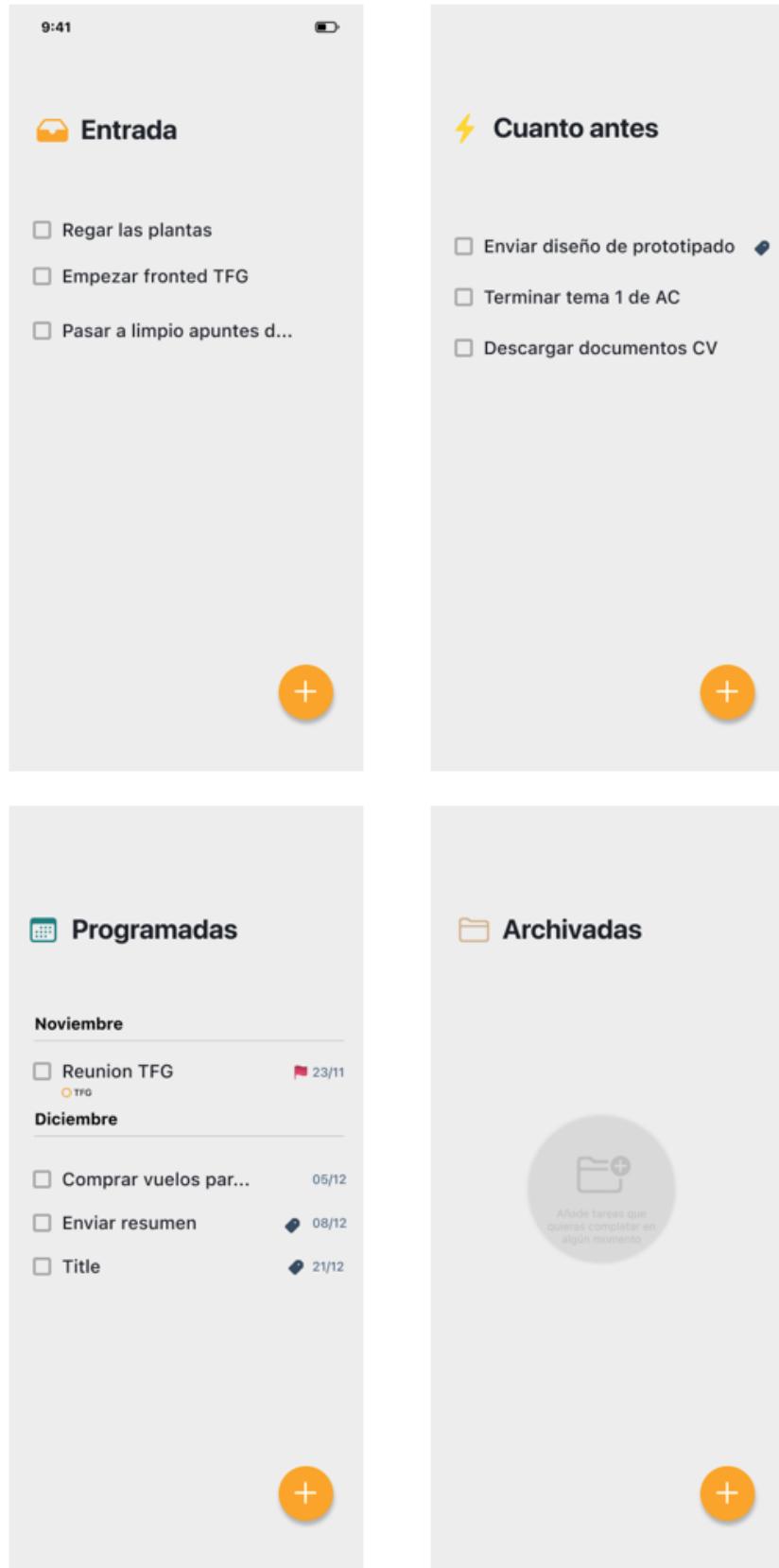


Figura 6.4: Prototipo de Bandeja de entrada, Cuanto antes, Programadas y Archivadas
51



Figura 6.5: Nombre de usuario y fecha

Asimismo, usamos gran parte de las opciones que ofrece para probar varios de los principios de diseño que hemos logrado implementar en la aplicación final, además de las fuentes y colores utilizados en la misma.

Finalmente, una vez creados todos los prototipos junto a sus componentes, probamos el flujo de diseño que habíamos creado inicialmente, mediante el uso de la aplicación móvil, llegando a emular fielmente el comportamiento final de la app.

A continuación presentamos todos las interfaces que componen SwiftDo, junto a una breve explicación que las relaciona con el método GTD

En primer lugar tenemos las interfaces relacionadas con el registro e inicio de sesión:

En segundo lugar tenemos las tareas relacionadas con la gestión del flujo de *GTD* y sus acciones en dispositivos móviles 6.4, además del prototipo de escritorio:

En adición a las categorías de *GTD* mencionadas en el capítulo 2, hemos querido añadir una nueva categoría “**Hoy**” 8.8 que actúe como resumen diario inteligente. Este detectará cuyas tareas pertenecen al día de “hoy”, tareas atrasadas cuya finalización ha expirado y por último, en caso de tener pocas tareas para realizar en el día, se irán adjuntando, otras pertenecientes a la categoría “**cuanto antes**”.

En tercer lugar tenemos interfaces relacionadas con el menú o barra lateral y ajustes tanto en formato escritorio como en dispositivos móviles:

Por último tenemos interfaces más detalladas como son la de añadir tarea/proyecto, interfaz de añadir detalles a la tarea, pudiendo insertar markdown texto en formato markdown y distintos modales para la creación o edición de tareas.

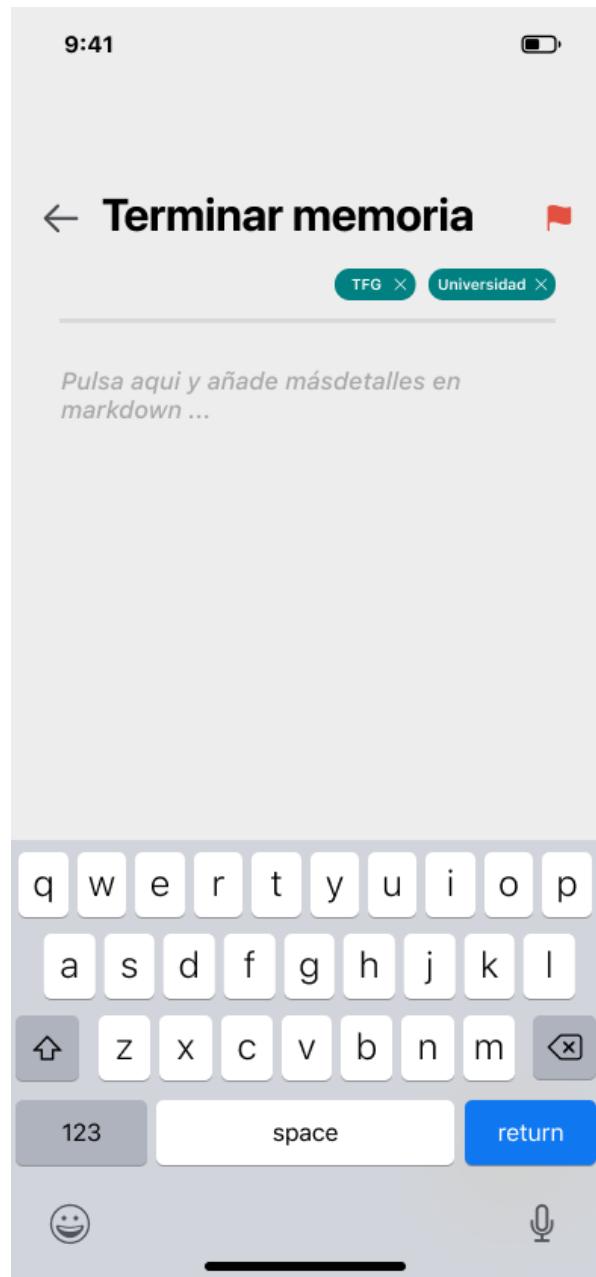


Figura 6.6: Sección para ver y editar aspectos más detallados de la tarea (incluso observaciones en Markdown)



Figura 6.7: Inicio de sesión y Registro

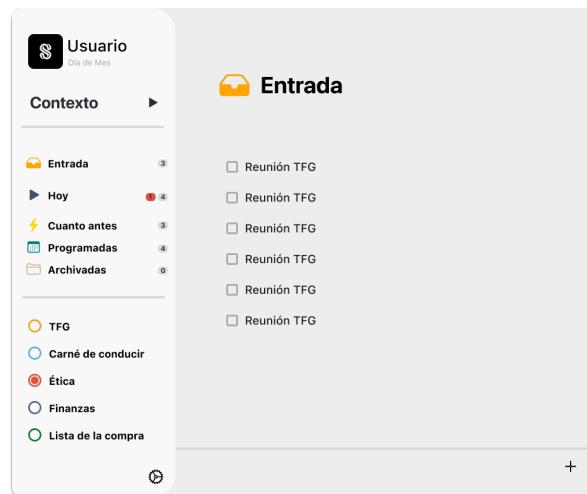


Figura 6.8: Escritorio

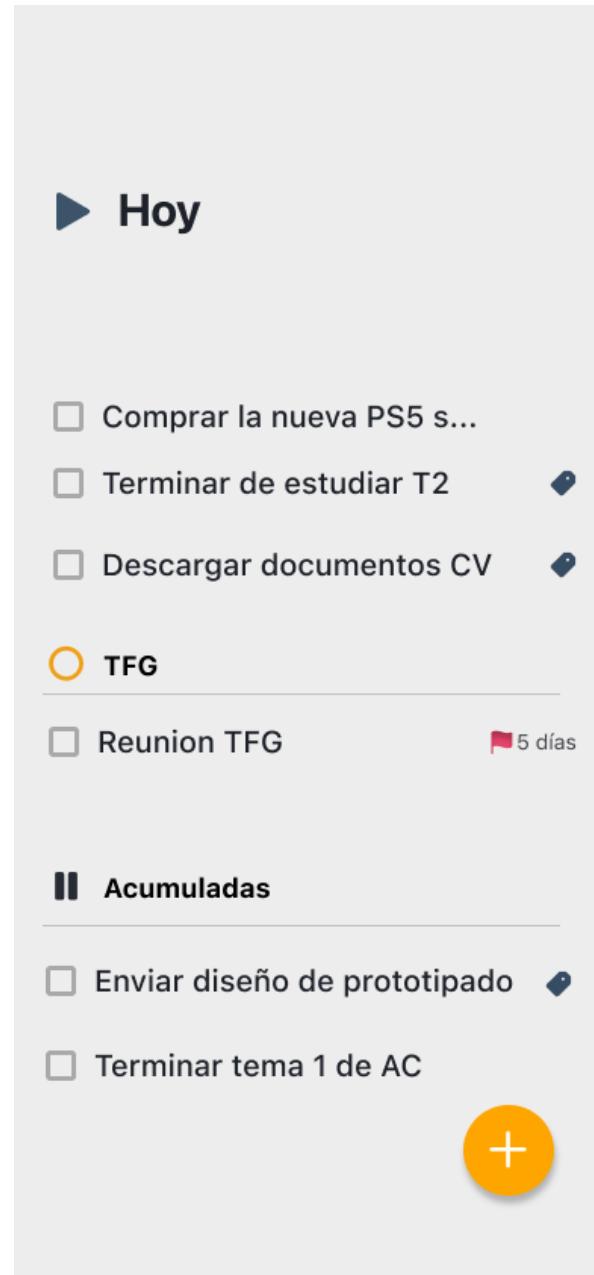


Figura 6.9: Pantalla de sección “hoy”



Figura 6.10: Ajustes - Móvil

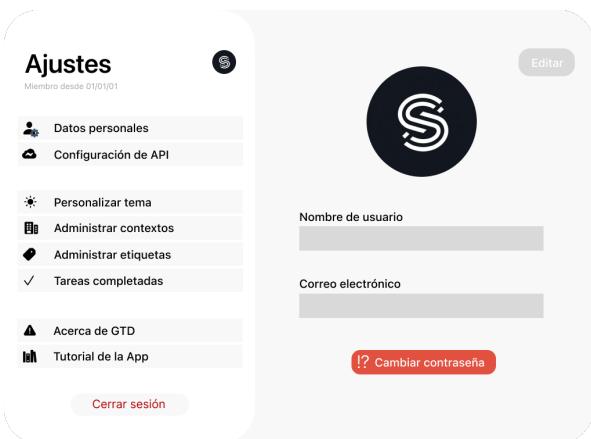


Figura 6.11: Ajustes - Escritorio

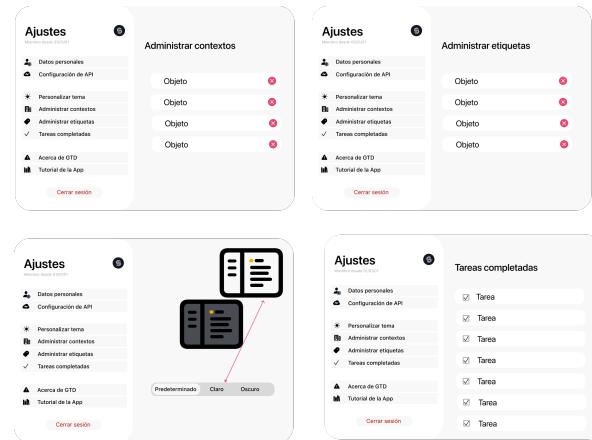


Figura 6.12: Opciones de ajustes



Figura 6.13: Editor de tareas

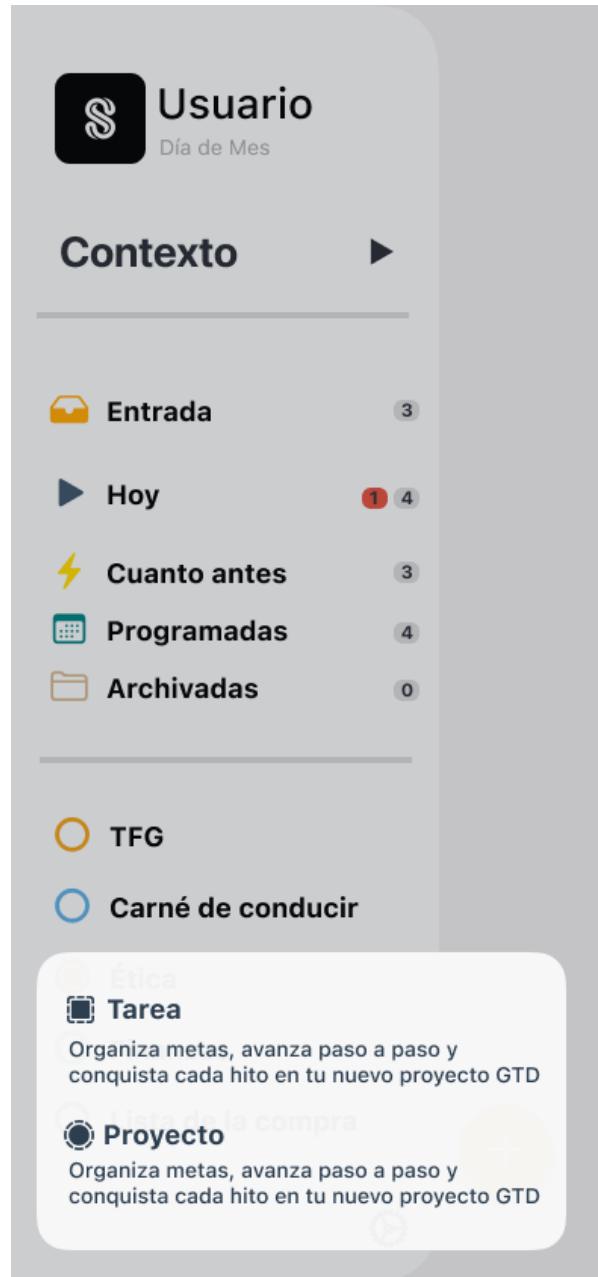


Figura 6.14: Añadir tareas o proyectos

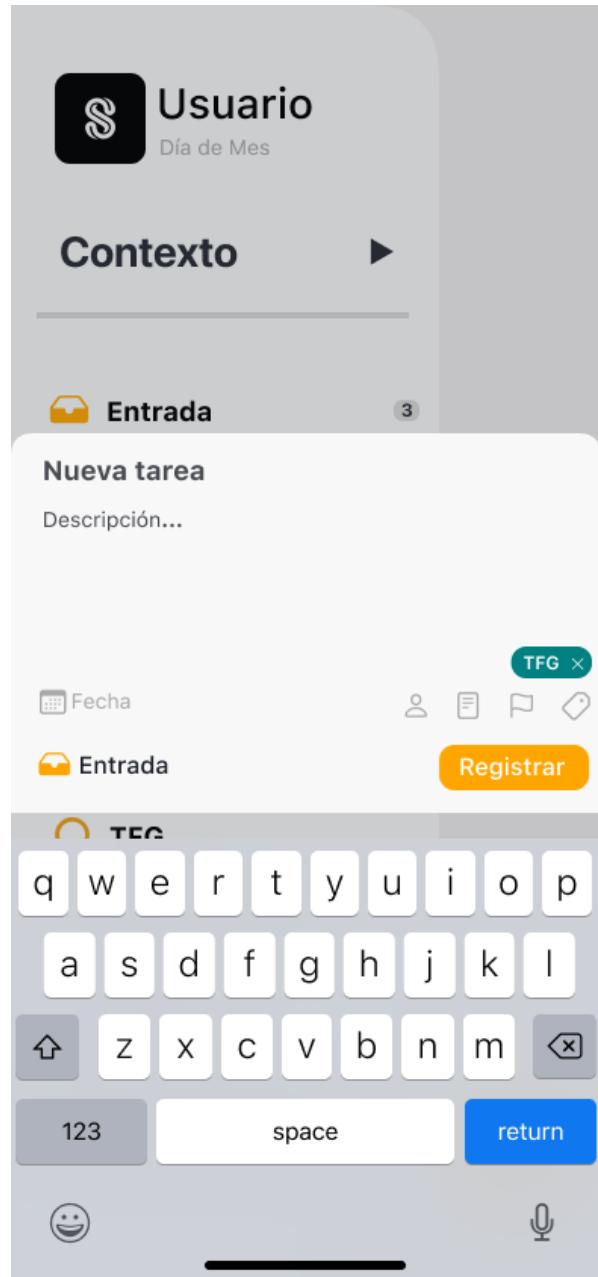


Figura 6.15: Menú para añadir tareas

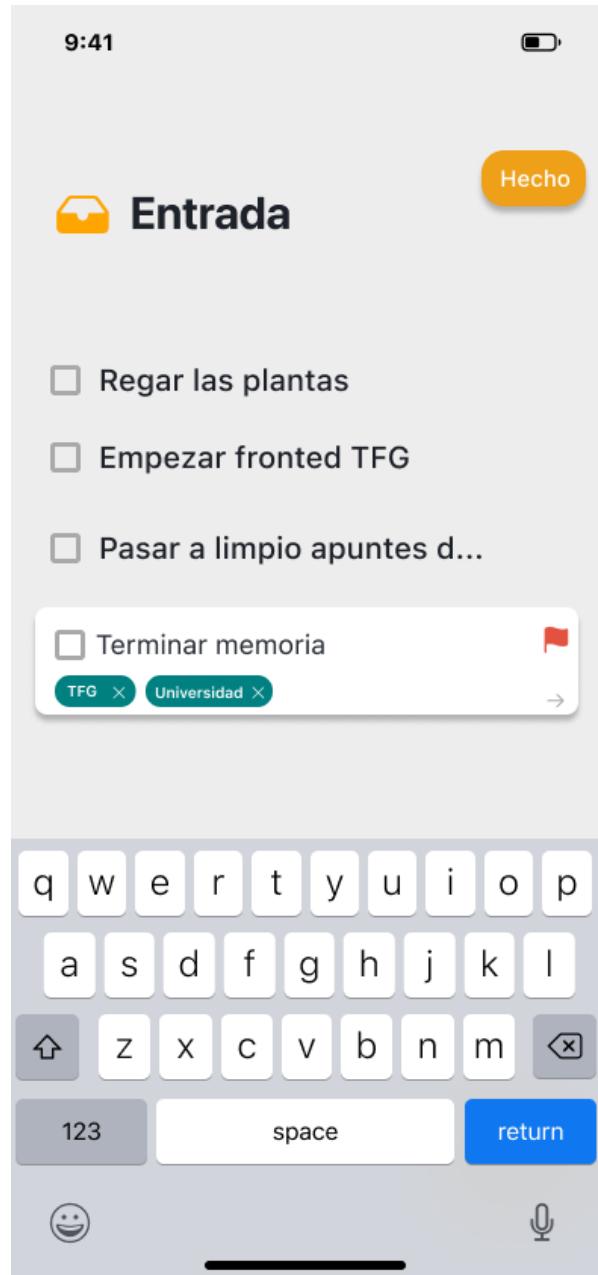


Figura 6.16: Visualización de tarea

6.4 Implementación: ¿Qué es React Native?

Para implementar el *frontend* de la app junto a las interfaces de usuario anteriormente descritas, necesitábamos hacer uso de un framework que pudiera adaptarse no solo a interfaces táctiles sino también a interfaces de escritorio. Para lograrlo hemos hecho uso de *React Native* y *Expo*.

Expo[9] es una plataforma que agiliza el desarrollo y despliegue de aplicaciones en *React Native* sin la necesidad de lidiar con configuraciones complejas.

Por otro lado, *React Native* es un framework desarrollado por Facebook que tiene por lema “*Learn once, write anywhere*”. Mediante el uso de *React* y *JavaScript*, permite la creación de aplicaciones multiplataforma, ya sea en *MacOS*, *iOS*, *Windows* o *Android*.

Este framework no solo destaca por la capacidad de reutilización de código sino por ofrecer un comportamiento nativo de los componentes utilizados, es decir, permitir un rendimiento similar al de las aplicaciones desarrolladas con lenguajes nativos (*Swift* para *iOS* o *Java/Kotlin* para *Android*).

Por último, destaca la gran comunidad que tiene, ya que además de la documentación que podemos consultar en la página web oficial de *React Native*, disponemos de bibliotecas externas que se detallan en otras páginas web que son útiles para la implementación de otros componentes que no se encuentran en las fuentes oficiales. Las bibliotecas externas que hemos empleado son las siguientes:

- **React Navigation** [10]: Esta librería permite definir y enrutar las diferentes pantallas de la aplicación para poder navegar entre ellas. Ha sido de gran importancia en la aplicación ya que la navegación es un aspecto clave en cualquier aplicación. Además este paquete es muy sencillo de entender y usar gracias a su buena documentación.
- **Drawer Navigation** [11]: Muestra un cajón de navegación en el costado de la pantalla que se puede abrir y cerrar mediante gestos.
- **Axios** [12]: Biblioteca de *JavaScript* que se utiliza para hacer solicitudes *HTTP* desde el navegador. Este paquete facilita la gestión y el envío de *requests* y *responses*, esto nos ha servido para gestionar los *headers* de autorización.
- **React Native Calendars** [13]: Un componente de calendario declarativo multiplataforma *React Native* para *iOS* y *Android*. Se ha utilizado para implementar la pantalla de Programadas.
- **React Native Async Storage** [14]: Este paquete ha sido de gran importancia en el desarrollo ya que permite guardar información en el dispositivo. Se ha utilizado para almacenar la información de las sesiones iniciadas como los *tokens* de acceso o la configuración aplicada como el tema seleccionado o los servidores configurados.
- **Markdown Display** [15]: Renderizador de *Markdown*.

-
- **Modern Datepicker** [16]: Se ha utilizado este paquete para implementar el selector de fecha para una tarea.
 - **Swipeable** [17]: Componente que permite implementar filas deslizables o interacción similar. Representa a sus hijos dentro de un contenedor que permite el deslizamiento horizontal hacia la izquierda y hacia la derecha.
 - **Wheel Color Picker** [18]: Componente seleccionador de colores. Se ha utilizado en el modal de crear proyecto para seleccionar el color del mismo.

Capítulo 7

Integración con agentes conversacionales

Desde el inicio del proyecto, nos propusimos habilitar la interacción mediante comandos de voz para mejorar la experiencia del usuario, para ello en este capítulo, profundizaremos en el proceso de integración de nuestra aplicación *SwiftDo* con la plataforma de voz de *Alexa* mediante el desarrollo de una *skill*. Para contextualizar, una *skill* en el ecosistema de *Alexa* es una capacidad o funcionalidad específica que permite a los usuarios interactuar con dispositivos habilitados para *Alexa*, utilizando comandos de voz.

Discutiremos los objetivos que persigue esta integración con *Alexa*, que incluyen proporcionar a los usuarios una forma intuitiva y práctica de administrar sus tareas diarias, así como mejorar la accesibilidad de *SwiftDo* para aquellos que prefieren la interacción por voz. Además, exploraremos la funcionalidad mínima que esperamos ofrecer a los usuarios a través de la *skill* de *Alexa*, centrándonos en la capacidad de añadir una tarea como funcionalidad básica de gestión de tareas que será compatible con esta integración.

7.1 Configuración de la skill de Alexa

El primer paso crucial en nuestra aplicación fue obtener acceso al *Amazon Developer Console* para crear una *skill* de *Alexa*. Esta *skill* fue meticulosamente configurada para permitir a los usuarios añadir tareas a nuestra aplicación *SwiftDo* utilizando comandos de voz de manera intuitiva y eficiente. Durante la configuración, definimos cuidadosamente los tipos de interacciones que la *skill* debería admitir, como la capacidad de añadir un título obligatorio para la tarea y la opción de proporcionar información adicional, como una descripción, fecha de vencimiento o nivel de importancia.

7.2 Vinculación de la cuenta del usuario

Para habilitar la interacción entre la *skill* de *Alexa* y nuestra aplicación, implementamos un sólido flujo de autorización *Oauth* realizado también en *Amazon Developer Console*.

Optamos por el método de autenticación de código de autorización (*Auth Code Grant*) para permitir que los usuarios vinculen sus cuentas de manera segura. Sin embargo, surgió un desafío significativo: para garantizar la seguridad de la transacción, *Amazon* requiere que las aplicaciones estén protegidas mediante *HTTPS*. Si bien la implementación de *HTTPS* estaba pendiente, este requisito nos incentivó a adelantar su incorporación a nuestra plataforma web, por lo tanto incorporamos *HTTPS* a nuestra plataforma web y permitir así la vinculación de las cuentas de usuario.

7.3 Implementación del flujo de autorización Oauth

Al configurar la *skill* de *Alexa*, seguimos un diagrama de secuencia detallado que guía al usuario a través del proceso de vinculación de la cuenta (véase la figura 7.1). Todo este flujo de autorización *Oauth* ya se prueba en la propia aplicación de *Alexa*. Para realizar pruebas y ajustes, es necesario iniciar sesión con el mismo usuario que en *Amazon Developer Console*, donde se define la *skill*.

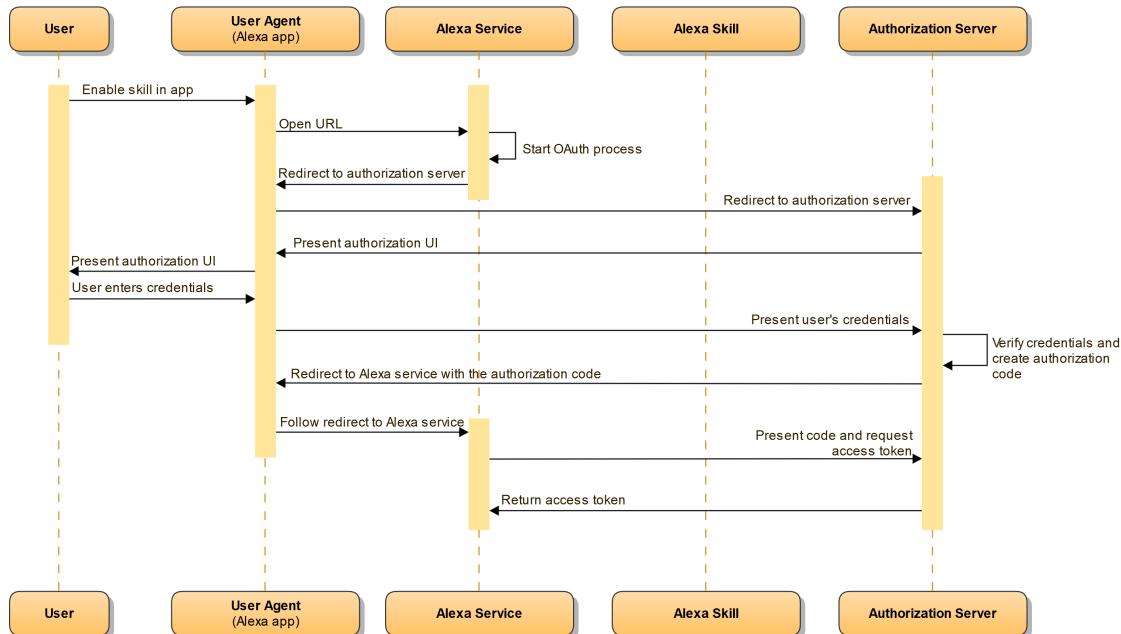


Figura 7.1: Diagrama de secuencia del proceso de vinculación de la cuenta. Fuente: [19]

El proceso comienza cuando el usuario habilita la *skill* en su dispositivo, la cual informa al usuario que necesita vincular su cuenta a la aplicación *SwiftDo* para que funcione.

Posteriormente, se redirige al usuario a la página de autorización de nuestro servidor. Aquí, el usuario introduce sus credenciales de la cuenta de la aplicación *SwiftDo*. Nuestro

servidor verifica las credenciales proporcionadas por el usuario y, una vez verificadas, genera un código de autorización y un estado.

Este código de autorización y el estado son devueltos por nuestro servidor a la *skill* de Alexa. La *skill* intenta entonces obtener un *token* de acceso utilizando el código de autorización proporcionado. Si hay un error en el proceso, la *skill* muestra un mensaje de error al usuario, y en caso de éxito, la *skill* vincula la cuenta del usuario y muestra un mensaje de confirmación.

7.4 Guía para vincular Alexa con SwiftDo

Para vincular Alexa con SwiftDo, debemos seguir una serie de pasos sencillos. Primero, accedemos a la aplicación de “Amazon Alexa” en nuestro dispositivo móvil (véase la figura 7.2). Una vez dentro de la aplicación nos dirigimos a la sección “Más”, ubicada en la parte inferior derecha de la pantalla. Desde ahí, seleccionamos “Skills y juegos” en el menú desplegable. Esta sección permite explorar y habilitar diversas *skills* que expanden las capacidades de Alexa tal y como se muestra en la figura 7.3.



Figura 7.2: Página principal aplicación “Amazon Alexa”

A continuación, utilizamos la barra de búsqueda para encontrar la *skill* de la aplicación “SwiftDo” (véase la figura 7.4). Una vez encontrada la *skill*, presionamos en “Permitir su uso” para habilitar la *skill* en nuestra cuenta de Alexa.

El siguiente paso es crucial para la vinculación de cuentas. Nos dirigimos a la opción “Configuración” dentro de la *skill* de SwiftDo y seleccionamos “Vincular cuenta”. Esto nos redirige a una página, mostrada en la figura 7.5, donde debemos introducir nuestro

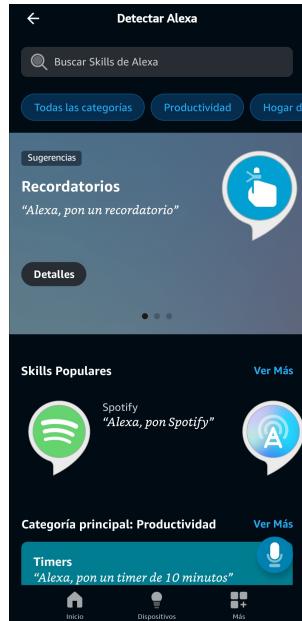


Figura 7.3: Página de “Skills y juegos” de la aplicación



Figura 7.4: Página de la Skill SwiftDo ya habilitada

“Correo electrónico” y “Contraseña” asociados a nuestra cuenta de *SwiftDo*. Una vez ingresados estos datos, presionamos “Vincular” para completar el proceso de autenticación.



Figura 7.5: Página redirigida para vinculación de Alexa con SwiftDo

Una vez vinculada la cuenta, es necesario activar la *skill* mediante el comando de voz: “Alexa, abre añadir tarea”. Este comando inicializa la *skill* y permite a *Alexa* la capacidad de añadir una nueva tarea a nuestra aplicación.

Para añadir una tarea, deberemos indicar el título de la tarea obligatoriamente. Por ejemplo, diciendo “Añade la tarea Prueba con Amazon”. Posteriormente, podemos añadir más información a la tarea, como una descripción diciendo “Añade de descripción Esto es una prueba con Amazon”. También podemos definir la importancia de la tarea con el comando “Si la tarea es importante” y establecer una fecha límite diciendo “Pon de fecha límite el 15 de junio”. Si no deseamos agregar más detalles podemos finalizar por ejemplo, con el comando “Eso es todo”.

Capítulo 8

Manual de usuario

A continuación mostraremos el manual de usuario de *SwiftDo* el cual mostrará de manera visual cómo podemos realizar las distintas acciones dentro de la app. Explicaremos como configurar el servidor, como iniciar sesión y registrarse, como filtrar dentro de la aplicación, como funcionan las pantallas de las distintas acciones que ofrece *SwiftDo* (inbox, hoy, cuanto antes, programadas, archivadas, proyectos), como utilizar la sidebar y los ajustes y, por último, cómo crear tareas y proyectos.

8.1 Configuración del servidor

Al iniciar la aplicación, el usuario se encontrará con una pantalla para configurar el servidor como podemos ver en la figura 8.1. Una vez damos al botón de configurar se abre otra pantalla en la cual debemos introducir el nombre del servidor y la url del mismo, pudiendo probar la conexión y, en el caso de que sea correcta, podamos conectarnos a dicho servidor. Una vez conectado, muestra una lista con los servidores configurados y permite crear nuevos o eliminar el deseado.

8.2 Inicio de sesión y registro

Una vez configurado y conectado el servidor, aparece una pantalla de inicio de sesión (ver figura 8.2), esta permite iniciar sesión introduciendo un correo y una contraseña o registrarse introduciendo un nombre de usuario, correo y contraseña en el caso de que no tengamos ninguna cuenta.

8.3 Inbox

Tras haber iniciado sesión, aparece la pantalla de *inbox*, la cual almacena las tareas sin organizar del usuario. Cómo podemos observar en la figura 8.3, en esta pantalla

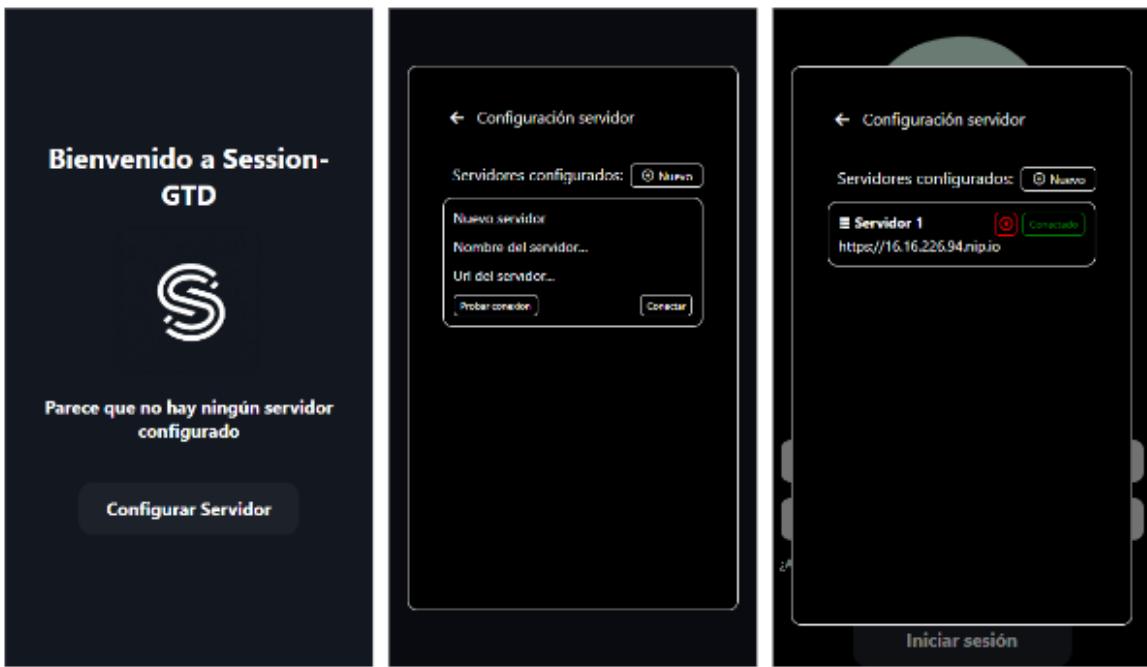


Figura 8.1: Configuración del servidor

aparecen varios elementos como el filtrado en la esquina superior derecha, la *sidebar* en la esquina superior izquierda, el botón de creación de tareas y proyectos en la esquina inferior derecha y las propias tareas en el centro.

8.4 Filtros

Presionando el botón de filtrado, nos encontramos con la pantalla que podemos observar en la figura 8.4, la cual nos permite filtrar las tareas por los contextos, proyectos y etiquetas existentes. Una vez seleccionados los filtros que deseé, el usuario podrá limpiarlos o aplicarlos.

8.5 Sidebar

Al presionar el botón de la *sidebar*, se despliega lo observado en la figura 8.5. La *sidebar* muestra el nombre de usuario, icono, la fecha actual, un botón de ajustes y un listado de los contextos, las distintas acciones que se pueden realizar: “entrada” o “inbox”, “hoy”, “cuanto antes”, “programadas” o “archivadas”. Por último se muestra un listado con todos los proyectos creados por el usuario.



Figura 8.2: Inicio de sesión y registro

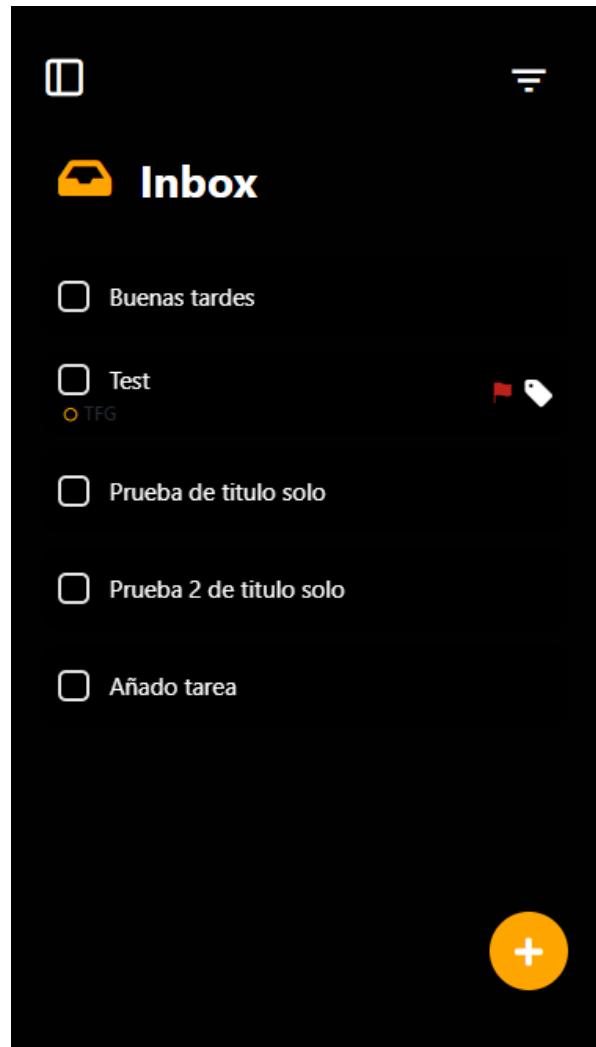


Figura 8.3: Inbox

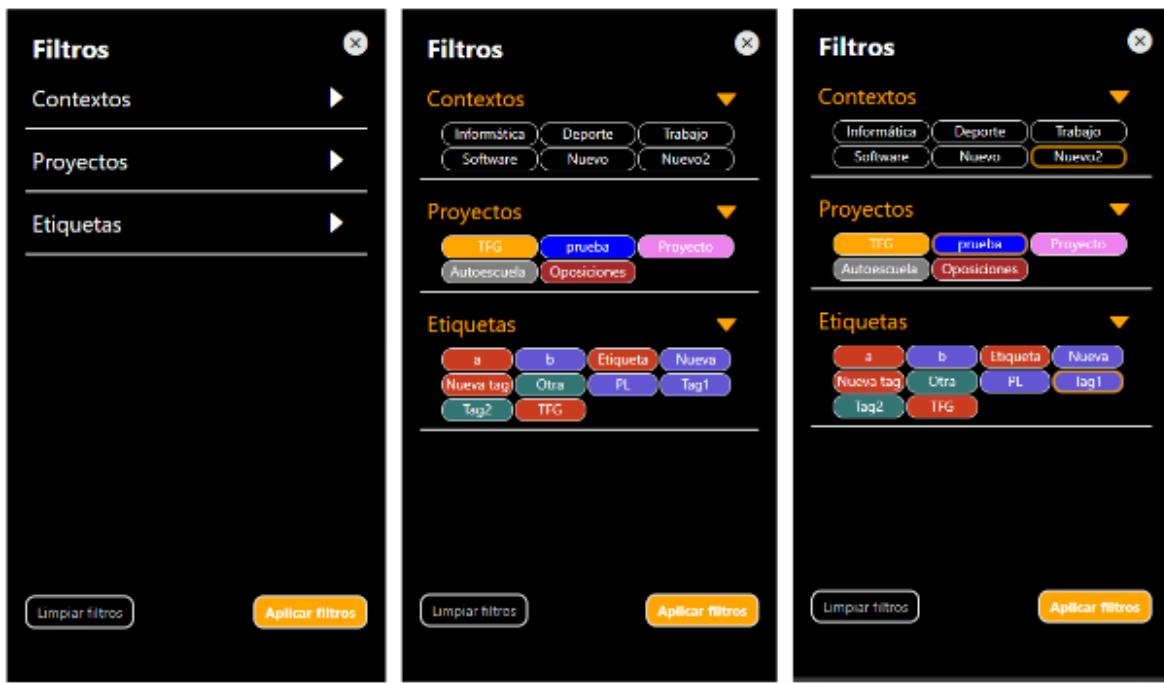


Figura 8.4: Filtros

8.6 Ajustes

Pinchando en el botón de ajustes, se abre el modal de la figura 8.6. Este modal permite actualizar los datos personales cambiando el nombre de usuario o correo, configurar el servidor (ver figura 8.1), personalizar el tema seleccionando modo claro u oscuro, eliminar contextos o etiquetas, ver una lista de las tareas completadas, conocer información acerca del GTD o ver un tutorial de la aplicación. Por último permite cerrar sesión.

8.7 Crear tarea y crear proyecto

Pulsando el botón naranja de la esquina inferior derecha aparece el modal para crear tarea que encontramos en la figura 8.7, el cual permite crear una tarea escribiendo el nombre en el texto Nueva Tarea, añadirle una descripción a la tarea, seleccionar una fecha en la que debe estar terminada, seleccionar un contexto pinchando el icono de usuario, marcar cómo importante presionando el icono de la bandera, añadirle una etiqueta y añadirla a un proyecto. Además, manteniendo presionado el botón naranja comentado anteriormente, permite elegir si crear una tarea desplegándose el mismo modal que acabamos de mencionar o crear un proyecto añadiendo un título de proyecto, una descripción y un color.

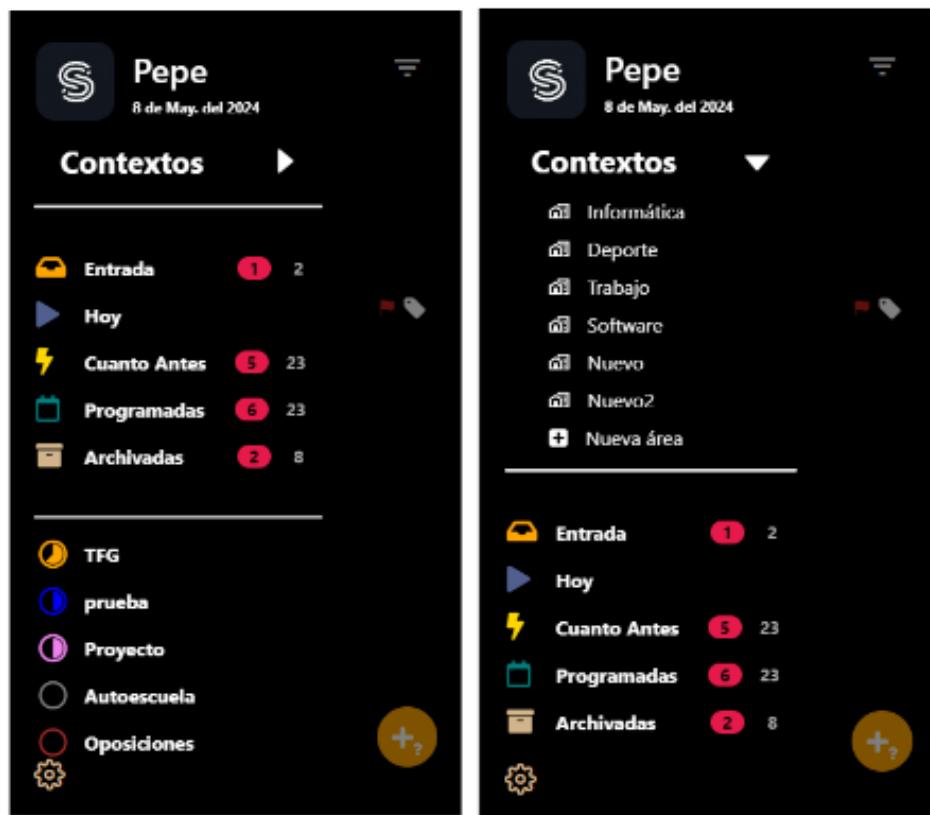


Figura 8.5: Sidebar

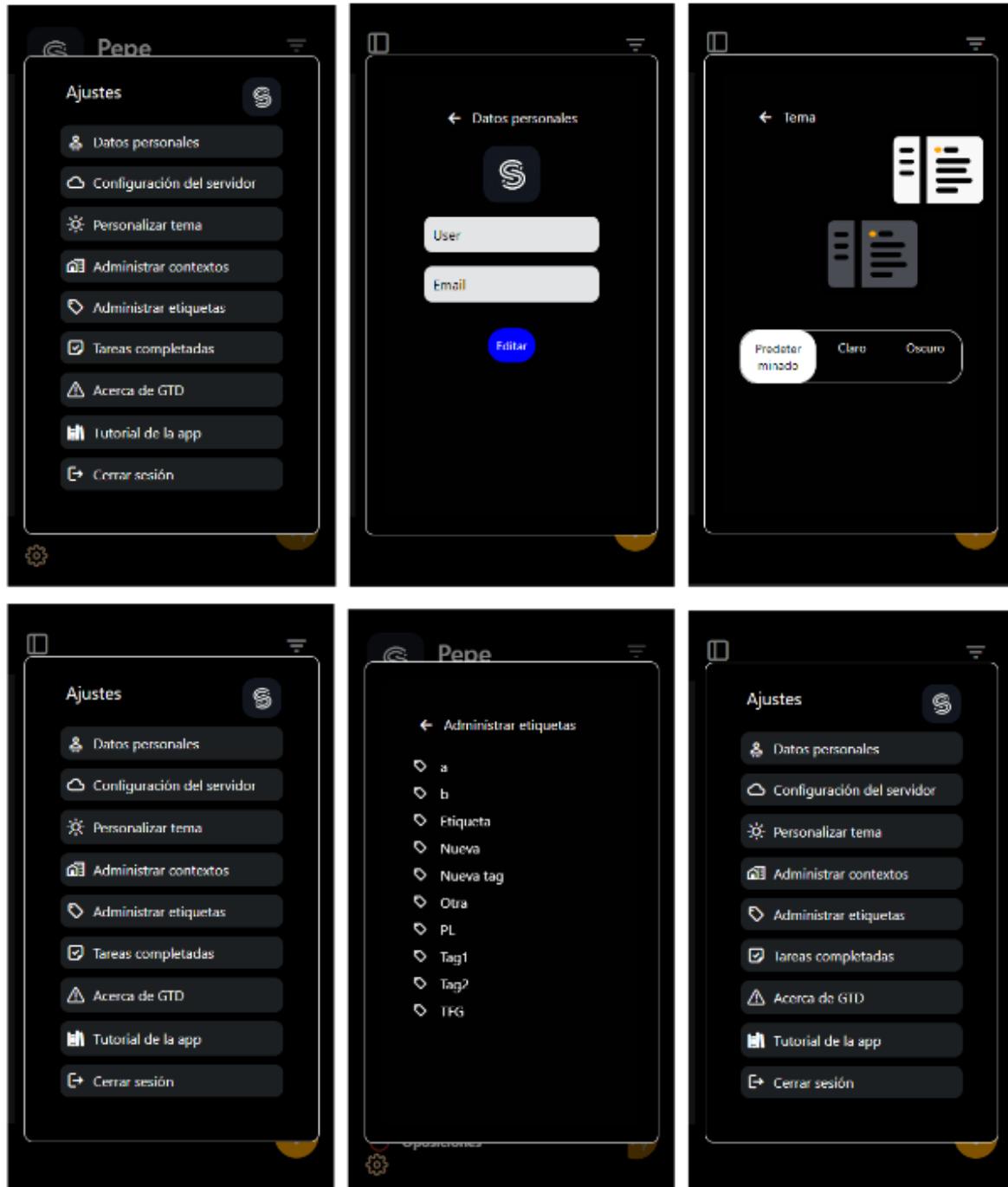


Figura 8.6: Ajustes

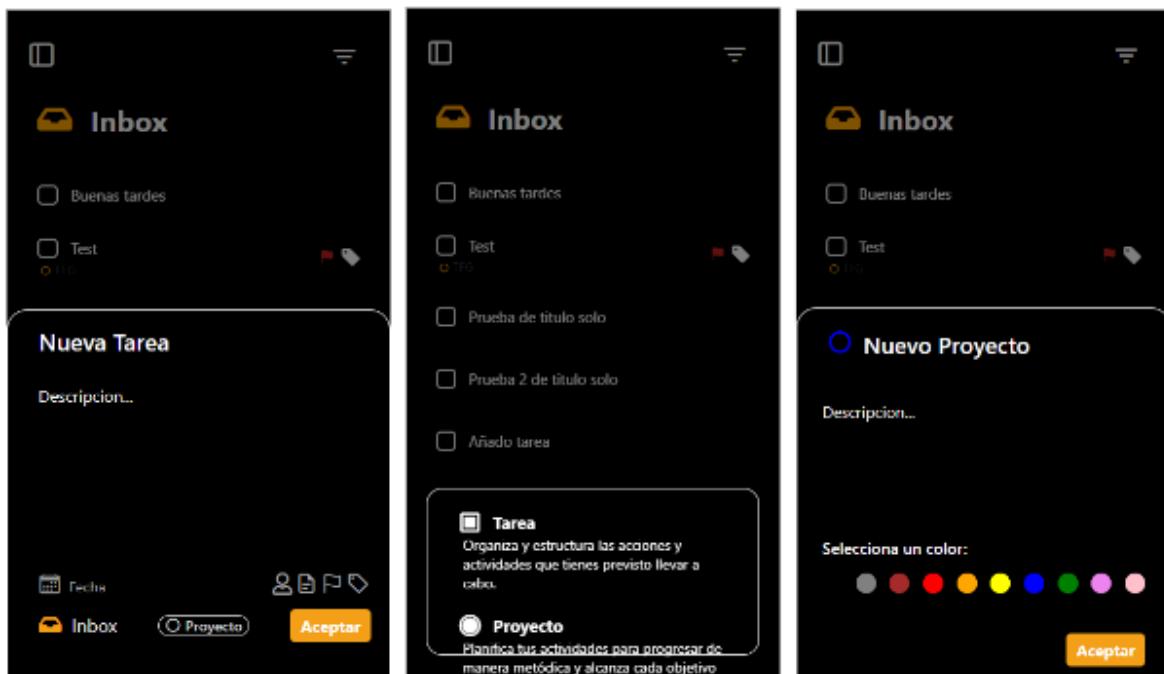


Figura 8.7: Crear tarea y crear proyecto

8.8 Hoy

Ahora nos centraremos en los elementos de la *sidebar*, al pinchar en Hoy podemos observar en la figura 8.8 cómo aparecen dos listados, uno con las tareas retrasadas y otro con las tareas acumuladas.

8.9 Cuanto antes

Pinchando en cuanto antes, podemos ver en la figura 8.9 cómo se despliega una pantalla con un listado de las tareas que deben ser realizadas de manera más urgente.

8.10 Programadas

Pinchando en Programadas, se muestra en la figura 8.10 cómo se abre una pantalla con las distintas tareas y la fecha programada de finalización de las mismas.

8.11 Archivadas

Pinchando en Archivadas (ver figura 8.11), se abre una pantalla con un listado de las tareas que no son urgentes y que pueden realizarse en cualquier momento.

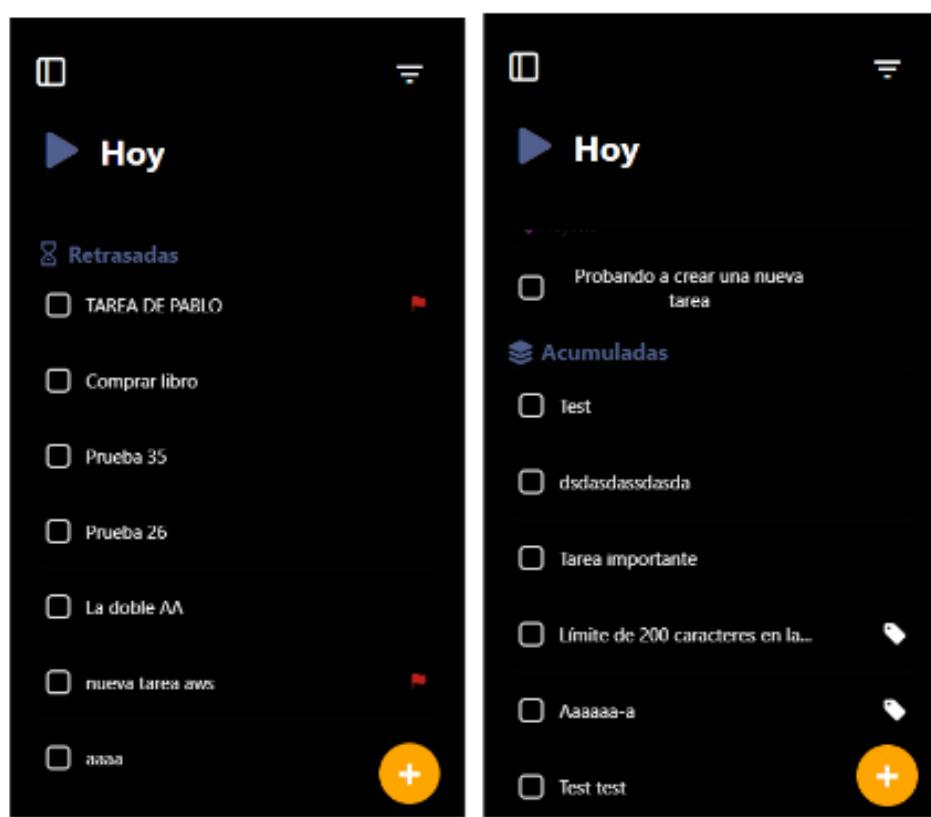


Figura 8.8: Hoy

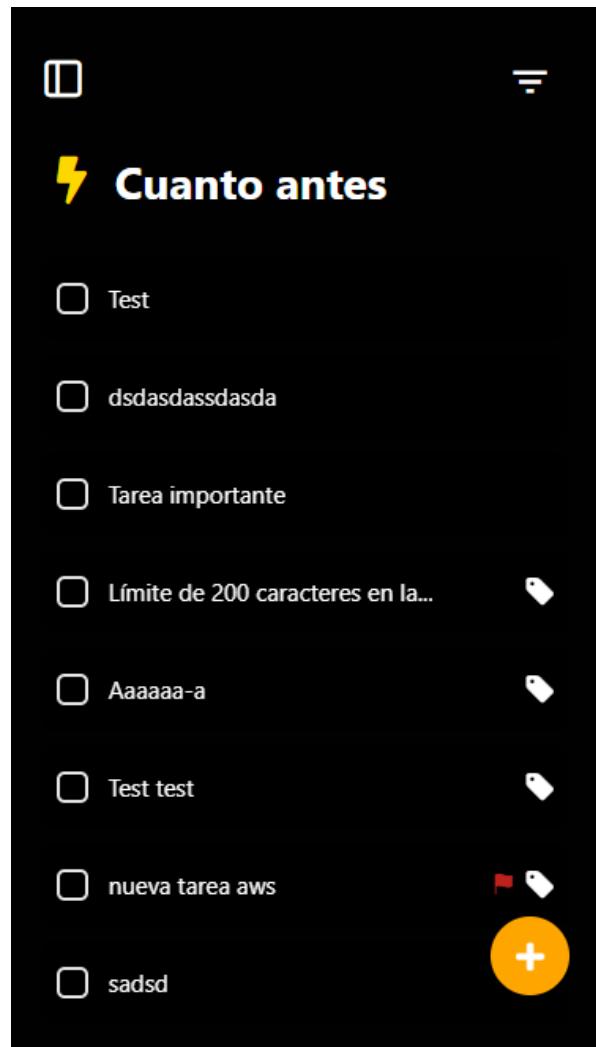


Figura 8.9: Cuanto antes



Figura 8.10: Programadas

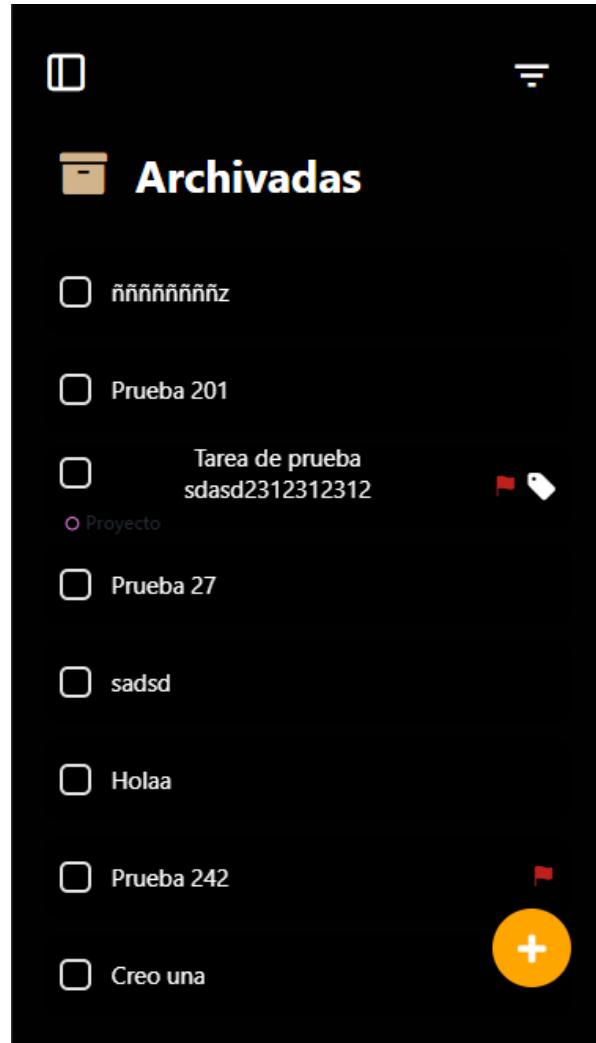


Figura 8.11: Archivadas

8.12 Proyectos

Por último, cuando pinchamos en cualquier proyecto (ver figura 8.12), se muestra una pantalla con un ícono que muestra el porcentaje aproximado de tareas finalizadas que tiene en la parte coloreada del mismo, el nombre del proyecto, un botón para editar el proyecto, la descripción en caso de que exista y un listado de las tareas pertenecientes a dicho proyecto.

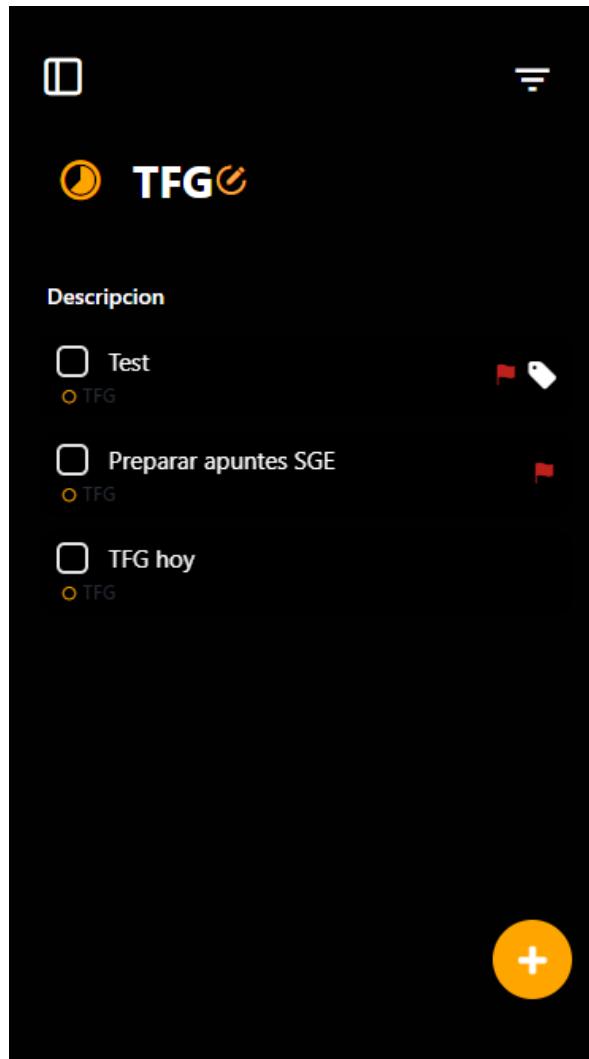


Figura 8.12: Proyectos

8.13 Tareas

En todos los casos en los que aparecen tareas, como podemos ver en la figura 8.13, los listados tienen tareas interactivas, de tal forma que si se arrastra el cursor hacia

la derecha aparece en verde la opción de mover la tarea mediante un modal o si se arrastra el cursor a la izquierda permite completar la tarea. Pinchando en el *checkbox* de la izquierda de cada tarea se abre una opción de selección múltiple para que si queremos, podamos mover o completar varias tareas a la vez.

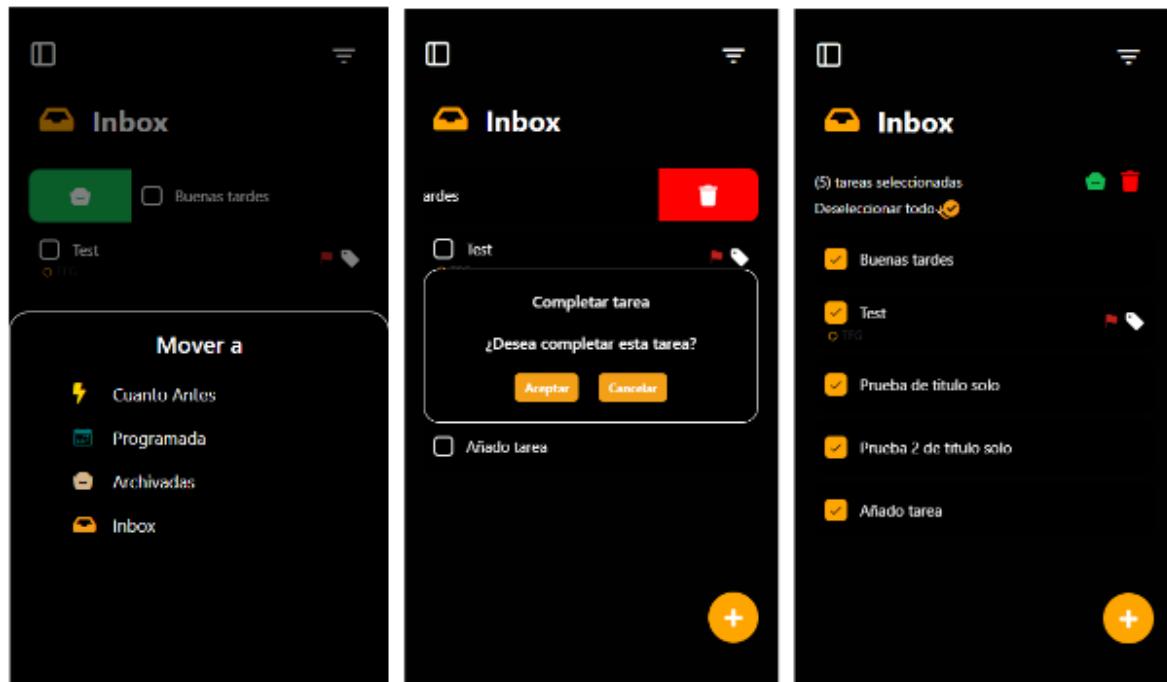


Figura 8.13: Funcionalidad tareas

Capítulo 9

Conclusiones y Trabajo futuro

9.1 Conclusiones

En el mundo actual, caracterizado por la constante presión del tiempo y la creciente demanda de eficiencia, contar con herramientas eficaces para la gestión de tareas se vuelve imprescindible. En este contexto, el método *GTD* destaca por su enfoque sistemático y estructurado, ayudando a los individuos a organizar sus actividades de manera efectiva y a mantener el enfoque en las tareas prioritarias.

Nuestra aplicación *SwiftDo* ha sido concebida con la intención de proporcionar una solución a las necesidades organizativas de los usuarios. En el desarrollo de esta aplicación, nos hemos centrado en la eficiencia y en objetivos claros así como en varios aspectos clave:

- **Aplicación de conceptos y valores de GTD:** *SwiftDo* consigue ser una herramienta efectiva para la gestión del tiempo y la organización personal, realizando con éxito todos los puntos que explica el autor del método *GTD*. Proporcionando un enfoque estructurado y minimalista, respaldado por un conjunto de técnicas y principios de diseño.
- **Infraestructura multiplataforma:** Empleando tecnologías como *React Native* para el desarrollo del cliente, hemos logrado crear una aplicación que puede ejecutarse en una amplia variedad de dispositivos y sistemas operativos. Esta característica garantiza que los usuarios puedan acceder a sus tareas y proyectos desde cualquier lugar y en cualquier momento, sin importar el dispositivo que utilicen.
- **Infraestructura REST:** Con la infraestructura implementada en la parte del *backend* hemos logrado desarrollar un servicio sencillo de comprender y mantener gracias a los principios de la arquitectura *REST*. Además lo hemos hecho en un entorno y con unas tecnologías ampliamente usadas en la actualidad como son *Node.js* y *Express.js*. Por ello este servicio, a parte de cumplir con su función podrá

seguir desarrollándose, mejorándose y ampliándose de cara al futuro.

- **Aplicación de estándares de seguridad:** Tener las medidas de seguridad adecuadas se ha vuelto un aspecto crucial para cualquier servicio y más si se trata con datos personales. Por ello hemos invertido gran parte del desarrollo de esta aplicación en incluir un sistema seguro y fiable de acceso a datos como es *OAuth2.0*, afrontando así uno de los retos más importantes en la actualidad de los servicios web como son la privacidad y la seguridad.
- **Modo offline:** Conscientes de que los usuarios pueden encontrarse en situaciones donde no dispongan de conexión a *Internet*, hemos diseñado nuestra aplicación para funcionar de manera eficiente incluso en modo *offline*. Los usuarios pueden acceder y actualizar sus tareas sin conexión, y una vez que recuperan la conexión, los cambios se sincronizan automáticamente con el servidor, garantizando una experiencia fluida y sin interrupciones.
- **Configuración en servidores personales:** Con el objetivo de satisfacer las necesidades específicas de nuestros usuarios, la aplicación puede ser desplegada y gestionada en cualquier infraestructura. Esto es especialmente útil para empresas u organizaciones que deseen mantener el control total sobre sus datos y procesos internos, ya que les permite utilizar nuestra aplicación en su propia infraestructura sin comprometer la seguridad o la privacidad de la información.

Estos logros son el resultado de un gran esfuerzo por parte del equipo en el desarrollo de esta aplicación, motivado por aportar nuestros conocimientos a la causa de solucionar un problema real. Estamos orgullosos de haber creado una aplicación que no solo ayuda a las personas a organizar sus vidas de manera más efectiva, sino que también les permite adaptarse y prosperar en un entorno en constante cambio.

9.2 Trabajo futuro

En esta sección, exploraremos las ideas y mejoras planificadas para el futuro de *Swift-Do*. Enumeramos a continuación las áreas en las que buscamos innovar y las nuevas características que planeamos implementar para seguir mejorando la experiencia del usuario.

- **Mejora de la integración con agentes conversacionales:** Nuestro objetivo es ampliar la funcionalidad de *Alexa*. Por ejemplo, planeamos implementar la capacidad de añadir proyectos y áreas, y el poder vincularlos a tareas. Además, exploraremos alternativas para ampliar la compatibilidad con otros asistentes virtuales para mejorar la accesibilidad y experiencia del usuario.
- **Inicio de sesión con cuentas de terceros:** Implementar la opción de inicio de sesión con *Google*, *Apple* u otras cuentas de terceros para simplificar el proceso de autenticación y mejorar la seguridad.

-
- **Mejora de la interfaz del usuario:** Refinar los cuadros de diálogo y la interfaz de creación de tareas para hacerla más intuitiva y fácil de usar.
 - **Soporte de aplicación de escritorio a más plataformas:** Desarrollar una versión de escritorio de *SwiftDo* compatible con plataformas como Linux para ampliar su alcance y accesibilidad.
 - **Editor de texto enriquecido:** Integrar un editor de texto enriquecido que permita a los usuarios dar formato a sus descripciones de tareas de manera más flexible y personalizada.

Guía de despliegue

En este apéndice se va a detallar cómo desplegar o compilar los distintos componentes de la aplicación. También se va a explicar la estructura de carpetas del proyecto para ayudar a realizar dicho proceso de despliegue/compilación de los distintos componentes.

9.3 Estructura de carpetas del proyecto

El proyecto está formado por 3 directorios principales. El primer directorio es *BBDD* y en este donde se encuentran los *scripts* de definición de las tablas de la aplicación. Después tenemos el directorio *Backend* que contiene todo el código referente al servidor de autorización *OAuth* y la *API REST*, además de un fichero *compose.yaml* el cual sirve para ejecutar los componentes con *docker*. Por último podemos encontrar el directorio *Frontend* el cual contiene el código fuente referente a la aplicación cliente, este código se encuentra a partir del subdirectorio */src*.

```
BBDD
    bdd.sql
    oauth.sql
Backend
    app.js
    bd
    compose.yaml
    instalation
    oauth
    package-lock.json
    package.json
    public
    routes
    services
Frontend
    app.json
    babel.config.js
    package-lock.json
```

```
package.json
src
desktop
README.md
```

9.4 Despliegue del *backend*

Requisitos

- *Docker* y *Docker-compose*

Para desplegar el *backend* debemos utilizar la herramienta *docker-compose* y para ello debemos definir un fichero *compose.yaml*. Este fichero puede definirse desde cero ajustándolo a las necesidades de infraestructura que tenga cada usuario, aun así proporcionamos la configuración que hemos utilizado nosotros el cual también se puede aplicar a cualquier necesidad. Dicho fichero se encuentra en la raíz del proyecto *Backend* y sigue la estructura que se puede ver en el siguiente definición:

```
services:
  backend:
    image: node:18-alpine
    command: sh -c "npm install && cp ./instalation/oauth/authorize-handler.js ./node_
    ports:
      - 3000:3000
    working_dir: /Backend
    volumes:
      - .:/Backend
    environment:
      PORT: 3000
      POSTGRES_HOST: db
      POSTGRES_USER: tfggtdUser
      POSTGRES_PASSWORD: nodenodito@69
      POSTGRES_DB: tfggtd

  db:
    image: postgres
    volumes:
      - posgresql-data:/var/lib/postgresql/data
      - ../BBDD:/docker-entrypoint-initdb.d/
    environment:
      POSTGRES_USER: tfggtdUser
      POSTGRES_PASSWORD: nodenodito@69
      POSTGRES_DB: tfggtd
```

```
ports:  
  - 127.0.0.1:5432:5432  
  
volumes:  
  postgresql-data:
```

Se definen dos servicios y por lo tanto dos contenedores separados, uno para la parte de la *API REST* utilizando una imagen de *Node* y otro para la base de datos utilizando una imagen de *PostgreSQL*. Dentro de cada servicio definimos las distintas variables de entorno que se utilizan en la aplicación, estas pueden modificarse a gusto del usuario. Cabe recalcar que las variables de entorno referentes a la conexión a la base de datos deben coincidir en ambos servicios.

El primer servicio llamado *backend* realizará tres acciones, la primera es instalar las dependencias con *npm install*, después realiza la sustitución de un fichero del paquete para implementar OAuth por el mismo pero modificado para la solución de un bug de este paquete. Dicho fichero modificado se encuentra en la carpeta del proyecto */Backend/installaton*. Por último se inicia el *backend* con *npm run start*.

En cuanto al segundo servicio, el llamado *db*, iniciará el sistema de gestión de bases de datos de *PostgreSQL*. Para que el sistema cree las tablas automáticamente al iniciar la ejecución, incluimos los *scripts* de definición de datos. Las tablas solo se crearán si no existen.

Por último, en el fichero *compose.yaml* también se define un volumen para los datos de la base de datos, esto permitirá que cuando finalicemos la ejecución de los contenedores, los datos no se borren y sigan estando cuando los levantemos de nuevo.

Para ejecutar el *backend* completo basta con ejecutar el siguiente comando:

```
docker-compose up -d
```

El flag *-d* ejecutará los contenedores en segundo plano liberando el terminal donde se haya ejecutado el comando.

9.5 Compilación de la aplicación web

Requisitos

- *Node.js* - (18.18.2)
- *npm* (9.8.1)

Las versiones indicadas son las que hemos utilizado nosotros, con versiones más modernas debería de funcionar correctamente, no así con versiones anteriores.

Para la compilación y generación de la aplicación web basta con ejecutar los siguientes

comandos situándonos en el directorio *Frontend*:

```
npm install  
npx expo export -p web
```

La ejecución de estos dos comandos generará una carpeta en el mismo directorio llamada *dist*, la cual contiene el *bundle* o paquete de la aplicación generada para web. Este contiene un *index.html* y los *scripts* necesarios. Por lo que para ejecutar la aplicación, bastará con iniciar cualquier servidor web sirviendo dicho directorio y el *index.html* como página de entrada.

9.6 Compilación de la aplicación de escritorio

- *Node.js* - (18.18.2)
- *npm* (9.8.1)

Las versiones indicadas son las que hemos utilizado nosotros, con versiones más modernas debería de funcionar correctamente, no así con versiones anteriores.

La versión de escritorio está desarrollada con *Electron*, un framework que permite ejecutar aplicaciones desarrolladas con tecnologías web en escritorio, esto se consigue incluyendo *Chromiuun* y *Node.js* en los binarios generados, dicho en otras palabras el binario generado con la aplicación web se ejecuta en un navegador incrustado en el mismo binario. *Electron* permite generar binarios para todas las principales plataformas, por ello nuestra aplicación puede ser compilada para cualquiera de estas.

Para realizar la generación del binario de la aplicación de escritorio debemos situarnos en el directorio *Frontent/desktop*, donde hemos incluido un script tanto en *bash* como en *powershell* que realiza los pasos necesarios. Una vez ejecutado el *script*, se generará en la misma carpeta un directorio llamado *out* el cual contendrá el ejecutable de la aplicación de escritorio. Es importante recalcar que tal compilación se hará para la misma plataforma donde se realice la misma, por ejemplo si deseamos compilar para *windows* se deberá de realizar este proceso en *windows*.

Endpoints

9.6.1 Tarea

Endpoint	/task/
Descripción	Crea una tarea para el usuario que lo solicita
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	201
Código HTTP (KO)	409

Parámetros	/task/	Tipo	Opcional
title (<i>Body</i>)	Titulo de la tarea	<i>String</i>	No
description (<i>Body</i>)	Descripción de la tarea	<i>String</i>	Si
state (<i>Body</i>)	Estado de la tarea	<i>Integer</i>	Si
important_fixed (<i>Body</i>)	Valor de importante	<i>Boolean</i>	Si
completed (<i>Body</i>)	Valor de completado	<i>Boolean</i>	Si
project_id (<i>Body</i>)	Proyecto de la tarea	<i>Integer</i>	Si
context_id (<i>Body</i>)	Contexto de la tarea	<i>Integer</i>	Si
date_limit (<i>Body</i>)	Fecha de la tarea	<i>Timestamp</i>	Si

Endpoint	/task/:id
Descripción	Modifica la tarea con el id indicado en la url del usuario que lo solicita
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200

Endpoint	/task/:id
Código HTTP (KO)	409

Parámetros	Endpoint	Tipo	Opcional
task_id (<i>Path</i>)	id de la tarea a modificar	<i>Integer</i>	No
title (<i>Body</i>)	Titulo de la tarea modificado	<i>String</i>	Si
description (<i>Body</i>)	Descripción de la tarea modificado	<i>String</i>	Si
state (<i>Body</i>)	Estado de la tarea modificado	<i>Integer</i>	Si
important_fixed (<i>Body</i>)	Valor de importante	<i>Boolean</i>	Si
completed (<i>Body</i>)	Valor de completado	<i>Boolean</i>	Si
project_id (<i>Body</i>)	Proyecto de la tarea	<i>Integer</i>	Si
context_id (<i>Body</i>)	Contexto de la tarea	<i>Integer</i>	Si
date_limit (<i>Body</i>)	Fecha de la tarea	<i>Timestamp</i>	Si

Endpoint	/task/:id
Descripción	Devuelve el contenido de tarea con el id solicitado
Método HTTP	GET
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Parámetros	Endpoint	Tipo	Opcional
task_id (<i>Path</i>)	id de la tarea a consultar	<i>Integer</i>	No

Endpoint	/task/
Descripción	Devuelve todas las tareas no completadas del usuario que las solicita
Método HTTP	GET
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Endpoint	/task/movelist
Descripción	Mueve de estado un listado de tareas
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	409

Parámetros	Endpoint	Tipo	Opcional
list_ids (Body)	Listado de ids de las tareas a mover de estado	array[Integer]	No
state (Body)	Estado al que se mueven las tareas	Integer	No

Endpoint	/task/completelist
Descripción	Completa un listado de tareas
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	409

Parámetros	Endpoint	Tipo	Opcional
list_ids (Body)	Listado de ids de las tareas a mover de estado	array[Integer]	No
comple- ted(Body)	Valor de completar al que modificar las tareas	Boolean	No

Endpoint	/task/addTag
Descripción	Añade una etiqueta a una tarea
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	409

Parámetros	/task/addTag	Tipo	Opcional
task_id (<i>Body</i>)	Id de la tarea a la cual se le añade la etiqueta	<i>Integer</i>	No
tag(<i>Body</i>)	Etiqueta a añadir a la tarea	<i>Object</i>	No

Endpoint	/task/:id/tags
Descripción	Devuelve el listado de etiquetas de la tarea con el id solicitado.
Método HTTP	GET
Cabecera de	Bearer token
Autorización	
Código HTTP (OK)	200
Código HTTP (KO)	404

Parámetros	/task/:id/tags	Tipo	Opcional
task_id (<i>Path</i>)	Id de la tarea a obtener sus etiquetas	<i>Integer</i>	No

Endpoint	/task/info
Descripción	Devuelve un resumen de la información de las tareas del usuario que lo solicita. Para mostrar en la sidebar.
Método HTTP	GET
Cabecera de	Bearer token
Autorización	
Código HTTP (OK)	200
Código HTTP (KO)	404

9.6.2 Usuario

Endpoint	/user/register
Descripción	Realiza el registro de un usuario creandolo en la BBDD
Método HTTP	POST
Cabecera de Autorización	
Código HTTP (OK)	200
Código HTTP (KO)	409

Parámetros	/user/register	Tipo	Opcional
email (<i>Body</i>)	E-mail del usuario	<i>String</i>	No
name (<i>Body</i>)	Nombre de usuario	<i>String</i>	No
password (<i>Body</i>)	Contraseña del usuario	<i>String</i>	No

9.6.3 Auth

Endpoint	/oauth/authorize		
Descripción	Obtiene el código de autorización de <i>OAuth</i> si la autenticación es correcta.		
Método HTTP	POST		
Cabecera de Autorización			
Código HTTP (OK)	304		
Código HTTP (KO)	401		

Parámetros	/oauth/authorize	Tipos	Opcional
client_id (<i>Body</i>)	id del cliente a autorizar	<i>Integer</i>	No
response_type (<i>Body</i>)	code por defecto ya que estamos obteniendo el código de autorización	<i>String</i>	No
email (<i>Body</i>)	E-mail del usuario	<i>String</i>	No
password (<i>Body</i>)	Contraseña del usuario	<i>String</i>	No

Endpoint	/oauth/token		
Descripción	Si el código de autorización es correcto devuelve el token de acceso. También permite realizar la renovación del token.		
Método HTTP	POST		
Cabecera de Autorización			
Código HTTP (OK)	200		

Endpoint	/oauth/token
Código HTTP (KO)	401

Parámetros	Op-cional	Tipo
client_id (<i>Body</i>) el cliente desde donde se requiere el token	<i>In-</i> <i>te-</i> <i>ger</i>	No
client_secret secreto del cliente autorizado	<i>String</i>	No
grant_type (<i>Body</i>) el proceso de <i>OAuth</i> puede ser <i>authorization_code</i> para obtener el token a partir de un código o <i>refresh_token</i> para refrescar un token ya existente	<i>String</i>	No
code (<i>Body</i>) Si estamos en el proceso <i>authorization_code</i> es el código de autorización obtenido previamente. Si no, no es necesario	<i>String</i>	Si
redirect_uri a la que redireccionará al obtener el token	<i>String</i>	No
refresh_token (<i>Body</i>) Si estamos en el proceso <i>refresh_token</i> es el token de renovación necesario para obtener un nuevo token. Si no, no es necesario.	<i>String</i>	Si
email E-mail del usuario	<i>String</i>	No
password (<i>Body</i>) Contraseña del usuario	<i>String</i>	No

9.6.4 Proyecto

Endpoint	/project/
Descripción	Devuelve todos los proyectos del usuario que lo requiere.
Método HTTP	GET
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Endpoint	/project/
Descripción	Crea un proyecto para el usuario que lo requiere.
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	409

Parámetros	/project/	Tipo	Opcional
title (<i>Body</i>)	Titulo del proyecto	String	No
description (<i>Body</i>)	Descripción del proyecto	String	Si
color (<i>Body</i>)	Color del proyecto	String	No

Endpoint	/project/:id/complete
Descripción	Completa el proyecto referente al id pasado por la url para el usuario que lo requiere. Si tiene tareas asociadas sin completar, también las completa.
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Parámetros	/project/:id/complete	Tipo	Opcional
project_id (<i>Path</i>)	Id del proyecto	Integer	No

Endpoint	/project/:id
Descripción	Devuelve toda la información referente al proyecto del id pasado por la url.
Método HTTP	GET

Endpoint	/project/:id
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Parámetros	/project/:id	Tipo	Opcional
project_id (<i>Path</i>)	Id del proyecto	<i>Integer</i>	No

Endpoint	/project/:id
Descripción	Modifica el proyecto con id pasado por la url.
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Parámetros	/project/	Tipo	Opcional
title (<i>Body</i>)	Titulo del proyecto	<i>String</i>	Si
description (<i>Body</i>)	Descripción del proyecto	<i>String</i>	Si
color (<i>Body</i>)	Color del proyecto	<i>String</i>	Si
completed (<i>Body</i>)	Valor de completado	<i>Boolean</i>	Si

9.6.5 Contexto

Endpoint	/context/
Descripción	Crea un context para el usuario que lo requiere.
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Parámetros	/context/	Tipo	Opcional
name (<i>Body</i>)	Nombre del contexto	<i>String</i>	No

Endpoint	/context/
Descripción	Devuelve todos los contextos del usuario que lo requiere.
Método HTTP	GET
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Endpoint	/context/:id
Descripción	Elimina el contexto con el id pasado por la url. Si ese contexto tiene tareas, se modifican para que no tengan contexto.
Método HTTP	DELETE
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Parámetros	/context/:id	Tipo	Opcional
context_id (<i>Path</i>)	id del contexto	<i>Integer</i>	No

Endpoint	/context/:id
Descripción	Modifica el contexto con el id pasado por la url.
Método HTTP	POST
Cabecera de Autorización	Bearer token
Código HTTP (OK)	200
Código HTTP (KO)	404

Parámetros	/context/:id	Tipo	Opcional
context_id (<i>Path</i>)	id del contexto	<i>Integer</i>	No

Parámetros	/context/:id	Tipo	Opcional
name (<i>Body</i>)	Nuevo nombre del contexto	<i>String</i>	Si

9.6.6 Etiqueta

Endpoint	/tag/		
Descripción	Crea una etiqueta para el usuario que lo requiere.		
Método HTTP	POST		
Cabecera de Autorización	Bearer token		
Código HTTP (OK)	200		
Código HTTP (KO)	409		

Parámetros	/tag/	Tipo	Opcional
name (<i>Body</i>)	Nombre de la etiqueta	<i>String</i>	No

Endpoint	/tag/gettags		
Descripción	Obtiene las etiquetas del usuario que lanza la operación.		
Método HTTP	GET		
Cabecera de Autorización	Bearer token		
Código HTTP (OK)	200		
Código HTTP (KO)	404		

Endpoint	/tag/:name		
Descripción	Elimina la etiqueta con el nombre pasado por parametro.		
Método HTTP	DELETE		
Cabecera de Autorización	Bearer token		
Código HTTP (OK)	200		
Código HTTP (KO)	404		

Parámetros	/tag/	Tipo	Opcional
name (<i>Path</i>)	Nombre de la etiqueta a eliminar	<i>String</i>	No

Bibliografía

- [1] D. Allen, *Getting Things Done. The Art of Stress-Free Productivity*. Penguin, 2003.
- [2] «Getting Things Done: The Art of Stress Free Productivity by David Allen (Nicole Schlinger Book Review)». <https://medium.com/@nicoleschlinger/getting-things-done-the-art-of-stress-free-productivity-by-david-allen-nicole-schlinger-book-688e1440290d>.
- [3] J. L. V. Grau, «Enfoque Adaptativo o predictivo, ¿cuál elegir?» <https://www.linkedin.com/pulse/enfoque-adaptativo-o-predictivo-cu%C3%A1l-elegir-juan-luis-vila-grau/>.
- [4] «Librería node.bcrypt.js». <https://www.npmjs.com/package/bcrypt>.
- [5] R. T. Fielding y R. N. Taylor, «Architectural styles and the design of network-based software architectures», Tesis doctoral, University of California, Irvine, 2000.
- [6] «Express.js». <https://expressjs.com/>.
- [7] D. Hardt, «The OAuth 2.0 Authorization Framework». RFC 6749; RFC Editor, oct-2012.
- [8] D. A. Norman, *The Design of Everyday Things*. USA: Basic Books, Inc., 2002.
- [9] «Expo». <https://expo.dev/>.
- [10] «React Navigation». <https://reactnavigation.org/>.
- [11] «Drawer Navigation». <https://reactnavigation.org/docs/drawer-navigator/>.
- [12] «Axios». <https://axios-http.com/es/docs/intro>.
- [13] «React Native Calendars». <https://github.com/wix/react-native-calendars>.
- [14] «React Native Async Storage». <https://www.npmjs.com/package/@react-native-async-storage/async-storage>.
- [15] «Markdown Display». <https://www.npmjs.com/package/react-native-markdown-display>.
- [16] «Modern Datepicker». <https://kiarash-z.github.io/react-modern-calendar-datepicker/>.

-
- [17] «Swipeable». <https://docs.swmansion.com/react-native-gesture-handler/docs/components/swipeable/>.
 - [18] «Wheel Color Picker». <https://www.npmjs.com/package/react-native-wheel-color-picker>.
 - [19] «Authorization code grant flow». <https://developer.amazon.com/en-US/docs/alexa/account-linking/configure-authorization-code-grant.html>.