Universidad de Cantabria

Facultad de Ciencias

SERVICIO DE AUTOMATIZACIONES EN CORREOS ELECTRÓNICOS INTEGRADAS EN EL ENTORNO DE GOOGLE

(Automated Email Workflow Service within the Google Ecosystem)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: David Ruiz del Corro

Directora: Cristina Tîrnăucă

Junio – 2025

Índice general

Índice de figuras

Índice de cuadros

Resumen

Este Trabajo de Fin de Grado se centra en el desarrollo de una aplicación web orientada a la automatización de tareas a partir de correos electrónicos recibidos en Gmail. Dicha automatización se basa en un sistema de macros, acciones configurables por el usuario que se activan mediante etiquetas de Gmail.

Gracias a los filtros de Gmail, el usuario puede etiquetar automáticamente los correos entrantes, generando de esta forma una integración muy flexible con el sistema de macros, de modo que se puedan optimizar procesos rutinarios en contextos personales y profesionales.

La plataforma ofrece una interfaz intuitiva desde la cual es posible ver y gestionar macros, así como activar o desactivar la monitorización del buzón según las necesidades del usuario.

Para su implementación, se ha empleado Angular 19 en el desarrollo del *front-end* y Node.js con Express en el *back-end*, integrando las APIs de Gmail, Google Drive, Google Calendar y Google Docs, con diversos servicios de Google Cloud como Pub/Sub, Cloud Functions, Firebase Authentication y Firestore, una base de datos NoSQL.

Palabras clave: Computación en la nube, Aplicación web, Inteligencia Artificial, Servicios de Google, Automatización de procesos.

Abstract

This Final Degree Project focuses on the development of a web application aimed at automating tasks based on incoming emails received in Gmail. The automation is implemented through a system of macros, user configurable actions that are triggered by Gmail labels.

Thanks to Gmail filters, users can automatically tag incoming emails, enabling seamless integration with the macro system and optimizing routine actions in both personal and professional settings.

The platform provides an intuitive interface from which users can view and manage macros, as well as enable or disable mailbox monitoring according to their preferences.

The implementation uses Angular 19 for the front-end and Node.js with Express for the back-end, integrating Gmail, Google Drive, Google Calendar, and Google Docs APIs, along with several Google Cloud services such as Pub/Sub, Cloud Functions, Firebase Authentication, and Firestore, a NoSQL database.

Keywords: Cloud computing, Web application, Artificial Intelligence, Google Services, Process automation.

Capítulo 1

Introducción

La automatización de tareas en entornos digitales es un campo en constante evolución, con soluciones que van desde simples filtros en clientes de correo hasta sistemas avanzados integrados con múltiples servicios externos. En este contexto, el auge de la inteligencia artificial (IA en adelante) ha ampliado las posibilidades, permitiendo **automatizar tareas complejas** como la clasificación, respuesta o extracción de información de correos electrónicos.

Paralelamente, la proliferación de herramientas basadas en IA ha facilitado tareas como el **resumen de textos**, la **clasificación de contenido** y la **extracción de información no estructurada**. Un claro ejemplo es ChatGPT, capaz de analizar correos extensos, identificar sus puntos clave, clasificar su contenido y extraer datos relevantes como fechas, nombres o compromisos, incluso cuando estos no están explícitamente estructurados.

En los últimos años han surgido numerosas plataformas que permiten automatizar tareas entre aplicaciones y servicios digitales. Dos de las más destacadas son IFTTT y Zapier, que permiten conectar distintas aplicaciones mediante reglas personalizadas y flujos de trabajo automatizados.

En este contexto, el presente Trabajo de Fin de Grado (TFG en adelante) aborda una necesidad específica no cubierta por las soluciones anteriormente mencionadas. Mientras que estas plataformas se centran en la automatización genérica entre múltiples aplicaciones, esta propuesta se orienta exclusivamente a la gestión inteligente del correo electrónico en Gmail. Integra funcionalidades avanzadas como el resumen de contenido, el almacenamiento de archivos adjuntos o la detección de eventos mediante IA, todo ello con una integración profunda en el ecosistema de Google.

1.1. Objetivos

El objetivo general de este TFG es desarrollar una aplicación web que permita automatizar tareas relacionadas con la gestión del correo electrónico, con el fin de mejorar la productividad y eficiencia de los usuarios en entornos personales y profesionales.

Para alcanzar este objetivo general, se plantean los siguientes objetivos específicos:

- Diseñar una interfaz de usuario (*User Interface*, UI) moderna y *responsive*, basada en principios de diseño moderno, que garantice una experiencia de usuario (*User Experience*, UX) sencilla e intuitiva, que permita su uso sin necesidad de conocimientos técnicos avanzados, ofreciendo una experiencia clara, accesible y visualmente atractiva.
- Implementar, mediante el uso de IA, funcionalidades de automatización aplicadas al correo electrónico tales como la clasificación, resumen de contenido o extracción de información relevante.
- Integrar la aplicación con servicios del ecosistema de Google como Gmail, Google Drive, Google Docs y Google Calendar, a fin de extender la funcionalidad y adaptarse a diferentes tipos de uso profesional y personal.
- Garantizar la seguridad y privacidad de los datos mediante la implementación de protocolos seguros como OAuth 2.0, el uso de JSON Web Tokens para el acceso a la API, y el almacenamiento seguro en Firestore, cumpliendo buenas prácticas de desarrollo.
- Aplicar una arquitectura modular y escalable basada en servicios de Google Cloud como Pub/Sub, Cloud Functions, Firebase Authentication y Firestore, que permita una gestión eficiente de eventos asíncronos y la futura ampliación del sistema sin comprometer su rendimiento.
- Desplegar la aplicación utilizando tecnologías *serverless* que faciliten su mantenimiento, reduzcan los costes en entornos de baja demanda y aseguren su disponibilidad global.
- Fomentar el aprendizaje y la aplicación práctica de tecnologías actuales como Angular 19 en el *front-end*, Node.js con Express en el *back-end*, y herramientas de control de versiones como Git y GitHub para una gestión eficiente del desarrollo.

1.2. Planificación y metodología

La planificación del desarrollo de este TFG ha estado condicionada por el proceso de aprendizaje progresivo de las tecnologías empleadas. Si bien contaba con cierta experiencia previa en JavaScript, no tenía conocimientos sólidos sobre herramientas clave como Angular, Node.js o Firebase, ni sobre el entorno de Google Cloud. Por ello, el avance del proyecto ha estado estrechamente ligado al descubrimiento y comprensión de estas tecnologías.

Durante las primeras semanas, el desarrollo se combinó con una intensa fase de investigación y aprendizaje práctico. A medida que adquiría nuevos conocimientos, los aplicaba directamente en el desarrollo de funcionalidades que más adelante serían parte integral de la aplicación final. Debido a este enfoque autodidacta e iterativo, no fue posible establecer desde el inicio una metodología de desarrollo formal. Sin embargo, para garantizar una evolución ordenada y medible, se establecieron objetivos a corto y medio plazo, con fechas estimadas de cumplimiento, los cuales fueron registrados y gestionados mediante la herramienta Trello.

La planificación del proceso puede dividirse claramente en cuatro etapas diferenciadas, de acuerdo con los avances técnicos y conceptuales alcanzados en cada una de ellas:

Etapa 1: Definición de la idea y objetivos generales (octubre 2024 – noviembre 2024)

En esta primera fase se partía de una idea inicial poco desarrollada. Durante estas semanas se llevó a cabo un proceso de maduración del concepto, explorando distintas posibilidades de aplicación, casos de uso y objetivos funcionales. También se realizaron cuestionarios a estudiantes universitarios para conocer sus propuestas de mejora del correo electrónico. Al finalizar esta etapa, se contaba con una visión mucho más clara y concreta de qué debía hacer la aplicación y qué necesidades cubriría.

Etapa 2: Investigación tecnológica y primeros desarrollos (noviembre 2024 – enero 2025)

Con el concepto ya definido, se inició una investigación exhaustiva sobre los aspectos técnicos necesarios para hacer viable el proyecto. Las principales cuestiones abordadas en esta etapa incluyeron el acceso al correo electrónico de los usuarios, el tratamiento automatizado de los mensajes entrantes y la autenticación sin intervención constante del usuario. Paralelamente, se desarrollaron pequeñas funcionalidades con el objetivo de afianzar los conocimientos adquiridos y validar conceptos clave.

Etapa 3: Desarrollo estructurado de la aplicación (enero 2025 – abril 2025)

Antes de iniciar esta etapa, se llevó a cabo una planificación detallada que permitió definir tareas concretas y distribuirlas en el tiempo mediante herramientas de gestión ágil (véase la Figura 1.1). Esta organización facilitó el avance ordenado del desarrollo y la priorización de funcionalidades clave.

Durante esta fase se establecieron las bases técnicas y conceptuales de la plataforma, consolidando su arquitectura general y asegurando el correcto funcionamiento de los elementos fundamentales como la autenticación, el sistema de seguimiento de correos y la gestión de macros. A partir de este punto, se adoptó una metodología de desarrollo incremental, centrada en la implementación modular de funcionalidades. Gracias a la estructura flexible del sistema, fue posible incorporar nuevas acciones (macros) como bloques independientes que no interferían con las funcionalidades ya existentes. Este enfoque permitió evolucionar la plataforma de forma controlada, asegurando su estabilidad y facilitando futuras ampliaciones.

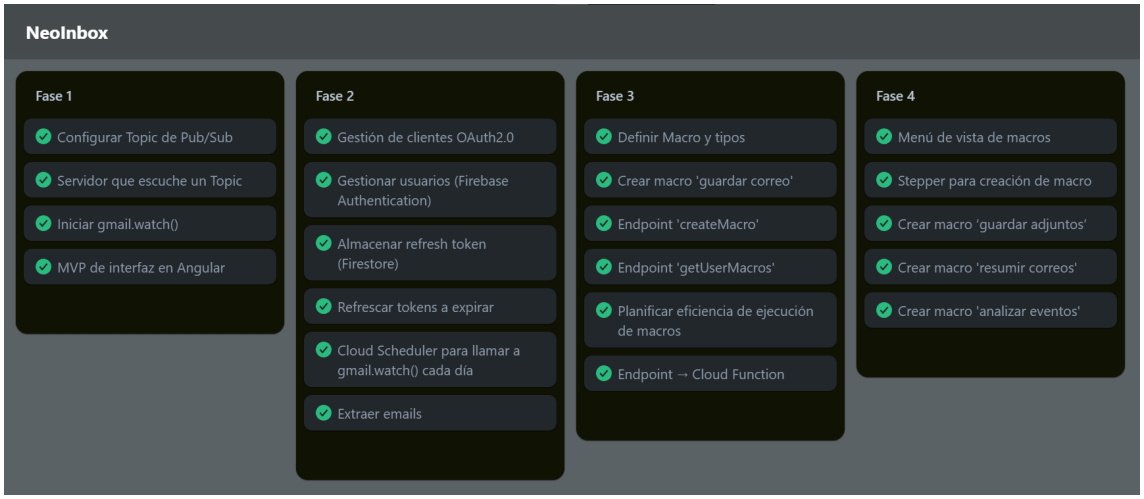

Figura 1.1: Planificación en Trello de la Etapa 3

Etapa 4: Optimización y ajustes finales (abril 2025 – mayo 2025)

Con el grueso de la aplicación ya desarrollado, esta última fase se centró en mejorar la experiencia de usuario, ajustar aspectos visuales y corregir errores menores. Se realizaron pruebas de uso para asegurar un funcionamiento fluido y se aplicaron mejoras de rendimiento y usabilidad.

Para ofrecer una visión más clara y resumida del proceso seguido, en la Tabla 1.1 se presenta un esquema del cronograma asociado a las distintas etapas de desarrollo descritas anteriormente.

Etapa	Periodo	Objetivos principales
1. Definición del proyecto	oct. 2024 – nov. 2024	▪ Definición de la idea inicial y objetivos generales. ▪ Análisis preliminar de viabilidad y funcionalidades. ▪ Concreción del enfoque de la aplicación.
2. Investigación técnica y pruebas	nov. 2024 – ene. 2025	▪ Investigación sobre tecnologías clave (APIs, OAuth2, automatización). ▪ Desarrollo de módulos funcionales simples. ▪ Validación de conceptos técnicos.
3. Desarrollo principal	ene. 2025 – abr. 2025	▪ Implementación de las funcionalidades principales. ▪ Diseño modular del sistema de macros. ▪ Integración con servicios de Google.
4. Optimización y ajustes finales	abr. 2025 – may. 2025	▪ Mejora visual y experiencia de usuario. ▪ Corrección de errores y detalles finales. ▪ Validación funcional y pruebas de uso.

Tabla 1.1: Cronograma del desarrollo del proyecto

Aunque no forma parte del desarrollo técnico del software, la etapa de documentación del progreso y redacción de la memoria ha sido fundamental para completar el TFG. Desde los primeros compases del proyecto, se asumió la importancia de mantener un registro sistemático de los avances logrados en cada sesión de trabajo. Al finalizar cada jornada, se dedicaba un breve espacio de tiempo a anotar los progresos realizados, los problemas encontrados, las soluciones aplicadas y aquellas cuestiones pendientes que requerían una revisión posterior. Este hábito de documentación no solo sirvió para tener una visión clara de la evolución del proyecto, sino que fue clave en el momento de afrontar la redacción final de la memoria.

Una vez consolidada la estructura y funcionalidad de la aplicación, se dio paso a la elaboración de este documento. Gracias a las anotaciones recogidas durante los meses previos, el proceso de redacción fue más ágil y ordenado, permitiendo reflejar con mayor precisión tanto el desarrollo técnico como las decisiones de diseño que marcaron la evolución del trabajo.

El resto del documento se estructura como sigue: en el capítulo 2, se detallan los requisitos del sistema, tanto funcionales como no funcionales, además de tratar las historias de usuario. El capítulo 3 presenta la arquitectura general de la solución, explicando aspectos clave como la monitorización del buzón de correo, la gestión de *tokens* de refresco, el cliente OAuth 2.0, la estructura de base de datos, la optimización del procesamiento de macros y las integraciones con servicios externos. A continuación, en el capítulo 4, además de tratar aspectos de seguridad y control de versiones, se presenta la implementación del sistema, diferenciando entre el *back-end*, centrado en Google Cloud y la API REST, y el *front-end*, enfocado en la interfaz de usuario. El capítulo 5 recoge los distintos tipos de pruebas realizadas, así como un análisis de su cobertura y resultados. Por último, el capítulo 6 resume las conclusiones extraídas del trabajo realizado y propone posibles mejoras y líneas de desarrollo futuras.

Capítulo 2

Requisitos del sistema

El diseño de cualquier aplicación debe comenzar por una definición clara de los requisitos que debe cumplir, ya que de ellos dependen tanto su funcionalidad como su calidad y viabilidad futura. En este proyecto, se buscó no solo cubrir las necesidades básicas de un usuario individual, sino también preparar el servicio para un posible despliegue comercial en entornos profesionales.

Para ello, durante las fases iniciales del trabajo se consultó tanto a usuarios particulares como a profesionales del ámbito empresarial, analizando sus expectativas respecto al uso del correo electrónico y las oportunidades de automatización. Este enfoque permitió recoger requisitos variados que abarcan desde funcionalidades esenciales hasta aspectos relacionados con la seguridad, la escalabilidad, la usabilidad y la mantenibilidad del sistema.

En esta sección se recogen tanto los requisitos funcionales como los no funcionales que guían el desarrollo del sistema. Tal y como expone Ian Sommerville en Software Engineering (2011), "los requisitos funcionales describen lo que debe hacer el sistema, mientras que los no funcionales definen restricciones sobre cómo debe hacerlo" [4]. Esta diferenciación es esencial para afrontar el diseño de cualquier aplicación de forma estructurada y eficiente, ya que permite no solo delimitar el alcance funcional del sistema, sino también establecer criterios de calidad, rendimiento, seguridad o usabilidad.

2.1. Requisitos funcionales

Los requisitos funcionales describen las capacidades que debe ofrecer el sistema desde el punto de vista del usuario final. Para su definición, se optó por utilizar el enfoque de historias de usuario, el cual trataremos en profundidad en la sección 2.3.

En la Tabla 2.1 se detallan los principales requisitos funcionales del sistema, organizados por módulos que los agrupan en base a su funcionalidad.

ID	Descripción del requisito funcional
Servicio - 01	
01.01	El usuario debe poder activar el seguimiento de su cuenta de Gmail para permitir la ejecución de macros sobre los correos recibidos.
01.02	El usuario debe poder desactivar de forma indefinida el seguimiento, pausando así la ejecución automática de macros.
Sesión - 02	
02.01	El sistema debe permitir el registro de usuarios mediante su cuenta de Google por medio del protocolo OAuth 2.0.
02.02	El usuario debe poder autenticarse para acceder a plataforma.
02.03	El usuario debe poder finalizar su sesión de manera segura.
Macros - 03	
03.01	El sistema debe permitir al usuario crear macros especificando su nombre, tipo y una o más etiquetas de Gmail que activarán dicha macro.
03.02	El usuario debe poder consultar un catálogo con todas las macros existentes.
03.03	El sistema debe permitir al usuario editar los parámetros de sus macros existentes.
03.04	El usuario debe poder eliminar una macro.

Tabla 2.1: Requisitos funcionales del sistema

2.2. Requisitos no funcionales

Además de los requisitos funcionales, se han tenido en cuenta diversos requisitos no funcionales que recogen algunos de los aspectos clave relacionados con la calidad del producto (véanse las Tablas 2.2, 2.3, 2.4 y 2.5).

Pese a haberlos tenido muy en cuenta, la velocidad de la página o la optimización de recursos son requisitos en los que no haré hincapié debido a que son un básico en los proyectos de calidad. Así pues, presentaré otros que puedan resultar más interesantes.

ID	RNF-01
Nombre	Escalabilidad del proyecto
Descripción	El sistema deberá estar diseñado para soportar cientos de miles de usuarios simultáneamente. Cada usuario dispondrá de su propio espacio de trabajo en la base de datos, lo que facilitará la separación lógica de datos y permitirá escalar de forma horizontal sin conflictos. Además, la estructura deberá estar preparada para extenderse a otros servicios como el entorno de Microsoft en el futuro.

Tabla 2.2: Requisito no funcional: escalabilidad del proyecto

ID	RNF-02
Nombre	Seguridad mediante JSON Web Tokens
Descripción	Las peticiones al *back-end* estarán protegidas mediante el uso de JSON Web Tokens (JWT). Cada usuario recibirá un *token* tras autenticarse, que se utilizará para verificar su identidad en cada petición a la API. Esto permitirá mantener la sesión de forma segura y sin necesidad de autenticar en cada interacción.

Tabla 2.3: Requisito no funcional: seguridad mediante JWT

ID	RNF-03
Nombre	Gestión de errores y tolerancia a fallos
Descripción	El sistema no interrumpirá el funcionamiento global en caso de que se produzca un error en la ejecución de una macro (por ejemplo, por un fallo puntual en la API de Gmail). En su lugar, esperará a que llegue un nuevo correo para volver a ejecutar la macro, garantizando así una experiencia resiliente y sin pérdidas de funcionalidad.

Tabla 2.4: Requisito no funcional: gestión de errores y tolerancia a fallos

ID	RNF-04
Nombre	Accesibilidad y diseño
Descripción	La aplicación se adaptará correctamente a distintos tamaños de pantalla, incluyendo dispositivos móviles. Además, deberá cuidar la accesibilidad mediante el uso de colores con buen contraste, etiquetas `alt` en imágenes y una estructura clara para todos los elementos de la interfaz.

Tabla 2.5: Requisito no funcional: accesibilidad y diseño

2.3. Historias de usuario

Durante las fases iniciales del proyecto, antes de definir formalmente los requisitos funcionales, se empleó la técnica de historias de usuario para recoger y organizar las necesidades del sistema desde el punto de vista del usuario final. Este enfoque centrado en el usuario permitió identificar de manera clara y concisa las funcionalidades esenciales, facilitando posteriormente su transformación en requisitos técnicos más estructurados.

Las historias de usuario consisten en descripciones breves y narrativas que explican lo que un usuario desea lograr con el sistema. Esta técnica no solo ayudó a priorizar funcionalidades, sino que también permitió alinear el desarrollo con las expectativas reales de uso, mejorando así la experiencia general del usuario.

Para ilustrar con más detalle en qué consisten, en la Tabla 2.6 se describe la historia de usuario correspondiente a la creación de una nueva macro. Esta historia refleja tanto el objetivo funcional como los criterios de aceptación que debe cumplir la funcionalidad para ser considerada completa y operativa.

ID	03.01
Título	Crear nueva macro
Descripción	La aplicación debe permitir al usuario crear una nueva macro personalizada que automatice una serie de acciones en función de los correos electrónicos recibidos. Esta funcionalidad facilita el ahorro de tiempo y la optimización de tareas repetitivas mediante reglas definidas por el propio usuario.

Criterios de aceptación	**03.01.01 - Crear macro correctamente**
	▪ El usuario puede acceder a la sección de creación de macros desde:
	  • el *dashboard*, o
	  • el menú de listado de macros.
	▪ El usuario completa correctamente cada paso del formulario: nombre, etiquetas, acción y parámetros.
	▪ Al confirmar, la macro aparece listada correctamente en el menú de macros.
	03.01.02 - Validación del formulario
	▪ Si el usuario no rellena todos los campos obligatorios, el sistema impide guardar la macro.
	▪ Se informa claramente al usuario de los campos que deben ser completados antes de continuar.
	03.01.03 - Fallo de conexión o *back-end*
	▪ Si durante el proceso de creación de una macro se detecta un fallo de conexión o un error del servidor, la aplicación muestra un mensaje informativo.
	▪ No se crea ninguna macro en la base de datos si la operación no ha podido completarse correctamente, garantizando la integridad de la información.
	▪ El usuario puede volver a intentar crear la macro una vez restablecida la conexión.

Tabla 2.6: Historia de usuario 03.01 - Crear nueva macro

Aunque el número de acciones que un usuario puede realizar en la plataforma es reducido, este hecho representa precisamente una de las principales fortalezas del servicio. La simplicidad de las operaciones disponibles facilita la experiencia de uso y reduce la curva de aprendizaje, mientras que las posibilidades de automatización que ofrece el sistema permiten abordar un amplio abanico de escenarios.

Capítulo 3

Arquitectura y diseño del sistema

El diseño de la arquitectura de un sistema es un aspecto fundamental que condiciona tanto su funcionamiento interno como su escalabilidad, mantenimiento y facilidad de evolución futura. En este proyecto, se ha optado por una arquitectura distribuida basada en los servicios de Google Cloud, combinando mecanismos de mensajería asíncrona y ejecución de funciones *serverless* para optimizar el uso de recursos.

A lo largo de esta sección se describen las decisiones de arquitectura más relevantes adoptadas durante el desarrollo del proyecto. Se explican las alternativas técnicas que se consideraron en cada caso, los criterios utilizados para seleccionar la solución final, y la forma en la que se han integrado distintos componentes como el seguimiento de correos electrónicos, la gestión de *tokens* de acceso, la autenticación OAuth 2.0 y la autorización incremental de permisos.

La arquitectura propuesta busca lograr un equilibrio entre eficiencia, robustez y facilidad de ampliación, siempre poniendo en el centro las necesidades del usuario final y los requisitos técnicos derivados del uso de APIs externas.

3.1. Visión general de la arquitectura

Una vez completado el proceso de registro, activado el seguimiento del correo y definidas una o más macros por parte del usuario, el sistema queda completamente operativo. A partir de ese momento, cada nuevo correo recibido en la cuenta de Gmail del usuario puede desencadenar automáticamente un flujo de ejecución que aplica distintas automatizaciones, tal y como se muestra en la Figura 3.1.

El flujo comienza cuando se recibe un nuevo mensaje en la bandeja de entrada. Gracias a la suscripción previa al buzón de correo del usuario, este evento genera una notificación en un tema de Google Pub/Sub. Dicha notificación contiene un

identificador, denominado **`historyId`**, que permite determinar los cambios recientes en el estado del buzón.

A raíz de esta notificación, se ejecuta una función desplegada en Cloud Run, encargada de orquestar todo el proceso. En primer lugar, esta función recupera desde Firestore el último **`historyId`** almacenado para el usuario y lo actualiza con el nuevo. A continuación, mediante la API de Gmail, solicita todos los correos electrónicos recibidos desde la última sincronización.

Identificados los nuevos mensajes, el sistema analiza sus etiquetas, ya que cada macro definida por el usuario se activa mediante una o varias etiquetas. Se consulta Firestore para obtener las macros que coinciden con esas etiquetas.

Antes de su ejecución, se aplica una capa de optimización que permite agrupar acciones redundantes. Este mecanismo, descrito en la sección 3.6, permite evitar operaciones duplicadas, reduciendo el uso de recursos y el tiempo de respuesta del sistema.

Una vez optimizadas, las macros se ejecutan. Cada macro puede requerir el uso de uno o varios servicios externos en función de su funcionalidad. Entre ellas encontramos Google Drive para almacenar documentos, Gemini para resumir o analizar texto, Google Calendar para validar disponibilidad ante eventos, entre otros.

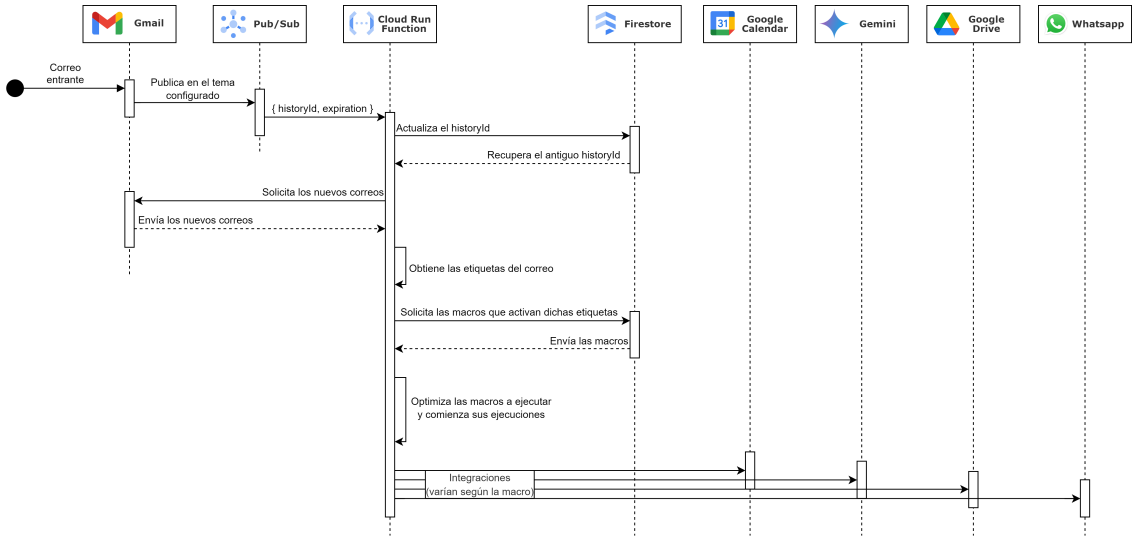

Figura 3.1: Flujo general del sistema tras la recepción de un nuevo correo

3.2. Seguimiento del correo electrónico

El servicio desarrollado se basa en la necesidad de realizar un seguimiento eficiente del correo electrónico de cada usuario. Desde un primer momento se identificaron distintas estrategias posibles para implementar esta funcionalidad, por lo que fue necesario analizar las alternativas disponibles y valorar cuál era la más adecuada para los requisitos del proyecto.

De manera similar a los esquemas que se utilizan en sistemas de sensorización, las opciones principales eran dos: realizar una encuesta periódica (*polling*) o utilizar un sistema basado en interrupciones. El método de *polling* consiste en consultar de forma regular si ha habido cambios, fijando un intervalo de tiempo entre cada verificación. Es un enfoque sencillo de programar, pero resulta ineficiente en términos de consumo de recursos, ya que genera un gran número de consultas innecesarias cuando no hay novedades.

Por el contrario, el modelo basado en interrupciones implica que el sistema permanezca en estado de escucha hasta que se produzca un evento relevante, momento en el que se activa una notificación que desencadena el procesamiento. Aunque más complejo de implementar, este enfoque permite optimizar el uso de recursos y mejorar el tiempo de respuesta ante la llegada de nuevos correos electrónicos.

Tras investigar en la documentación oficial de la API de Gmail, se identificó el método **`watch`** como una solución adecuada para implementar un sistema basado en interrupciones. Este método permite establecer, mediante ciertos parámetros de configuración, la generación de notificaciones *push* cada vez que se recibe un correo que cumple determinadas condiciones.

Las notificaciones generadas por **`watch`** se envían a través del servicio Pub/Sub de Google Cloud.

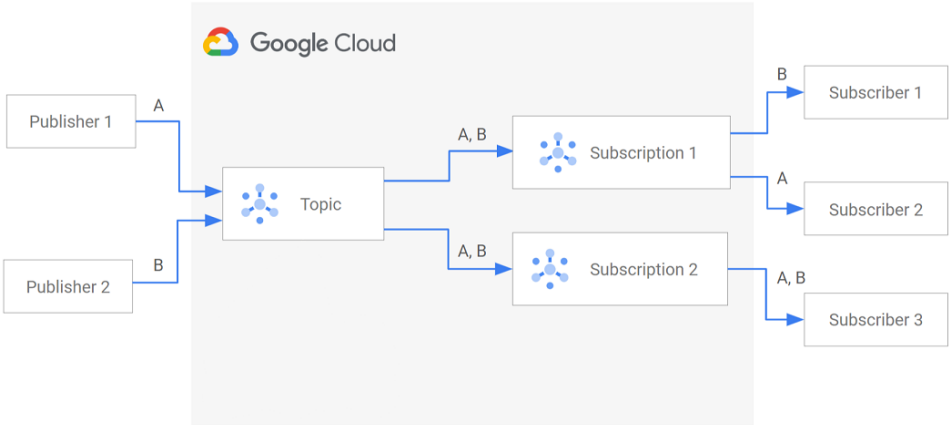

Figura 3.2: Esquema general de funcionamiento de Pub/Sub.[1]

La Figura 3.2 muestra el funcionamiento de este servicio de una forma clara y visual. En este sistema, un productor de eventos (*Pub / Publisher*) publica mensajes en un tema (*topic*), mientras que uno o varios consumidores (*Subs / Subscribers*) se suscriben a dichos temas para recibir las notificaciones correspondientes cada vez que se publica un nuevo evento en ellos.

En el contexto de este proyecto, se ha configurado un único tema de Pub/Sub denominado **`gmail-watch-topic`**. Además, se desplegó una función en la nube (Cloud Run Function) que no se activa mediante peticiones HTTP tradicionales, sino que se desencadena automáticamente al recibir una publicación en dicho tema. De este modo, cada vez que el *Publisher* genera un nuevo mensaje en el tema **`gmail-watch-topic`**, la función se ejecuta, recibiendo el contenido del mensaje como entrada y permitiendo iniciar el procesamiento correspondiente.

3.3. *Token* de refresco y renovación

El método **`watch`**, descrito anteriormente, permite establecer una suscripción al correo electrónico del usuario para recibir notificaciones de nuevos eventos. Sin embargo, su uso introduce ciertas limitaciones que fue necesario gestionar para garantizar el correcto funcionamiento a largo plazo del servicio.

Cada vez que se invoca el método **`watch`**, si la operación resulta exitosa, se recibe una respuesta HTTP con código 200 y un cuerpo de respuesta con la siguiente estructura:

{ **`historyId`**: *string*, **`expiration`**: *string* }

El campo **`historyId`** representa el identificador del estado actual de la bandeja de entrada del usuario. Se trata de un número que únicamente puede incrementarse (aunque no necesariamente de forma secuencial) y que cambia cada vez que se produce un evento relevante, como la recepción, eliminación o movimiento de un correo.

Por su parte, el campo **`expiration`** indica el momento en el que la suscripción caducará si no se renueva. Esta fecha se expresa en formato de tiempo Unix y, por defecto, establece una validez máxima de siete días desde la activación del **`watch`**. Google recomienda, como buena práctica, que las renovaciones se realicen de manera diaria para evitar riesgos de expiración inesperada.

Dado que se requería un sistema autónomo, en el que el usuario no tuviera que intervenir manualmente para mantener activo el servicio, fue necesario diseñar un

[1]Fuente: `https://cloud.google.com/pubsub/docs/pubsub-basics?hl=es-419`

mecanismo de renovación automática del `watch`. Para ello, era imprescindible contar con credenciales que permitieran realizar nuevas llamadas autenticadas a la API de Gmail en nombre del usuario incluso en su ausencia.

Este requisito de autenticación continua planteó un desafío específico durante el proceso de autorización mediante OAuth 2.0. Si no se especifica explícitamente el tipo de acceso, el comportamiento por defecto es otorgar un acceso de tipo *online*. En este modo, la aplicación recibe únicamente un *access token*, que permite realizar peticiones autenticadas durante un periodo de tiempo muy limitado y, generalmente, con un único uso permitido. Esta restricción resulta insuficiente para un sistema que requiere operaciones recurrentes en segundo plano, sin intervención activa del usuario.

Para solucionar esta limitación, se optó por solicitar un acceso de tipo *offline*. En este escenario, además del *access token* de corta duración, se obtiene un *refresh token*, un elemento fundamental para mantener la continuidad del servicio. El *refresh token* puede considerarse como un generador de nuevos *access tokens*, permitiendo que la aplicación obtenga credenciales válidas de forma indefinida, siempre que el servicio necesite realizar una acción en nombre del usuario.

Una vez establecido este flujo, el sistema guarda el *refresh token* asociado a cada usuario en la base de datos Firestore. Posteriormente, se implementó una Cloud Run Function denominada `/watch-renew`, encargada de renovar periódicamente las suscripciones activas. La ejecución de esta función está programada mediante el servicio Cloud Scheduler, que permite definir tareas recurrentes siguiendo la sintaxis de unix-cron, una sintaxis compuesta por cinco campos que representan `minuto`, `hora`, `día del mes`, `mes` y `día de la semana`, respectivamente. En este caso, se configuró la expresión (0 0 * * *), que indica una ejecución diaria a las 00:00. De este modo, todas las suscripciones activas se renuevan automáticamente cada veinticuatro horas, sin requerir intervención del usuario, y respetando las buenas prácticas recomendadas por Google.

3.4. Cliente OAuth 2.0 y permisos graduales

Para gestionar la autenticación y el acceso a los servicios de Google, este proyecto utiliza el protocolo OAuth 2.0, que permite a las aplicaciones obtener acceso limitado a las cuentas de usuario de forma segura y controlada.

Según se recoge en la documentación oficial de OAuth 2.0 de Google, el proceso de autorización sigue una secuencia bien definida: la aplicación redirige al navegador del usuario a una URL de Google que contiene parámetros específicos, como el tipo de acceso y los permisos solicitados. Google gestiona la autenticación del usuario, la selección de la cuenta y el consentimiento de los permisos. Como resultado, se genera un código de autorización que la aplicación puede intercambiar posteriormente por

un *access token* y, en caso de haber solicitado acceso de tipo *offline*, también por un *refresh token*.

La aplicación debe almacenar el *refresh token* de forma segura para su uso futuro. El *access token* se utilizará para acceder a las APIs de Google, y, una vez que expire, se podrá utilizar el *refresh token* para obtener uno nuevo automáticamente, evitando que el usuario tenga que autenticarse de nuevo.

La Figura 3.3 ilustra de forma simplificada el flujo completo de autorización utilizando el protocolo OAuth 2.0.

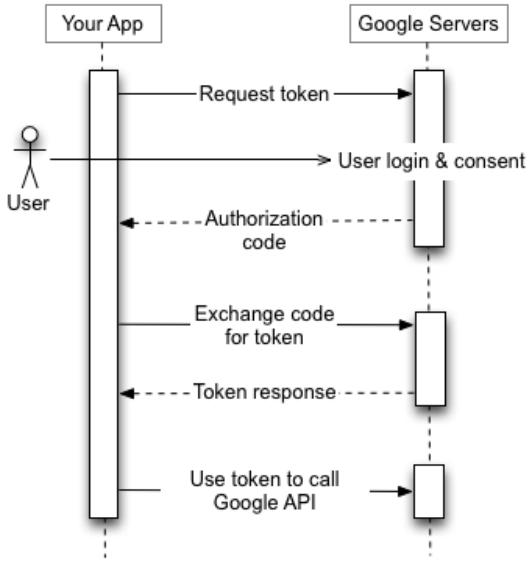

Figura 3.3: Flujo de autorización en OAuth 2.0.[2]

A partir de este modelo estándar, el proyecto implementa el flujo de autenticación mediante dos *endpoints* específicos que permiten integrarlo de manera controlada y segura dentro de la arquitectura del servicio.

En primer lugar, desde el cliente web se realiza una petición al *endpoint* `/auth-url`, enviando en el cuerpo de la solicitud los permisos o *scopes* que se desean solicitar. Este *endpoint* utiliza las credenciales configuradas para generar un cliente OAuth 2.0 asociado al servicio.

Cada cliente OAuth 2.0 tiene definida previamente una lista cerrada de *scopes* permitidos, establecida durante su configuración. Antes de generar la URL de autorización, el *endpoint* verifica que los *scopes* solicitados en la petición estén incluidos en esa lista de *scopes* autorizados para el cliente. De este modo, se evita que la aplicación solicite permisos no previstos o excesivos.

[2] `https://developers.google.com/identity/protocols/oauth2?hl=es-419#scenarios`

Si los *scopes* solicitados son válidos, se genera y devuelve un objeto que contiene la URL de autorización personalizada a la que el usuario deberá acceder para otorgar los permisos. Dicho objeto tiene la forma:

{ **data**: { **url**: *string* }; **message**: *string* }

Una vez recibida la URL, en el cliente se abre una nueva ventana emergente (*pop-up*) en la que el usuario puede completar el proceso de autorización. Tras aceptar los permisos, Google redirige al usuario a una URL previamente configurada en el cliente OAuth 2.0, añadiendo el código de autorización como parámetro.

En este proyecto, la URL de redirección configurada es `URL_SERVICIO/redirect`, una ruta específica gestionada en el sistema de *routing* de la aplicación Angular. Esta ruta está asociada a un componente sencillo cuya única funcionalidad consiste en extraer automáticamente el código de autorización de la URL y enviarlo de vuelta a la ventana principal de la aplicación, permitiendo así completar el proceso de autenticación de forma transparente para el usuario.

Posteriormente, el cliente realiza una segunda llamada, enviando el código recibido, al *endpoint* `URL_API/auth-token`. Este *endpoint* valida el código de autorización con los servidores de Google y, si es correcto, intercambia el código por un conjunto de *tokens*. Este conjunto incluye el *access token*, el *refresh token* y la fecha de expiración del primero. El sistema almacena el *refresh token* en la base de datos Firestore para permitir futuras renovaciones automáticas del acceso, y devuelve al cliente un objeto, con los *tokens* y el identificador de usuario, de la forma:

{ **data**: { **tokens**: *Tokens*; **userId**: *string* }; **message**: *string* }

En resumen, el *endpoint* `URL_API/auth-url` se encarga de generar la URL de autorización dados unos *scopes* determinados, y el *endpoint* `URL_API/auth-token` se ocupa de validar el código de autorización, obtener los *tokens*, almacenarlos adecuadamente y retornar la información necesaria para la continuidad del servicio.

El diseño de estos dos *endpoints* es intencionadamente genérico, lo que permite solicitar permisos adicionales de forma incremental conforme se vayan necesitando, aplicando las mejores prácticas en lo que a otorgación de permisos se refiere.

Este enfoque, denominado autorización incremental, consiste en solicitar únicamente los permisos estrictamente necesarios en cada momento, en lugar de pedir todos los permisos posibles desde el inicio del registro del usuario. Por ejemplo, inicialmente el servicio puede solicitar únicamente permisos de lectura de correo, y, en caso de que más adelante se requiera acceso a archivos de Google Drive, solicitar entonces los permisos pertinentes de manera diferenciada.

Además, la correcta definición de los *scopes* resulta esencial para limitar adecuadamente el nivel de acceso otorgado a la aplicación. En el caso concreto de Google Drive, existen más de trece *scopes* específicos, que permiten definir niveles de acceso como lectura, escritura o administración completa. Así, si el objetivo es únicamente visualizar archivos, bastará con solicitar el *scope* de lectura, reduciendo de esta manera la exposición innecesaria de información sensible.

3.5. Diseño de la base de datos

La persistencia de la información en el proyecto se gestiona mediante Cloud Firestore, una base de datos NoSQL orientada a documentos ofrecida como parte del ecosistema de Google Cloud a través de Firebase. La elección de Firestore responde a varios factores estratégicos tales como su elevada escalabilidad, la facilidad de integración con otros servicios de Google y su modelo flexible de almacenamiento, que se adapta perfectamente a las necesidades de una aplicación en fase de crecimiento.

Una de las principales características de Firestore es su capacidad para permitir accesos directos desde aplicaciones cliente, gracias a su sistema de reglas de seguridad basado en condiciones de autenticación y validación de datos. No obstante, en este proyecto se ha optado por un enfoque de tres capas, donde el *back-end* actúa como intermediario entre el *front-end* y la base de datos. Esta decisión responde a criterios de seguridad y control de acceso, permitiendo centralizar la lógica de validación y minimizar la exposición directa de Firestore a usuarios externos. En futuras iteraciones, se podría optimizar aún más la arquitectura implementando reglas avanzadas que permitan delegar ciertas operaciones directamente al cliente de forma segura.

Firestore organiza sus datos en una estructura jerárquica basada en:

- **Colecciones**: conjuntos de documentos que agrupan información de una misma naturaleza.
- **Documentos**: unidades básicas de almacenamiento, compuestas por pares clave-valor, similares al formato JSON.
- **Subcolecciones**: colecciones anidadas dentro de documentos, que permiten representar relaciones más complejas.

Como podemos ver en la Figura 3.4, la estructura general de la base de datos es la siguiente. Cada usuario registrado en el sistema se representa mediante un documento único dentro de la colección `users`. Este documento almacena información relevante como el `refresh_token` asociado a la sesión OAuth, el estado de activación del servicio de seguimiento de correos (`watch`) y la colección de macros personalizadas que el usuario haya creado. A su vez, cada macro incluye una subcolección de etiquetas,

que permiten asociar la ejecución de acciones a determinadas categorías de correos electrónicos.

Esta estructura de base de datos está diseñada para favorecer la escalabilidad horizontal, permitiendo añadir nuevas clases de macros o integraciones con más servicios sin necesidad de rediseñar la arquitectura existente.

Estructura de la base de datos

```
users (colección)
 │
 ├─ userID_1 (documento)
 └─ userID_2 (documento)
      │
      ├─ refresh_token: '1//03xDRpm....5kw5jPSoI'
      │
      ├─ watch
      │    ├── enabled: true
      │    ├── expiration: '2025-04-28T19:51:24.601Z'
      │    └── historyId: 1348647
      │
      └── macros (subcolección)
           ├── macroID_1 (documento)
           └── macroID_2 (documento)
                │
                ├── name: 'Macro1'
                ├── content
                │     ├── service: 'google-drive'
                │     └── type: 'attachment'
                │
                └── labels (subcolección)
                      ├── labelID_1 (documento)
                      └── labelID_2 (documento)
                            │
                            ├── id: 'jdkasd98h2'
                            └── name: 'custom_name'
```

Figura 3.4: Estructura jerárquica de documentos en Firestore

3.6. Optimización del procesamiento de macros

Tal y como se muestra en la Figura 3.5, cuando un correo electrónico llega a la cuenta del usuario, puede activar varias macros de forma simultánea. Ejecutar cada una de ellas por separado puede resultar ineficiente y redundante, sobre todo cuando realizan operaciones similares o que implican un alto coste computacional, como la generación de resúmenes mediante modelos de lenguaje.

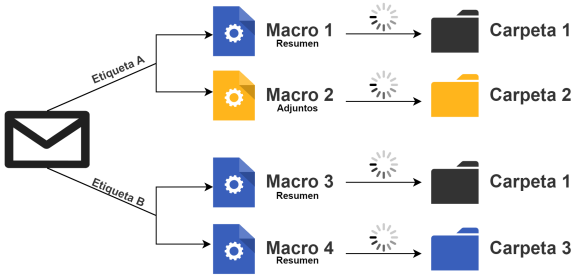

Figura 3.5: Ejecución no optimizada de las macros

Para abordar este problema, se ha implementado una estrategia de optimización que agrupa y fusiona las macros activadas por un mismo correo electrónico. Esta optimización, ilustrada en la Figura 3.6, permite reducir drásticamente el número de operaciones redundantes y aumentar la eficiencia del sistema.

El flujo optimizado comienza igualmente con la recepción de un nuevo mensaje. A partir de las etiquetas que este contiene, se detectan las macros activadas. Sin embargo, antes de proceder a su ejecución, las macros se agrupan por tipo de acción (por ejemplo, “generar resumen”, “extraer adjuntos”, etc.). Esta agrupación permite detectar operaciones repetidas sobre el mismo contenido, evitando así la ejecución múltiple de tareas idénticas.

El resultado generado por cada acción se reutiliza para todas las macros implicadas. Por ejemplo, si varias macros solicitan guardar un mismo archivo en distintas carpetas, el sistema realiza una única extracción del adjunto y genera una operación de escritura en lote que lo distribuye automáticamente a todos los destinos requeridos.

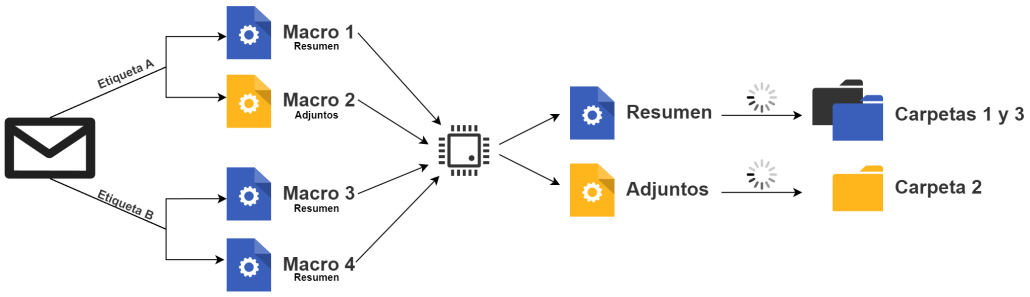

Figura 3.6: Agrupación y ejecución optimizada de las macros

3.7. Integraciones externas

El desarrollo de esta aplicación ha requerido la integración con múltiples servicios externos, principalmente del ecosistema de Google, lo que ha supuesto tanto un reto técnico como una oportunidad para explorar el potencial de sus herramientas. A continuación, se describen las integraciones más relevantes, su propósito dentro del sistema y las principales dificultades encontradas durante su implementación.

3.7.1. Gmail

Gmail constituye el núcleo funcional sobre el que gira la lógica de esta aplicación. A través de su API, se monitorizan los correos electrónicos entrantes del usuario, analizando las etiquetas asociadas a cada mensaje para determinar si activan alguna de las macros configuradas. Esta lógica de activación, basada en etiquetas, es la base sobre la que se articulan las automatizaciones.

Para recibir notificaciones de nuevos mensajes, se emplea el método **`watch`**, que activa la monitorización del buzón mediante Google Pub/Sub. Cada vez que se detecta un cambio, Gmail publica una notificación que incluye un identificador único llamado **`historyId`**, el cual representa el estado más reciente del buzón. A partir de dicho identificador, es posible consultar los mensajes modificados desde el último evento utilizando el *endpoint* **`history.list`**.

Uno de los principales desafíos técnicos fue interpretar correctamente la estructura interna de los mensajes. El contenido de un correo puede presentarse en múltiples formatos y su estructura varía dependiendo de si el mensaje contiene archivos adjuntos. Para manejar esta variabilidad, se desarrolló una lógica específica que analiza la jerarquía de partes del mensaje (**`message.payload.parts`**), permitiendo extraer de forma precisa tanto el contenido textual como los adjuntos:

- Si el mensaje no contiene adjuntos:
  - **`parts[0]`** contiene el texto plano.
  - **`parts[1]`** contiene el contenido HTML.
- Si el mensaje incluye adjuntos:
  - **`parts[0].parts[0]`** contiene el texto plano.
  - **`parts[0].parts[1]`** contiene el HTML.
  - **`parts[1 ... n]`** contienen los distintos archivos adjuntos.

Además, fue necesario implementar funcionalidades adicionales para recuperar dinámicamente el conjunto de etiquetas disponibles en la cuenta del usuario. Esta

información se utiliza en el *front-end* para alimentar un menú desplegable que facilita la selección de etiquetas durante el proceso de configuración de nuevas macros.

Esta integración supuso un importante esfuerzo de análisis y pruebas, dada la complejidad de los datos devueltos por la API y su sensibilidad a pequeñas variaciones estructurales.

3.7.2. Google Drive

Google Drive actúa como sistema de almacenamiento externo y es utilizado por distintas macros para guardar tanto archivos adjuntos de correos electrónicos como resúmenes generados a partir de su contenido. Gracias a esta integración, el usuario puede seleccionar desde la interfaz de la aplicación cualquiera de sus carpetas disponibles en Drive como destino final para almacenar estos elementos.

Uno de los retos más significativos en esta integración fue conservar el formato original del contenido de los correos electrónicos (estilos como negrita, enlaces, listas, colores, etc.). Para lograrlo, se extrajo el cuerpo del mensaje en formato `HTML`, asegurando así que toda la información enriquecida pudiera ser representada fielmente. Posteriormente, se llevó por un proceso de conversión dinámica de `HTML` a documentos de tipo `.doc`, que son los que finalmente se suben a Drive. Esta conversión garantiza que tanto la estructura como la apariencia original del mensaje se mantengan en el archivo generado.

Adicionalmente, como se detalla en la sección 3.6, se implementó un mecanismo de operaciones en lote que permite almacenar un mismo archivo en múltiples carpetas cuando es requerido por varias macros simultáneamente. Esta optimización no solo reduce el número de llamadas a la API de Drive, sino que también mejora significativamente el rendimiento del sistema y disminuye los costes computacionales asociados.

3.7.3. Gemini

La integración de capacidades de inteligencia artificial ha sido una de las claves diferenciales de esta aplicación. Se realizó un estudio comparando diferentes modelos de lenguaje, incluyendo versiones de GPT (OpenAI) [3], modelos desarrollados por Google como Gemini [5] e incluso opciones de código abierto como OLLama [1]. Además, se diseñó una arquitectura flexible, capaz de integrar diferentes modelos de IA con facilidad para no depender de un modelo concreto. Sin embargo, se optó finalmente por utilizar **Gemini 2.0 Flash Lite**. Esta decisión se basó en el punto de equilibrio que ofrece entre velocidad de respuesta, precisión semántica y coste de generación de *tokens*, convirtiéndolo en una opción especialmente adecuada para

tareas como:

- Resumir correos electrónicos extensos, extrayendo sus ideas principales.
- Identificar eventos y reuniones descritos en lenguaje natural.

Cada tipo de macro basada en IA se configura con parámetros específicos adaptados a su objetivo. En el caso de la generación de resúmenes, se emplea un *prompt* detallado que incluye instrucciones sobre el nivel de compresión deseado (entre un 50 y un 70 % del contenido original), así como requerir el uso de un tono informativo y conciso.

La tarea de extracción de eventos es significativamente más compleja, especialmente debido a la ambigüedad temporal presente en muchos correos electrónicos. Frases como "mañana a las 11", "el jueves por la tarde" o "dentro de dos semanas" requieren que el modelo comprenda el contexto temporal del mensaje, incluyendo la fecha en que se escribió el correo y los husos horarios del usuario.

Para resolver este reto, se diseñó un *prompt* complejo que incluye ejemplos positivos y negativos, así como instrucciones detalladas que guían al modelo para interpretar la información de forma coherente. Además, fue configurado con un esquema en formato JSON que define la estructura que debe seguir su respuesta, de modo que esta pueda ser fácilmente manejada por el sistema, posteriormente.

Por último, se realizó un ajuste fino de diferentes hiperparámetros como la Temperatura, Top-K o Top-P, los cuales definen el grado de creatividad, diversidad y relevancia del modelo. Tal y como señalan Renze y Guven (2024), estos parámetros influyen directamente en la capacidad de los modelos de lenguaje para resolver problemas, afectando el equilibrio entre exploración y coherencia en las respuestas generadas. Tras múltiples iteraciones, se alcanzó una configuración equilibrada que permite obtener resultados consistentes en tiempos razonables, manteniendo la calidad necesaria para su explotación en contextos reales [2].

3.7.4. Google Calendar

La integración con Google Calendar desempeña un papel clave en la automatización de la gestión de eventos extraídos de correos electrónicos. En concreto, su funcionalidad se activa una vez que el sistema ya ha transformado la información del mensaje en uno o varios rangos temporales normalizados, previamente definidos por el modelo de lenguaje Gemini.

A partir de estos rangos, se realiza una petición al *endpoint* `calendar.freebusy` de la API de Google Calendar, con el objetivo de verificar si en ese intervalo de tiempo

el usuario ya tiene alguna actividad registrada en su calendario. Esta verificación de disponibilidad permite detectar solapamientos con otros eventos existentes.

Si se detecta que el intervalo propuesto está libre, el sistema marca la sugerencia como viable, permitiendo que el usuario acepte la propuesta. En caso de conflicto, se le indica el solapamiento, lo que permite al usuario evaluar si desea reprogramar, rechazar o contactar al emisor.

Este flujo de trabajo no solo automatiza la gestión de compromisos, sino que también aporta valor contextual, ayudando al usuario a tomar decisiones más informadas y en menor tiempo. La lógica se ha diseñado para ser extensible, permitiendo en un futuro no solo validar disponibilidad, sino también proponer horarios alternativos o gestionar directamente la invitación de asistentes mediante la API de Calendar.

3.7.5. WhatsApp

Aunque no se ha implementado de forma definitiva, se exploró la posibilidad de integrar la aplicación con WhatsApp mediante su API oficial WhatsApp Business. Esta integración permitiría enviar notificaciones automáticas en situaciones críticas, como la detección de correos urgentes o la propuesta de reuniones.

Sin embargo, la integración presenta varios retos, ya que se requiere una cuenta verificada, un número asociado al servicio y la aprobación de Meta para el uso de plantillas de mensajes. Además, existen costes por mensaje enviado, lo que limita su uso en proyectos académicos. Por estos motivos, la funcionalidad ha sido diseñada pero permanece desactivada, esperando una futura implementación en caso de evoluciones comerciales del proyecto.

Capítulo 4

Deasarrollo e implementación

Una vez definidos los requisitos y diseñada la arquitectura del sistema, se procedió a la fase de implementación, en la que se materializó el desarrollo de la aplicación tanto en su parte de servidor (*back-end*) como en la parte cliente (*front-end*). Esta etapa ha supuesto la puesta en práctica de las decisiones técnicas previamente analizadas, integrando tecnologías modernas y servicios en la nube para lograr un producto funcional, escalable y fácilmente mantenible.

Además, el desarrollo del proyecto ha estado apoyado desde el inicio en el uso de control de versiones mediante Git, una herramienta ampliamente consolidada que permite gestionar de forma eficiente el historial de cambios, coordinar el trabajo entre distintos entornos y mantener la integridad del código fuente.

Para ello, se creó un repositorio principal alojado en la plataforma GitHub, accesible públicamente a través de la URL: `https://github.com/daavid6/neoinbox`. Este repositorio contiene la última versión del proyecto, estructurada en carpetas para diferenciar los distintos componentes (angular-web y cloud-functions).

Durante las primeras fases del proyecto, se trabajó en un repositorio provisional que servía como entorno de pruebas. Posteriormente, una vez definida la arquitectura y establecido el nombre definitivo del servicio (NeoInbox), se migró todo aquel trabajo útil al repositorio actual. Esta separación permitió experimentar libremente durante las fases iniciales sin comprometer la estructura final.

En cuanto a la estrategia de trabajo con Git, el desarrollo se realizó principalmente sobre una única rama principal (**`main`**), realizando *commits* frecuentes que recogían los avances incrementales de cada funcionalidad. Aunque esta forma de trabajo ha sido válida para el desarrollo en solitario, a lo largo del proyecto se identificaron algunas prácticas que podrían mejorarse en futuros desarrollos, especialmente si se trabaja en equipo.

Una de ellas sería la adopción de un modelo basado en ramas, por ejemplo, utilizando una rama por funcionalidad o tarea. Este enfoque, combinado con una metodología ágil como Scrum, permitiría dividir mejor el trabajo en *sprints* y facilitar tanto la revisión como la integración continua. Asimismo, en caso de trabajar con múltiples entornos, el uso de ramas de desarrollo, pruebas y producción facilitaría la detección de errores antes del despliegue.

Cabe destacar que, al mantener siempre el repositorio sincronizado con GitHub, fue posible trabajar desde distintos dispositivos y localizaciones sin pérdida de información. Esta característica ha resultado especialmente útil en un proyecto desarrollado en distintas ubicaciones (universidad, casa, empresa), y ha contribuido a mantener una copia de seguridad continua y actualizada del proyecto.

En definitiva, Git y GitHub han constituido elementos clave durante todo el proceso de desarrollo, facilitando no solo la gestión del código fuente, sino también la organización, la trazabilidad y el mantenimiento del proyecto.

A continuación, se detallan las principales características del desarrollo, comenzando por el *back-end* del sistema.

4.1. *Back-end*

El *back-end* del proyecto ha sido concebido como el núcleo lógico de la plataforma, encargado de gestionar la comunicación con las APIs de Google, procesar las peticiones provenientes del cliente, interactuar con la base de datos y garantizar la correcta ejecución de las automatizaciones definidas por los usuarios.

Para su desarrollo se ha utilizado Node.js en combinación con el framework Express.js, desplegando los servicios en Google Cloud mediante tecnologías *serverless*. El diseño de la API, la estrategia de despliegue y las herramientas de documentación han sido elegidas para ofrecer un servicio eficiente, escalable y fácilmente integrable con el *front-end* de la aplicación.

4.1.1. Google Cloud

El despliegue de la API desarrollada en este proyecto se ha apoyado en los servicios de Google Cloud Platform, aprovechando las ventajas que ofrece la computación en la nube en términos de escalabilidad, disponibilidad y facilidad de integración con otros servicios. A lo largo del desarrollo, la arquitectura de despliegue evolucionó a través de distintas fases, conforme se identificaban necesidades específicas de rendimiento, mantenimiento y optimización de recursos.

Fase 1: Desarrollo local con Node.js y Express.js

En la etapa inicial, la API se desarrolló y ejecutó en un entorno local, utilizando Node.js junto con el *framework* Express.js. Esta aproximación permitía iterar de forma ágil sobre las funcionalidades, realizar pruebas inmediatas y depurar el código de manera sencilla, sin la complejidad añadida de desplegar en un entorno en la nube.

Además, trabajar de forma local proporcionaba la tranquilidad de no incurrir en costes asociados al consumo de servicios en la nube, especialmente relevantes en etapas tempranas donde los ciclos de prueba y error son frecuentes. Sin embargo, esta solución presentaba limitaciones inherentes, como la falta de un entorno de ejecución persistente para mantener levantada la API.

Fase 2: Despliegue de una Cloud Run Function por cada *end-point*

El primer paso hacia la nube consistió en migrar la API a Google Cloud Functions, desplegando cada *endpoint* como una función independiente. Este enfoque aprovechaba el modelo *serverless*, donde el proveedor gestiona automáticamente la infraestructura subyacente, escalando las funciones en base a la demanda y reduciendo los costes en periodos de inactividad.

No obstante, surgieron varios inconvenientes importantes. El primero fue el denominado *cold start*, un fenómeno característico de las funciones *serverless*, donde, tras un periodo de inactividad (quince minutos en este caso), la función entra en un estado de suspensión para ahorrar recursos y debe ser reactivada antes de atender nuevas peticiones, introduciendo una latencia adicional. En servicios con tráfico constante esto puede no ser un problema, pero dado el bajo tráfico actual del servicio, este retardo era frecuente. Como cada *endpoint* era una función independiente, el *cold start* ocurría por separado en cada uno, multiplicando el tiempo de espera, lo que afectaba negativamente a la experiencia de usuario.

El segundo problema estaba relacionado con la gestión del código. Cada función contenía una copia completa del proyecto, lo que generaba redundancia, un consumo ineficiente de memoria y almacenamiento, y una mayor complejidad en el proceso de actualización, al requerir el despliegue individual de múltiples funciones ante cualquier cambio.

Estas limitaciones evidenciaron la necesidad de replantear la arquitectura hacia un modelo más eficiente y mantenible.

Fase 3: Unión en una única Cloud Run Function

La solución adoptada consistió en juntar todos los *endpoints* de la API dentro de un único servidor Node.js, desplegado como una Cloud Run Function. Este servidor, basado en Express.js, se encarga de enrutar internamente cada petición a su controlador correspondiente.

Esta nueva arquitectura resolvió los problemas detectados en fases anteriores. Al centralizar la lógica en una sola instancia, se reduce significativamente el impacto del *cold start*, ya que solo existe un único servicio que permanece activo para atender todas las solicitudes. Además, se elimina la duplicidad de código, optimizando el uso de recursos y simplificando el proceso de despliegue, ya que ahora, cualquier cambio en la API requiere únicamente la actualización de una única función.

Otra ventaja importante es la posibilidad de ejecutar el servidor de forma local con la misma estructura que en producción, lo que facilita el desarrollo, la depuración y la implementación de pruebas automatizadas antes del despliegue definitivo.

En conjunto, esta evolución progresiva hacia un modelo basado en Cloud Run Functions ha permitido construir una infraestructura robusta, eficiente y alineada con las mejores prácticas de desarrollo de aplicaciones web modernas en la nube.

4.1.2. Diseño de la API REST

Durante el desarrollo del *back-end* del proyecto, una de las primeras decisiones arquitectónicas fue la selección del estilo de comunicación entre el cliente (*front-end*) y el servidor. Existen diversas alternativas para estructurar una API, cada una con ventajas e inconvenientes en función del tipo de aplicación y sus requisitos específicos, por lo que resultaba fundamental escoger la opción más adecuada para el sistema a desarrollar.

Entre las alternativas consideradas se encontraba SOAP, un protocolo tradicional ampliamente utilizado en entornos corporativos que destaca por su robustez, pero cuya complejidad resultaba excesiva para una aplicación web moderna centrada en tareas concretas. También se valoraron tecnologías más recientes como GraphQL, que permite al cliente especificar exactamente los datos que necesita, y gRPC, que ofrece una alta eficiencia y velocidad de comunicación, especialmente orientada a arquitecturas de microservicios. Sin embargo, tanto GraphQL como gRPC implicaban una curva de aprendizaje adicional, la necesidad de herramientas específicas para su consumo y una complejidad que no se justificaba en este proyecto, especialmente considerando que el *front-end* ya estaba definido en Angular y que el ecosistema web se integra de manera natural con APIs REST.

Por todo ello, se optó por una arquitectura basada en REST, que ofrece una implementación sencilla mediante el *framework* Express.js y una estructuración clara de los recursos y las operaciones. Con este modelo, cada funcionalidad del sistema se expone a través de un *endpoint* accesible mediante los métodos estándar del protocolo HTTP, como `GET`, `POST` o `DELETE`. Este planteamiento no solo simplifica el desarrollo y el mantenimiento, sino que también facilita las pruebas y la validación del sistema, permitiendo interactuar con la API mediante herramientas como Postman, Swagger UI o incluso directamente a través del navegador.

La API desarrollada expone una serie de *endpoints* que permiten gestionar las principales funcionalidades del sistema, incluyendo la activación y desactivación del servicio, la creación, consulta y eliminación de macros, y el proceso de autenticación con Google. Un ejemplo representativo es el *endpoint* `POST /watch-enable`, cuya función es activar el seguimiento de correos electrónicos en la cuenta del usuario. Este *endpoint* recibe un cuerpo JSON que incluye los campos **`userId`**, **`historyId`** y **`expiration`**. El campo **`userId`** identifica de forma única al usuario en la base de datos, **`historyId`** indica el punto a partir del cual Gmail debe comenzar a detectar cambios en la bandeja de entrada, y **`expiration`** establece la fecha límite del seguimiento, expresada en tiempo Unix. Una vez procesada la petición, el *back-end* actualiza la información correspondiente en Firestore y responde confirmando la operación.

Para organizar de forma lógica los diferentes *endpoints*, se adoptó una convención de prefijado en las rutas. Así, las rutas relacionadas con el seguimiento de correos comienzan por `/watch-`, las de autenticación por `/auth-` y las correspondientes a la gestión de macros por `/macro-`. Esta estrategia de agrupación, aunque sencilla, ha resultado eficaz para mantener el código ordenado y facilitar la identificación rápida de la funcionalidad asociada a cada *endpoint*. De hecho, esta aproximación podría escalar fácilmente hacia una arquitectura basada en módulos o controladores separados en proyectos de mayor envergadura.

Una de las ventajas más destacables de haber seguido un diseño REST es la facilidad de integración con el *front-end* desarrollado en Angular. La existencia de operaciones claramente definidas como recursos o acciones ha permitido crear de forma sencilla servicios en el cliente que consumen la API de manera eficiente.

Adicionalmente, todos los *endpoints* siguen una estructura de respuesta coherente, diseñada para facilitar su consumo desde el *front-end* y mejorar la legibilidad del código. En caso de éxito, la respuesta contiene un objeto JSON con dos campos principales: **`data`**, que incluye los datos devueltos, y **`message`**, un texto breve asociado al estado de la operación. En caso de error, la respuesta incluye un objeto con los campos **`error`**, indicando el código de error, y **`errorMessage`**, que proporciona una descripción detallada del problema. Esta convención uniforme ha resultado clave para simplificar el tratamiento de errores en el cliente y mantener una comunicación clara, predecible y robusta entre el *front-end* y el *back-end*.

4.1.3. Documentación de la API con Swagger

En el desarrollo de una API no basta únicamente con garantizar su correcto funcionamiento. Es fundamental que esté bien documentada, especificando de manera precisa su comportamiento, los datos de entrada esperados, las respuestas posibles y la forma correcta de interactuar con ella. Una documentación clara y actualizada no solo facilita la tarea a futuros desarrolladores que trabajen sobre el sistema, sino que

también resulta crucial durante el propio desarrollo para realizar pruebas, validar comportamientos y mantener la coherencia a medida que la aplicación evoluciona.

Con el objetivo de cubrir esta necesidad, se decidió incorporar Swagger, una herramienta ampliamente utilizada para la documentación de APIs REST de manera dinámica e interactiva. Swagger permite describir de forma estructurada cada *endpoint* en un archivo de configuración (en este caso, `swagger.json`), donde se especifican, entre otros aspectos, el método HTTP utilizado, los parámetros esperados, el formato de las respuestas y los posibles errores.

A partir de esta definición, se genera automáticamente una interfaz web accesible desde el *endpoint* `URL_API/docs`, utilizando el *middleware* `swagger-ui-express`. Esta interfaz no solo permite consultar la documentación de manera visual e intuitiva, sino también interactuar directamente con la API, enviando peticiones de prueba y obteniendo respuestas en tiempo real, lo que agiliza significativamente las tareas de validación y depuración.

La Figura 4.1 muestra una captura de la documentación generada mediante Swagger en el proyecto.

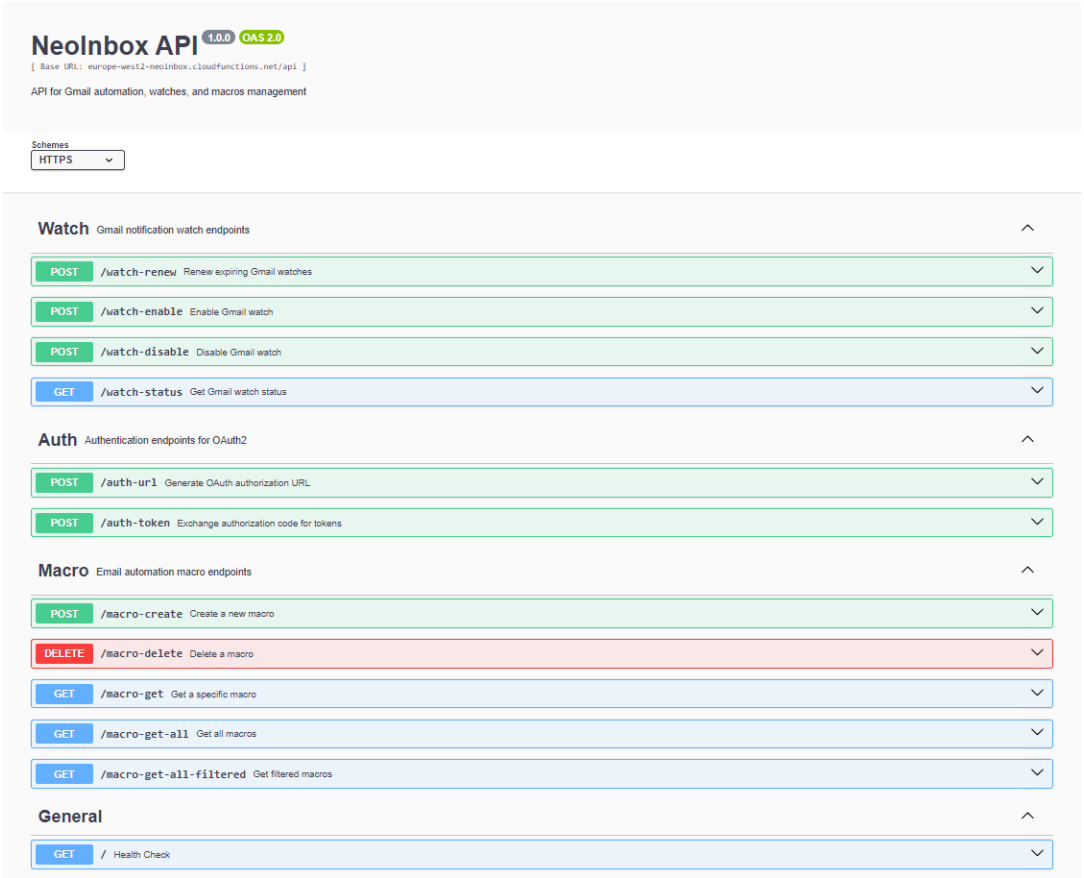

Figura 4.1: Documentación generada con Swagger

La integración de Swagger ha supuesto una mejora sustancial en la calidad del *back-end*, proporcionando una referencia centralizada, clara y siempre sincronizada con el código fuente. Además, su presencia facilitará la futura escalabilidad y el mantenimiento del sistema, ya que cualquier nuevo desarrollador podrá comprender de forma rápida el funcionamiento y las especificaciones de cada *endpoint*, reduciendo el riesgo de errores y mejorando la eficiencia del trabajo en equipo.

4.2. *Front-end*

El desarrollo del lado cliente ha sido concebido con el objetivo de ofrecer una experiencia de usuario fluida, accesible y visualmente coherente. Se ha optado por una arquitectura moderna basada en tecnologías web actuales, que permite estructurar la aplicación de forma modular, facilitar el mantenimiento y asegurar su escalabilidad. En los siguientes apartados se detallan aspectos clave del diseño de la interfaz, así como los mecanismos implementados para garantizar la seguridad y consistencia en la navegación.

4.2.1. Interfaz

La interfaz de usuario de la aplicación ha sido desarrollada utilizando Angular 19, un *framework* moderno y robusto que permite construir aplicaciones web dinámicas y escalables. Angular ofrece un enfoque basado en componentes reutilizables, enrutamiento integrado y soporte completo para TypeScript, lo cual aporta una gran seguridad a nivel de tipado y ayuda a detectar errores en tiempo de desarrollo.

En este proyecto se ha seguido una organización modular basada en la separación de responsabilidades entre componentes y servicios, tal como promueve la arquitectura de Angular. Aunque debido al tamaño reducido de la aplicación muchos componentes no requerían reutilización, se ha procurado, sobre todo, mantener una estructura clara y escalable que facilite futuras expansiones o refactorizaciones.

Para la capa visual se ha empleado Angular Material, una biblioteca de componentes UI desarrollada por Google basada en los principios de Material Design. Esto ha permitido construir una interfaz moderna, intuitiva y alineada con los principios fundamentales del diseño de sistemas de información.

Además, durante el diseño y desarrollo del proyecto se tuvo en cuenta que la aplicación debía ser completamente *responsive*, con el objetivo de ofrecer una experiencia de usuario óptima en el mayor número posible de dispositivos. Como se puede observar en la Figura 4.2, la interfaz se adapta tanto a pantallas de ordenador como a dispositivos móviles, ajustando el diseño, los elementos visuales y la disposición del contenido para garantizar una navegación cómoda y accesible en cualquier entorno.

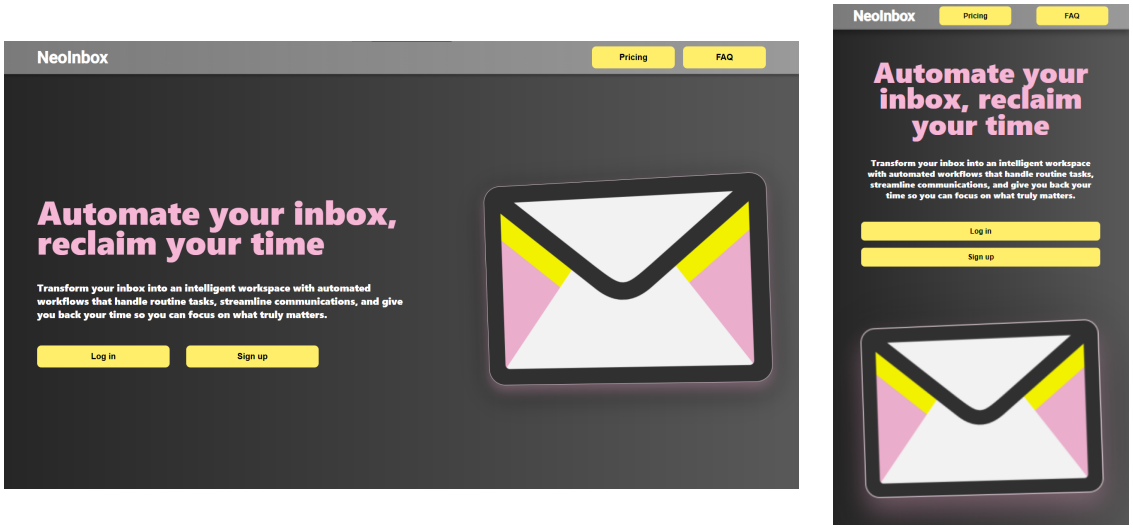

Figura 4.2: Vista del módulo de autenticación en versión escritorio y móvil.

Los componentes de la interfaz siguen una línea de diseño coherente, basada en formas rectangulares con bordes suavemente redondeados, lo que contribuye a una estética uniforme y profesional en toda la aplicación. En cuanto a la paleta cromática, se ha optado por una combinación armónica de tonos grisáceos, rosa y amarillo (véase la Figura 4.3). Esta elección garantiza un contraste adecuado, mejorando la legibilidad y la accesibilidad para usuarios con deficiencias visuales. Adicionalmente, todas las imágenes empleadas en la interfaz incluyen etiquetas alternativas (`alt`), reforzando así el compromiso del proyecto con las buenas prácticas en accesibilidad.

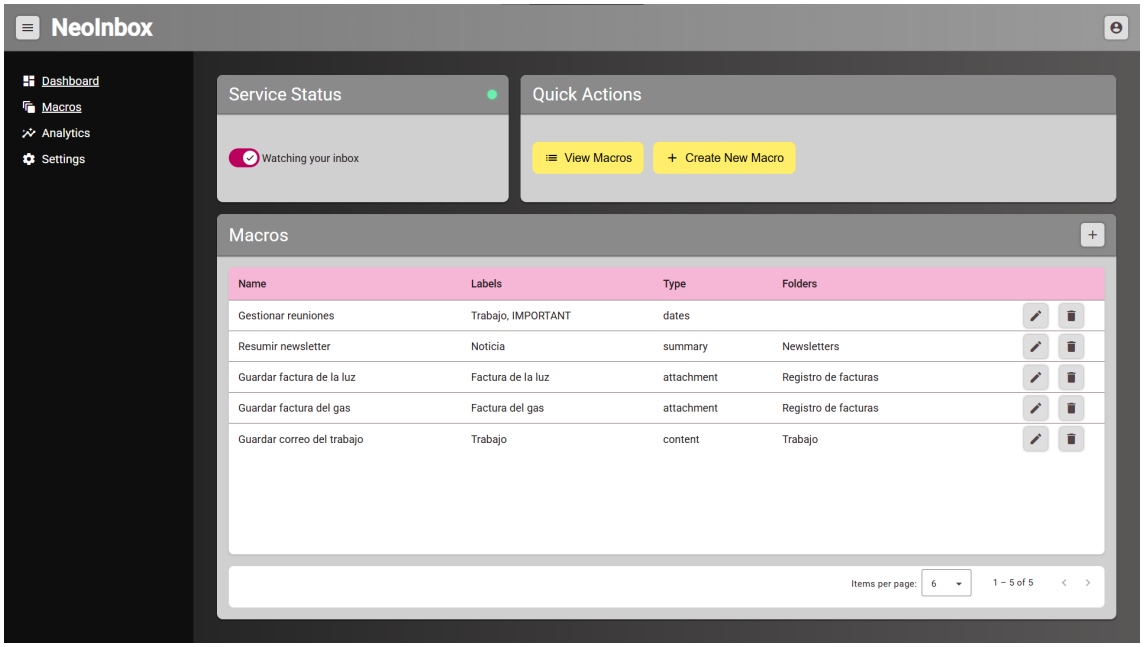

Figura 4.3: Centro de gestión principal del servicio NeoInbox

Desde el punto de vista funcional, la aplicación ofrece una experiencia de usuario clara y accesible desde el primer momento. Tal y como se muestra en la Figura 4.2, al acceder, el usuario se encuentra con una pantalla de bienvenida que presenta de forma directa las opciones de inicio de sesión y registro. Esta vista inicial ha sido diseñada con un enfoque moderno y atractivo, cuidando tanto la estética como la usabilidad, ya que constituye la primera impresión del sistema. Además, se incluye un acceso a una sección de precios, simulando el comportamiento de un servicio real orientado al público general (véase la Figura 4.4).

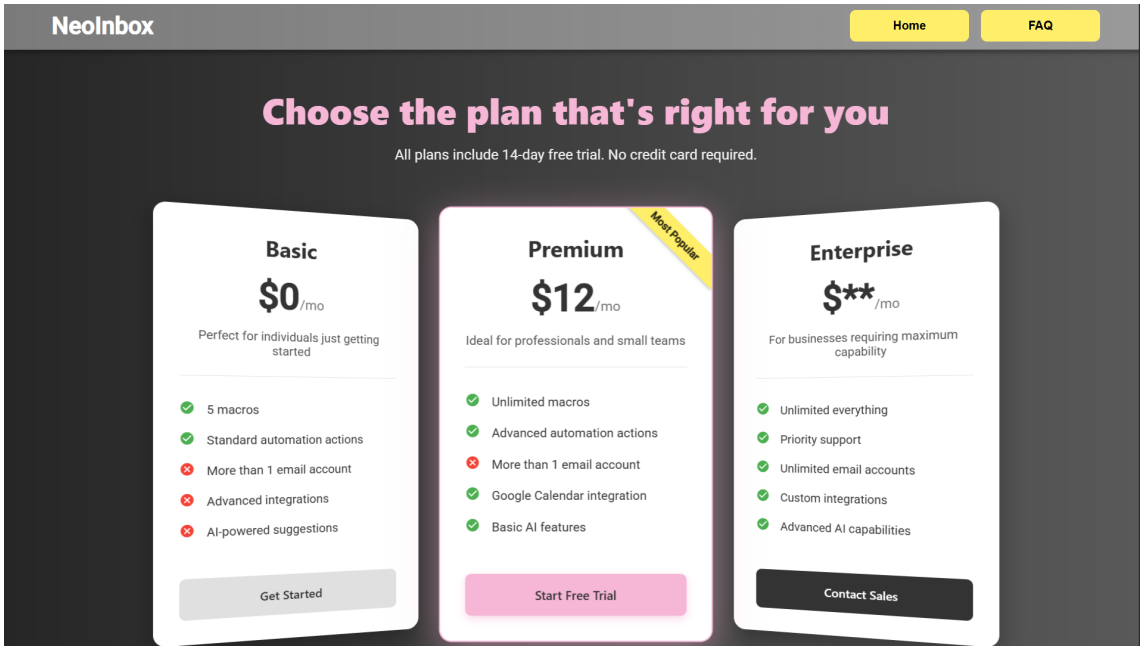

Figura 4.4: Módulo de planes del servicio y formulario de contacto

Tras la autenticación, el usuario accede al panel principal de la aplicación, mostrado en la Figura 4.3. En esta vista se le ofrece una interfaz clara desde la que puede activar o desactivar el servicio de monitorización mediante un interruptor central. Esta funcionalidad permite gestionar de forma sencilla cuándo el sistema debe comenzar a recibir y procesar correos electrónicos entrantes para ejecutar las macros correspondientes.

Justo debajo del interruptor se presenta una tabla con las últimas macros creadas por el usuario, lo que facilita el acceso visual a la actividad reciente y permite gestionar de forma eficiente las automatizaciones existentes (edición, eliminación, etc.).

Asimismo, se incluye una sección de acciones rápidas, desde la que se puede acceder a la creación de nuevas macros.

Como se observa en la Figura 4.5, el proceso está guiado a través de un componente tipo *stepper*, diseñado para facilitar la experiencia del usuario y reducir errores durante la configuración. El flujo se divide en varias etapas secuenciales que incluyen la asignación de un nombre descriptivo a la macro, la selección de las etiquetas de Gmail que actuarán como disparadores, la elección del tipo de acción que ejecutará la macro y la configuración de parámetros adicionales, los cuales varían según el tipo de macro seleccionada.

Este enfoque modular y guiado no solo simplifica el proceso de creación, sino que además contribuye a que el usuario comprenda el funcionamiento de las automatizaciones sin necesidad de conocimientos técnicos avanzados.

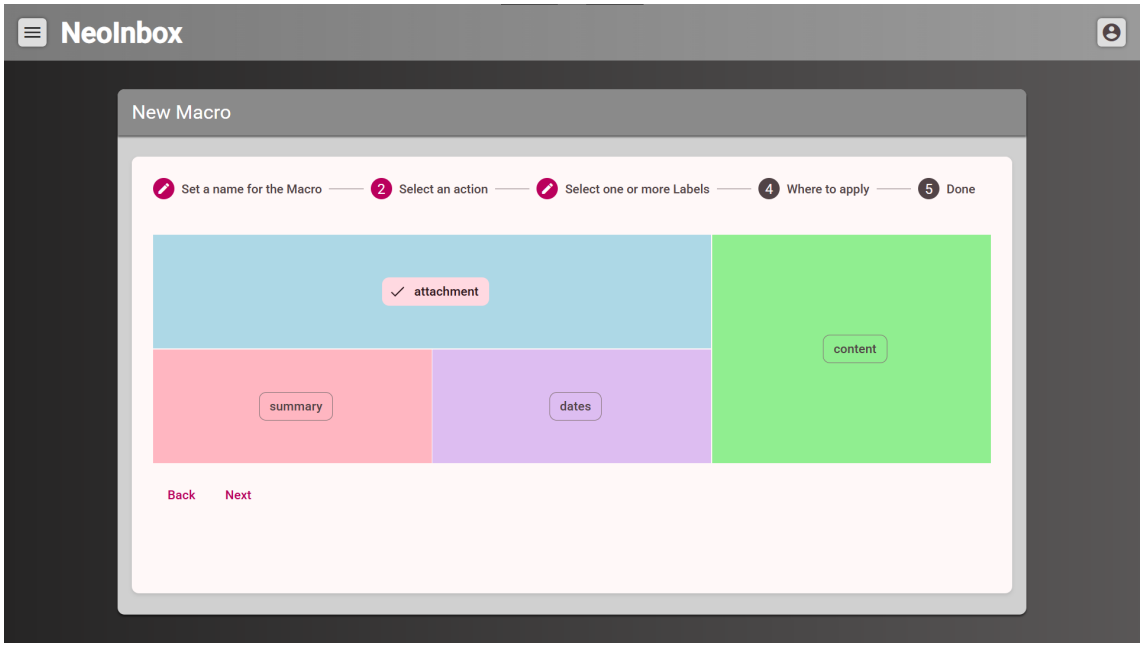

Figura 4.5: Componente responsable de la creación de nuevas macros

En lo relativo a la navegación, Angular permite definir rutas que estructuran el flujo de la aplicación. La configuración de rutas en este proyecto incluye páginas como `/authenticate`, `/watch-control`, `/macro-menu` o `/macro-create`, así como una ruta comodín que redirige a `/watch-control` en caso de acceder a rutas no definidas.

Para la comunicación con el *back-end* se han implementado servicios específicos que encapsulan las llamadas HTTP hacia la API. Estos servicios permiten una separación clara entre lógica de presentación y lógica de negocio. La URL base de la API está parametrizada para facilitar el cambio entre entornos de desarrollo (local) y producción (Cloud Run Function).

4.3. Seguridad y autenticación

La seguridad en el acceso a los recursos del sistema es un aspecto fundamental, especialmente en aplicaciones que interactúan con servicios externos, como las APIs de Google, y que gestionan información sensible del usuario. Para ello, en este proyecto se ha implementado un sistema de control de acceso basado en JSON Web Tokens, una de las soluciones más extendidas en entornos web por su sencillez, eficiencia y compatibilidad con arquitecturas sin estado.

4.3.1. Gestión de rutas y autenticación en el servidor

En el servidor desarrollado con Node.js y Express.js, se ha definido una separación clara entre rutas públicas y rutas protegidas mediante la creación de dos *routers* distintos. Además, las rutas protegidas están sujetas a un *middleware* de autenticación que intercepta las peticiones antes de llegar al controlador correspondiente.

Este *middleware* se encarga de verificar que la cabecera `Authorization` incluya un *token* válido. Para ello, extrae el JWT del encabezado, comprueba su validez mediante la clave secreta del servidor (almacenada de forma segura en variables de entorno) y, en caso de que sea correcto, permite que la petición continúe su curso. Si el *token* es inválido, ha expirado o no está presente, la respuesta del servidor es un código `401 Unauthorized`, denegando el acceso al recurso.

Este esquema garantiza que únicamente los usuarios autenticados puedan acceder a las rutas protegidas, reduciendo el riesgo de accesos no autorizados y minimizando el uso innecesario de recursos del *back-end*.

4.3.2. Restricciones de acceso en la web

Para garantizar que únicamente los usuarios autenticados puedan acceder a determinadas secciones de la aplicación, se ha implementado un sistema de control de acceso basado en la protección de rutas. Esta medida responde a la necesidad de restringir la navegación dentro del sistema según el estado de autenticación del usuario, asegurando que la experiencia y los datos queden limitados únicamente a quienes dispongan de las credenciales adecuadas.

El enfoque adoptado permite interceptar los intentos de acceso antes de que se carguen los contenidos protegidos. De este modo, se evalúa en tiempo real si el usuario cumple con los requisitos necesarios para continuar con la navegación. En caso contrario, se redirige al usuario a un punto de entrada seguro desde el cual puede iniciar sesión.

La lógica que determina si un usuario puede acceder o no se encuentra centralizada, lo que facilita tanto su mantenimiento como su extensión futura. Esta arquitectura permite, por ejemplo, incorporar más adelante otros niveles de acceso, como permisos basados en roles o restricciones contextuales más complejas.

En definitiva, esta estrategia contribuye a mantener la integridad del sistema desde el punto de vista de la experiencia de usuario y de la protección de recursos internos.

4.3.3. Gestión del *token* en el cliente

Desde el lado del cliente, la autenticación también se gestiona de forma transparente. Una vez que el usuario ha completado el proceso de autenticación, se recibe desde el *back-end* un JWT que se almacena en el `localStorage` del navegador, con una validez configurada de 24 horas. Esta estrategia permite mantener la sesión del usuario activa durante ese período sin necesidad de volver a autenticarse.

Para incluir automáticamente el *token* en todas las peticiones protegidas, se ha desarrollado un interceptor HTTP. Este interceptor se encarga de capturar todas las peticiones salientes desde el cliente, leer el *token* almacenado y añadirlo como cabecera `Authorization: Bearer <token>`, siempre que el recurso solicitado requiera autenticación.

Este mecanismo permite mantener una separación clara entre la lógica de negocio y los aspectos de seguridad, reduciendo la complejidad en los servicios del *front-end* y garantizando una comunicación segura entre el cliente y el servidor.

Capítulo 5

Pruebas y verificación

Este capítulo tiene como objetivo describir las pruebas realizadas para verificar el correcto funcionamiento del sistema. Las pruebas buscan asegurar que el comportamiento de la aplicación se ajusta a los requisitos definidos, y que la interacción entre sus distintos módulos es robusta, segura y eficiente.

5.1. Tipos de pruebas realizadas

Para garantizar la calidad, estabilidad y funcionalidad del sistema desarrollado, se llevaron a cabo diferentes tipos de pruebas a lo largo de todo el ciclo de vida del proyecto. Cada una de ellas se diseñó con un enfoque complementario, abarcando desde la validación unitaria de componentes aislados hasta la verificación del sistema completo en condiciones reales de uso.

Las pruebas realizadas incluyen tanto pruebas unitarias, centradas en verificar el correcto comportamiento de funciones y componentes individuales del *front-end*, como pruebas de integración orientadas a validar la correcta interacción entre los diferentes módulos del *back-end*. También se realizaron pruebas de extremo a extremo (E2E), simulando el flujo completo que sigue un usuario desde la interfaz hasta la ejecución de funcionalidades complejas.

Además, se contemplaron escenarios de error y recuperación para verificar la robustez del sistema ante fallos imprevistos, así como pruebas de rendimiento para evaluar los tiempos de respuesta y optimizar la experiencia de usuario.

5.1.1. Pruebas unitarias

Se desarrollaron pruebas unitarias para todos los elementos del *front-end* utilizando Jasmine y el entorno de pruebas proporcionado por Angular. Estas pruebas tienen como objetivo validar el funcionamiento de cada unidad de forma aislada, es decir, sin depender de otros módulos, servicios o el estado general de la aplicación. Se cubren todos los componentes visuales, así como los servicios, interceptores y *guards* definidos en el proyecto.

Para cada componente se creó un archivo específico `.spec.ts`, siguiendo una estructura organizada y modular. Dentro de estos archivos, se definieron bloques `describe()` individuales para cada método del componente, utilizando como nombre del bloque el propio nombre de la función evaluada. Esta práctica mejora la legibilidad de las pruebas y facilita el mantenimiento. Dentro de cada bloque, se incluyeron múltiples bloques `it()` que prueban distintas condiciones de entrada, valores límite o casos especiales, garantizando así un análisis exhaustivo del comportamiento esperado.

Esta misma metodología se aplicó también a servicios, *guards* e interceptores, simulando dependencias y utilizando técnicas como la inyección de servicios *mock* para verificar el correcto funcionamiento de cada unidad bajo diferentes condiciones.

La ejecución de las 105 pruebas definidas se realiza mediante el navegador, utilizando Karma, lo que permite una visualización clara del resultado de cada prueba en tiempo real (véase la Figura 5.1). Además, se genera automáticamente un informe de cobertura que proporciona métricas detalladas sobre el grado de cobertura alcanzado por las pruebas unitarias.

En el momento de la redacción de esta memoria, la cobertura alcanzada por el conjunto de pruebas del *front-end* es la siguiente:

- **Sentencias**: 79.86 % (365 de 457)
- **Ramas**: 72.81 % (75 de 103)
- **Funciones**: 77.41 % (96 de 124)
- **Lineas**: 79.67 % (341 de 428)

Estos resultados reflejan un nivel de cobertura satisfactorio en relación con los objetivos definidos para el proyecto, considerando la complejidad funcional de la aplicación y la adopción de una arquitectura modular que promueve la mantenibilidad y escalabilidad del sistema.

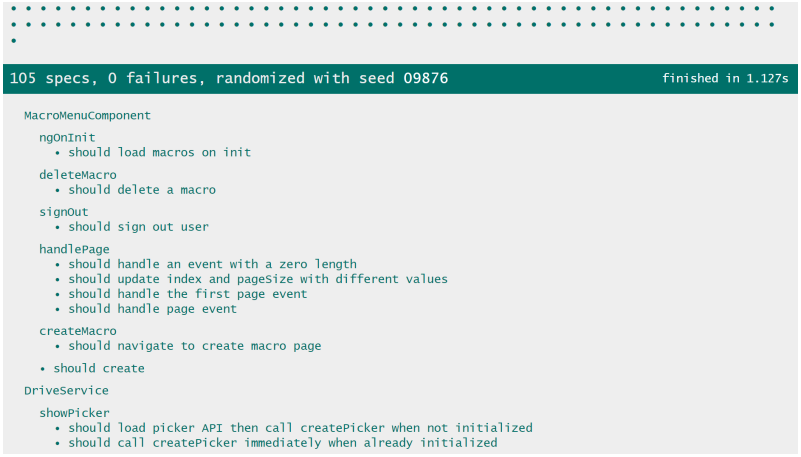

Figura 5.1: Resumen de pruebas unitarias ejecutadas con Karma

5.1.2. Pruebas de integración

Las pruebas de integración tienen como objetivo verificar que los distintos componentes del sistema interactúan correctamente entre sí. En el contexto de este proyecto, dichas pruebas se centraron en validar la comunicación entre la lógica del *back-end*, la base de datos y servicios externos como Pub/Sub o las APIs de Google.

Para su implementación se empleó el entorno de pruebas Jest, que permitió ejecutar funciones reales del servidor en un entorno controlado, sin necesidad de pasar por la interfaz del *front-end*. Esta aproximación facilita la detección de errores en la lógica de negocio y en las operaciones de persistencia, como por ejemplo durante la creación, consulta o eliminación de macros en Firestore.

A diferencia de las pruebas unitarias, estas pruebas no utilizan dobles de prueba ni *mocks*. En su lugar, los distintos módulos del *back-end* (controladores, servicios y funciones de acceso a datos) interactúan entre sí a través de sus interfaces reales, utilizando una instancia de base de datos de pruebas configurada específicamente para estos escenarios.

Los casos de prueba se diseñaron para cubrir flujos reales, como por ejemplo la creación de una macro y su posterior recuperación desde la base de datos, asegurando la coherencia de los datos procesados. También se validó el correcto funcionamiento de las interacciones asíncronas que dependen del sistema de mensajería Pub/Sub, incluyendo el tratamiento de eventos internos desencadenados por la recepción de correos electrónicos.

Gracias a este enfoque, se consiguió verificar el comportamiento del sistema en condiciones cercanas al entorno de producción, asegurando una integración robusta entre los diferentes componentes y minimizando el riesgo de errores en tiempo de ejecución.

5.1.3. Pruebas de extremo a extremo (E2E)

Las pruebas *end-to-end* (E2E) fueron desarrolladas utilizando el framework Cypress, el cual permite simular interacciones reales del usuario dentro del navegador. Estas pruebas desempeñan un papel fundamental para verificar que todos los flujos funcionales de la aplicación se comportan correctamente en conjunto, desde la vista del usuario final.

Se definió un archivo de pruebas Cypress para cada ruta principal de la aplicación web (`/authenticate`, `/watch-control`, `/macro-menu`, `/macro-create`, etc.). En cada uno de estos archivos se comprobó tanto la navegación como la interacción con los distintos elementos visuales y lógicos de la interfaz.

Para evitar la duplicación de código y mejorar el mantenimiento de las pruebas, se utilizaron **`fixtures`**, archivos donde se almacenan datos de ejemplo devueltos por los *endpoints* de la API. De este modo, es posible simular respuestas del servidor y validar comportamientos sin necesidad de depender de un *back-end* en ejecución.

Asimismo, se utilizaron *mocks* para interceptar las llamadas HTTP realizadas desde el cliente Angular, devolviendo respuestas controladas y verificando que el *front-end* reacciona adecuadamente ante diferentes escenarios (respuestas exitosas, errores, datos vacíos, etc.).

Además, con el objetivo de garantizar la correcta visualización y usabilidad de la aplicación en diferentes dispositivos, se configuró Cypress para ejecutar cada conjunto de pruebas bajo dos tamaños de dispositivo: uno correspondiente a un entorno de escritorio y otro adaptado a un dispositivo móvil. Esto permitió validar que el diseño *responsive* funciona correctamente y que la experiencia de usuario es adecuada tanto en pantallas grandes como en teléfonos móviles.

El uso de Cypress permitió generar informes detallados facilitando la identificación de errores y la validación visual de cada prueba. Como podemos ver en la Figura 5.2 se han elaborado y ejecutado correctamente un total de 54 pruebas.

√	authenticate.cy.ts	00:23	6	6
√	macro-create.cy.ts	00:23	2	2
√	macro-menu.cy.ts	00:30	10	10
√	pricing.cy.ts	00:19	8	8
√	watch-control.cy.ts	01:25	28	28
√	All specs passed!	03:03	54	54

Figura 5.2: Ejecución de pruebas E2E con Cypress

5.1.4. Pruebas de error y recuperación

Uno de los aspectos clave del desarrollo de esta aplicación ha sido garantizar su robustez frente a errores y su capacidad de recuperación ante situaciones inesperadas. Para ello, se han implementado múltiples validaciones tanto en el tratamiento de las peticiones HTTP como en el procesamiento de los datos recibidos.

En primer lugar, se verifica que el tipo de petición HTTP sea el correcto para cada uno de los *endpoints*, respondiendo con el código de estado adecuado si no se cumple esta condición. Asimismo, se comprueba la existencia y validez de los argumentos requeridos en el cuerpo o los parámetros de la petición, devolviendo los correspondientes códigos de error.

Además, las secciones críticas del código están encapsuladas en bloques `try/catch` que permiten capturar y manejar excepciones, devolviendo mensajes de error informativos en caso de fallos imprevistos. Esto garantiza que el servidor no se detenga abruptamente ante errores, y que el cliente reciba siempre una respuesta clara sobre el estado de la operación.

Estas medidas permiten detectar errores tempranamente, mejorar la tolerancia a fallos y, sobre todo, ofrecer una experiencia más segura y controlada al usuario final.

5.1.5. Pruebas de rendimiento

Aunque no se realizaron pruebas de carga formales, se monitorizó el rendimiento de las funciones desplegadas en Cloud Run mediante las herramientas de *logging* y métricas proporcionadas por Google Cloud. En general, los tiempos de respuesta observados fueron prácticamente instantáneos, especialmente tras aplicar una mejora clave, la ya comentada consolidación de los distintos *endpoints* en una única función, lo cual redujo significativamente el impacto del *cold start* y mejoró la eficiencia general del sistema.

En cuanto a los procesos más exigentes, aquellos que implican el uso de inteligencia artificial, como el resumen automático de correos electrónicos o la identificación de eventos expresados en lenguaje natural, fueron analizados detenidamente. Se evaluaron distintas configuraciones de modelo con el fin de equilibrar tres factores clave, como son el coste por petición, tiempo de respuesta y calidad de los resultados obtenidos.

Finalmente, se optó por una configuración que ofrece una buena precisión en las tareas mencionadas, manteniendo un tiempo de respuesta inferior a los 30 segundos, valor considerado razonable para el tipo de interacción del usuario. En cualquier caso, el sistema está preparado para admitir configuraciones más avanzadas si en el futuro se desea mejorar aún más la velocidad o la calidad del análisis automático.

5.2. Conclusiones de las pruebas

El proceso de verificación y validación del sistema ha permitido garantizar que la aplicación cumple con los requisitos establecidos, tanto funcionales como no funcionales, y que su comportamiento es estable bajo diferentes escenarios de uso.

Las pruebas unitarias, desarrolladas con Jasmine y ejecutadas mediante Karma, han permitido alcanzar una cobertura superior al 75 %, asegurando la fiabilidad de los componentes del *front-end* en aislamiento. Las pruebas *end-to-end*, por su parte, ofrecieron una visión integral del flujo de navegación e interacción desde el punto de vista del usuario, permitiendo detectar errores de integración visual o funcional antes del despliegue.

Las pruebas de integración, realizadas con Jest sobre el *back-end*, han sido esenciales para validar la comunicación entre servicios clave como Firestore, Pub/Sub y las APIs de Google, asegurando que la lógica de negocio se mantiene coherente y funcional en entornos reales. Asimismo, se han verificado escenarios de error y recuperación, incorporando buenas prácticas de manejo de excepciones y respuestas informativas que fortalecen la robustez general del sistema.

Por último, el análisis del rendimiento en entorno de producción ha demostrado que la arquitectura es eficiente y escalable. A pesar de no haberse realizado pruebas de carga formalizadas, el sistema muestra tiempos de respuesta aceptables, incluyendo aquellos procesos que involucran el uso de inteligencia artificial.

En conjunto, el sistema se considera estable, fiable y preparado para ser desplegado en un entorno real. Las pruebas realizadas no solo han permitido validar el estado actual del software, sino también sentar las bases para un proceso de mejora continua, facilitando la incorporación de nuevas funcionalidades y el mantenimiento a largo plazo de la aplicación.

Capítulo 6

Conclusiones y trabajo futuro

Este trabajo ha dado lugar al desarrollo de una aplicación web funcional que permite automatizar acciones a partir de correos electrónicos recibidos en Gmail, lo que representa una solución versátil y útil para múltiples contextos personales o profesionales. A lo largo del proyecto, se han puesto en práctica conocimientos avanzados en desarrollo web *full-stack*, integración con APIs de Google, uso de servicios en la nube y principios de diseño de interfaces accesibles y escalables.

No obstante, este proyecto puede considerarse una base sobre la que construir nuevas mejoras y funcionalidades en el futuro. Una de las primeras acciones previstas sería la publicación oficial del servicio, lo que implicaría contratar un dominio personalizado y desplegar tanto el *front-end* como el *back-end* en servicios de alojamiento en la nube, asegurando su disponibilidad pública para cualquier usuario interesado.

Otro paso fundamental será la integración de una pasarela de pago, probablemente mediante la plataforma Stripe, lo que permitiría monetizar el servicio, implementar planes por suscripción y escalar su uso de forma sostenible.

Además, se contempla la expansión del sistema para trabajar no solo con el ecosistema de Google, sino también con otros proveedores como Microsoft. La integración con servicios como Outlook, OneDrive o Word permitiría ampliar el abanico de usuarios y diversificar los casos de uso, consolidando así una plataforma más completa y transversal.

En cuanto a la funcionalidad, se plantean múltiples mejoras. Una de las más destacadas es la posibilidad de analizar la prioridad de los correos recibidos mediante técnicas de procesamiento de lenguaje natural o inteligencia artificial, de modo que el sistema pueda asignar automáticamente etiquetas diferenciadas en función de dicha prioridad. Estas etiquetas servirían, a su vez, como nuevos disparadores (*triggers*) para la ejecución de macros, enriqueciendo aún más la lógica del sistema.

Finalmente, para mejorar la experiencia del usuario y facilitar la interacción con

el sistema, sería interesante desarrollar una extensión o *add-on* para Gmail. Esto permitiría acceder a las funcionalidades del servicio directamente desde la interfaz de Gmail, sin necesidad de abandonar el entorno habitual de trabajo del usuario.

Bibliografía

[1] Francisco S. Marcondes et al. "Using Ollama". En: *Natural Language Analytics with Generative Large-Language Models: A Practical Approach with Ollama and Open-Source LLMs*. Cham: Springer Nature Switzerland, 2025, págs. 23-35. ISBN: 978-3-031-76631-2. DOI: 10.1007/978-3-031-76631-2_3. URL: https://doi.org/10.1007/978-3-031-76631-2_3.

[2] Matthew Renze y Erhan Guven. "The Effect of Sampling Temperature on Problem Solving in Large Language Models". En: (feb. de 2024). arXiv preprint. URL: https://arxiv.org/html/2402.05201v1.

[3] Konstantinos I. Roumeliotis y Nikolaos D. Tselikas. "ChatGPT and Open-AI Models: A Preliminary Review". En: *Future Internet* 15.6 (2023). ISSN: 1999-5903. DOI: 10.3390/fi15060192. URL: https://www.mdpi.com/1999-5903/15/6/192.

[4] Ian Sommerville. *Software Engineering*. 9th. Boston: Addison-Wesley, 2011. ISBN: 978-0-13-703515-1.

[5] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2025. arXiv: 2312.11805 [cs.CL]. URL: https://arxiv.org/abs/2312.11805.