

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA E INFORMÁTICA
CONCEPCIÓN - CHILE



**“DISEÑO E IMPLEMENTACIÓN DEL BACKEND DE
EVERPLANAPP: APLICACIÓN DE ORGANIZACIÓN CON
ENFOQUE INCLUSIVO”**

SEBASTIÁN ALEJANDRO GALLARDO PÉREZ

**MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO EN INFORMÁTICA**

Profesor Guía: Cristian Antonio Lara Valenzuela

Diciembre – 2025



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título; ☐ Tesis de Postgrado;

Título del trabajo: DISEÑO E IMPLEMENTACIÓN DEL BACKEND DE EVERPLANAPP: APLICACIÓN DE ORGANIZACIÓN CON ENFOQUE INCLUSIVO

Nombre del candidato(a): Sebastián Alejandro Gallardo Pérez

Carrera / Grado: Ingeniería en informática

Campus: Concepción ; Departamento: Electronica e informatica

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Cristian Lara Valenzuela, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO** contiene información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

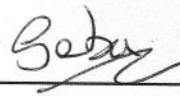
☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis: CRISTIAN ANTONIO LARA VALENZUELA
Fecha: 27/05/2026 ; Firma: LARA VALENZUELA
Firmado digitalmente por CRISTIAN ANTONIO LARA VALENZUELA
Fecha: 2026.05.27 01:05:29 -04'00'

Estudiante o Candidato(a): Sebastián Gallardo

Fecha: 27/05/2026 ; Firma: 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

DEDICATORIA

Este trabajo representa el esfuerzo y la perseverancia de varios años de estudio, dedicación y crecimiento personal.

Quiero dedicar esta memoria, en primer lugar, a mis profesores, quienes con su guía y conocimientos me ayudaron a desarrollar no solo habilidades técnicas, sino también una visión crítica y responsable frente a la informática. Cada clase, consejo y corrección aportó a mi formación y a este logro.

A mi familia, que siempre estuvo a mi lado, brindándome su apoyo incondicional en cada etapa del camino.

Y, especialmente, a mi madre. Gracias por tu paciencia, por creer en mí incluso cuando yo no lo hacía, y por darme fuerzas para seguir adelante. Tu amor, tus sacrificios y tu ejemplo son los pilares sobre los que hoy puedo construir mis metas. Esta memoria también es tuya.

Sebastián Alejandro Gallardo Pérez

AGRADECIMIENTOS

Quiero expresar mi sincero agradecimiento a todas las personas que hicieron posible la realización de este proyecto y que, de una u otra forma, me acompañaron durante mi etapa universitaria.

En primer lugar, agradezco especialmente al profesor José Luis Carrasco Sáez, por su guía constante, sus consejos y su disposición para orientarme a lo largo del desarrollo de esta memoria. Su apoyo fue fundamental para dar forma a este trabajo y para enfrentar con mayor seguridad los distintos desafíos que fueron surgiendo.

Asimismo, deseo expresar un agradecimiento especial a Cristian Lara, Javier Maldonado, Rodrigo Gutiérrez y José Arellano, quienes fueron una parte muy importante de mi formación académica. Cada uno de ellos me entregó conocimientos valiosos y perspectivas distintas que enriquecieron mi aprendizaje, y sus clases no solo fueron útiles, sino también motivadoras. Para mí siempre fue un gusto asistir, aprender y desarrollar mis habilidades bajo su guía.

También quiero agradecer a mis compañeros de equipo de EverPlanAPP, por su compromiso, colaboración y trabajo en conjunto. Compartir este desafío con ustedes hizo que el proceso fuera más enriquecedor y permitió que esta experiencia fuera no solo académica, sino también humana.

Agradezco de manera especial a mis amigos, quienes hicieron de mi vida universitaria una etapa entretenida, significativa e inolvidable: Alan Mac-Kay, Alejandro Valenzuela, Felipe Badilla, Tomás Solar, Fernando Cruces, Javier Choque, Ignacio Hormazábal, Dismar Quiroz y Matías Torres. Gracias por el apoyo, las risas, los momentos difíciles y los buenos recuerdos que marcaron estos años.

Finalmente, quiero agradecer de todo corazón a mi familia, por su apoyo incondicional, su paciencia y su comprensión a lo largo de todo este proceso. Gracias por acompañarme en los momentos de cansancio, por confiar en mí y por brindarme el espacio, la tranquilidad y el ánimo necesarios para poder concentrarme y seguir adelante. Su respaldo y cariño fueron fundamentales para poder llegar hasta aquí.

RESUMEN

Resumen—Esta memoria presenta el desarrollo del *backend* y la base de datos de *EverPlanApp*, una aplicación web y móvil orientada a apoyar la organización académica de estudiantes universitarios, especialmente aquellos con condiciones neurodivergentes. En primer lugar, se realizó un proceso de investigación y análisis comparativo de distintas tecnologías y arquitecturas, con el fin de seleccionar las herramientas más adecuadas para los objetivos del proyecto. Posteriormente, se diseñó e implementó una solución *backend* basada en Django y su sistema de Mapeo Objeto-Relacional (ORM), utilizando una arquitectura monolítica modular que permitió mantener una estructura clara, escalable y mantenible. Se definió un modelo de base de datos relacional acorde a los requerimientos funcionales del sistema, integrando entidades como usuarios, tareas, horario de clases, recordatorios, estados de ánimo, estadísticas y sugerencias de organización. Las *APIs REST* desarrolladas fueron evaluadas con *Postman*, validando su funcionalidad, seguridad y consistencia. Como resultado, se construyó un *backend* estructurado y listo para su integración con el *frontend* de la aplicación.

Palabras clave—*Backend*; Arquitectura modular; Django; Base de datos relacional; Inclusión educativa.

ABSTRACT

Abstract—This thesis presents the development of the backend and database for *EverPlanApp*, a web and mobile application aimed at supporting the academic organization of university students, especially those with neurodivergent conditions. The work began with research and comparative analysis of various backend technologies and architectural approaches, in order to select the most appropriate tools for the project's objectives. Based on the findings, a backend solution was implemented using Django and its Object-Relational Mapping (ORM) system, following a modular monolithic architecture to ensure a clear, scalable, and maintainable structure. A relational database model was designed to reflect the system's functional requirements, including entities such as users, tasks, class schedule, reminders, mood tracking, statistics, and organizational suggestions. The REST APIs were tested with *Postman*, validating their functionality, security, and consistency. As a result, a well-structured backend was built, ready to be integrated with the application's frontend.

Keywords— Backend; Modular architecture; Django; Relational database; Educational inclusion.

GLOSARIO

AbstractUser: Clase base de Django que permite personalizar modelos de usuario.

Access Token: Token utilizado para acceder a recursos protegidos.

APIClient: Herramienta para simular solicitudes a una API en pruebas.

APIs (Application Programming Interfaces): Interfaces que permiten la comunicación entre diferentes aplicaciones.

API REST (Representational State Transfer): Arquitectura para crear servicios web que usan HTTP para intercambiar datos.

APITestCase: Clase de Django utilizada para pruebas de APIs.

Apps: Aplicaciones que forman parte de un sistema o plataforma.

Asíncrono: Modelo de ejecución que permite realizar múltiples tareas sin bloquear el flujo principal del sistema.

AWS (Amazon Web Services): Plataforma de servicios en la nube de Amazon.

AWS EventBridge: Servicio de AWS para gestionar eventos y flujos de trabajo.

Backend: Parte de una aplicación encargada de la lógica del sistema, el procesamiento de datos y la comunicación con la base de datos.

Blacklist: Lista de tokens invalidados por razones de seguridad.

Cache: Mecanismo de almacenamiento temporal que permite acelerar el acceso a datos usados frecuentemente.

Celery: Sistema de tareas en segundo plano para aplicaciones distribuidas.

Clickjacking: Ataque que engaña al usuario para que haga clic en elementos ocultos o manipulados.

Compose: Herramienta de Docker para definir y ejecutar aplicaciones con múltiples contenedores.

Cron Job: Tarea programada que se ejecuta automáticamente en horarios definidos.

CRUD (Create, Read, Update, Delete): Operaciones básicas de gestión de datos.

Cross-site Request Forgery (CSRF): Ataque que obliga a un usuario autenticado a ejecutar acciones no deseadas en una aplicación web.

Cross-site Scripting (XSS): Ataque que permite inyectar código malicioso en páginas web vistas por otros usuarios.

DELETE: Verbo HTTP utilizado para eliminar recursos del servidor.

DevOps: Cultura y prácticas que integran desarrollo y operaciones para automatizar y mejorar el despliegue de software.

Django: Framework web de alto nivel para Python que permite desarrollar aplicaciones web de forma rápida y segura.

Docker: Plataforma para crear, distribuir y ejecutar aplicaciones en contenedores.

Endpoints: Puntos de acceso de una API donde se realizan solicitudes.

FastAPI: Framework moderno de Python para crear APIs rápidas, seguras y con alto rendimiento.

Flask: Microframework de Python utilizado para crear aplicaciones web ligeras y flexibles.

ForeignKey: Relación que conecta una tabla con otra en una base de datos.

Frontend: Parte de una aplicación que interactúa directamente con el usuario, incluyendo interfaces gráficas y experiencia visual.

Framework: Conjunto de herramientas, librerías y reglas que facilitan el desarrollo de aplicaciones de software.

GET: Verbo HTTP utilizado para solicitar datos a un servidor.

Google Cloud Scheduler: Servicio de Google Cloud para programar tareas.

HTTP (HyperText Transfer Protocol): Protocolo que regula la comunicación entre clientes y servidores web.

Inyección SQL: Ataque que permite ejecutar comandos SQL maliciosos a través de entradas no validadas.

Jinja2: Motor de plantillas utilizado en Flask para generar contenido HTML dinámico.

JSON (JavaScript Object Notation): Formato ligero de intercambio de datos basado en texto.

Machine Learning: Rama de la inteligencia artificial que permite a los sistemas aprender a partir de datos.

makemigrations: Comando de Django que genera cambios en la base de datos.

ManyToManyField: Tipo de relación en Django que conecta múltiples registros entre dos entidades.

MER (Modelo Entidad–Relación): Diagrama que representa la estructura lógica de una base de datos.

Microframework: Framework liviano que provee solo las funciones básicas para desarrollar aplicaciones.

Microservicios: Arquitectura que divide un sistema en servicios pequeños e independientes.

Model: Capa que representa los datos y su estructura en una aplicación.

MongoDB: Base de datos NoSQL orientada a documentos.

Monolítico: Arquitectura donde toda la aplicación se desarrolla como una sola unidad.

MySQL: Sistema de gestión de bases de datos relacional.

NoSQL: Tipo de base de datos no relacional que permite almacenar datos de forma flexible y escalable.

OAuth 2.0: Protocolo de autorización que permite a las aplicaciones acceder a recursos de un usuario sin exponer sus credenciales.

OpenAPI: Estándar que define cómo describir y documentar APIs REST.

ORM (Object Relational Mapping): Herramienta que permite mapear objetos de programación a tablas de bases de datos.

pg_cron: Extensión de PostgreSQL que permite programar tareas dentro de la base de datos.

pgModeler: Software para diseñar modelos de bases de datos PostgreSQL.

PostgreSQL: Sistema de base de datos relacional de código abierto.

POST: Verbo HTTP utilizado para enviar datos y crear recursos.

Pydantic: Librería de Python usada para validar y estructurar datos en APIs.

Python: Lenguaje de programación interpretado, ampliamente utilizado en desarrollo web, ciencia de datos y automatización.

RabbitMQ: Sistema de mensajería utilizado para la comunicación entre servicios.

Refresh Token: Token que permite obtener un nuevo access token sin volver a iniciar sesión.

RRULE (Recurrence Rule): Regla que define la repetición de eventos en sistemas de calendario.

scheduled_at: Campo que indica la fecha y hora programada para una acción.

Serializer: Componente que convierte objetos en formatos como JSON.

Signals: Mecanismo de Django que permite ejecutar acciones cuando ocurre un evento.

SimpleJWT: Librería de Django para autenticación basada en JSON Web Tokens.

SQL (Structured Query Language): Lenguaje utilizado para gestionar y consultar bases de datos relacionales.

TDAH (Trastorno por Déficit de Atención e Hiperactividad): Condición neurobiológica que afecta la capacidad de concentración, organización y control de impulsos.

TEA (Trastorno del Espectro Autista): Condición del neurodesarrollo que influye en la comunicación, la interacción social y el comportamiento.

Testing: Proceso de verificación del funcionamiento correcto de un sistema mediante pruebas automáticas o manuales.

Token: Cadena de seguridad utilizada para autenticar usuarios.

URL (Uniform Resource Locator): Dirección que identifica un recurso en internet.

Uvicorn: Servidor ASGI de alto rendimiento utilizado para ejecutar aplicaciones FastAPI.

View: Componente que procesa las solicitudes del usuario y retorna respuestas.

WebSockets: Protocolo que permite comunicación bidireccional en tiempo real.

Workbench: Herramienta gráfica para administrar bases de datos MySQL.

INDICE DE CONTENIDOS

RESUMEN.....	5
ABSTRACT	5
INDICE DE FIGURAS	13
INTRODUCCIÓN.....	16
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	17
1.1 Contexto General.....	17
1.2 Situación Actual	17
1.3 Actores Involucrados	18
1.4 Problema específico de la memoria	18
1.5 Motivación e Interés del Proyecto	18
1.6 Objetivo General	19
1.7 Objetivos Específicos	19
CAPÍTULO 2: MARCO CONCEPTUAL	20
2.1 ¿Qué es un framework?	20
2.2 Principales frameworks backend en Python.....	20
2.2.1 ¿Qué es Django?	21
2.2.2 ¿Qué es Flask?	22
2.2.3 ¿Qué es FastAPI?.....	23
2.2.4 Análisis final y elección del framework	24
2.3 ¿Qué es la arquitectura de software?	25
2.3.1 Evaluación de alternativas de arquitectura	25
2.3.1.1 ¿Qué es una API?	26
2.3.1.2 ¿Qué es la arquitectura monolítica?	26
2.3.1.3 ¿Qué es la arquitectura basada en microservicios?	28
2.3.2 Análisis final y elección de la arquitectura de software	30
2.3.2.1 ¿Qué es una Arquitectura monolítica modular?	30

2.4 Diseñar la estructura del sistema y modelo de datos.....	32
2.4.1 ¿Qué tipo de base de datos deberíamos elegir?.....	32
2.4.1.1 ORM en Django: Definición y funcionamiento	32
2.4.1.2 ¿Qué es una base de datos relacional?	33
2.4.2 ¿Que son las herramientas de modelado y el modelo Entidad-Relación?.....	33
2.4.3 Análisis y utilidad del modelo entidad-relación (ER) en EverPlanAPP.....	34
2.4.4 Optimizar la base de datos con normalización	34
2.5 Analizar mecanismos de seguridad para el backend	35
2.6 Criterios de validación del backend	36
2.6.1 Herramientas de validación	36
CAPÍTULO 3: PROPUESTA DE SOLUCION	38
3.1 Diseño de la arquitectura general de EverPlanAPP	38
3.2 Requerimientos funcionales del sistema	39
3.3 Diseño funcional de los módulos del backend	40
3.4 Arquitectura del Backend de EverPlanAPP.....	42
3.4.1 Estructura general de los módulos del backend	43
3.5 Diseño del modelo de datos (MER).....	44
3.6 Diseño e implementación de los módulos del backend	45
3.6.1 Módulo de autenticación y gestión de usuarios (auth_app)	46
3.6.2 Módulo de gestión de tareas (tarea_app).....	53
3.6.3 Módulo de gestión de rutinas (rutinas_app).....	59
3.6.4 Módulo de gestión de horarios personales (horarioPersonal_app).....	65
3.6.5 Módulo de notificaciones en tiempo real (notificacion_app).....	70
3.7 Librerías y herramientas utilizadas en el backend	74
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	76
4.1 Validación del módulo auth_app	77
4.1.1 Ejecución de las pruebas unitarias de auth_app	79
4.2 Validación del módulo tarea_app	81

4.2.1 Ejecución de las pruebas unitarias de tarea_app	84
4.3 Validación del módulo rutinas_app	84
4.3.1 Ejecución de las pruebas unitarias de rutinas_app	87
4.4 Validación del módulo horarioPersonal_app	88
4.4.1 Ejecución de las pruebas unitarias de horarioPersonal_app	90
4.5 Validación del módulo notificaciones_app	91
4.5.1 Limitaciones actuales y mejoras futuras del módulo notificaciones_app	91
4.6 Apoyo de herramientas en el proceso de validación	92
CAPÍTULO 5: CONCLUSIONES	96
REFERENCIAS BIBLIOGRÁFICAS	98

INDICE DE FIGURAS

Figura 1: Arquitectura monolítica. Fuente: Atlassian	27
Figura 2: Arquitectura basada en microservicios. Fuente: Atlassian	28
Figura 3: Arquitectura monolítica modular. Fuente: GeeksforGeeks	30
Figura 4: Comparativa entre arquitecturas. Fuente: VCSOFT	31
Figura 5: Arquitectura de EverPlanAPP. Fuente: Elaboración propia	39
Figura 6: Arquitectura del backend de EverPlanAPP. Fuente: Elaboración propia	42
Figura 7: Estructura del proyecto Django de EverPlanAPP y organización interna de una aplicación. Fuente: Elaboración propia	44
Figura 8: Modelo Entidad-Relación (MER) de EverPlanAPP. Fuente: Elaboración propia	45
Figura 9: Diagrama de secuencia del proceso de registro del usuario. Fuente: Elaboración propia	48
Figura 10: Diagrama de secuencia del proceso de inicio de sesión del usuario. Fuente: Elaboración propia	48
Figura 11: Diagrama de secuencia del proceso de cierre de sesión del usuario. Fuente: Elaboración propia	49
Figura 12: Diagrama de secuencia del proceso de ver perfil del usuario. Fuente: Elaboración propia	50
Figura 13: Diagrama de secuencia del proceso de modificar perfil del usuario. Fuente: Elaboración propia	50
Figura 14: Diagrama de secuencia del proceso de cambio de contraseña del usuario. Fuente: Elaboración propia	51
Figura 15: Diagrama de secuencia del proceso de eliminación de cuenta de usuario. Fuente: Elaboración propia	52
Figura 16: Diagrama de secuencia del proceso de creación de una tarea. Fuente: Elaboración propia	55
Figura 17: Diagrama de secuencia del proceso de listar tareas. Fuente: Elaboración propia	56
Figura 18: Diagrama de secuencia del proceso de ver detalle de una tarea. Fuente: Elaboración propia	56
Figura 19: Diagrama de secuencia del proceso de editar una tarea. Fuente: Elaboración propia	57
Figura 20: Diagrama de secuencia del proceso de eliminar una tarea. Fuente: Elaboración propia	57
Figura 21: Diagrama de secuencia del proceso de eliminar todas las tareas. Fuente: Elaboración propia	58
Figura 22: Diagrama de secuencia del proceso de estadísticas motivacionales. Fuente: Elaboración propia	58
Figura 23: Diagrama de secuencia del proceso de creación de una rutina. Fuente: Elaboración propia	61

Figura 24: Diagrama de secuencia del proceso de listar rutinas. Fuente: Elaboración propia.....	61
Figura 25: Diagrama de secuencia del proceso de ver detalle de una rutina. Fuente: Elaboración propia....	62
Figura 26: Diagrama de secuencia del proceso editar una rutina. Fuente: Elaboración propia.....	62
Figura 27: Diagrama de secuencia del proceso de eliminar una rutina. Fuente: Elaboración propia.....	63
Figura 28: Diagrama de secuencia del proceso de eliminar todas las rutinas. Fuente: Elaboración propia..	63
Figura 29: Diagrama de secuencia del proceso de listar ocurrencias de rutinas. Fuente: Elaboración propia.	64
Figura 30: Diagrama de secuencia del proceso de creación de un evento. Fuente: Elaboración propia.	66
Figura 31: Diagrama de secuencia del proceso de listar eventos. Fuente: Elaboración propia.	66
Figura 32: Diagrama de secuencia del proceso de ver detalle de un evento. Fuente: Elaboración propia. ...	67
Figura 33: Diagrama de secuencia del proceso de editar un evento. Fuente: Elaboración propia.	67
Figura 34: Diagrama de secuencia del proceso de eliminar un evento. Fuente: Elaboración propia.	68
Figura 35: Diagrama de secuencia del proceso de eliminar todos los eventos. Fuente: Elaboración propia.	68
Figura 36: Diagrama de secuencia de la señal para tareas. Fuente: Elaboración propia.	71
Figura 37: Diagrama de secuencia del proceso de la señal para rutinas. Fuente: Elaboración propia.	71
Figura 38: Diagrama de secuencia del proceso de envió de notificaciones desde el backend hacia el frontend. Fuente: Elaboración propia.	73
Figura 39: Diagrama de secuencia del proceso listar notificaciones. Fuente: Elaboración propia.....	73
Figura 40: Creación de la base de datos test_everplanapp. Fuente: Elaboración propia.	80
Figura 41: Figura 41: Pruebas unitarias del módulo auth_app. Fuente: Elaboración propia.....	80
Figura 42: Pruebas unitarias del módulo tarea_app. Fuente: Elaboración propia.	84
Figura 43: Pruebas unitarias del módulo rutinas_app. Fuente: Elaboración propia.....	87
Figura 44: Pruebas unitarias del módulo horarioPersonal_app. Fuente: Elaboración propia.	90
Figura 45: Endpoints de EverPlanAPP en Postman. Fuente: Elaboración propia.	93
Figura 46: Ejemplo de validación con Postman Registrar usuario. Fuente: Elaboración propia.	94
Figura 47: Administrador de Django. Fuente: Elaboración propia.	94
Figura 48: Ejemplo administrador de Django consultar tabla de usuarios. Fuente: Elaboración propia.	95
Figura 49: Ejemplo administrador de Django añadir nuevo usuario. Fuente: Elaboración propia.....	95

INDICE DE TABLAS

Tabla 1: Objetivos específicos, marco conceptual y propuesta de solución del subproyecto Backend de EverPlanApp. Fuente: Elaboración propia.....	19
Tabla 2: Resumen comparativo de Django, Flask y FastAPI para desarrollo backend. Fuente: Elaboración propia en base a documentación oficial y comparativas técnicas.	24
Tabla 3: Endpoints del módulo auth_app. Fuente: Elaboración Propia.	53
Tabla 4: Endpoints del módulo tarea_app. Fuente: Elaboración Propia.	59
Tabla 5: Endpoints del módulo rutinas_app. Fuente: Elaboración Propia.	64
Tabla 6: Endpoints del módulo horarioPersonal_app. Fuente: Elaboración Propia.	69
Tabla 7: Endpoints del módulo notificacion_app. Fuente: Elaboración Propia.	73

INTRODUCCIÓN

La presente memoria aborda el diseño, implementación y validación del *backend* de *EverPlanAPP*, una aplicación orientada a la planificación y organización para estudiantes universitarios con neurodivergencia, especialmente aquellos que presentan dificultades en la gestión de horarios, tareas y rutinas. Si bien la plataforma está pensada principalmente para este grupo, también resulta útil para personas sin una condición neurodivergente que igualmente presentan problemas de organización y planificación. La aplicación surge como una herramienta de apoyo para estructurar el día a día, facilitar la autorregulación y promover una mayor autonomía en el ámbito académico y personal.

El área de trabajo de esta memoria se enmarca en el desarrollo *backend* de aplicaciones, donde se gestionan la lógica del sistema, la persistencia de los datos y la seguridad de la información. En el contexto de *EverPlanAPP*, el principal desafío fue contar con una infraestructura capaz de manejar de forma confiable la información asociada a usuarios, tareas, rutinas, horarios personales, pictogramas, ejercicios de apoyo a la autorregulación (como respiración y pausas guiadas) y estadísticas motivacionales. Estos elementos son fundamentales para el funcionamiento de la aplicación y requieren una base sólida que asegure consistencia, privacidad y disponibilidad de los datos.

Para dar respuesta a esta necesidad, se implementó un *backend* que centraliza y organiza todas las funcionalidades de la plataforma. La solución se desarrolló utilizando *Django* y *Django Rest Framework*, junto con una base de datos *PostgreSQL*, lo que permitió modelar y relacionar adecuadamente los distintos componentes del sistema. Dado que el *frontend* de *EverPlanAPP* corresponde a una aplicación móvil (APK) y se encuentra desacoplado del *backend*, se definió una *API REST* como mecanismo de comunicación entre ambos, permitiendo que la aplicación cliente consuma los servicios de forma segura y estructurada. A través de esta *API*, *EverPlanAPP* puede satisfacer de manera efectiva las necesidades de organización, planificación y apoyo que requieren sus usuarios. Asimismo, se incorporaron mecanismos de autenticación y autorización mediante *tokens JWT*, garantizando que cada estudiante acceda únicamente a su propia información.

La metodología empleada para el desarrollo de esta memoria incluyó el levantamiento y análisis de requerimientos, el diseño del modelo de datos y de la arquitectura del *backend*, la implementación progresiva de las funcionalidades y la validación del sistema mediante pruebas sobre la *API*. Este enfoque permitió construir una solución estructurada y coherente con los objetivos de *EverPlanAPP*, asegurando que cada módulo respondiera a las necesidades reales de los usuarios.

CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA

1.1 Contexto General

En el contexto de la educación superior, la organización académica representa un desafío constante para gran parte del estudiantado. Esta dificultad se acentúa en estudiantes neurodivergentes, como aquellos que presentan Trastorno por Déficit de Atención e Hiperactividad (TDAH) o Trastorno del Espectro Autista (TEA), quienes suelen enfrentar barreras adicionales relacionadas con la gestión del tiempo, la estructuración de tareas y la autorregulación emocional.

A pesar de la existencia de múltiples herramientas digitales orientadas a la planificación, muchas de ellas no consideran las particularidades cognitivas ni sensoriales de este grupo de estudiantes. Interfaces recargadas, elementos de gamificación o flujos poco intuitivos pueden generar sobrecarga sensorial o desmotivación, dificultando aún más su experiencia académica.

Frente a esta realidad, surge la necesidad de desarrollar soluciones tecnológicas inclusivas que no solo faciliten la organización académica, sino que también fomenten la autonomía, la autorregulación y el bienestar emocional de los usuarios. En este escenario, se enmarca el desarrollo de *EverPlanAPP*, una aplicación diseñada específicamente para apoyar la planificación académica de estudiantes neurodivergentes. La presente memoria se centra en la construcción del *backend* y la base de datos de dicha aplicación, componentes clave para garantizar su funcionamiento robusto, seguro y escalable, en línea con los principios de accesibilidad e inclusión educativa.

1.2 Situación Actual

Actualmente, herramientas como *Todoist*, *MyStudyLife* o *Focus Plant* son ampliamente utilizadas por estudiantes para organizar sus tareas, rutinas y compromisos académicos. Sin embargo, muchas de estas plataformas pueden representar una barrera para personas con condiciones neurodivergentes, como aquellas dentro del espectro autista. Esto se debe a que algunas de estas aplicaciones incorporan elementos de gamificación o interfaces visualmente cargadas con el objetivo de hacer más atractiva la experiencia de usuario. No obstante, este tipo de estímulos pueden resultar sobre estimulantes o generar rechazo en usuarios autistas, quienes suelen preferir entornos visuales más simples, con menos elementos distractores, colores suaves y estructuras claras. En general, estas aplicaciones carecen de un enfoque inclusivo y un diseño sensorialmente accesible que minimice la sobrecarga cognitiva y emocional.

1.3 Actores Involucrados

Los principales beneficiarios de esta solución son estudiantes universitarios neurodivergentes, quienes enfrentan mayores desafíos en la gestión de sus actividades académicas. No obstante, la aplicación también puede ser útil para cualquier estudiante que requiera apoyo en su organización personal. Asimismo, se consideran actores secundarios a tutores, docentes y profesionales del ámbito psicopedagógico, quienes podrían utilizar la información generada por la aplicación como una herramienta complementaria para fortalecer el acompañamiento académico y la toma de decisiones pedagógicas.

1.4 Problema específico de la memoria

El problema específico que se aborda en esta memoria es la necesidad de desarrollar un *backend* funcional, estructurado y escalable para *EverPlanAPP*, capaz de integrar y gestionar adecuadamente los datos y funcionalidades requeridas por la aplicación. Sin una base *backend* sólida, la aplicación carecería de una infraestructura que permita almacenar, organizar y procesar información de forma segura y eficiente, lo que limitaría su funcionamiento, impediría la persistencia de datos entre sesiones y dificultaría la comunicación entre el *frontend* y la lógica del sistema.

Este *backend* debe responder a criterios de eficiencia, seguridad, mantenibilidad y compatibilidad con el *frontend*, permitiendo el correcto funcionamiento a las distintas funcionalidades que conforman la aplicación, como la gestión de tareas, horarios, recordatorios, estados de ánimo, estadísticas, entre otras. Además, debe considerar buenas prácticas en el diseño de arquitecturas y modelos de datos, asegurando una base tecnológica escalable para el crecimiento futuro del sistema.

1.5 Motivación e Interés del Proyecto

La motivación principal para llevar a cabo esta memoria surge de la necesidad de contribuir, desde el desarrollo tecnológico, a la creación de herramientas inclusivas que respondan a realidades diversas dentro del entorno universitario. La falta de aplicaciones adaptadas a las particularidades de estudiantes neurodivergentes representa una brecha significativa en términos de equidad y acceso al aprendizaje. Desde una perspectiva tanto profesional como personal, existe un interés genuino en aportar a una solución concreta que permita mejorar la experiencia académica de quienes enfrentan mayores barreras organizativas y sensoriales.

Además, el desarrollo del *backend* y la base de datos de *EverPlanAPP* ofrece una oportunidad formativa valiosa para aplicar conocimientos adquiridos durante la carrera, enfrentando desafíos reales relacionados con arquitectura de software, seguridad, escalabilidad y diseño centrado en el usuario.

1.6 Objetivo General

Desarrollar un *backend* funcional y seguro que permita gestionar usuarios, tareas, rutinas, recordatorios, horario de clases, estadísticas y sugerencias de organización en *EverPlanAPP*, asegurando integridad de datos, accesibilidad y posibilidad de escalabilidad.

1.7 Objetivos Específicos

Tabla 1: Objetivos específicos, marco conceptual y propuesta de solución del subproyecto Backend de EverPlanApp.

Fuente: Elaboración propia.

Objetivo Específico	Marco Conceptual	Propuesta de Solución
Evaluar <i>frameworks</i> para el backend	Principales <i>frameworks</i> en Python (Django, Flask, FastAPI). Comparación en rendimiento, escalabilidad, documentación, curva de aprendizaje y comunidad.	Analizar las características de cada <i>framework</i> para recomendar la opción más adecuada según los requerimientos del sistema.
Evaluar alternativas de arquitectura	Tipos de arquitectura de software para el <i>backend</i> : monolítica y basada en microservicios. Ventajas, desventajas y Análisis comparativo según complejidad, mantenibilidad y escalabilidad.	Comparar las distintas arquitecturas para seleccionar la más conveniente en términos de complejidad, mantenimiento, y escalabilidad futura.
Diseñar la estructura del sistema y modelo de datos	Bases de datos relacionales y NoSQL; modelos entidad-relación; herramientas de modelado; conceptos de normalización; integración con el <i>backend</i> .	Definir una estructura lógica del sistema y proponer un modelo de datos que se adapte a las necesidades funcionales y técnicas del proyecto.
Analizar mecanismos de seguridad para el <i>backend</i>	Seguridad web: autenticación (básica, token), control de acceso, buenas prácticas de manejo de datos sensibles, modularidad.	Identificar los mecanismos de seguridad más apropiados para proteger los datos y el acceso a las funcionalidades del sistema.
Establecer criterios de validación del <i>backend</i>	Principios de pruebas funcionales; herramientas como Postman o alternativas; validación de flujo, errores y seguridad; <i>testing</i> en desarrollo <i>backend</i> .	Definir una estrategia de validación que permita evaluar si el sistema cumple con sus requerimientos técnicos y funcionales.

CAPÍTULO 2: MARCO CONCEPTUAL

2.1 ¿Qué es un framework?

En ingeniería y programación de software, un *framework* (o marco de trabajo) es una estructura reutilizable que proporciona componentes, herramientas y reglas predefinidas para facilitar el desarrollo de aplicaciones. Su objetivo es estandarizar y agilizar el proceso de creación de software, permitiendo que los desarrolladores se enfoquen en la lógica específica del proyecto en lugar de construir todo desde cero.

Al igual que en otras ramas de la ingeniería donde se reutilizan componentes estandarizados, los *frameworks* permiten construir sobre bases sólidas y probadas, promoviendo buenas prácticas de diseño y arquitectura.

Ventajas de utilizar frameworks:

- Mayor eficiencia: reducen el tiempo de desarrollo al incluir funcionalidades comunes ya implementadas (como conexión a bases de datos, manejo de rutas, seguridad, etc.).
- Mejor calidad de código: fomentan estructuras claras y reutilizables, lo que mejora la legibilidad y reduce errores.
- Seguridad incorporada: muchos *frameworks* incluyen medidas de seguridad listas para usar.
- Facilitan el trabajo en equipo: al seguir una estructura común, varios desarrolladores pueden colaborar con mayor fluidez.
- Escalabilidad y flexibilidad: permiten adaptar o reemplazar módulos fácilmente sin reescribir toda la aplicación.

Referencia web: Amazon Web Services, 2025. *What is a Framework in Programming and Engineering?* Recuperado de https://aws.amazon.com/what-is/framework/?nc1=h_ls

2.2 Principales frameworks backend en Python

Para el presente proyecto se definió previamente el lenguaje de programación que se utilizaría durante el desarrollo. En este caso, se optó por Python, principalmente por la familiaridad del autor con este lenguaje y considerando el tiempo disponible para la implementación del sistema. Esta decisión también se justifica por la versatilidad y robustez que Python ofrece en el desarrollo de aplicaciones backend, junto con su amplia disponibilidad de bibliotecas, documentación y una comunidad activa de soporte.

Dado que Python será el lenguaje principal del proyecto, a continuación, se evaluarán los frameworks de backend más utilizados en este entorno: Django, Flask y FastAPI.

2.2.1 ¿Qué es Django?

Django es un framework web de alto nivel en Python que adopta un enfoque “todo incluido”, ideal para desarrollar aplicaciones robustas de forma rápida. Utilizado por empresas como Instagram, Spotify y Dropbox, Django se destaca por su madurez, estabilidad y desarrollo activo desde 2005. Está basado en la arquitectura Modelo-Vista-Plantilla (MVT), donde los modelos gestionan los datos, las vistas controlan la lógica del negocio y las plantillas presentan la información al usuario. Es especialmente adecuado para aplicaciones como redes sociales, sitios de noticias y plataformas de comercio electrónico.

Ventajas de Django:

- Enfoque “todo incluido”: Django ofrece funciones integradas como autenticación, almacenamiento en caché, validación de datos y gestión de sesiones.
- Principio DRY (*Don't Repeat Yourself*): promueve un desarrollo limpio y reutilizable
- ORM incorporado: facilita el trabajo con bases de datos relacionales como PostgreSQL, MySQL y SQLite sin necesidad de SQL explícito.
- Seguridad: Django tiene defensas integradas contra vulnerabilidades comunes como el *cross-site-scripting* (XSS), la inyección SQL y el *clickjacking*, lo que ayuda a proteger la aplicación desde el inicio del desarrollo.
- Escalabilidad: soporta técnicas como caché, tareas asíncronas y balanceo de carga para entornos de alto tráfico.
- Documentación y amplia comunidad: clara, oficial y constantemente actualizada, además de una activa desde 2005, con numerosos recursos, plugin y soporte comunitario.

Desventaja de Django:

- Peso del framework: su enfoque “todo incluido” puede resultar excesivo para proyectos pequeños o simple.
- Curva de aprendizaje: mayor complejidad inicial debido a la cantidad de herramientas y convenciones integradas.
- Rendimiento: Django suele ser más lento que otros marcos de trabajo como Flask o FastAPI, pero el almacenamiento en caché integrado y el procesamiento asíncrono pueden ayudar a mejorar los tiempos de respuesta.

Referencia web: JetBrains, 2000. Which Is the Best Python Web Framework: Django, Flask, or FastAPI? Recuperado de <https://blog.jetbrains.com/pycharm/2025/02/django-flask-fastapi/>

2.2.2 ¿Qué es Flask?

Flask es un microframework de desarrollo web en Python, diseñado con un núcleo simple y flexible basado en Werkzeug y Jinja2. A pesar de su nombre, no se limita a aplicaciones pequeñas y ha sido utilizado por empresas como Netflix, Airbnb y Reddit. A diferencia de *frameworks* más estructurados como Django, Flask sigue una filosofía minimalista, permitiendo a los desarrolladores elegir libremente sus componentes. Por ello, no incluye por defecto un ORM ni una capa de base de datos, ofreciendo mayor libertad y personalización en la construcción de aplicaciones.

Ventajas de Flask:

- Diseño ligero y minimalista: ideal para aplicación pequeñas o medianas, evitando el exceso de herramientas preintegradas. Sin embargo, Flask no se limita a proyectos pequeños, ya que puede ampliarse según las necesidades del sistema.
- Flexibilidad: Flask permite elegir libremente bibliotecas y componentes según las necesidades del proyecto.
- Escalabilidad: el diseño modular de Flask facilita la escalabilidad horizontal. El uso de una capa de base de datos NoSQL puede mejorar aún más la escalabilidad.
- Curva de aprendizaje suave: fácil de comenzar, especialmente útil para desarrolladores con menos experiencia.
- Documentación clara y comunidad activa: aunque más pequeña que la de Django, su comunidad crece constantemente y ofrece soporte técnico sólido.

Desventajas de Flask:

- Tráigalo todo usted mismo: al ser un microframework, requiere que el desarrollador implemente o integre manualmente funcionalidades clave como validación de datos, gestión de sesiones y almacenamiento en caché. Aunque esta flexibilidad puede ser beneficiosa, también puede ralentizar el proceso de desarrollo.
- Seguridad: Flask incluye pocas protecciones por defecto, por lo que es responsabilidad del desarrollador aplicar buenas prácticas y mantener actualizadas las librerías utilizadas.
- Rendimiento: aunque Flask rinde ligeramente mejor que Django, se queda por detrás de FastAPI. Flask ofrece algo de compatibilidad con ASGI (el estándar utilizado por FastAPI), pero está más estrechamente ligado a WSGI.

Referencia web: JetBrains, 2000. Which Is the Best Python Web Framework: Django, Flask, or FastAPI? Recuperado de <https://blog.jetbrains.com/pycharm/2025/02/django-flask-fastapi/>

2.2.3 ¿Qué es FastAPI?

FastAPI es un microframework moderno para el desarrollo de API web de alto rendimiento con Python. Lanzado en 2018, ha ganado gran popularidad por su enfoque asíncrono y su compatibilidad nativa con OpenAPI para la documentación automática. Basado en Uvicorn (ASGI) y Starlette, facilita la validación de datos, la serialización y el desarrollo eficiente de APIs RESTful. Está especialmente orientado a aplicaciones que requieren respuestas rápidas y concurrencia, como plataformas de mensajería, paneles en tiempo real o servicios que integran machine learning o procesamiento intensivo de datos.

Ventajas de FastAPI:

- Rendimiento: diseñado para ofrecer rapidez, FastAPI admite procesamiento asíncrono y sockets web bidireccionales (gracias a Starlette). Superó tanto a Django como a Flask en las pruebas de referencia, por lo que es ideal para aplicaciones de alto tráfico.
- Escalabilidad: al igual que Flask, FastAPI es muy modular, por lo que es fácil de escalar e ideal para despliegues en contenedores.
- Compatible con estándares: integra OAuth 2.0, OpenAPI y JSON schema, lo que simplifica la autenticación y la generación de documentación.
- Facilidad de uso: el uso de Pydantic por parte de FastAPI para sugerencias de tipo y validación acelera el desarrollo al proporcionar comprobaciones de tipo, autocompletado y validación de solicitudes.
- Documentación: FastAPI cuenta con amplia documentación oficial y recursos de terceros en aumento, útil para desarrolladores de todos los niveles.

Desventajas de FastAPI:

- Menor madurez: al ser más reciente, su comunidad es más pequeña y la experiencia de desarrollo puede ser menos pulida que en *frameworks* más consolidados.
- Compatibilidad: requiere integrar herramientas externas para implementar funcionalidades avanzadas. Dispone de menos bibliotecas compatibles en comparación con Django o Flask.

Referencia web: JetBrains, 2000. Which Is the Best Python Web Framework: Django, Flask, or FastAPI? Recuperado de <https://blog.jetbrains.com/pycharm/2025/02/django-flask-fastapi/>

2.2.4 Análisis final y elección del framework

Tabla 2: Resumen comparativo de Django, Flask y FastAPI para desarrollo backend.
Fuente: Elaboración propia en base a documentación oficial y comparativas técnicas.

Criterio	Django	Flask	FastAPI
Tipo	Framework completo para apps web con BD relacional.	Microframework ligero y flexible.	Microframework orientado a APIs rápidas y modernas.
Facilidad de uso	“Todo incluido”, pero con curva de aprendizaje alta.	Fácil de aprender, requiere más configuración manual.	Intermedio; validación y tipado facilitan el desarrollo.
Extensibilidad	Mayor cantidad de paquetes y extensiones disponibles.	Buena compatibilidad con librerías externas.	Menos paquetes compatibles que Flask o Django.
Rendimiento	Bueno, pero más lento que los otros dos.	Mejor que Django, pero inferior a FastAPI.	El más rápido de los tres.
Escalabilidad	Diseñado para manejar cargas altas y aplicaciones grandes.	Puede escalar bien, pero requiere más configuración.	Muy alta, ideal para microservicios y alto tráfico.
Seguridad	Amplias protecciones integradas.	Requiere agregar medidas adicionales manualmente.	Compatible con OAuth 2.0; requiere configurar otras protecciones.
Madurez	Activo desde 2005.	Activo desde 2010.	Activo desde 2018.
Comunidad	Muy amplia y consolidada.	Activa y en crecimiento.	Más reducida, pero en expansión.
Documentación	La más robusta y completa.	Extensa y clara.	Menos extensa, pero en crecimiento.

Después de revisar y comparar los principales frameworks backend en Python: Django, Flask y FastAPI, y considerar aspectos clave como el rendimiento, la escalabilidad, la facilidad de uso, la seguridad, la madurez y el respaldo comunitario, se concluye que Django es la opción más adecuada para el desarrollo de este proyecto.

La decisión se basa en su enfoque “todo incluido”, que proporciona desde el inicio una estructura completa con herramientas integradas como autenticación, gestión de sesiones, ORM y panel de administración. Esto no solo agiliza el desarrollo, sino que

también permite mantener una organización clara del proyecto sin depender en exceso de herramientas externas.

Aunque *Flask* y *FastAPI* ofrecen ventajas importantes en términos de flexibilidad o rendimiento en contextos específicos, Django se ajusta mejor a las necesidades actuales del proyecto, ya que se busca una solución segura, práctica y escalable para el *backend* de *EverPlanAPP*.

2.3 ¿Qué es la arquitectura de software?

La arquitectura de software define la organización fundamental de un sistema y, en términos más simples, define una solución estructurada. Determina cómo se ensamblan los diversos componentes de un sistema de software, cómo se relacionan entre sí y cómo se comunican. En esencia, sirve como modelo para la aplicación y como base sobre la que el equipo de desarrollo puede construir.

La arquitectura de software define una lista de cosas que hacen que muchas cosas sean más fáciles en el proceso de desarrollo de software.

- Estructura del sistema: la organización y disposición de los componentes.
- Comportamiento del sistema: La funcionalidad y el rendimiento esperados.
- Relaciones de componentes: cómo interactúan las diferentes partes del sistema.
- Estructura de comunicación: La forma en que los componentes se comunican entre sí.
- Equilibrio entre las partes interesadas: satisfacer las necesidades y expectativas de todas las partes interesadas.
- Estructura del equipo: cómo se organiza y coordina el equipo de desarrollo.
- Decisiones de diseño tempranas: tomar decisiones importantes desde el principio para guiar el desarrollo.

Referencia web: GeeksforGeeks, Sanchhaya Education Private Limited, 2025. Fundamentals of Software Architecture. Recuperado de <https://www.geeksforgeeks.org/fundamentals-of-software-architecture/>

2.3.1 Evaluación de alternativas de arquitectura

Una vez teniendo claro el lenguaje de programación y haber seleccionado el *framework backend* más adecuado, es fundamental analizar las distintas arquitecturas de software que pueden adoptarse para el desarrollo del sistema.

En este proyecto ya se ha definido el uso de una arquitectura desacoplada, en la que el *frontend* y el *backend* funcionan como sistemas independientes y se comunican mediante *APIs*. Esta decisión fue acordada junto al encargado del desarrollo *frontend*, con el objetivo de separar de forma más clara las responsabilidades de cada equipo. Por lo tanto, el análisis debe centrarse ahora en definir qué tipo de arquitectura se implementará en el *backend*: mantener un enfoque monolítico o adoptar una arquitectura basada en microservicios.

2.3.1.1 ¿Qué es una API?

Una API (*Application Programming Interface*) es un conjunto de normas, protocolos y herramientas que permite la comunicación entre diferentes sistemas de software. Actúa como un intermediario que define cómo los programas solicitan y reciben datos o funcionalidades de otros servicios, sin que el consumidor tenga que entender la implementación interna del proveedor del servicio

En el desarrollo web, las *APIs REST* son especialmente comunes: utilizan métodos HTTP (como GET, POST, PUT y DELETE) para acceder y manipular recursos identificados mediante URLs. Los datos se intercambian generalmente en formato JSON, lo que permite una integración sencilla entre *frontend* y *backend*.

Este tipo de API ofrece varias ventajas esenciales:

- Promueven una arquitectura de cliente-servidor, donde el *frontend* y el *backend* están desacoplados.
- Facilitan la interoperabilidad, permitiendo que distintas aplicaciones o servicios trabajen juntos sin la necesidad de conocer los detalles internos de cada uno.
- Mejoran la escalabilidad y mantenibilidad, al seguir estándares de diseño como REST, que promueven interfaces uniformes y manejo correcto del estado mediante métodos HTTP.

Referencia web: IBM, 2025. What is a REST API? Recuperado de <https://www.ibm.com/think/topics/rest-apis>

2.3.1.2 ¿Qué es la arquitectura monolítica?

Una arquitectura monolítica es aquella donde todos los componentes de una aplicación, como la interfaz de usuario, la lógica de negocios y el acceso a datos, están integrados en un único bloque de código. Todo el sistema se desarrolla, prueba y despliega como una única unidad.

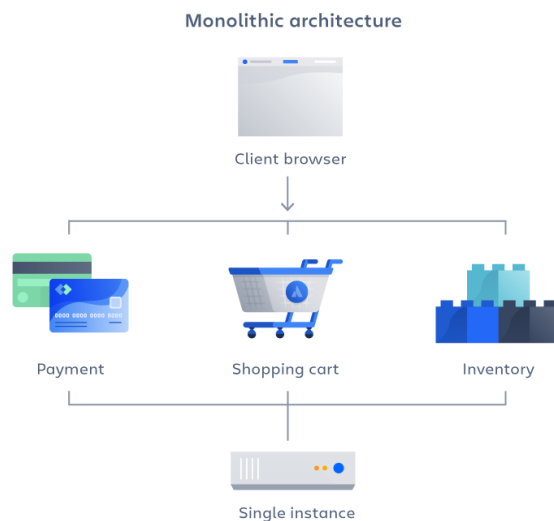


Figura 1: Arquitectura monolítica.

Fuente: Atlassian

Este modelo ofrece ventajas significativas en proyecto que requieren rapidez y simplicidad. Al centralizar todo en una sola base de código, se facilita la implementación, el desarrollo y el proceso de prueba y depuración, lo que reduce la complejidad técnica en etapas iniciales del proyecto.

Entre sus principales ventajas se encuentran:

- Implementación sencilla: se despliega como un único archivo o directorio.
- Desarrollo centralizado: facilita la gestión del código y la colaboración.
- Buen rendimiento: una sola *API* se puede realizar tareas que, en microservicios, requerirían múltiples llamadas.
- Pruebas más rápidas: al ser una unidad única, las pruebas integrales son más simples.
- Depuración directa: al tener todo el código en un mismo lugar, es más fácil rastrear errores.

Aun que una arquitectura monolítica puede ser efectiva en etapas iniciales, su escalabilidad y flexibilidad se vuelven limitadas a medida que la aplicación comienza a crecer. Esto puede dificultar la implementación de prácticas ágiles y aumentar los tiempos de desarrollo.

Principales desventajas:

- Desarrollo más lento: una aplicación grande y monolítica hace que el desarrollo sea más complejo y lento.
- Escalabilidad: los módulos no pueden escalarse de forma independiente.
- Poca confiabilidad: un fallo en un componente puede afectar a toda la aplicación.
- Barrera para la adopción de tecnología: cambiar el lenguaje o *framework* impacta todo el sistema.
- Falta de flexibilidad: un monolito está limitado por las tecnologías ya utilizadas en él.
- Implementación: cualquier modificación requiere recompilar y volver a desplegar toda la aplicación.

Referencia web: Atlassian, 2025. Microservices vs. monolithic architecture. Recuperado de <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>

2.3.1.3 ¿Qué es la arquitectura basada en microservicios?

La arquitectura de microservicios consiste en descomponer una aplicación en servicios pequeños e independientes, cada uno con su propia lógica y base de datos, que se comunican entre sí a través de APIs.

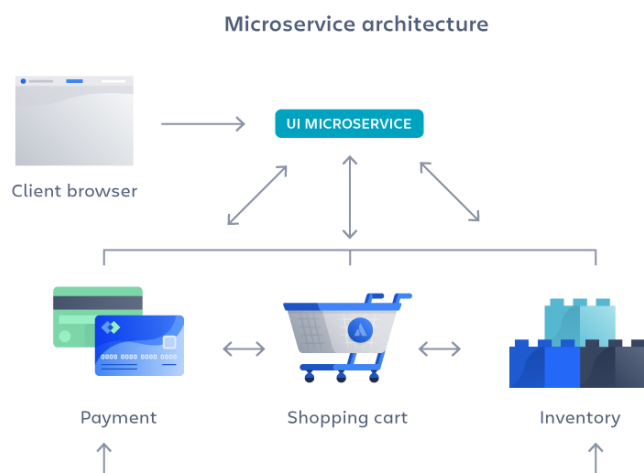


Figura 2: Arquitectura basada en microservicios.

Fuente: Atlassian.

Este enfoque permite una organización modular del sistema, lo que brinda múltiples beneficios para el desarrollo, escalabilidad y mantenimiento del software, especialmente en entornos complejos o en crecimiento.

Entre sus principales ventajas se encuentran:

- **Agilidad:** promover formas ágiles de trabajar con equipos pequeños que se implementan con frecuencia.
- **Escalabilidad flexible:** permite aumentar instancias de servicios específicos según la demanda, mejorando la capacidad del sistema.
- **Implementación continua:** facilita actualizaciones frecuentes y rápidas, agilizando el ciclo de entrega.
- **Altamente mantenible y *testable*:** permite probar, actualizar y revertir funciones de forma aislada, acelerando el desarrollo y solución de errores.
- **Implementación independiente:** cada función puede desplegarse por separado sin afectar al resto del sistema.
- **Flexibilidad tecnológica:** los equipos pueden elegir las herramientas y lenguajes que mejor se adapten a cada servicio.
- **Alta confiabilidad:** un fallo en un servicio no compromete toda la aplicación.
- **Equipos más satisfechos:** favorece la autonomía y reduce los tiempos de espera entre equipos.

Principales desventajas:

- **Expansión del desarrollo:** aumenta la complejidad por la cantidad de servicios distribuidos y equipos involucrados, lo que puede ralentizar el desarrollo si no se gestiona bien.
- **Costos de infraestructura exponenciales:** cada microservicio requiere recursos propios para pruebas, despliegue, alojamiento y monitoreo, elevando los costos.
- **Mayor carga organizativa:** se necesita más coordinación entre equipos para mantener la coherencia y sincronización entre servicios.
- **Desafíos de depuración:** la distribución de servicios y registros dificulta rastrear errores y procesos de negocio.
- **Falta de estandarización:** la libertad de tecnológica puede generar diferencias en lenguajes, monitoreo y registros entre servicios.

- Falta de propiedad clara: con muchos servicios y equipos, se vuelve difícil identificar responsables y puntos de contacto para soporte.

Referencia web: Atlassian, 2025. Microservices vs. monolithic architecture. Recuperado de <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>

2.3.2 Análisis final y elección de la arquitectura de software

Después de analizar en profundidad la arquitectura monolítica y de microservicios, y considerando que el *framework* elegido para este proyecto es Django, se identificó una alternativa que combina lo mejor de ambos mundos: la arquitectura monolítica modular. Esta opción representa un equilibrio ideal, ya que permite mantener una única base de código y un despliegue simplificado, sin la complejidad operativa que conlleva una infraestructura distribuida como la de los microservicios.

Al mismo tiempo, gracias a la estructura de *apps* que ofrece Django, es posible dividir la aplicación en módulos internos con responsabilidades bien definidas. Esto favorece la separación de funcionalidades, mejora el mantenimiento y facilita el trabajo colaborativo entre equipos. En la práctica, cada *app* puede desarrollarse, probarse y escalarse de forma más organizada, lo que permite alcanzar beneficios similares a los de los microservicios, pero dentro de un entorno más simple y coherente.

2.3.2.1 ¿Qué es una Arquitectura monolítica modular?

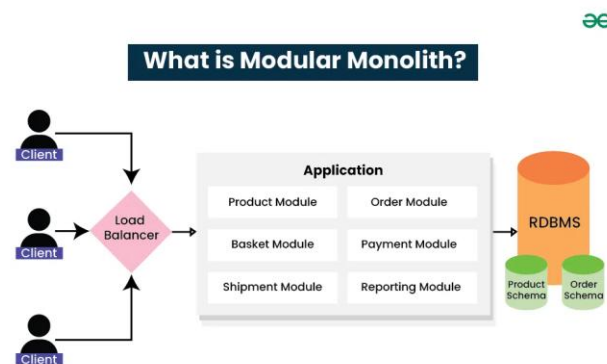


Figura 3: Arquitectura monolítica modular.

Fuente: GeeksforGeeks.

Una arquitectura modular monolítica modular es un estilo de diseño de software que combina lo mejor de las arquitecturas anteriormente mencionadas, mantiene una estructura monolítica (una sola base de código, un único despliegue), pero está organizada internamente en módulos independientes y bien definidos.

Cada módulo en este tipo de arquitectura representa una unidad funcional concreta (por ejemplo, autenticación, gestión de usuarios, tareas) y no depende directamente de otros módulos, sino que interactúa a través de interfaces o contratos bien establecidos. Aunque todos los módulos viven en el mismo repositorio y se despliegan juntos, están aislados lógicamente.

Principales ventajas de la arquitectura monolítica modular:

- Simplicidad: una sola base de código facilita el desarrollo y gestión.
- Modularidad: los módulos independientes permiten una mejor organización y mantenimiento.
- Escalabilidad: es posible escalar módulos específicos sin complicar el sistema.
- Mantenibilidad: se simplifica la incorporación de mejoras y correcciones.
- Despliegue sencillo: solo requiere gestionar un artefacto, reduciendo la complejidad.

Referencia web: GeeksforGeeks, Sanchhaya Education Private Limited, 2024. What Is a Modular Monolith? Recuperado de <https://www.geeksforgeeks.org/system-design/what-is-a-modular-monolith/>

Comparación entre arquitecturas:

Ítem	Monolito	Microservicio	Monolito Modular
Agilidad del equipo	✗	✓	✓
Tolerancia a fallos a nivel de proceso	✗	✓	✗
Escalabilidad modular	✗	✓	✗
Contexto delimitado forzado	✗	✓	✓
Implementaciones protegidas	✗	✓	✓
Testeable por separado	✗	✓	✓
Trazabilidad distribuida	✗	✓	✓
Fácil de redefinir límites	✗	✓	✓
Evitar dispersión de repositorios Git	✓	✗	✓
Evitar gestión de sistemas distribuidos	✓	✗	✓

Figura 4: Comparativa entre arquitecturas.

Fuente: VCSOFT.

2.4 Diseñar la estructura del sistema y modelo de datos

En esta sección analizaremos la estructura lógica del sistema y las posibles alternativas para diseñar un modelo de datos que responda adecuadamente a los requerimientos funcionales y técnicos del proyecto. Para ello, abordaremos conceptos fundamentales como el uso de bases de datos relacionales y NoSQL, la aplicación de modelos entidad-relación (MER), principios de normalización, herramientas de modelado, y su integración con el *backend*. El objetivo es identificar qué enfoque de modelado y tipo de base de datos resulta más adecuado, considerando criterios como mantenibilidad, escalabilidad y compatibilidad con el *framework* previamente seleccionado.

2.4.1 ¿Qué tipo de base de datos deberíamos elegir?

Dado que el *framework* seleccionado para el desarrollo del sistema es Django, resulta natural optar por una base de datos relacional (SQL), ya que Django tiene el beneficio de incorporar ORM (*Object Relational Mapping*) que permite modelar, consultar y administrar datos de manera eficiente mediante objetos de Python. Este enfoque no solo permite agilizar el desarrollo, sino que también garantiza que la integración entre el *backend* y nuestra base de datos sea fluida. Si bien existen alternativas basadas en NoSQL, como MongoDB, que pueden ser adecuadas para sistemas que manejen grandes cantidades de datos no estructurados o que requieran alta escalabilidad horizontal, estas no se ajustan al escenario actual de nuestro proyecto *EverPlanApp*. En este caso, se trata de una aplicación organizacional de pequeña escala, con datos estructurados y relaciones bien definidas, por lo que una base de datos relacional representa la solución más adecuada, simple y coherente con los objetivos funcionales y técnicos de nuestro sistema.

Referencia web: Django, 2005-2025. Making queries. Recuperado de <https://docs.djangoproject.com/en/5.2/topics/db/queries/>

2.4.1.1 ORM en Django: Definición y funcionamiento

El ORM (*Object Relational Mapping*), o Mapeo Objeto Relacional, es una técnica que permite trabajar con bases de datos relacionales mediante clases y objetos, en lugar de usar sentencias SQL directamente. Django incorpora un ORM que simplifica esta tarea, permitiendo crear, leer, actualizar y eliminar datos (operaciones CRUD) desde el propio código Python.

En Django, cada modelo (una clase definida por el desarrollador) representa una tabla en la base de datos, y cada instancia de esa clase equivale a una fila de la tabla. De esta forma, es posible interactuar con la base de datos de forma intuitiva, sin necesidad de conocer la sintaxis de SQL.

Esta abstracción no solo mejora la legibilidad del código, sino que también agiliza el desarrollo, facilita el mantenimiento y ayuda a prevenir errores comunes, como la

inyección de SQL. Gracias al ORM, las operaciones con datos se vuelven más seguras, estructuradas y alineadas con el lenguaje del proyecto.

Referencia web: Documentación oficial de Django, 2005-2025. *Making queries*. Recuperado de <https://docs.djangoproject.com/en/5.2/topics/db/queries/>

Referencia web: Documentación oficial de Django, 2005-2025. *Models*. Recuperado de <https://docs.djangoproject.com/en/5.2/topics/db/models/>

2.4.1.2 ¿Qué es una base de datos relacional?

Una base de datos relacional es una forma de guardar información de manera ordenada, como si fuera una hoja de cálculo. La información se organiza en tablas, donde cada tabla tiene filas (que representan registros, como una persona o un producto) y columnas (que representan los datos específicos, como nombre, edad o precio).

Lo que hace especial a este tipo de base de datos es que las tablas pueden estar conectadas entre sí, lo que permite relacionar información fácilmente. Además, se usa un lenguaje llamado SQL que ayuda a buscar, agregar o cambiar datos de forma rápida y precisa.

Gracias a esta estructura, las bases de datos relacionales son muy útiles cuando es importante que la información esté bien organizada y sea confiable, como en sistemas de bancos, inventarios o aplicaciones que manejan muchos datos.

Referencia web: Atlan, 2024. *Relational vs NoSQL Databases: Which is Right for You in 2024?* Recuperado de <https://atlan.com/relational-database-vs-nosql/>

2.4.2 ¿Que son las herramientas de modelado y el modelo Entidad-Relación?

Las herramientas de modelado de datos permiten representar de forma estructurada y visual la organización de los datos que utilizará un sistema. Este proceso se desarrolla en tres niveles:

- Modelado conceptual: identifica las entidades y sus relaciones principales sin entrar en detalles técnicos.
- Modelo lógico: define los atributos, claves y restricciones que regulan los datos.
- Modelo físico: traduce el modelo lógico a un sistema gestor de base de datos, considerando tipos de datos y aspectos de rendimiento.

El modelo entidad-relación (ER) es una metodología de modelado conceptual fundamental para el diseño de bases de datos relacionales. Fue introducido por [Chen(1976)], y desde entonces ha sido ampliamente adoptado en la ingeniería de software y sistemas de información.

Este modelo describe de manera clara las entidades (como personas, productos o cursos), sus características (atributos) y las relaciones entre ellas. Gracias a su simplicidad y utilidad, el modelo ER se sigue usando en herramientas modernas para planificar y construir bases de datos bien estructuradas.

Entre las herramientas más usadas para modelar datos destacan *Lucidchart*, *draw.io*, *dbdiagram.io*, *MySQL Workbench* y *pgModeler*, las cuales permiten crear y validar diagramas entidad-relación de forma visual y eficiente.

2.4.3 Análisis y utilidad del modelo entidad-relación (ER) en EverPlanAPP

El modelo entidad-relación (ER) es una herramienta importante que permitirá organizar correctamente las entidades que implementaremos en el modelo de Django. Su uso permitirá:

- Definir con claridad los datos del sistema: ya que nos permite identificar qué entidades serán necesarias, como usuarios, tareas, rutinas, estados de ánimo, estadísticas, etc. Además, permite visualizar sus atributos, por ejemplo: nombre, fecha, prioridad o estado.
- Establecer relaciones correctas entre los datos: el modelo ER permite representar cómo se conectan las entidades. Por ejemplo: Un usuario puede tener muchas tareas.
- Evitar errores y duplicación de información: diseñar bien las relaciones y claves permite mantener la base de datos ordenada, sin datos repetidos ni inconsistencias.
- Facilitar la implementación en Django mediante ORM: a partir del modelo ER, se puede construir la base de datos usando modelos en Django (*model.py*), aplicando relaciones como *ForeignKey* o *ManyToManyField*, de forma clara y coherente.
- Servir como respaldo técnico: el modelo ER es útil para documentar la estructura lógica del sistema y justificar cómo se diseñó la base de datos según las necesidades del proyecto.

2.4.4 Optimizar la base de datos con normalización

La normalización es una técnica que ayuda a organizar bien los datos en una base de datos, evitando repeticiones y errores. En *EverPlanApp*, aplicar normalización permite tener tablas bien estructuradas, con relaciones claras entre usuarios, tareas, rutinas, estados de ánimo, entre otros.

Usar las primeras formas normales (1NF, 2NF y 3NF) ayuda a mantener la información ordenada y sin duplicados, lo que mejora la velocidad de búsqueda, facilita futuros cambios en el sistema y evita inconsistencias. En resumen, la normalización mejora el funcionamiento del *backend* y ofrece una experiencia más segura y estable para los usuarios.

Libro: Elmasri, R., & Navathe, S. (2020). *Fundamentals of Database Systems (7th Edition)*. Pearson.

2.5 Analizar mecanismos de seguridad para el backend

En *EverPlanApp*, la seguridad del *backend* es un aspecto clave, ya que la aplicación maneja información personal, académica y emocional de los usuarios. Proteger estos datos es especialmente importante al estar dirigida a estudiantes neurodivergentes, quienes necesitan una plataforma confiable y segura. Para lograrlo, se implementan diversos mecanismos de seguridad web, muchos de los cuales son facilitados por el *framework* Django. Entre los principales se incluyen:

- Autenticación segura: mediante autenticación básica, por tokens o con JWT.
- Control de acceso: definición de permisos y roles para limitar el acceso según el perfil del usuario.
- Cifrado de contraseñas y datos sensibles: uso de algoritmos de hash y almacenamiento seguro.
- Protección contra ataques comunes: como inyección SQL, XSS (*cross-site scripting*) y CSRF (*cross-site request forgery*)
- Validación de entradas: para evitar que datos maliciosos afecten la integridad del sistema.
- Uso de HTTPS: para asegurar la transmisión cifrada de la información.
- Modularidad del sistema: separación de funciones mediante middleware y apps, lo que mejora la organización y el control de seguridad.

Aplicar estas medidas no solo protege la aplicación frente a posibles fallos de seguridad, sino que también contribuye a mantener el sistema estable y confiable.

Referencia web: Documentación oficial de Django, 2005-2025. *Security in Django*
Recuperado de <https://docs.djangoproject.com/en/5.2/topics/security/>

2.6 Criterios de validación del backend

En el desarrollo de aplicaciones web, es fundamental establecer criterios de validación que permitan asegurar que el *backend* cumple correctamente con sus funciones. Estos criterios se utilizan para evaluar si el sistema responde adecuadamente ante diferentes tipos de uso, errores y condiciones inesperadas. Entre ellos se incluyen:

- Cumplimiento funcional: verificar que cada funcionalidad del sistema se comporta como fue definida.
- Validación del flujo lógico: asegurar que las acciones del sistema sigues una secuencia coherente y esperada.
- Manejo de errores: comprobar que el *backend* responde correctamente frente a entradas inválidas, errores del cliente o del servidor.
- Seguridad básica: revisar que no se pueda acceder a datos o funcionalidades protegidas sin autorización.
- Correcta comunicación con la base de datos: validar que los datos se almacenan, actualizan y consultan de forma consistente.
- Códigos de respuesta HTTP adecuados: evaluar que el servidor devuelva los códigos de estado correctos ante cada solicitud.

Estas validaciones pueden realizarse con herramientas que permiten simular peticiones al *backend*, evaluar sus respuestas y detectar fallos. Aplicar criterios claros de validación es clave para asegurar la calidad, seguridad y estabilidad del sistema antes de su implementación final.

Referencia web: PYNT, 2025. *API Functional Testing: Practical Guide with Examples* Recuperado de <https://www.pynt.io/learning-hub/api-security-testing-guides/api-functional-testing>

Referencia web: STACKHAWK, 2025. *Functional API Testing: A Complete Guide* Recuperado de <https://www.stackhawk.com/blog/functional-api-testing/>

2.6.1 Herramientas de validación

Para asegurar la calidad del *backend* en desarrollo web, se utilizan diversas herramientas de validación. Estas permiten comprobar el correcto funcionamiento, la integridad del API y el comportamiento ante distintos escenarios. A continuación, se describen tres herramientas ampliamente reconocidas:

Postman

Postman es una herramienta muy popular que se usa para probar si una API funciona correctamente. Permite enviar distintos tipos de solicitudes (como GET o POST), revisar si las respuestas son correctas y simula errores o casos especiales. También se pueden automatizar pruebas y comprobar que todo siga funcionando como se espera.

Swagger

Swagger Validator es una herramienta que revisa si la documentación de tu API está bien escrita y sigue las reglas del estándar *OpenAPI*. Ayuda a que lo que está documentado coincida con lo que realmente hace el *backend*, evitando errores y malentendidos entre el *frontend* y el *backend*.

Jest

Jest es una herramienta que sirve para probar que las funciones del *backend* hacen lo que deberían. También te permite simular respuestas de la API y revisar que los datos devueltos sean correctos. Es muy usada cuando el *backend* está hecho en *JavaScript* (como con Node.js).

Referencia web: Documentación oficial de Postman, 2025. *Postman documentation overview* Recuperado de <https://learning.postman.com/docs/introduction/overview/>

Referencia web: Documentación oficial de Swagger, 2025. *OpenAPI Specification* Recuperado de <https://swagger.io/specification/>

Referencia web: Documentación oficial de Jest, 2025. *Getting Started* Recuperado de <https://jestjs.io/docs/getting-started>

CAPÍTULO 3: PROPUESTA DE SOLUCION

3.1 Diseño de la arquitectura general de EverPlanAPP

La solución propuesta para *EverPlanAPP* se basa en una arquitectura de software cliente-servidor, donde el *frontend* y el *backend* se comunican a través de una *API REST*. Esta arquitectura separa claramente la capa de presentación de la lógica de negocio, permitiendo que cada parte del sistema se desarrolle y mantenga de forma independiente.

Esta separación facilita el trabajo en equipo, ya que el *frontend* y el *backend* se desarrollan en repositorios independientes utilizando control de versiones con Git. De esta manera, los cambios realizados en una parte del sistema no afectan directamente a la otra, lo que reduce conflictos y mejora la organización del proceso de desarrollo.

A nivel de *backend*, *EverPlanAPP* utiliza una arquitectura monolítica modular, implementada con el *framework Django*. El sistema cuenta con una única base de código y un único despliegue, pero se encuentra organizado internamente en módulos funcionales bien definidos, como autenticación, gestión de usuarios o gestión de tareas. Esta organización permite incorporar nuevas funcionalidades de forma más sencilla y facilita la detección y corrección de errores.

Para la persistencia de datos se utiliza una base de datos *PostgreSQL*, a la cual accede exclusivamente el *backend* mediante el *ORM* de *Django*. Este *ORM* permite definir la estructura de la base de datos directamente desde los modelos del *backend*, evitando la necesidad de escribir consultas *SQL* de forma manual.

A partir de estos modelos, *Django* genera las migraciones correspondientes mediante el comando *makemigrations*, y posteriormente crea o actualiza las tablas en la base de datos utilizando el comando *migrate*. De esta forma, la estructura de la base de datos se construye y mantiene de manera automática y consistente a partir del código del *backend*.

Tanto el *backend* como la base de datos se ejecutan dentro de contenedores *Docker*, los cuales son gestionados mediante *Docker Compose*. Esto permite levantar y administrar ambos servicios de forma conjunta, facilitando el entorno de desarrollo y asegurando una configuración reproducible entre distintos entornos.

En el ámbito de *DevOps* y despliegue, el *backend* se encuentra alojado en un servidor en la nube sobre *Amazon Web Services (AWS)*, utilizando un sistema operativo *Ubuntu*. El proceso de integración y despliegue continuo se apoya en *Jenkins*, herramienta que permite automatizar tareas relacionadas con la construcción, prueba y despliegue del *backend*. Por su parte, el *frontend* se desarrolla como una aplicación móvil utilizando *React Native* y *Expo*, a partir de la cual se genera un archivo *APK* que es instalado en el dispositivo del usuario final.



Fuente: Elaboración propia.

Los requerimientos funcionales de *EverPlanAPP* se definieron a partir de la necesidad de apoyar la organización personal, la gestión del tiempo y la formación de hábitos, especialmente en personas con neurodivergencia. A partir del análisis realizado en el marco conceptual y de las necesidades detectadas en los usuarios, se identificaron las funcionalidades esenciales que el sistema debía ofrecer. Estos requerimientos sirvieron como base para la definición de los módulos del *backend* y del posterior diseño del modelo de datos.

De manera general, los requerimientos funcionales considerados para el *backend* del sistema son los siguientes:

1. **Gestión de usuarios y autenticación:** El sistema debe permitir el registro y la autenticación de los usuarios, definiendo un mecanismo de control de acceso a las distintas funcionalidades de la aplicación. Este requerimiento es fundamental, ya que permite identificar a cada usuario y asegurar que la información gestionada sea personal y privada.
2. **Gestión de tareas:** El sistema debe permitir a los usuarios crear, editar, eliminar y listar tareas personales, entendidas como actividades puntuales que ocurren una sola vez. Cada tarea debe asociarse a información temporal, como fecha y hora, además de atributos descriptivos que faciliten su organización.
3. **Gestión de rutinas recurrentes:** El sistema debe permitir la creación y gestión de rutinas, las cuales representan actividades que se repiten en el tiempo. Este requerimiento considera la configuración de distintos tipos de recurrencia, como diaria, semanal o mensual, así como la definición de un período de duración.
4. **Gestión de horarios personales:** El sistema debe permitir gestionar horarios personales o académicos mediante la creación de eventos organizados en bloques temporales. Cada evento puede incorporar información descriptiva y visual, facilitando la comprensión de la distribución del tiempo del usuario.
5. **Notificaciones programadas:** El sistema debe generar notificaciones asociadas a tareas y rutinas, las cuales se activan automáticamente cuando se cumple la fecha y hora definidas por el usuario, apoyando el cumplimiento de las actividades planificadas.
6. **Visualización de estadísticas básicas:** El sistema debe permitir visualizar estadísticas simples relacionadas con el cumplimiento de tareas, con un enfoque motivacional, reutilizando la información existente sin aumentar la complejidad del sistema en esta etapa del desarrollo.

3.3 Diseño funcional de los módulos del backend

Los módulos que componen el *backend* de *EverPlanAPP* se definieron a partir de los requerimientos funcionales descritos anteriormente y de las necesidades detectadas en los usuarios a los que está orientada la aplicación. Para organizar de mejor manera la lógica del sistema, se optó por dividir el *backend* en módulos independientes, cada uno enfocado en una funcionalidad específica.

En base a lo anterior, el *backend* se estructuró en los siguientes módulos funcionales:

1. **Autenticación y gestión de usuarios (auth_app):** Este módulo define el modelo de usuario de la aplicación e incluye los campos necesarios para el proceso de autenticación. Su objetivo es controlar el acceso a las distintas funcionalidades del sistema, permitiendo diferenciar entre tipos de usuarios, como administradores o usuarios normales, y utilizando autenticación mediante tokens JWT para asegurar que solo los usuarios registrados puedan utilizar la aplicación.

2. **Gestión de tareas (tarea_app):** Este módulo surge de la necesidad de que los usuarios puedan organizar sus actividades diarias de manera simple. A través de él es posible crear, editar, eliminar y listar tareas, además de asociarles fechas, horas y recordatorios. Su diseño busca facilitar el seguimiento de las tareas personales y permitir que cada usuario adapte su uso según sus propias necesidades.
3. **Gestión de rutinas (rutinas_app):** Este módulo se diferencia del módulo de tareas debido a su carácter recurrente. Las rutinas representan actividades que se repiten en el tiempo, ya sea de forma diaria, semanal o mensual, considerando fechas de inicio y término, así como horarios definidos. Esta diferencia hizo necesario tratarlas como un módulo independiente, tanto a nivel lógico como de modelado de datos.
4. **Gestión de horarios personales (horarioPersonal_app):** Este módulo permite a los usuarios gestionar sus horarios, como por ejemplo horarios de clases u otras actividades planificadas. Permite almacenar información adicional, como el nombre de la actividad, una descripción, la sala o incluso una imagen de referencia, con el fin de facilitar una mejor organización y comprensión visual del horario.
5. **Gestión de notificaciones (notificacion_app):** Este módulo se encarga de gestionar los avisos asociados a tareas y rutinas. Su función principal es notificar al usuario cuando se cumple una fecha y hora previamente establecida, permitiendo que las notificaciones se activen directamente en el dispositivo móvil. Este módulo se relaciona con otros módulos del sistema, ya que depende de la información generada por las tareas y las rutinas.

Adicionalmente, durante el desarrollo de la aplicación se consideraron otras funcionalidades asociadas a este enfoque inclusivo, tales como el uso de pictogramas y herramientas de autorregulación. Sin embargo, estas funcionalidades no requirieron implementación a nivel de *backend*, ya que fueron desarrolladas completamente en el *frontend*. Aun así, no se descarta que, en futuras versiones de la aplicación, sea necesario incorporar persistencia de datos para este tipo de características, lo que implicaría la creación de nuevas entidades dentro del modelo de datos.

En cuanto al módulo de estadísticas, este no fue implementado como un módulo independiente en esta etapa del proyecto. Las estadísticas con enfoque motivacional se integraron directamente dentro del módulo de gestión de tareas, reutilizando la información existente asociada a estas. Esta decisión permitió implementar una solución simple y funcional como prueba inicial, evitando aumentar la complejidad del sistema en una primera versión.

Finalmente, no se definió un módulo específico para la gestión del perfil de usuario, ya que esta funcionalidad fue implementada directamente en el *frontend* utilizando la información proporcionada por el módulo de autenticación (*auth_app*).

3.4 Arquitectura del Backend de EverPlanAPP

El *backend* de *EverPlanAPP* es responsable de implementar la lógica de negocio del sistema y de gestionar la persistencia de los datos. En esta capa se concentran las operaciones asociadas a las principales funcionalidades de la aplicación, tales como la gestión de usuarios, tareas, rutinas, horarios personales y notificaciones, asegurando un funcionamiento coherente y controlado del sistema.

Siguiendo la arquitectura general definida en la sección anterior, el *backend* se estructura como una aplicación monolítica modular desarrollada con Django. Esta organización permite que, a pesar de existir una única base de código y un único despliegue, el sistema se encuentre dividido internamente en módulos funcionales independientes, denominados *apps*, cada uno orientado a una responsabilidad específica dentro del dominio de la aplicación.

En la siguiente figura se presenta la organización interna del *backend* de *EverPlanAPP*, donde se observa la separación del proyecto en distintas aplicaciones Django, tales como autenticación, gestión de tareas, rutinas, horarios personales y notificaciones. Esta estructura modular facilita la mantenibilidad del código, reduce el acoplamiento entre funcionalidades y permite la evolución progresiva del sistema a medida que se incorporan nuevas características.

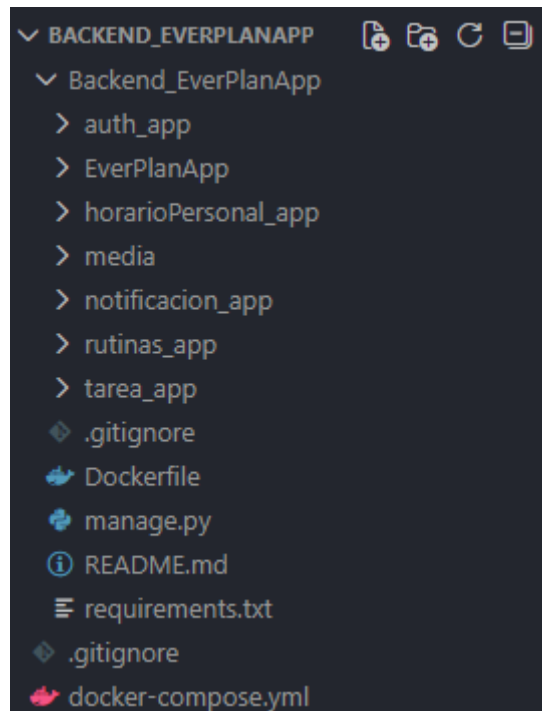


Figura 6: Arquitectura del backend de EverPlanAPP.

Fuente: Elaboración propia.

3.4.1 Estructura general de los módulos del backend

Con el fin de comprender cómo se implementan los módulos definidos anteriormente, en esta sección se describe la estructura general utilizada en el *backend*. El *backend* de *EverPlanAPP* se estructura siguiendo la organización estándar de *Django*, en la cual cada funcionalidad del sistema se implementa mediante aplicaciones independientes (*apps*). Esta disposición permite agrupar de forma coherente los distintos componentes asociados a una misma responsabilidad funcional.

De manera general, cada *app* presenta una estructura interna común. El archivo *models.py* define las entidades del dominio y su correspondencia con la base de datos, mientras que *views.py* contiene la lógica asociada a cada funcionalidad, implementando los *endpoints* que serán consumidos por el *frontend*. Esta separación permite mantener una clara distinción entre la representación de los datos y el procesamiento de las solicitudes.

El intercambio de información entre el *backend* y el *frontend* se gestiona mediante el uso de *serializers.py*, encargado de transformar los modelos de datos en formatos adecuados para su transmisión. Por su parte, el archivo *urls.py* define las rutas específicas de cada módulo, contribuyendo a una organización clara y mantenible de los *endpoints* expuestos por la aplicación.

Adicionalmente, cada módulo incorpora archivos como *admin.py*, destinado al registro de los modelos en la interfaz administrativa de *Django*, y *tests.py*, utilizado para la validación del comportamiento de las funcionalidades implementadas. Esta estructura interna facilita el mantenimiento del código y permite una evolución progresiva del sistema.

La configuración global del *backend* se centraliza en la carpeta principal del proyecto *Django*, donde se definen los parámetros generales de funcionamiento y las rutas principales del sistema. Complementariamente, el proyecto incorpora archivos de configuración asociados a *Docker* y *Docker Compose*, los cuales permiten estandarizar el entorno de ejecución y simplificar el despliegue del *backend*.

```
backend/
|
├── .gitignore           # Archivos y carpetas excluidos del control de versiones
├── Dockerfile           # Definición de la imagen Docker del backend
├── docker-compose.yml   # Configuración de servicios y contenedores
├── manage.py            # Punto de entrada para comandos de Django
├── requirements.txt      # Dependencias del proyecto
|
├── backend/             # Configuración principal del proyecto Django
|   ├── __init__.py
|   ├── asgi.py           # Configuración para servidores ASGI
|   ├── settings.py       # Configuración global del proyecto
|   ├── urls.py           # Definición de rutas principales del backend
|   └── wsgi.py           # Configuración para servidores WSGI
|
├── auth_app/           # Aplicación de autenticación
|   ├── __init__.py
|   ├── admin.py         # Registro de modelos en la interfaz administrativa
|   ├── apps.py          # Configuración de la aplicación
|   ├── models.py        # Definición de modelos de datos
|   ├── serializers.py    # Transformación de datos para la API
|   ├── urls.py          # Rutas propias de la aplicación
|   ├── views.py         # Lógica de procesamiento de solicitudes
|   └── tests.py          # Pruebas automatizadas de la aplicación
|
├── tarea_app/          # Aplicación de gestión de tareas
|
├── rutinas_app/         # Aplicación de gestión de rutinas
|
├── horarioPersonal_app/ # Aplicación de gestión de horarios
|
├── notificacion_app/    # Aplicación de notificaciones
|
└── media/              # Almacenamiento de archivos
```

Figura 7: Estructura del proyecto Django de EverPlanAPP y organización interna de una aplicación.

Fuente: Elaboración propia.

3.5 Diseño del modelo de datos (MER)

En el desarrollo de *EverPlanAPP*, el diseño del modelo de datos se inició con la elaboración de un Modelo Entidad–Relación conceptual preliminar, el cual tuvo como objetivo identificar las principales entidades del sistema y servir como guía para la definición de los módulos del *backend*. Este modelo inicial permitió establecer una base estructural para el desarrollo, orientando la organización de las funcionalidades y la posterior implementación en *Django*.

A partir de esta base conceptual, el modelo de datos fue implementado utilizando el *ORM* de *Django*, definiendo las entidades y relaciones directamente en el código. Durante el proceso de desarrollo, el modelo fue ajustándose de manera iterativa conforme se incorporaban nuevas funcionalidades y se refinaban los requerimientos del sistema, lo que se reflejó en modificaciones sobre los modelos implementados en el *backend*.

El Modelo Entidad–Relación (*MER*) presentado en este capítulo corresponde al modelo final del sistema, obtenido directamente a partir de la estructura real de la base de datos

la *API REST*. Asimismo, se analizan los módulos más relevantes del *backend*, detallando su responsabilidad dentro del sistema, los mecanismos de seguridad aplicados y los *endpoints* que permiten la interacción con el *frontend*.

3.6.1 Módulo de autenticación y gestión de usuarios (*auth_app*)

Rol del módulo

El módulo de autenticación y gestión de usuarios, implementado en *auth_app*, es el encargado de gestionar la identidad de los usuarios dentro del sistema y controlar el acceso a las distintas funcionalidades del *backend*. Este módulo cumple un rol fundamental, ya que permite identificar a cada usuario de forma única y asegurar que las operaciones realizadas en la aplicación se encuentren asociadas correctamente a su propietario.

A través de este módulo se centraliza la lógica relacionada con el registro, la autenticación y la administración de credenciales, estableciendo la base sobre la cual se apoyan el resto de los módulos del sistema

Modelo de datos del usuario

Para la gestión de usuarios se utilizó como base el modelo de usuario provisto por *Django*, extendiéndolo mediante la herencia de *AbstractUser*. Esta decisión permitió reutilizar las funcionalidades de autenticación ya implementadas por el *framework*, como el manejo seguro de contraseñas y permisos, evitando desarrollar estas características desde cero.

Sobre este modelo base se incorporaron campos adicionales necesarios para la aplicación, como el nombre del usuario y un avatar, los cuales permiten una experiencia más personalizada desde el *frontend*. Asimismo, se definieron restricciones como la unicidad del correo electrónico y configuraciones adicionales, tales como índices, que contribuyen a optimizar las consultas relacionadas con los usuarios.

Serialización y validación de datos

La comunicación entre el *backend* y el *frontend* se gestiona mediante un *serializer* específico para el modelo de usuario. Este componente cumple un rol clave, ya que se encarga de validar los datos recibidos desde el cliente, transformar las entidades del sistema en formatos adecuados para su transmisión y aplicar reglas de seguridad sobre la información sensible.

Durante los procesos de registro y actualización de datos del usuario, el *serializer* valida que las contraseñas cumplan con los criterios de seguridad definidos por *Django* y garantiza que estas sean almacenadas de forma cifrada, evitando en todo momento el manejo de contraseñas en texto plano.

Autenticación basada en JWT

Para el control de acceso al *backend* se implementó un mecanismo de autenticación basado en tokens *JWT* (*JSON Web Tokens*), utilizando la librería *SimpleJWT*. Este enfoque permite validar cada solicitud de forma segura y desacoplada, sin necesidad de mantener sesiones activas en el servidor.

Una vez que el usuario se autentica correctamente, el sistema genera un token de acceso y un *token* de refresco. Estos *tokens* son utilizados por el *frontend* para acceder a los *endpoints* protegidos del *backend*, asegurando que solo los usuarios autenticados puedan interactuar con las funcionalidades del sistema.

Funcionalidades del módulo de autenticación

- **Registro de usuarios:** El sistema expone un *endpoint* que permite el registro de nuevos usuarios, creando una cuenta a partir de los datos definidos en el modelo de usuario. Al finalizar exitosamente el proceso de registro, el *backend* genera automáticamente los *tokens JWT* necesarios, permitiendo que el usuario comience a utilizar la aplicación sin requerir un inicio de sesión adicional.
- **Inicio de sesión:** El inicio de sesión se realiza mediante un *endpoint* que valida las credenciales ingresadas por el usuario. En caso de ser correctas, el sistema retorna los *tokens JWT* correspondientes, habilitando el acceso a los *endpoints* protegidos del *backend*.
- **Cierre de sesión:** El módulo incorpora un *endpoint* de cierre de sesión que permite invalidar el *token* de refresco del usuario en el servidor. Esta acción refuerza la seguridad del sistema, evitando que un *token* pueda ser reutilizado una vez que el usuario decide cerrar su sesión.

Con el objetivo de representar de manera clara la interacción entre el usuario y los distintos componentes del sistema, a continuación, se presentan los diagramas de secuencia correspondientes a los procesos de registro de usuario, inicio de sesión (*login*) y cierre de sesión (*logout*).

Estos diagramas permiten visualizar el flujo de acciones, mensajes e intercambios de información entre el *frontend*, el *backend* y los servicios involucrados, facilitando la comprensión del funcionamiento interno del sistema y la forma en que se gestionan los procesos de autenticación.

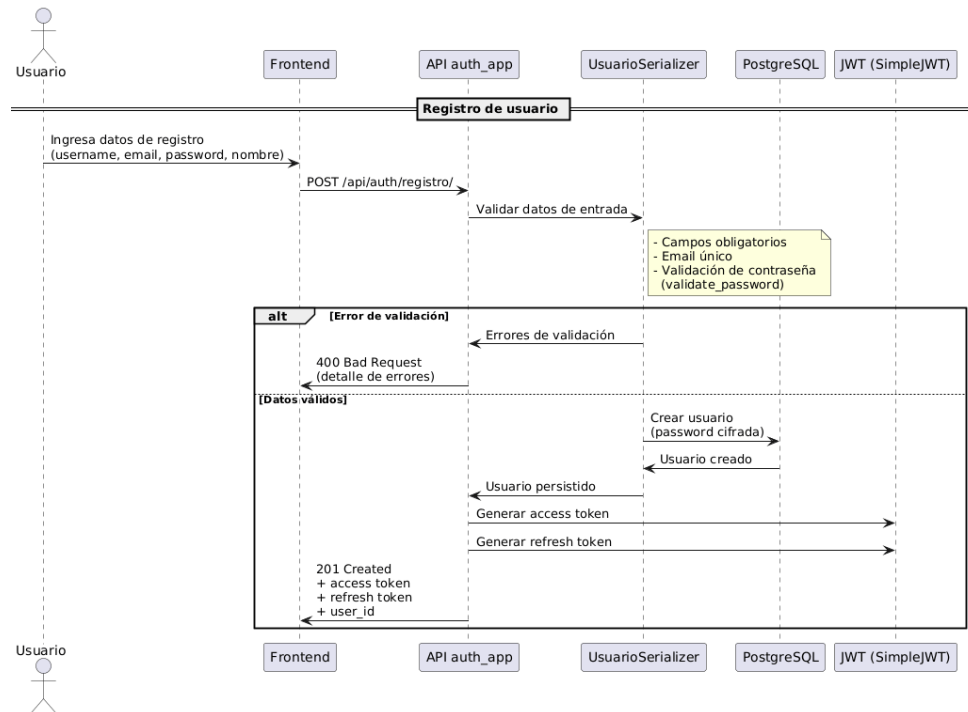


Figura 9: Diagrama de secuencia del proceso de registro del usuario.
Fuente: Elaboración propia.

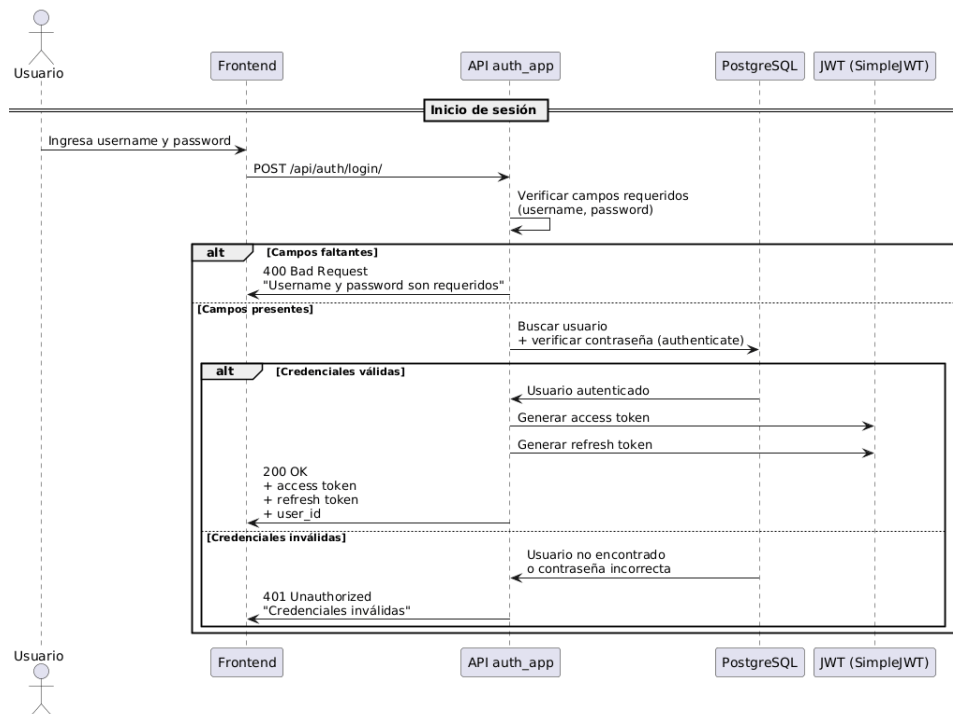


Figura 10: Diagrama de secuencia del proceso de inicio de sesión del usuario.
Fuente: Elaboración propia.

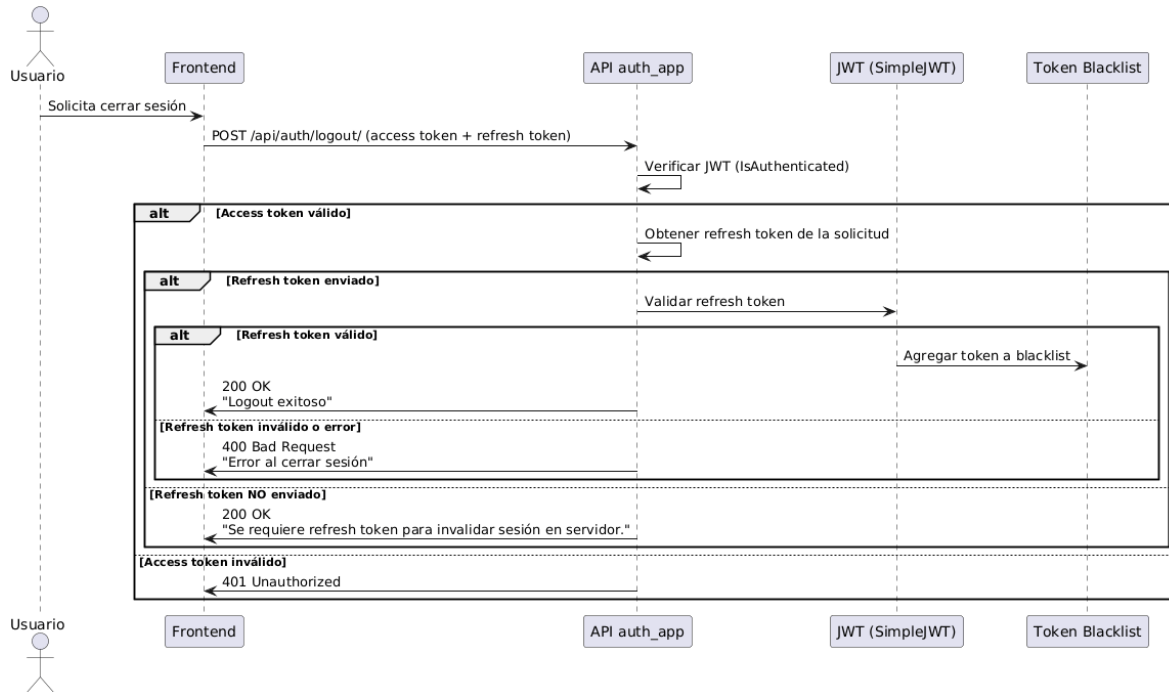


Figura 11: Diagrama de secuencia del proceso de cierre de sesión del usuario. Fuente: Elaboración propia.

Gestión del perfil del usuario

Adicionalmente, el módulo permite que cada usuario autenticado acceda y modifique su propia información personal mediante un *endpoint* de perfil. Esta funcionalidad reutiliza directamente la información del modelo de usuario y permite actualizar campos como nombre, correo electrónico o avatar.

No se definió un módulo independiente para la gestión de perfiles, ya que la visualización y organización del perfil se realiza en el *frontend* utilizando los datos proporcionados por este módulo, evitando duplicar lógica innecesaria en el *backend*.

A continuación, se presentan los diagramas de secuencia correspondientes a la visualización y edición del perfil del usuario, los cuales describen la interacción entre el usuario, el *frontend* y el *backend* durante la consulta y actualización de la información. El sistema permite realizar estas operaciones mediante solicitudes de tipo *GET*, para la obtención de los datos del perfil, y *PUT* o *PATCH*, para su modificación, considerando actualizaciones completas o parciales según corresponda.

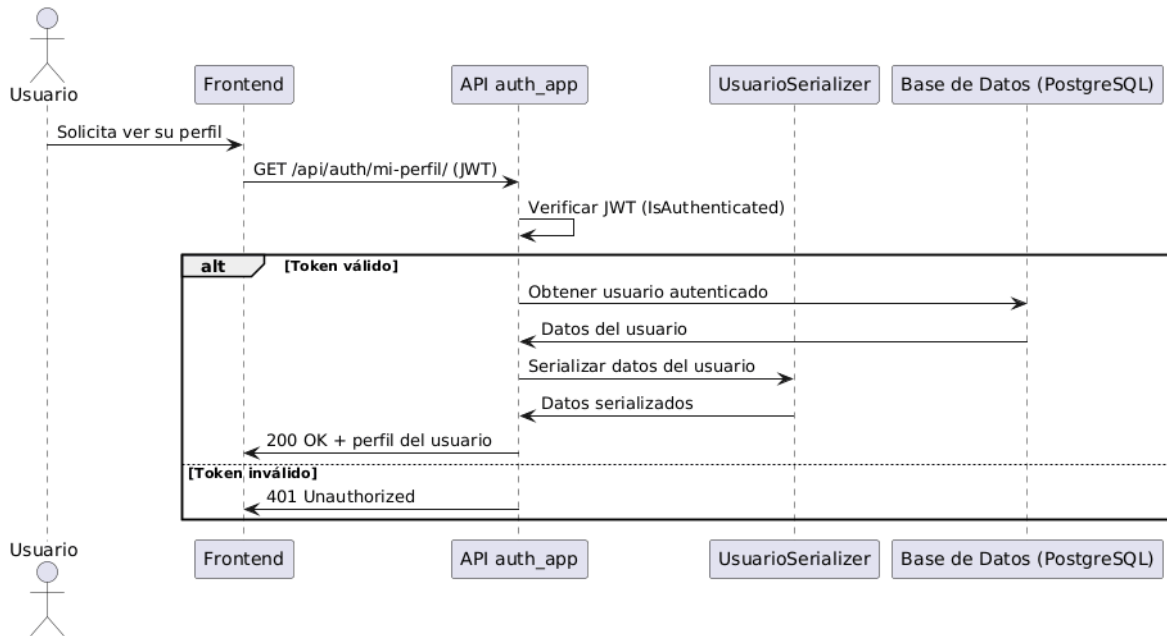


Figura 12: Diagrama de secuencia del proceso de ver perfil del usuario.
Fuente: Elaboración propia.

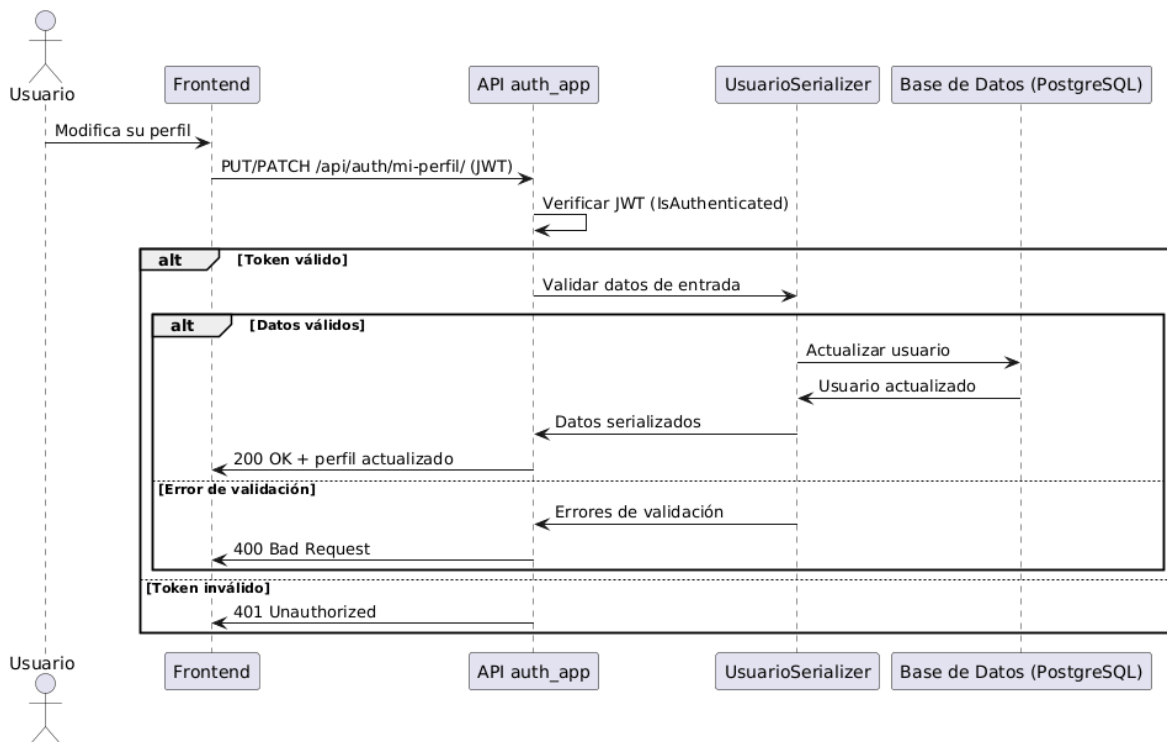


Figura 13: Diagrama de secuencia del proceso de modificar perfil del usuario.
Fuente: Elaboración propia.

Seguridad y gestión de credenciales

El módulo contempla funcionalidades adicionales orientadas a la seguridad del usuario. Entre estas se incluye el cambio de contraseña, el cual valida la contraseña actual, aplica reglas de seguridad y puede invalidar *tokens* activos para prevenir accesos no autorizados.

Asimismo, se implementó la eliminación de cuenta, la cual requiere la confirmación explícita de la contraseña del usuario antes de ejecutar la acción. Este mecanismo busca prevenir eliminaciones accidentales y asegurar que solo el propietario de la cuenta pueda realizar esta operación.

A continuación, se presentan los diagramas de secuencia correspondientes a los procesos de cambio de contraseña y eliminación de cuenta.

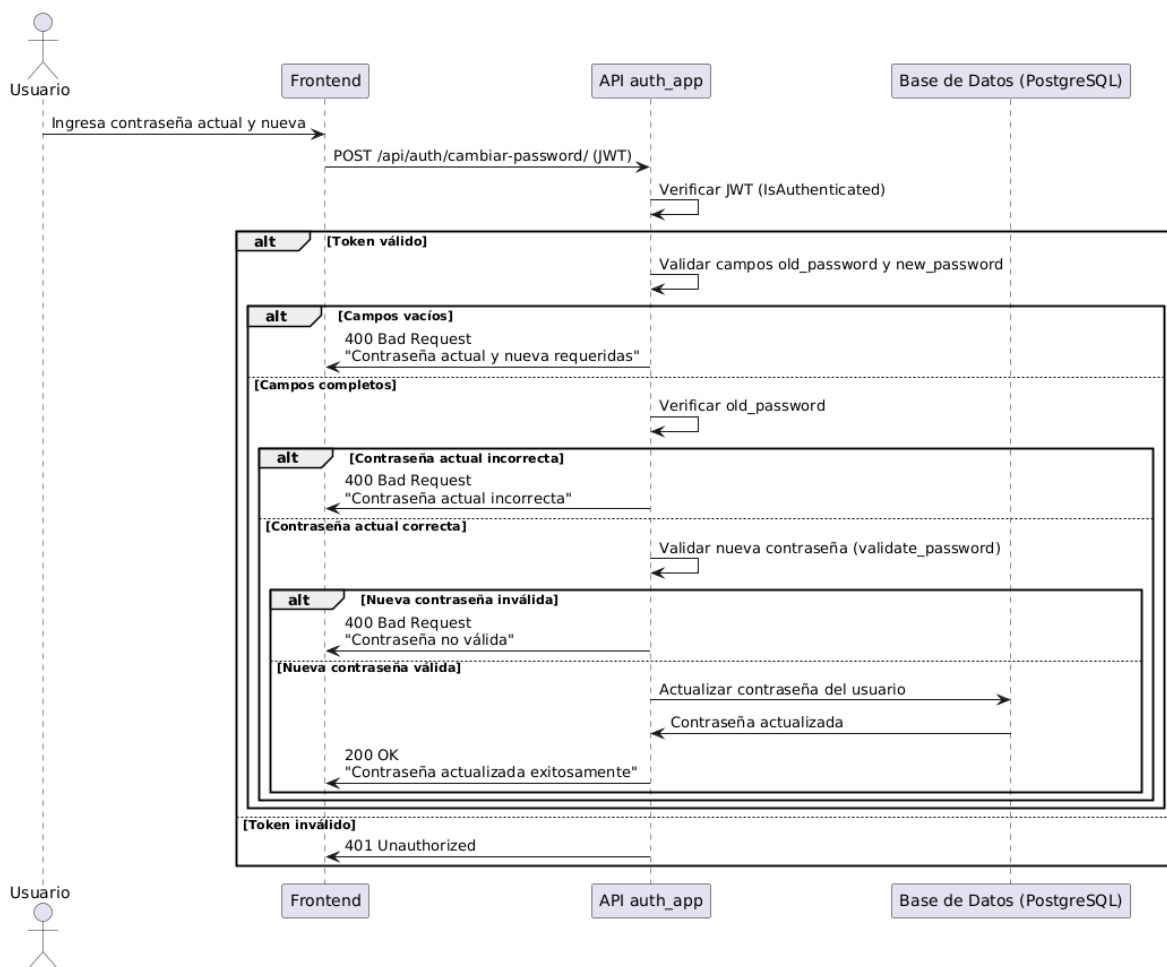


Figura 14: Diagrama de secuencia del proceso de cambio de contraseña del usuario.

Fuente: Elaboración propia.

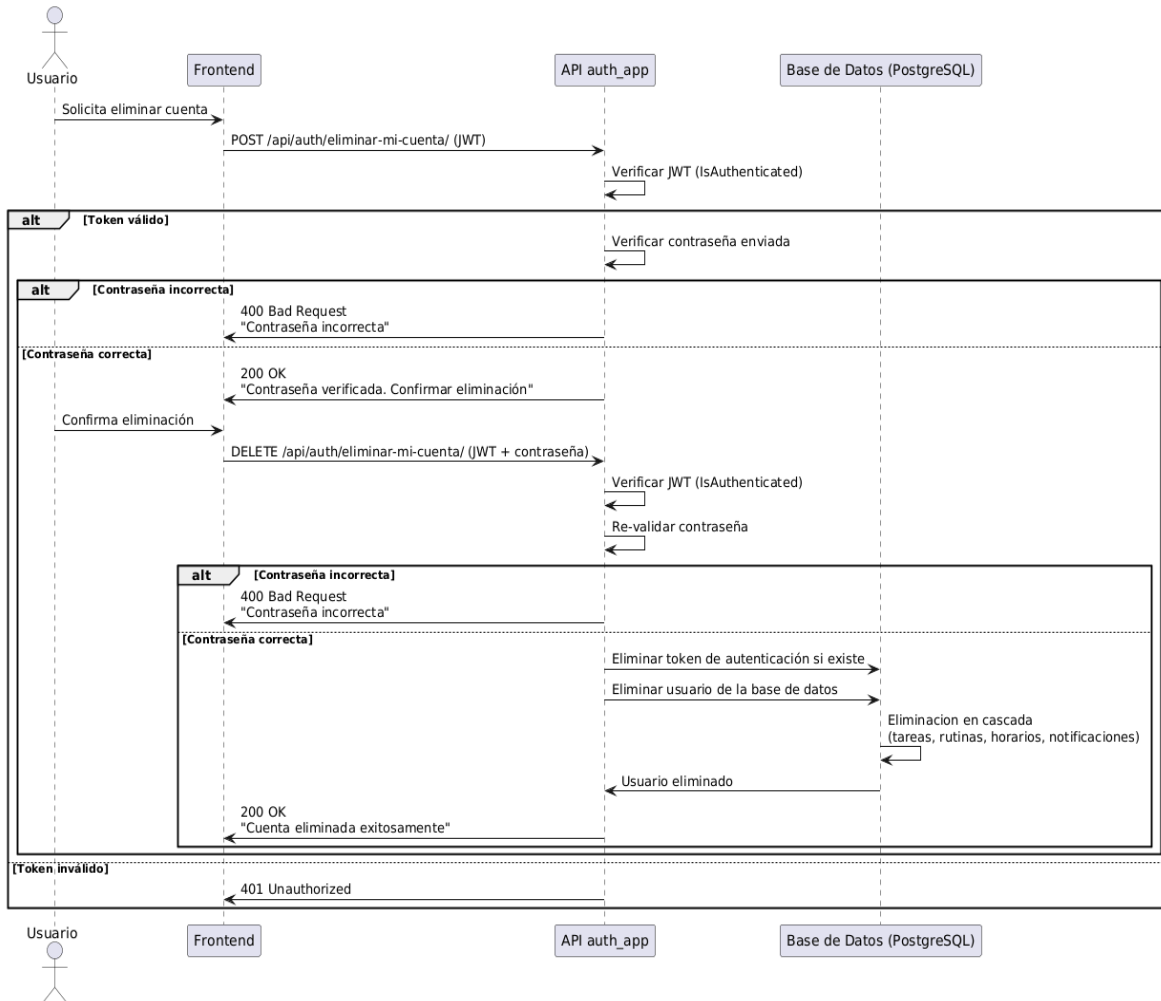


Figura 15: Diagrama de secuencia del proceso de eliminación de cuenta de usuario.

Fuente: Elaboración propia.

Al eliminar la cuenta, el sistema realiza una eliminación en cascada, es decir, todos los datos asociados al usuario en la base de datos se eliminan automáticamente. Esto incluye cualquier registro relacionado con la cuenta, como tareas, rutinas u otros módulos que dependan del usuario, garantizando que no queden datos huérfanos.

Además, si la solicitud de eliminación incluye un *refresh token*, este se invalida en el servidor mediante la lista negra (*blacklist*), evitando que pueda ser utilizado posteriormente para generar nuevos tokens de acceso. En caso de que no se proporcione un *refresh token*, la invalidación no ocurre, por lo que el *frontend* debe eliminar localmente cualquier token almacenado para completar el cierre seguro de sesión.

Endpoints del módulo de autenticación

Tabla 3: Endpoints del módulo auth_app.
Fuente: Elaboración Propia.

Endpoint	Método	Descripción	Seguridad
/api/auth/registro/	POST	Registro de nuevos usuarios y generación automática de tokens JWT	Acceso público
/api/auth/login/	POST	Autenticación de usuarios y obtención de tokens JWT	Acceso público
/api/auth/logout/	POST	Cierre de sesión e invalidación del token de refresco	Usuario autenticado
/api/auth/mi-perfil/	GET / PUT / PATCH	Consulta y actualización de información personal del usuario	Usuario autenticado
/api/auth/cambiar-password/	POST	Cambio de contraseña con validaciones de seguridad	Usuario autenticado
/api/auth/eliminar-mi-cuenta/	POST / DELETE	Eliminación de la cuenta con confirmación de contraseña	Usuario autenticado
/api/auth/token/refresh/	POST	Renovación del token de acceso mediante token de refresco	Token de refresco válido

3.6.2 Módulo de gestión de tareas (tarea_app)

Rol del módulo

El módulo de gestión de tareas, implementado en *tarea_app*, permite a los usuarios organizar y administrar sus actividades puntuales dentro de la aplicación. Una tarea representa una actividad que ocurre una sola vez y que se encuentra asociada a un momento específico, lo que la diferencia de otros elementos del sistema como las rutinas.

Este módulo constituye una parte central del *backend*, ya que soporta gran parte de la interacción diaria del usuario con la aplicación y sirve como base para funcionalidades complementarias como las notificaciones y las estadísticas motivacionales.

Modelo de datos de tareas

El modelo de datos del módulo se representa mediante la entidad Tarea, la cual se encuentra asociada directamente al usuario autenticado a través de una relación de tipo uno a muchos. Cada tarea pertenece a un único usuario, y esta relación fue definida de modo que, si un usuario es eliminado del sistema, todas sus tareas asociadas se eliminan automáticamente. Esta decisión permite mantener la consistencia de los datos y evita la existencia de registros huérfanos en la base de datos.

La entidad Tarea incluye los siguientes campos principales:

- **titulo:** identifica la tarea de forma breve.
- **descripcion:** permite detallar información adicional sobre la tarea.
- **fecha_hora:** define la fecha y hora en que la tarea debe realizarse.
- **todo_el_dia:** indica si la tarea puede ejecutarse en cualquier momento del día.
- **completado:** señala si la tarea ha sido finalizada.
- **color:** permite asociar un color a la tarea para facilitar su identificación visual.
- **notificacion_activa:** determina si la tarea debe generar una notificación.
- **creado_en:** registra el momento en que la tarea fue creada.

Estos campos permiten modelar de forma clara el comportamiento de una tarea y responder a los requerimientos funcionales definidos para la aplicación.

Reglas de negocio y validaciones

Para mantener la coherencia de los datos, el módulo implementa reglas de negocio que impiden crear o modificar tareas con fechas u horas pasadas. Esta validación se aplica a nivel de modelo y se refuerza en el proceso de serialización, evitando que se registren tareas con información temporal inválida.

Este enfoque asegura que la lógica de negocio se mantenga protegida desde el *backend*, independientemente de las validaciones que puedan existir en el *frontend*.

Serialización y control de datos

La comunicación entre el *backend* y el *frontend* se gestiona mediante un *serializer* específico para la entidad Tarea. Además de transformar los datos del modelo en un formato adecuado para su transmisión, este *serializer* define configuraciones que regulan el acceso y la modificación de la información.

En particular, el *serializer*:

- Restringe la modificación de campos críticos como el usuario asociado y la fecha de creación.
- Valida que el campo *fecha_hora* no corresponda a una fecha u hora pasada.
- Permite controlar de forma explícita qué campos pueden ser enviados y actualizados desde el *frontend*.

Gracias a estas configuraciones, el *serializer* no solo cumple una función de transmisión de datos, sino que también actúa como una capa adicional de validación y seguridad.

Funcionalidades principales del módulo

El módulo de tareas implementa las operaciones básicas de creación, listado, consulta, actualización y eliminación de tareas. Todas estas operaciones se encuentran protegidas mediante autenticación, garantizando que cada usuario solo pueda acceder y modificar sus propias tareas.

Además, se incluye una funcionalidad que permite eliminar todas las tareas asociadas al usuario autenticado, facilitando la gestión completa de la información en casos específicos.

A continuación, se presentan los diagramas de secuencia correspondientes a los procesos de las operaciones básicas.

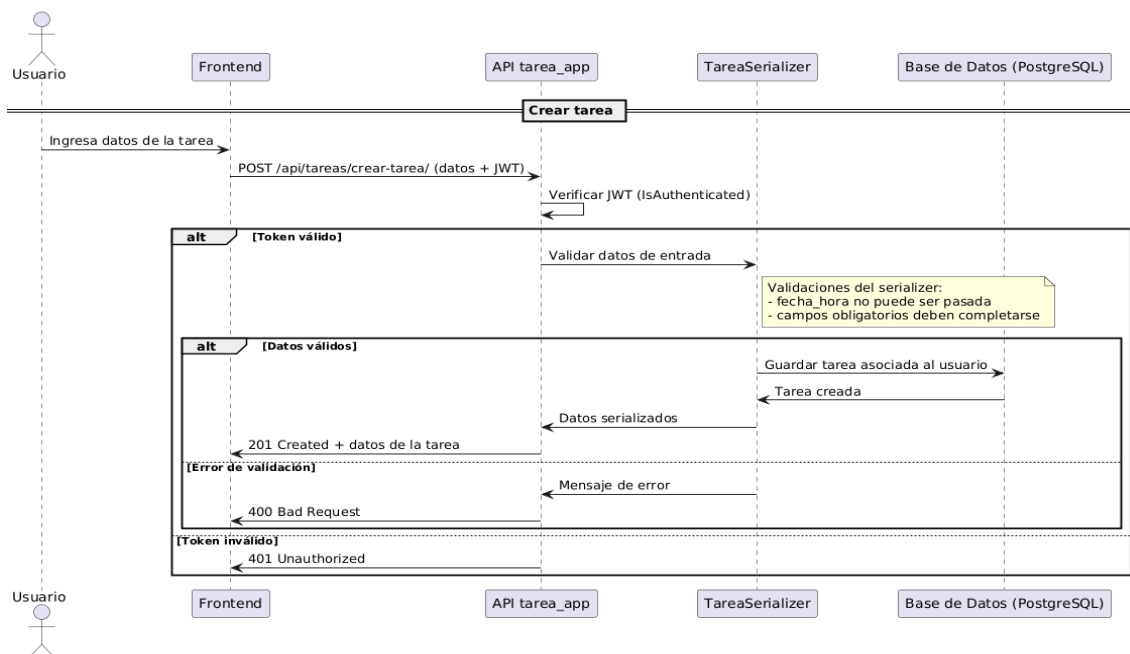


Figura 16: Diagrama de secuencia del proceso de creación de una tarea. Fuente: Elaboración propia.

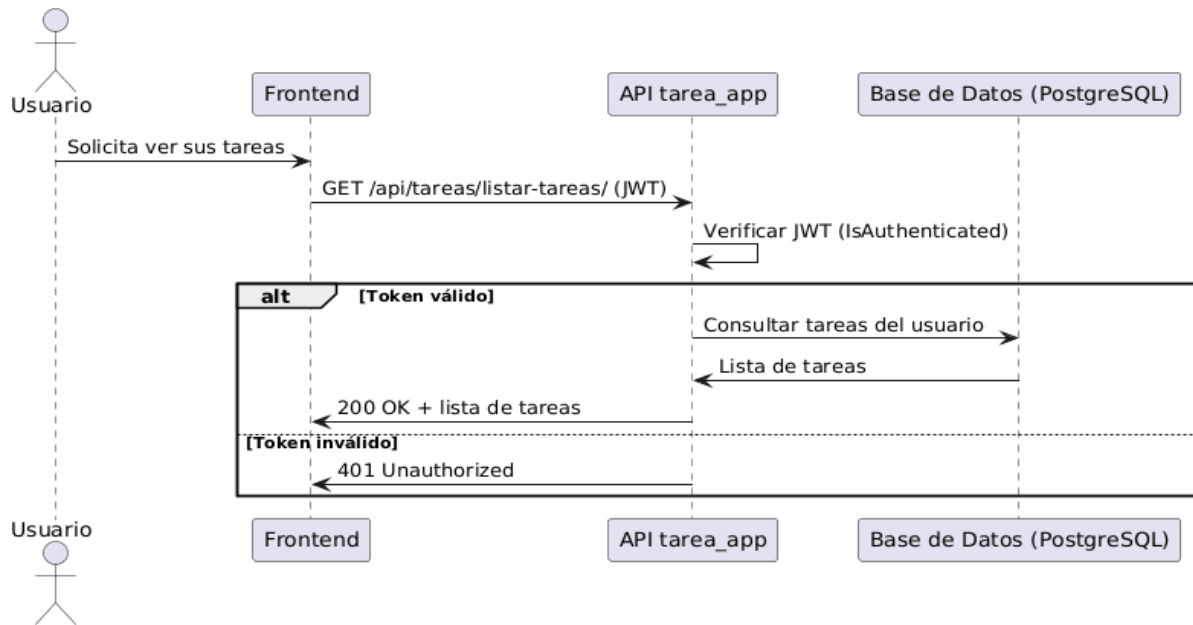


Figura 17: Diagrama de secuencia del proceso de listar tareas. Fuente: Elaboración propia.

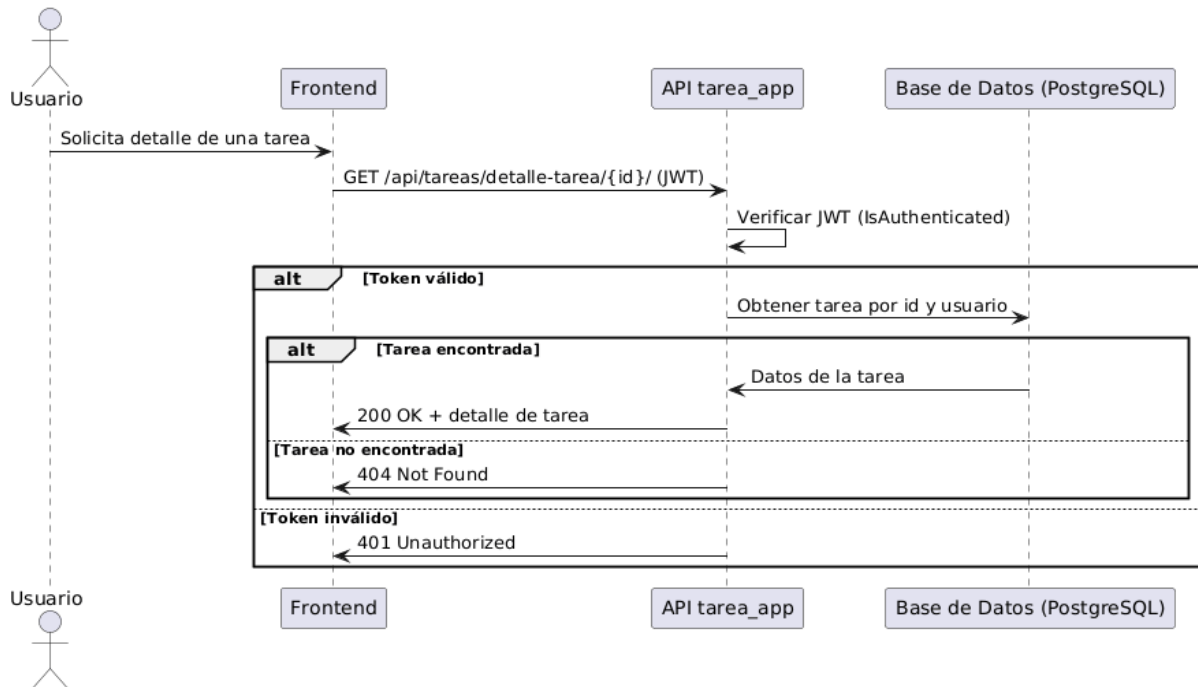


Figura 18: Diagrama de secuencia del proceso de ver detalle de una tarea. Fuente: Elaboración propia.

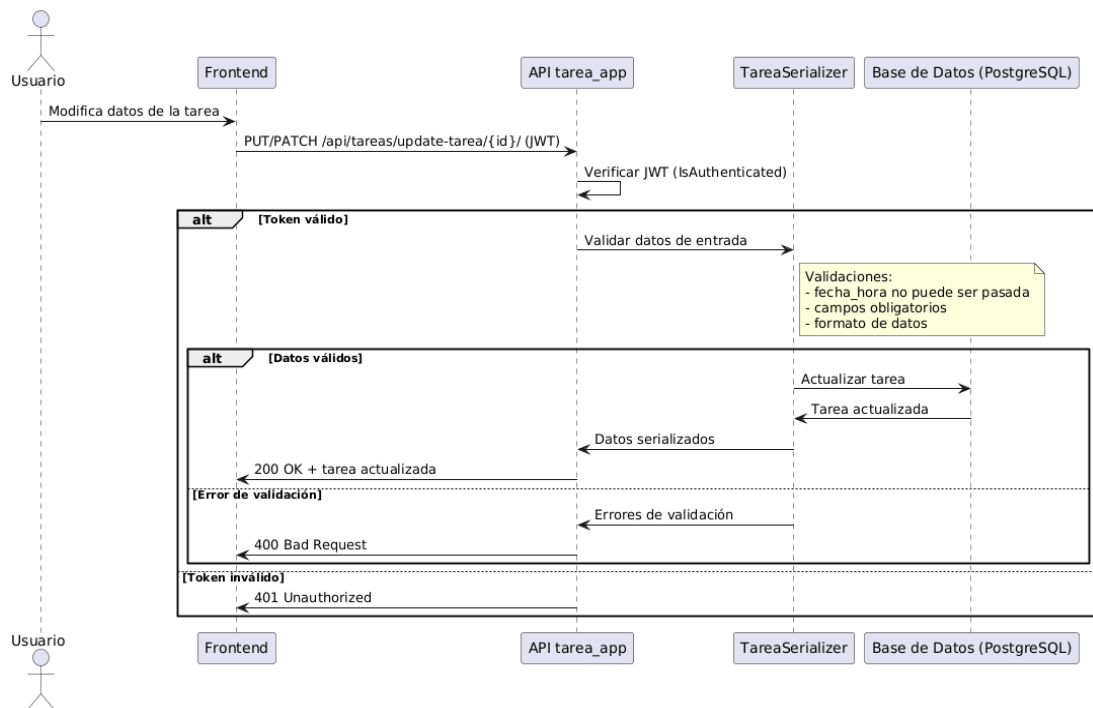


Figura 19: Diagrama de secuencia del proceso de editar una tarea.

Fuente: Elaboración propia.

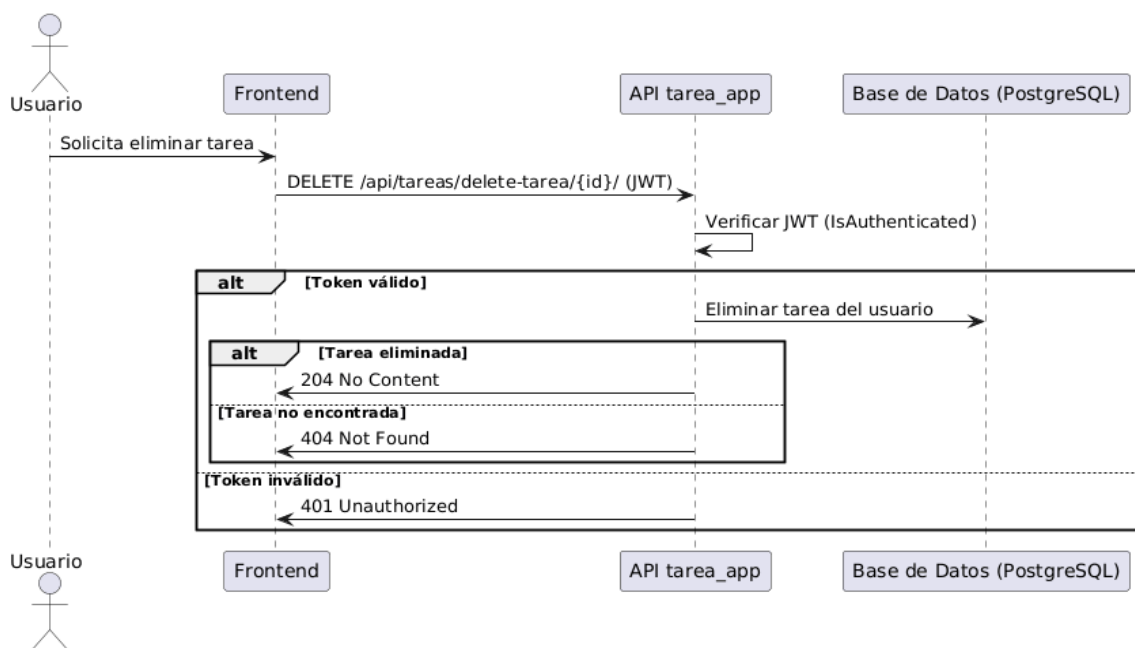


Figura 20: Diagrama de secuencia del proceso de eliminar una tarea.

Fuente: Elaboración propia.

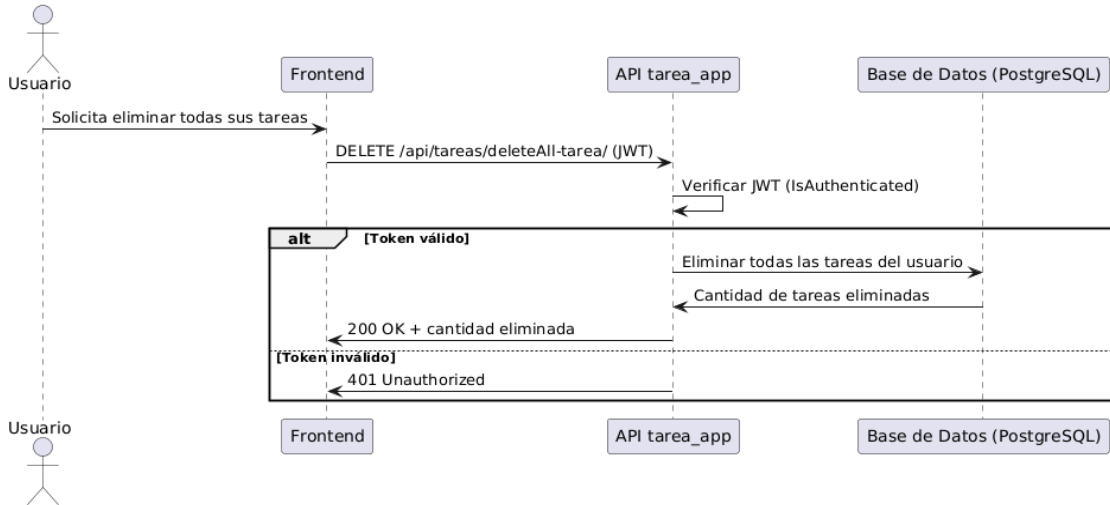


Figura 21: Diagrama de secuencia del proceso de eliminar todas las tareas.

Fuente: Elaboración propia.

Estadísticas motivacionales basadas en tareas

Como parte de una primera aproximación a las estadísticas del sistema, el módulo de tareas incorpora una funcionalidad que permite visualizar información básica sobre el desempeño del usuario. Estas estadísticas se calculan a partir de las tareas marcadas como completadas y entregan indicadores como tareas realizadas en la última semana, promedio diario y rachas de cumplimiento.

El objetivo de esta funcionalidad es entregar retroalimentación motivacional de manera simple, reutilizando la información existente y evitando aumentar la complejidad del *backend* en esta etapa inicial del proyecto.

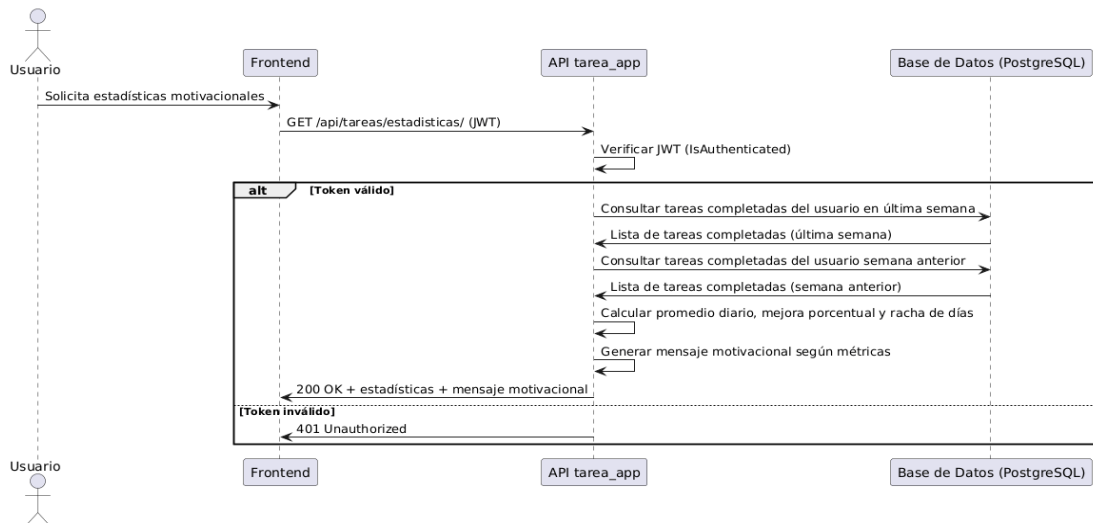


Figura 22: Diagrama de secuencia del proceso de estadísticas motivacionales.

Fuente: Elaboración propia.

Endpoints del módulo de tareas

Tabla 4: Endpoints del módulo tarea_app.
Fuente: Elaboración Propia.

Endpoint	Método	Descripción	Seguridad
/api/tareas/crear-tarea/	POST	Permite crear una nueva tarea asociada al usuario autenticado.	Usuario autenticado
/api/tareas/listar-tareas/	GET	Retorna el listado de tareas del usuario autenticado.	Usuario autenticado
/api/tareas/detalle-tarea/{id}/	GET	Obtiene el detalle de una tarea específica del usuario autenticado.	Usuario autenticado
/api/tareas/update-tarea/{id}/	PUT / PATCH	Permite actualizar una tarea existente del usuario autenticado.	Usuario autenticado
/api/tareas/delete-tarea/{id}/	DELETE	Permite eliminar una tarea específica del usuario autenticado.	Usuario autenticado
/api/tareas/deleteAll-tarea/	DELETE	Permite eliminar todas las tareas del usuario autenticado.	Usuario autenticado
/api/tareas/estadisticas/	GET	Retorna estadísticas motivacionales basadas en tareas completadas.	Usuario autenticado

3.6.3 Módulo de gestión de rutinas (rutinas_app)

Rol del módulo

El módulo de gestión de rutinas, implementado en *rutinas_app*, tiene como objetivo permitir a los usuarios definir actividades recurrentes que se repiten en el tiempo, como hábitos diarios, rutinas semanales o actividades periódicas. Este módulo complementa al sistema de tareas, ya que mientras las tareas representan acciones puntuales, las rutinas permiten modelar comportamientos repetitivos dentro de la aplicación.

Su principal función es almacenar la definición base de una rutina y generar, a partir de esta información, las distintas ocurrencias temporales que luego pueden ser consumidas por el *frontend* para su visualización.

Modelo de datos de rutinas

El modelo principal del módulo es la entidad Rutina, la cual se encuentra asociada directamente a un usuario mediante una relación de uno a muchos. Cada rutina pertenece a un único usuario y, de la misma forma que en otros módulos del sistema, si un usuario es eliminado, todas sus rutinas asociadas se eliminan automáticamente, asegurando la integridad de los datos.

La entidad Rutina está compuesta por los siguientes campos principales:

- **titulo:** identifica la rutina de forma clara.
- **descripcion:** permite agregar información adicional de la rutina.
- **fecha_hora_inicio:** indica el momento en que comienza la rutina.
- **fecha_hora_fin:** define el término de la rutina.
- **recurrencia:** establece la periodicidad de la rutina, permitiendo valores como diaria, semanal o mensual.
- **creado_en:** registra la fecha de creación de la rutina.

Adicionalmente, el modelo incorpora propiedades calculadas como la duración de la rutina y la regla de recurrencia (RRULE), las cuales no se almacenan directamente en la base de datos, pero permiten representar de forma estructurada el comportamiento repetitivo de la rutina.

Como medida de validación, el modelo asegura que la fecha y hora de finalización siempre sea posterior a la de inicio, evitando configuraciones inválidas desde el punto de vista lógico.

Serialización y validación de datos

La comunicación entre el *backend* y el *frontend* se gestiona mediante un *serializer* específico para rutinas, el cual se encarga de transformar los datos del modelo en un formato adecuado para su consumo a través de la *API REST*.

Además de exponer los campos principales de la rutina, el *serializer* incorpora configuraciones adicionales, como:

- Validación de coherencia temporal, verificando que la fecha de término sea posterior a la fecha de inicio.
- Exposición del campo *RRULE* como un atributo de solo lectura, permitiendo que el *frontend* conozca la regla de recurrencia sin necesidad de recalcularla.
- Restricción de campos de solo lectura, como el usuario asociado y la fecha de creación, evitando modificaciones no autorizadas.

Funcionalidades principales del módulo

El módulo de rutinas implementa las operaciones básicas de creación, listado, consulta, actualización, eliminación de rutinas y la generación de ocurrencias. Todas estas operaciones se encuentran protegidas mediante autenticación, garantizando que cada usuario solo pueda acceder y modificar sus propias rutinas.

A continuación, se presentan los diagramas de secuencia correspondientes a los procesos de las operaciones básicas.

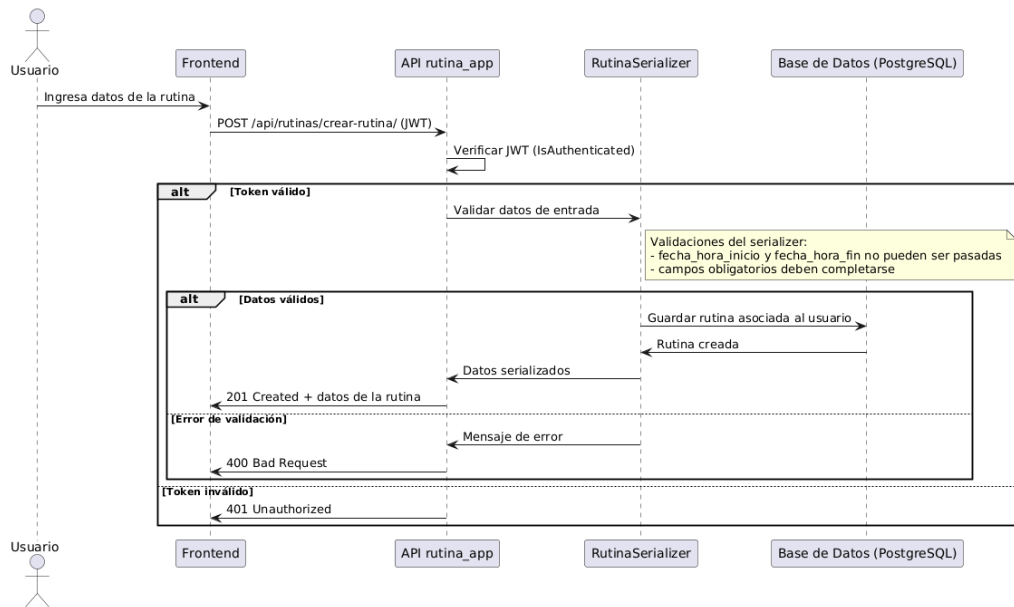


Figura 23: Diagrama de secuencia del proceso de creación de una rutina.

Fuente: Elaboración propia.

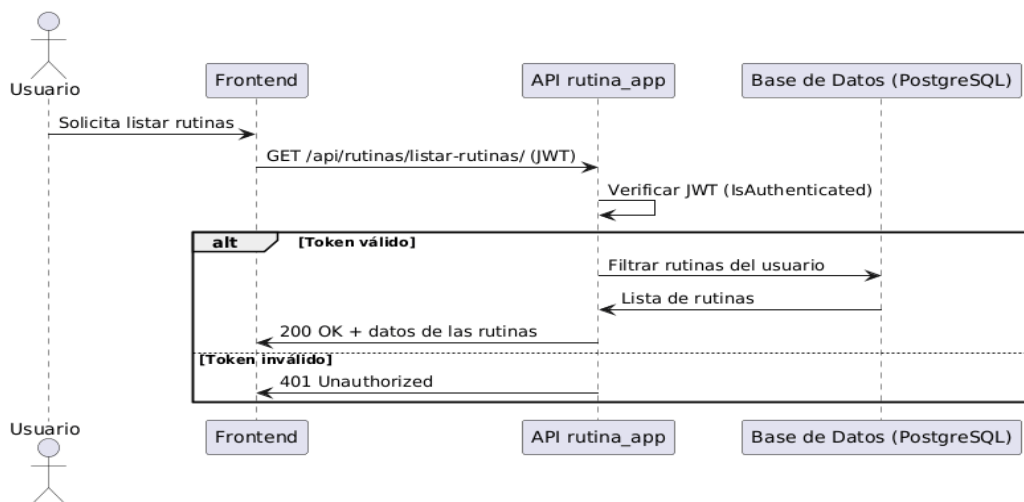


Figura 24: Diagrama de secuencia del proceso de listar rutinas.

Fuente: Elaboración propia.

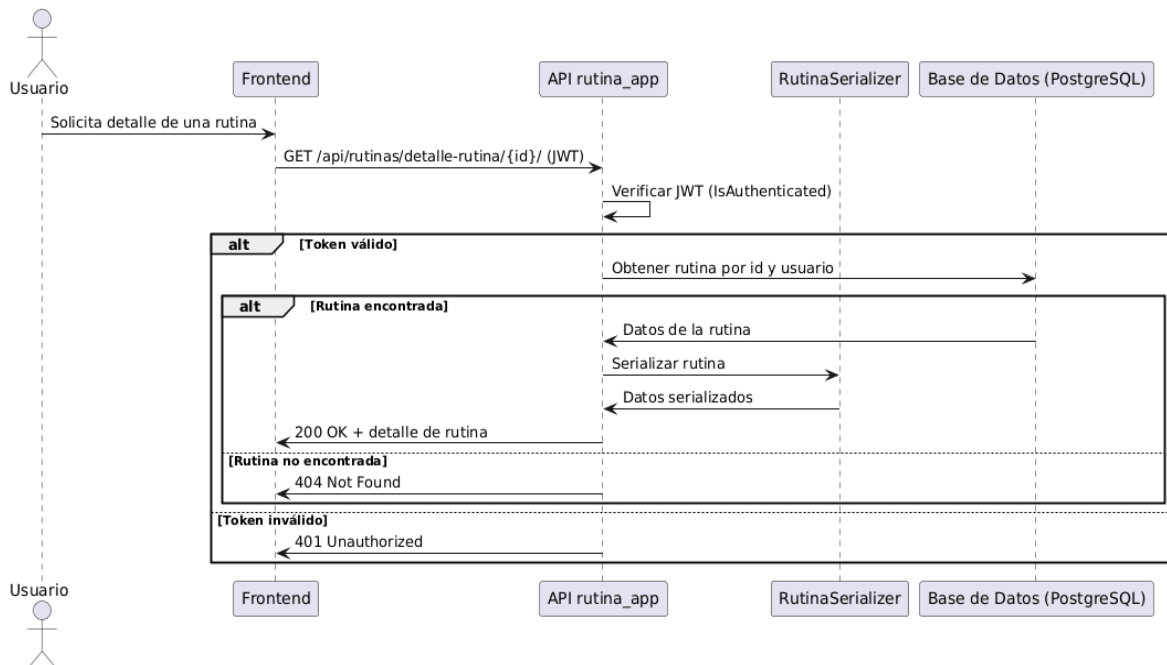


Figura 25: Diagrama de secuencia del proceso de ver detalle de una rutina.
Fuente: Elaboración propia.

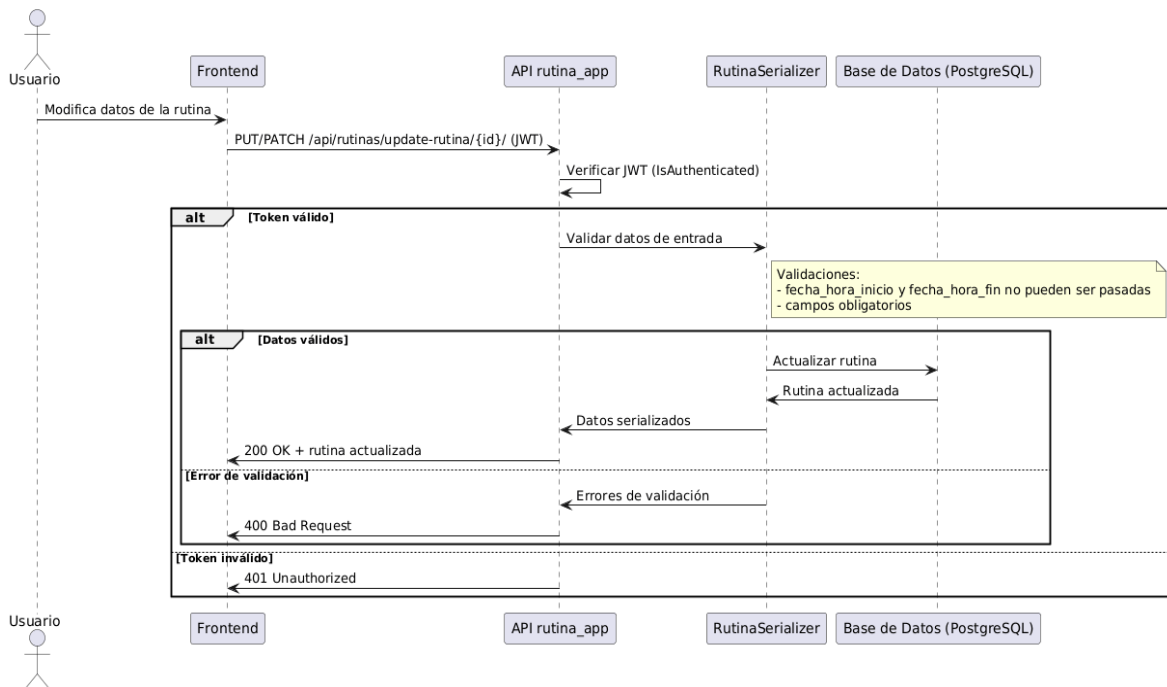


Figura 26: Diagrama de secuencia del proceso editar una rutina.
Fuente: Elaboración propia.

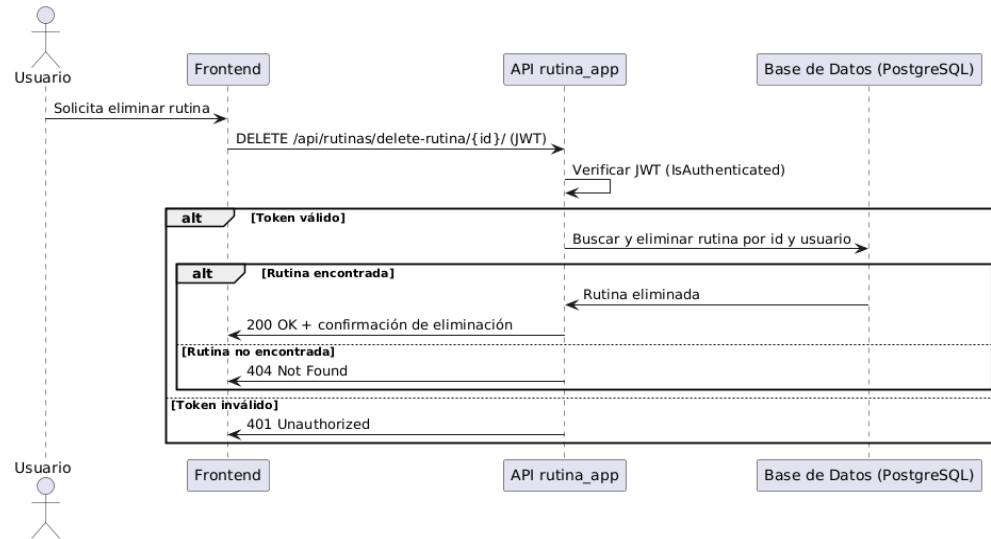


Figura 27: Diagrama de secuencia del proceso de eliminar una rutina.
Fuente: Elaboración propia.

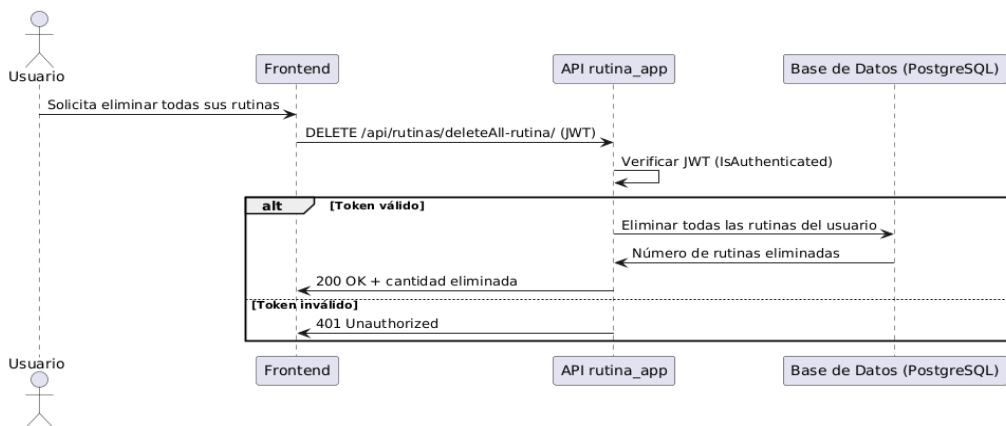


Figura 28: Diagrama de secuencia del proceso de eliminar todas las rutinas.
Fuente: Elaboración propia.

Generación de ocurrencias de rutinas

Una de las funcionalidades más relevantes de este módulo es la generación de ocurrencias de las rutinas. A partir de la definición base de cada rutina, el *backend* calcula las distintas instancias temporales según su tipo de recurrencia.

Para ello, el sistema procesa las rutinas del usuario autenticado y genera una lista de ocurrencias con sus respectivas fechas de inicio y término. Esta información es entregada al *frontend* para su visualización en calendarios o vistas temporales, sin necesidad de almacenar cada ocurrencia de forma individual en la base de datos.

Este enfoque permite un uso más eficiente de los datos y facilita la modificación de rutinas sin duplicar información.

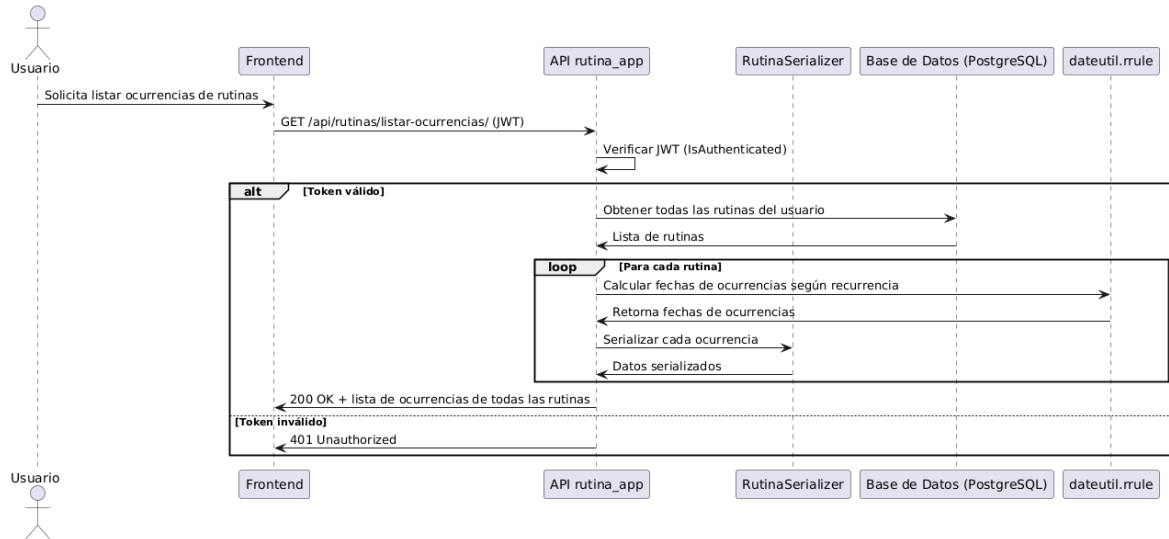


Figura 29: Diagrama de secuencia del proceso de listar ocurrencias de rutinas.

Fuente: Elaboración propia.

Endpoints del módulo de rutinas

Tabla 5: Endpoints del módulo rutinas_app.

Fuente: Elaboración Propia.

Endpoint	Método	Descripción	Seguridad
/api/rutinas/crear-rutina/	POST	Permite crear una nueva rutina asociada al usuario autenticado.	Usuario autenticado
/api/rutinas/listar-rutinas/	GET	Obtiene el listado de rutinas pertenecientes al usuario autenticado.	Usuario autenticado
/api/rutinas/detalle-rutina/{id}/	GET	Retorna el detalle de una rutina específica del usuario.	Usuario autenticado
/api/rutinas/update-rutina/{id}/	PUT / PATCH	Permite actualizar los datos de una rutina existente.	Usuario autenticado
/api/rutinas/delete-rutina/{id}/	DELETE	Elimina una rutina específica del usuario autenticado.	Usuario autenticado
/api/rutinas/deleteAll-rutina/	DELETE	Elimina todas las rutinas asociadas al usuario autenticado.	Usuario autenticado
/api/rutinas/listar-ocurrencias/	GET	Genera y retorna las ocurrencias de las rutinas del usuario según su recurrencia.	Usuario autenticado

3.6.4 Módulo de gestión de horarios personales (*horarioPersonal_app*)

Rol del módulo

El módulo de gestión de horarios personales, implementado en *horarioPersonal_app*, permite a los usuarios organizar y visualizar sus actividades fijas dentro de la semana, como clases, horarios académicos u otras actividades recurrentes con un día y horario definido. A diferencia del módulo de rutinas, este módulo se enfoca en eventos semanales estructurados por día de la semana y rangos horarios específicos.

Este módulo aporta una visión más estructurada del tiempo del usuario, complementando la planificación diaria realizada mediante tareas y rutinas.

Modelo de datos de horarios personales

El modelo principal del módulo es la entidad *EventoHorario*, la cual representa un evento fijo dentro del horario semanal del usuario. Cada evento está asociado a un único usuario mediante una relación de uno a muchos. De esta forma, si un usuario es eliminado del sistema, todos sus eventos de horario se eliminan automáticamente, manteniendo la coherencia de los datos.

La entidad *EventoHorario* incluye los siguientes campos relevantes:

- **titulo:** nombre del evento dentro del horario.
- **descripcion:** información adicional opcional sobre la actividad.
- **día:** día de la semana en que ocurre el evento, representado mediante valores predefinidos.
- **hora_inicio** y **hora_fin:** delimitan el rango horario del evento.
- **ubicacion:** permite indicar un lugar físico, como una sala o aula.
- **color:** color asociado al evento para facilitar su visualización.
- **imagen:** campo opcional que permite asociar una imagen representativa, por ejemplo, del docente o la asignatura.
- **notificacion_activa:** indica si el evento tiene notificaciones asociadas.
- **creado_en:** registra la fecha de creación del evento.

El modelo incorpora validaciones que aseguran que la hora de término siempre sea posterior a la hora de inicio, evitando inconsistencias en la definición del horario. Además, se definió un ordenamiento por día y hora de inicio, facilitando la presentación de los eventos de forma cronológica.

Funcionalidades principales del módulo

El módulo de horario personal implementa las operaciones básicas de creación, listado, consulta, actualización y eliminación de eventos asociados al horario del usuario. Estas funcionalidades permiten gestionar actividades fijas dentro de la planificación semanal, como clases, reuniones u otros compromisos personales.

Todas las operaciones de este módulo se encuentran protegidas mediante autenticación, asegurando que cada usuario solo pueda acceder y modificar los eventos de su propio horario.

A continuación, se presentan los diagramas de secuencia correspondientes a los procesos de las operaciones básicas del módulo de horario personal, los cuales ilustran la interacción entre el usuario, el frontend y el backend durante la gestión de eventos.

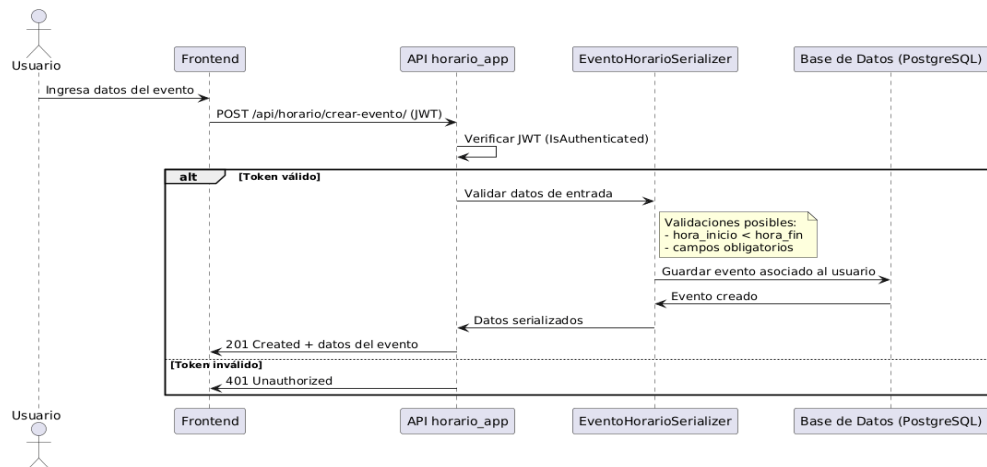


Figura 30: Diagrama de secuencia del proceso de creación de un evento.

Fuente: Elaboración propia.

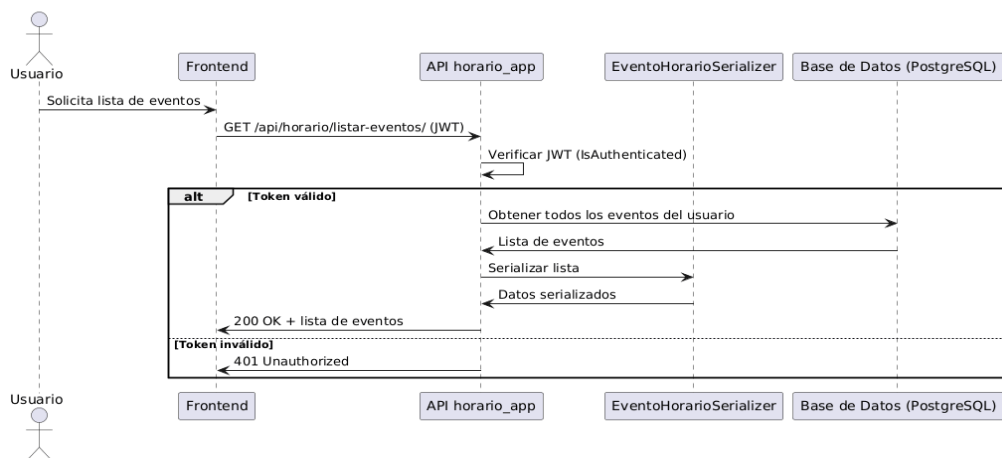


Figura 31: Diagrama de secuencia del proceso de listar eventos.

Fuente: Elaboración propia.

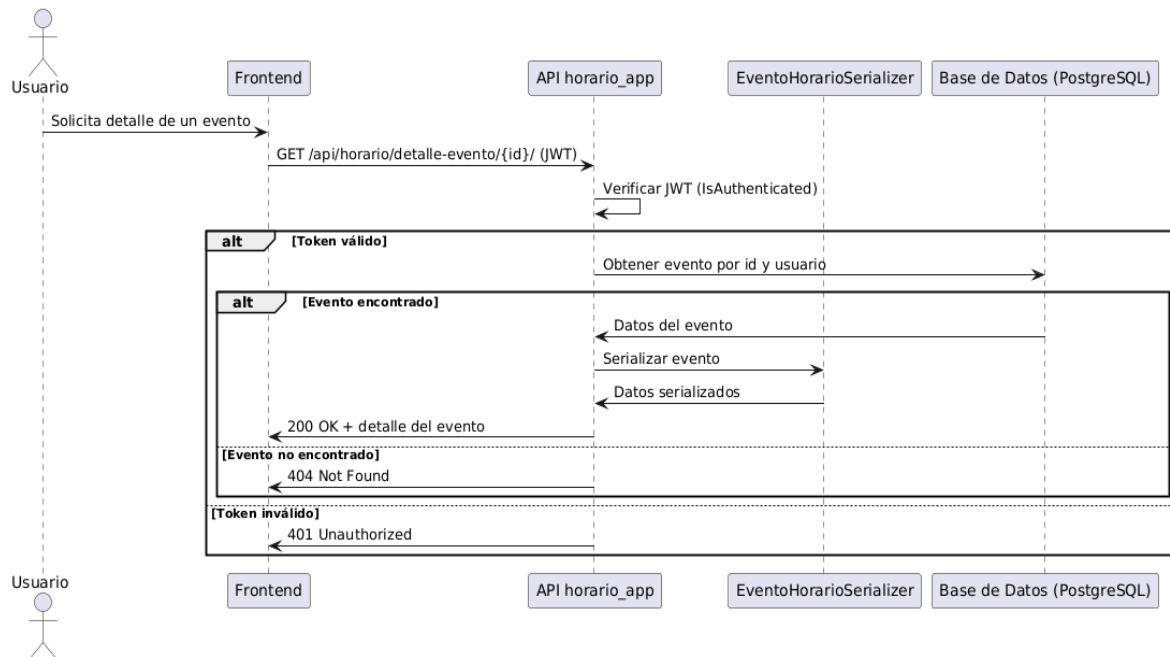


Figura 32: Diagrama de secuencia del proceso de ver detalle de un evento.

Fuente: Elaboración propia.

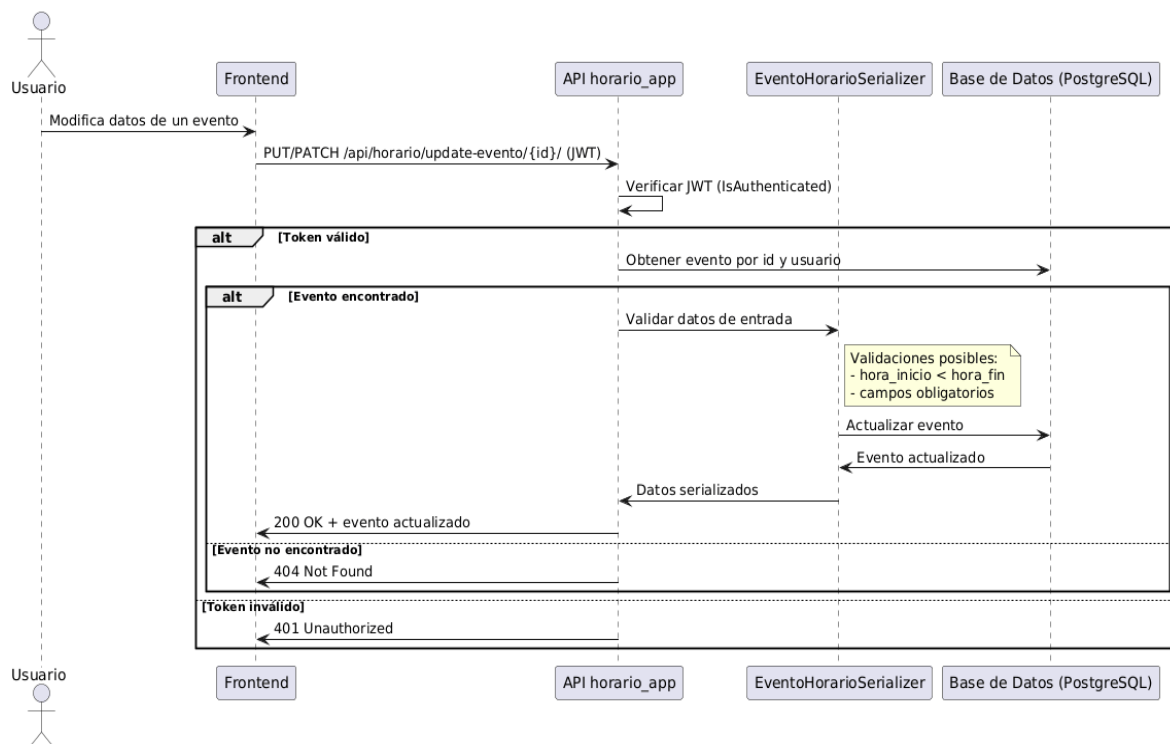


Figura 33: Diagrama de secuencia del proceso de editar un evento.

Fuente: Elaboración propia.

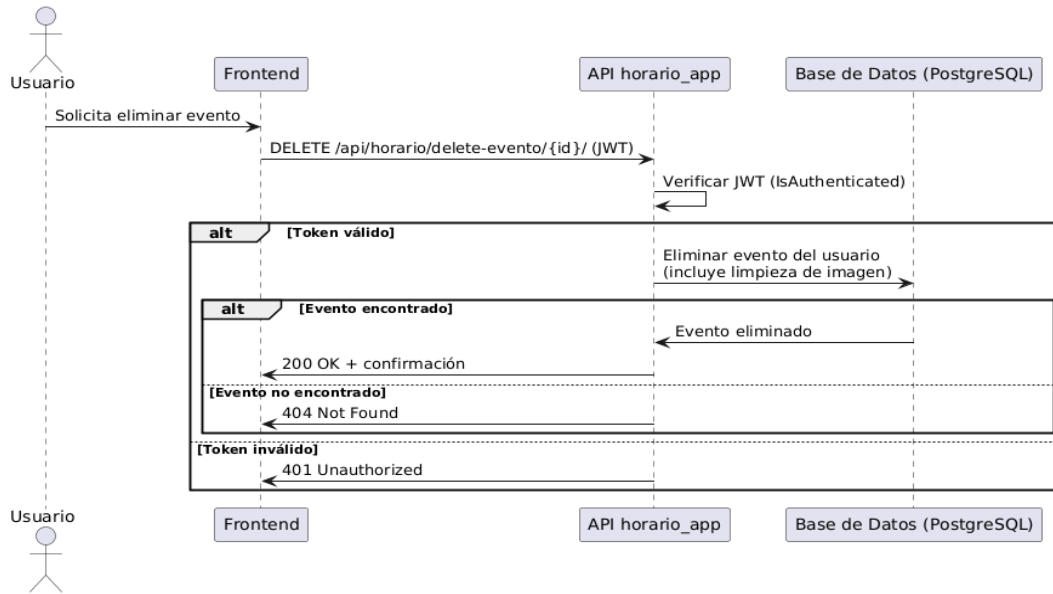


Figura 34: Diagrama de secuencia del proceso de eliminar un evento.

Fuente: Elaboración propia.

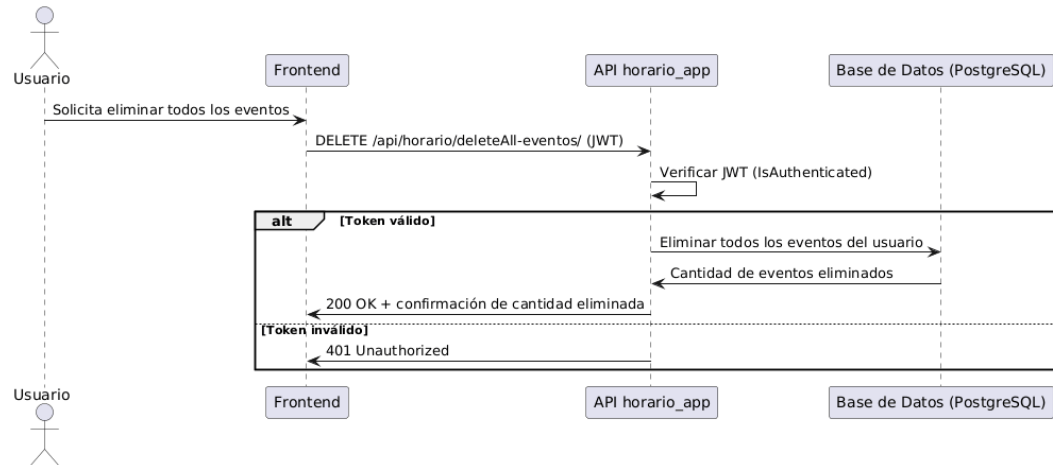


Figura 35: Diagrama de secuencia del proceso de eliminar todos los eventos.

Fuente: Elaboración propia.

Gestión de archivos e integridad de datos

Una característica relevante de este módulo es el manejo de imágenes asociadas a los eventos del horario. El sistema implementa mecanismos automáticos que permiten eliminar correctamente los archivos de imagen cuando un evento es modificado o eliminado.

De esta forma, cuando un evento cambia su imagen o es eliminado, el archivo anterior se borra del almacenamiento, evitando la acumulación de archivos innecesarios y mejorando la gestión de recursos del sistema.

Serialización y validación de datos

La comunicación entre el *backend* y el *frontend* se realiza mediante un *serializer* específico para eventos de horario, encargado de transformar los datos del modelo en un formato adecuado para su consumo por la *API REST*.

Además de exponer los campos principales del evento, el *serializer* incluye validaciones adicionales, entre las que se encuentra:

- Verificación de que la hora de fin sea posterior a la hora de inicio.
- Definición de campos de solo lectura, como el usuario asociado y la fecha de creación, evitando modificaciones no autorizadas.
- Soporte para el envío y recepción de archivos de imagen asociados al evento.

Estas validaciones aseguran que la información del horario personal sea consistente antes de ser almacenada en la base de datos.

Endpoints del módulo de horarios personales

Tabla 6: Endpoints del módulo horarioPersonal_app.
Fuente: Elaboración Propia.

Endpoint	Método	Descripción	Seguridad
/api/horario/crear-evento/	POST	Permite crear un nuevo evento dentro del horario personal del usuario autenticado.	Usuario autenticado
/api/horario/listar-eventos/	GET	Obtiene el listado de eventos del horario personal del usuario, ordenados por día y hora.	Usuario autenticado
/api/horario/detalle-evento/{id}/	GET	Retorna el detalle de un evento específico del horario personal.	Usuario autenticado
/api/horario/update-evento/{id}/	PUT / PATCH	Permite actualizar la información de un evento del horario personal.	Usuario autenticado
/api/horario/delete-evento/{id}/	DELETE	Elimina un evento específico del horario personal del usuario.	Usuario autenticado
/api/horario/deleteAll-evento/	DELETE	Elimina todos los eventos del horario personal asociados al usuario autenticado.	Usuario autenticado

3.6.5 Módulo de notificaciones en tiempo real (*notificacion_app*)

Rol del módulo

El módulo *notificacion_app* es el encargado de gestionar las notificaciones del sistema y entregarlas al *frontend* en el momento adecuado. Su objetivo principal es informar al usuario cuando una tarea o una rutina debe comenzar, reforzando el carácter proactivo de *EverPlanAPP*.

Este módulo actúa como un componente transversal, ya que se integra directamente con los módulos de tareas y rutinas, reaccionando a los cambios en estos y generando notificaciones de forma automática cuando corresponde.

Modelo de datos de notificaciones

La entidad central del módulo es *Notificacion*, la cual representa una alerta programada que debe ser enviada al usuario en una fecha y hora específicas.

Cada notificación está asociada obligatoriamente a un usuario, y opcionalmente a una tarea o una rutina, permitiendo reutilizar el mismo modelo para distintos tipos de eventos del sistema. Esta relación asegura que, si un usuario es eliminado, todas sus notificaciones asociadas se eliminan automáticamente, manteniendo la integridad de los datos.

El modelo *Notificacion* contempla los siguientes campos principales:

- **usuario:** usuario dueño de la notificación.
- **tarea:** referencia opcional a una tarea específica.
- **rutina:** referencia opcional a una rutina.
- **scheduled_at:** fecha y hora exacta en la que la notificación debe ser emitida.
- **enviada:** indica si la notificación ya fue procesada y enviada al *frontend*.
- **creado_en:** registra el momento en que se creó la notificación.

Creación automática de notificaciones mediante señales

Para desacoplar la lógica de notificaciones del resto del sistema, se utilizaron señales de Django, las cuales permiten reaccionar automáticamente ante cambios en otros modelos sin necesidad de llamadas explícitas entre módulos.

- **Tareas:** cada vez que una tarea es creada o modificada, el sistema evalúa si la notificación está activa y si la tarea no corresponde a una actividad de todo el día. En caso contrario, cualquier notificación asociada es eliminada. Si la tarea es válida

y futura, se crea o actualiza una notificación programada exactamente para la fecha y hora de inicio de la tarea.

- **Rutinas:** para las rutinas, el sistema calcula dinámicamente la siguiente ocurrencia futura, considerando el tipo de recurrencia configurado (diaria, semanal o mensual). A diferencia de generar notificaciones para todas las ocurrencias, el sistema mantiene únicamente la próxima notificación pendiente, lo que reduce la carga en la base de datos y simplifica la gestión del ciclo de vida de las notificaciones.

Este enfoque permite que el módulo de notificaciones funcione de manera automática y consistente, sin depender de lógica adicional en los *endpoints* de tareas o rutinas.

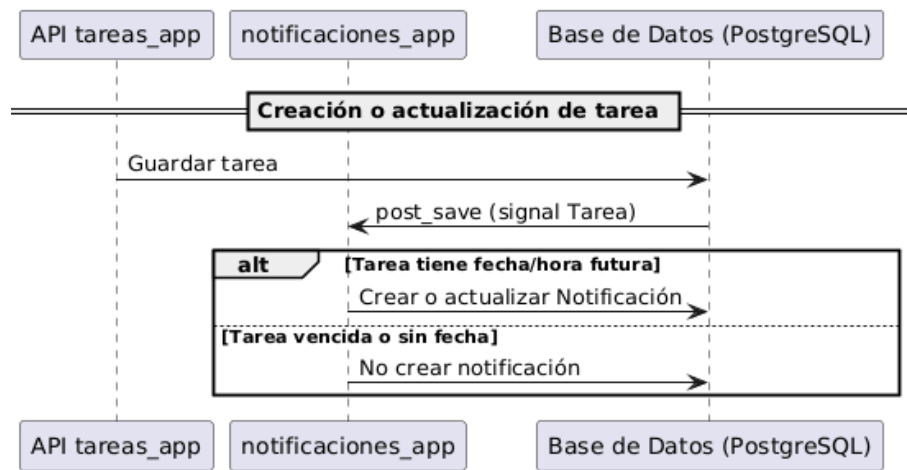


Figura 36: Diagrama de secuencia de la señal para tareas.

Fuente: Elaboración propia.

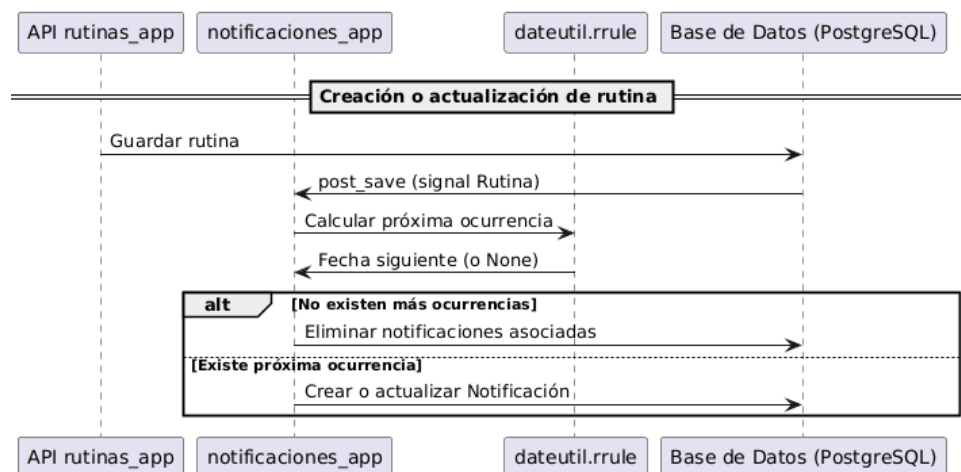


Figura 37: Diagrama de secuencia del proceso de la señal para rutinas.

Fuente: Elaboración propia.

Comunicación en tiempo real mediante WebSockets

Para entregar las notificaciones al usuario en tiempo real, se implementó un canal de comunicación basado en *WebSockets*, utilizando *Django Channels*.

Cada usuario dispone de un canal exclusivo identificado por su ID, al cual el *frontend* puede conectarse y permanecer a la espera de nuevas notificaciones. Cuando una notificación debe ser enviada, el *backend* la transmite directamente al canal correspondiente, permitiendo que el *frontend* la reciba y la muestre inmediatamente, sin necesidad de realizar consultas periódicas.

Este enfoque mejora significativamente la experiencia de usuario, ya que las notificaciones se reciben de forma instantánea mientras el usuario está activo en la aplicación.

Envío de notificaciones programadas

El envío efectivo de las notificaciones se realiza mediante un comando personalizado de *Django*, el cual puede ser ejecutado de forma periódica en el servidor (por ejemplo, mediante un *cron job*).

Este comando se encarga de:

- Buscar notificaciones pendientes cuya fecha y hora ya hayan sido alcanzadas.
- Enviar la notificación al canal *WebSocket* correspondiente al usuario.
- Marcar la notificación como enviada para evitar duplicados.
- En el caso de rutinas, calcular automáticamente la siguiente ocurrencia y generar una nueva notificación futura, manteniendo el ciclo activo hasta que la rutina finalice.

Este diseño permite que el *backend* controle completamente el flujo de notificaciones, incluso cuando el *frontend* no realiza solicitudes activas.

A continuación, se presenta el diagrama de secuencia que ilustra el flujo completo de este proceso, desde la ejecución periódica del cron en el servidor *AWS*, pasando por la ejecución del comando *enviar_notificaciones*, hasta la entrega final de la notificación en el *frontend*.

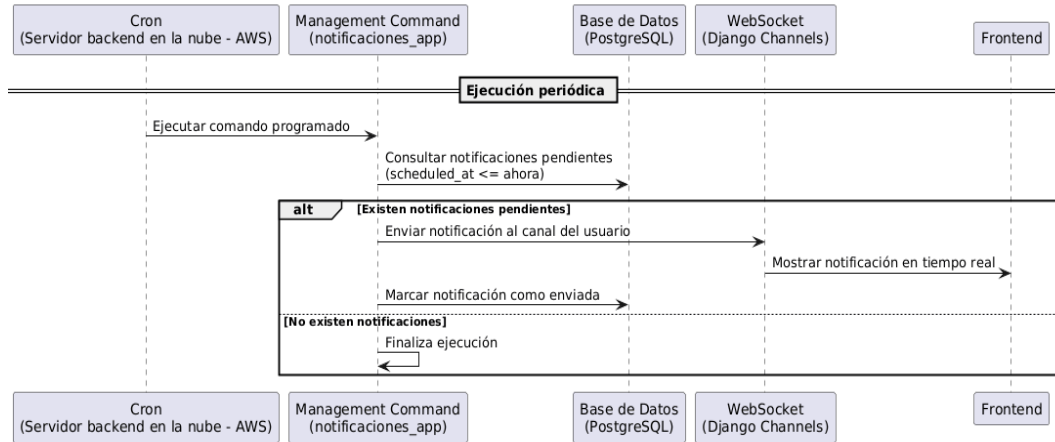


Figura 38: Diagrama de secuencia del proceso de envío de notificaciones desde el backend hacia el frontend.

Fuente: Elaboración propia.

Serialización y exposición de notificaciones

El módulo expone un *serializer* específico para las notificaciones, el cual permite transmitir al *frontend* la información necesaria para su visualización. Los campos relacionados con el estado interno del sistema, como si la notificación ya fue enviada, se definen como de solo lectura, evitando modificaciones indebidas desde el cliente.

Endpoint del módulo de notificaciones

Tabla 7: Endpoints del módulo notificacion_app.

Fuente: Elaboración Propia.

Endpoint	Método	Descripción	Seguridad
/api/notificaciones/listarNotificaciones/	GET	Retorna el listado de notificaciones del usuario	Usuario autenticado

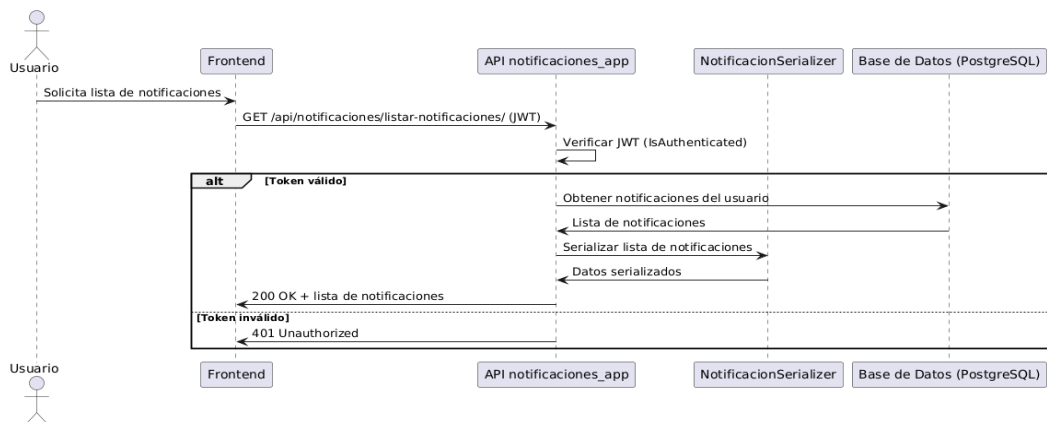


Figura 39: Diagrama de secuencia del proceso listar notificaciones.

Fuente: Elaboración propia.

3.7 Librerías y herramientas utilizadas en el backend

A continuación, se describen las principales librerías y herramientas empleadas en el desarrollo del *backend* de *EverPlanAPP*, indicando su propósito dentro del sistema y las razones de su utilización.

Django (5.2)

Framework principal utilizado para el desarrollo del *backend* de la aplicación, y se empleó para estructurar toda la aplicación, definir los modelos de datos, las vistas y la lógica de negocio, además de gestionar la comunicación con la base de datos a través de su *ORM*. Se optó por esta versión debido a que corresponde a una versión estable y con soporte oficial, lo que garantiza mayor seguridad, mantenimiento a largo plazo y compatibilidad con las funcionalidades más recientes del *framework*.

Django REST Framework – DRF (3.16.0)

Librería utilizada para la construcción de la *API REST*. Permitió definir los distintos *endpoints* del sistema, manejar los métodos *HTTP* (*GET*, *POST*, *PUT*, *DELETE*) y serializar la información intercambiada con el *frontend* de forma estructurada y segura.

django-rest-framework-simplejwt 5.5.0

Se utilizó para implementar la autenticación basada en *JSON Web Tokens (JWT)*. Esta librería permitió generar, validar e invalidar tokens de acceso y refresco, asegurando que solo los usuarios autenticados puedan acceder a los *endpoints* protegidos del sistema.

django-cors-headers 4.7.0

Librería utilizada durante la etapa de desarrollo web para permitir la comunicación entre el *backend* y el *frontend* ejecutándose en distintos orígenes. Su uso fue necesario mientras el *frontend* se desarrollaba como aplicación web, pero dejó de utilizarse cuando el proyecto pasó a una aplicación móvil, donde las restricciones *CORS* ya no eran requeridas.

PostgreSQL

Sistema gestor de base de datos utilizado para la persistencia de la información del sistema. El acceso a la base de datos se realiza exclusivamente a través del *ORM* de Django, permitiendo generar y mantener la estructura de las tablas mediante los comandos *makemigrations* y *migrate*, sin necesidad de escribir consultas *SQL* manuales.

psycopg2-binary 2.9.10

Adaptador utilizado para conectar *Django* con la base de datos *PostgreSQL*. Esta librería permite que el *ORM* de *Django* se comunique correctamente con la base de datos relacional.

dj-database-url 3.0.1

Librería empleada para simplificar la configuración de la base de datos mediante variables de entorno. Su uso facilitó la ejecución del proyecto en distintos entornos, como desarrollo local, contenedores *Docker* y despliegue en servidores.

Pillow 11.2.1

Librería utilizada para el manejo y procesamiento de imágenes dentro del *backend*. Fue necesaria para soportar la carga y gestión de imágenes asociadas a los eventos del horario personal, como imágenes de referencia o profesores.

python-dateutil 2.9.0

Librería empleada para el manejo avanzado de fechas y horas, especialmente en el módulo de rutinas. Permitted calcular recurrencias y ocurrencias de rutinas de forma flexible, evitando depender de servicios externos como *Google Calendar*, lo cual se ajusta al enfoque de un prototipo inicial.

Channels 4.3.1

Librería utilizada para implementar comunicación en tiempo real mediante *WebSockets*. Se empleó en el módulo de notificaciones para permitir que el *backend* envíe notificaciones al *frontend* de forma inmediata cuando estas se activan.

asgiref 3.8.1

Dependencia interna de *Django* y *Channels* que permite manejar aplicaciones asíncronas. Fue necesaria para soportar *WebSockets* y tareas asíncronas dentro del *backend*.

PyJWT 2.9.0

Librería utilizada internamente para la codificación y decodificación de *tokens JWT*. Es una dependencia fundamental para el funcionamiento del sistema de autenticación basado en *tokens*.

sqlparse 0.5.3

Dependencia interna de *Django* utilizada para el análisis y formateo de consultas *SQL* generadas por el *ORM*, contribuyendo al correcto funcionamiento de las operaciones sobre la base de datos.

Python 3.12.9

Lenguaje de programación utilizado para el desarrollo del *backend* de *EverPlanAPP*. Se trabajó con la versión 3.12.9 debido a su estabilidad, buen rendimiento y compatibilidad con las versiones recientes de *Django* y las librerías utilizadas en el proyecto.

Miniconda (entorno virtual)

Se utilizó *Miniconda* para la creación y gestión del entorno virtual del proyecto. Esta herramienta fue seleccionada ya que es el entorno recomendado y enseñado durante las clases, permitiendo aislar las dependencias del *backend* y asegurar que todas las librerías se ejecuten en versiones controladas y reproducibles.

CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN

En este capítulo se presenta el proceso de validación de la solución desarrollada, cuyo objetivo es comprobar que el *backend* cumple correctamente con los requerimientos funcionales, de seguridad y de comunicación definidos durante el diseño e implementación del sistema. La validación permite asegurar que la aplicación responde de manera adecuada ante distintos escenarios de uso, errores y condiciones esperadas, garantizando su correcto funcionamiento antes de su integración final con el *frontend*.

Para llevar a cabo este proceso, se utilizaron dos enfoques complementarios de validación. En primer lugar, se implementaron pruebas unitarias utilizando las herramientas nativas de *Django* y *Django Rest Framework*, las cuales permiten verificar de forma automática el comportamiento de los distintos *endpoints* del sistema. Estas pruebas se enfocan en validar la lógica interna del *backend*, el cumplimiento de los flujos definidos, el manejo de errores, la seguridad mediante autenticación *JWT* y el uso correcto de los códigos de respuesta *HTTP*.

Las pruebas unitarias fueron desarrolladas utilizando la clase *APITestCase* y el cliente de pruebas *APIClient*, el cual permite simular peticiones *HTTP* (*GET*, *POST*, *PUT*, *DELETE*) hacia los *endpoints* de la *API* sin necesidad de ejecutar el servidor de forma manual. A través de este cliente es posible enviar datos en formato *JSON*, incluir encabezados de autenticación como *tokens JWT* y analizar las respuestas del sistema, tales como el código de estado *HTTP* y el contenido retornado.

Durante la ejecución de los *tests*, *Django* utiliza una base de datos temporal de pruebas, en la cual se crean usuarios y datos de ejemplo de forma controlada. Esta base de datos se genera automáticamente antes de ejecutar los *tests* y se elimina al finalizar, lo que permite realizar validaciones sin afectar los datos reales del sistema. De esta manera, cada prueba se ejecuta en un entorno aislado, asegurando resultados consistentes y reproducibles.

En segundo lugar, se realizaron pruebas manuales de carácter visual mediante la herramienta *Postman*, la cual permitió simular peticiones *HTTP* a los distintos *endpoints* del *backend* de forma controlada. Estas pruebas facilitaron la validación del comportamiento de la *API* desde la perspectiva del cliente, permitiendo inspeccionar las respuestas del servidor, los datos retornados y los mensajes de error ante diferentes tipos de solicitudes.

Adicionalmente, *Postman* permitió la exportación y organización de colecciones de *endpoints*, lo que facilitó el trabajo colaborativo con el equipo de *frontend*. Gracias a esta funcionalidad, fue posible compartir los *endpoints* definidos y probados, permitiendo que el *frontend* pudiera realizar pruebas en un entorno local de manera más eficiente y alineada con el funcionamiento real del *backend*.

De esta forma, la combinación de pruebas unitarias automáticas mediante *APIClient* y pruebas manuales con Postman permitió realizar una validación integral del *backend*, asegurando la calidad, estabilidad y confiabilidad del sistema desarrollado.

4.1 Validación del módulo *auth_app*

Para validar el correcto funcionamiento del módulo de autenticación (*auth_app*), se implementaron pruebas unitarias utilizando el *framework* de pruebas de *Django Rest Framework*. Estas pruebas permiten verificar de forma automática el cumplimiento funcional, la seguridad básica, el manejo de errores y los códigos de respuesta *HTTP* definidos para cada *endpoint*.

A continuación, se describen las pruebas implementadas y sus criterios de validación.

I. Test: Registro de usuario

Método: *test_registro_usuario*

Objetivo: Verificar que un usuario pueda registrarse correctamente en el sistema proporcionando datos válidos.

Condiciones de éxito:

- El nombre de usuario no existe previamente.
- El correo electrónico es único.
- La contraseña cumple con las reglas de seguridad definidas por Django.
- Todos los campos obligatorios son enviados correctamente.

Condiciones de fallo:

- Falta alguno de los campos requeridos.
- El nombre de usuario o correo ya existe.
- La contraseña no cumple con los criterios de validación.

Resultado esperado: El servidor responde con código *201 Created* y retorna los *tokens* de autenticación *JWT* (*access* y *refresh*), confirmando la creación exitosa del usuario.

II. Test: Inicio de sesión con credenciales válidas

Método: *test_login_usuario_correcto*

Objetivo: Validar que el sistema permita el inicio de sesión cuando se proporcionan credenciales correctas.

Condiciones de éxito:

- El usuario existe en la base de datos.
- La contraseña ingresada coincide con la almacenada.

Condiciones de fallo:

- El sistema rechaza credenciales válidas.

Resultado esperado:

El servidor responde con código *200 OK* y entrega un token de acceso *JWT (access)* que permite autenticarse en *endpoints* protegidos.

III. Test: Inicio de sesión con credenciales inválidas

Método: *test_login_usuario_incorrecto*

Objetivo: Verificar que el sistema rechace correctamente un intento de inicio de sesión con credenciales inválidas.

Condiciones de éxito:

- El usuario existe, pero la contraseña ingresada es incorrecta.

Condiciones de fallo:

- El sistema permite iniciar sesión con credenciales erróneas.

Resultado esperado: El servidor responde con código *401 Unauthorized*, indicando que las credenciales proporcionadas no son válidas.

IV. Test: Acceso a endpoint protegido con token válido

Método: *test_acceso_inicio_con_token*

Objetivo: Validar que un usuario autenticado mediante *JWT* pueda acceder correctamente a un *endpoint* protegido.

Condiciones de éxito:

- El usuario inicia sesión correctamente.
- Se envía un *token JWT* válido en la cabecera de la solicitud (*Authorization: Bearer <token>*).

Condiciones de fallo:

- No se envía el *token*.
- El *token* es inválido o ha expirado.

Resultado esperado: El servidor responde con código *200 OK*, confirmando que el acceso a recursos protegidos solo es posible para usuarios autenticados.

V. Test: Cambio de contraseña del usuario autenticado

Método: *test_cambiar_password*

Objetivo: Comprobar que un usuario autenticado pueda cambiar su contraseña de forma segura.

Condiciones de éxito:

- El usuario está autenticado mediante *JWT*.
- La contraseña actual es correcta.
- La nueva contraseña cumple con las reglas de validación.

Condiciones de fallo:

- No se envía el token de autenticación.
- La contraseña actual es incorrecta.
- La nueva contraseña no cumple con las políticas de seguridad.

Resultado esperado: El servidor responde con código *200 OK*, confirmando que la contraseña fue actualizada correctamente.

VI. Test: Eliminación de cuenta de usuario

Método: *test_eliminar_cuenta*

Objetivo: Verificar que un usuario autenticado pueda eliminar su cuenta del sistema.

Condiciones de éxito:

- El usuario está autenticado mediante *JWT*.
- Se proporciona la contraseña correcta para confirmar la eliminación.

Condiciones de fallo:

- El token de autenticación es inválido o no se envía.
- La contraseña proporcionada es incorrecta.

Resultado esperado: El servidor responde con código *200 OK*, indicando que la cuenta fue eliminada exitosamente y, además, al eliminarse el usuario, todos los datos asociados a la cuenta se eliminan de forma consistente debido a la configuración de relaciones en cascada del modelo.

4.1.1 Ejecución de las pruebas unitarias de *auth_app*

Las pruebas unitarias fueron ejecutadas utilizando el comando: *python manage.py test auth_app*

La base de datos *test_everplanapp* siempre se creará de forma temporal cada vez que se ejecuten las pruebas unitarias y al final de estas se eliminará de forma automática.

```
Creating test database for alias 'default' ('test_everplanappdb')...
Operations to perform:
  Synchronize unmigrated apps: channels, messages, rest_framework, staticfiles
  Apply all migrations: admin, auth, auth_app, contenttypes, horarioPersonal_app, notificacion_app, rutinas_app, sessions, tarea_app, token_blacklist
Synchronizing apps without migrations:
  Creating tables...
  Running deferred SQL...
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying auth_app.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying horarioPersonal_app.0001_initial... OK
  Applying tarea_app.0001_initial... OK
  Applying rutinas_app.0001_initial... OK
  Applying notificacion_app.0001_initial... OK
  Applying sessions.0001_initial... OK
  Applying token_blacklist.0001_initial... OK
  Applying token_blacklist.0002_outstandingtoken_jti_hex... OK
  Applying token_blacklist.0003_auto_20171017_2007... OK
  Applying token_blacklist.0004_auto_20171017_2013... OK
  Applying token_blacklist.0005_remove_outstandingtoken_jti... OK
  Applying token_blacklist.0006_auto_20171017_2113... OK
  Applying token_blacklist.0007_auto_20171017_2214... OK
  Applying token_blacklist.0008_migrate_to_bigautofield... OK
  Applying token_blacklist.0010_fix_migrate_to_bigautofield... OK
  Applying token_blacklist.0011_linearizes_history... OK
  Applying token_blacklist.0012_alter_outstandingtoken_user... OK
```

Figura 40: Creación de la base de datos *test_everplanapp*.

Fuente: Elaboración propia.

```
System check identified no issues (0 silenced).
test_acceso_inicio_con_token (auth_app.tests.AuthAppTests.test_acceso_inicio_con_token)
Verifica acceso a endpoint protegido usando JWT. ... ok
test_cambiar_password (auth_app.tests.AuthAppTests.test_cambiar_password)
Verifica el cambio de contraseña del usuario autenticado. ... ok
test_eliminar_cuenta (auth_app.tests.AuthAppTests.test_eliminar_cuenta)
Verifica la eliminación de la cuenta del usuario. ... ok
test_login_usuario_correcto (auth_app.tests.AuthAppTests.test_login_usuario_correcto)
Verifica login con credenciales válidas. ... ok
test_login_usuario_incorrecto (auth_app.tests.AuthAppTests.test_login_usuario_incorrecto)
Verifica que el login falle con credenciales inválidas. ... ok
test_registro_usuario (auth_app.tests.AuthAppTests.test_registro_usuario)
Verifica que un usuario pueda registrarse correctamente. ... ok

-----
Ran 6 tests in 5.533s

OK
Destroying test database for alias 'default' ('test_everplanappdb')...
```

Figura 41: Pruebas unitarias del módulo *auth_app*.

Fuente: Elaboración propia.

4.2 Validación del módulo tarea_app

I. Test: Creación de tarea

Método: *test_crear_tarea*

Objetivo: Verificar que un usuario autenticado pueda crear una tarea válida en el sistema.

Condiciones de éxito:

- El usuario está autenticado mediante *JWT*.
- Se envía un título válido.
- La fecha y hora de la tarea es posterior al momento actual.
- Los datos cumplen las reglas del *TareaSerializer*.

Condiciones de fallo:

- El *token* de autenticación no se envía o es inválido.
- La fecha u hora enviada es anterior al momento actual.
- Faltan campos obligatorios.

Resultado esperado: El servidor responde con código *201 Created*, confirmando que la tarea fue creada correctamente y almacenada en la base de datos.

II. Test: Creación de tarea con fecha pasada

Método: *test_crear_tarea_fecha_pasada*

Objetivo: Verificar que el sistema no permita crear tareas con fechas u horas pasadas.

Condiciones de éxito:

- El usuario está autenticado.
- Se envía una fecha anterior al momento actual.

Condiciones de fallo:

- El sistema permite guardar una tarea en el pasado.

Resultado esperado: El servidor responde con código *400 Bad Request* y devuelve un mensaje de validación indicando que no se pueden crear tareas en fechas pasadas.

III. Test: Listado de tareas

Método: *test_listar_tareas*

Objetivo: Comprobar que un usuario autenticado pueda obtener solo sus propias tareas.

Condiciones de éxito:

- El usuario está autenticado mediante *JWT*.
- Existen tareas asociadas a su cuenta.

Condiciones de fallo:

- El *token* no se envía o es inválido.
- Se muestran tareas de otros usuarios.

Resultado esperado: El servidor responde con código *200 OK* y devuelve una lista con únicamente las tareas pertenecientes al usuario autenticado.

IV. Test: Detalle de tarea

Método: *test_detalle_tarea*

Objetivo: Verificar que un usuario pueda consultar el detalle de una tarea específica.

Condiciones de éxito:

- El usuario está autenticado.
- La tarea solicitada pertenece al usuario.

Condiciones de fallo:

- La tarea no existe.
- La tarea pertenece a otro usuario.
- El *token* no es válido.

Resultado esperado: El servidor responde con código *200 OK* y devuelve los datos completos de la tarea solicitada.

V. Test: Actualización de tarea

Método: *test_actualizar_tarea*

Objetivo: Verificar que un usuario pueda modificar los datos de una tarea propia.

Condiciones de éxito:

- El usuario está autenticado.
- La tarea pertenece al usuario.
- La nueva fecha y hora no es pasada.

Condiciones de fallo:

- La tarea no pertenece al usuario.
- Se intenta actualizar con una fecha pasada.
- El *token* es inválido.

Resultado esperado: El servidor responde con código *200 OK* y los datos de la tarea se actualizan correctamente en la base de datos.

VI. Test: Eliminación de tarea

Método: `test_eliminar_tarea`

Objetivo: Verificar que un usuario pueda eliminar una tarea específica.

Condiciones de éxito:

- El usuario está autenticado.
- La tarea pertenece al usuario.

Condiciones de fallo:

- La tarea no existe.
- La tarea pertenece a otro usuario.
- El *token JWT* es inválido.

Resultado esperado: El servidor responde con código *200 OK* o *204 No Content*, indicando que la tarea fue eliminada correctamente de la base de datos.

VII. Test: Eliminación de todas las tareas

Método: `test_eliminar_todas_las_tareas`

Objetivo: Verificar que un usuario pueda eliminar todas sus tareas de una sola vez.

Condiciones de éxito:

- El usuario está autenticado.
- Existen tareas asociadas a su cuenta.

Condiciones de fallo:

- El *token* es inválido o no se envía.

Resultado esperado: El servidor responde con código *200 OK* indicando cuántas tareas fueron eliminadas y confirmando que la cuenta del usuario quedó sin tareas asociadas.

VIII. Test: Estadísticas motivacionales

Método: `test_estadisticas_motivacionales`

Objetivo: Verificar que el sistema calcule correctamente las estadísticas de productividad del usuario.

Condiciones de éxito:

- El usuario está autenticado.
- Existen tareas completadas en los rangos de tiempo evaluados.

Condiciones de fallo:

- El usuario no está autenticado.

Resultado esperado: El servidor responde con código *200 OK* y devuelve valores como tareas de la última semana, promedio diario, racha de días y un mensaje motivacional generado automáticamente.

4.2.1 Ejecución de las pruebas unitarias de *tarea_app*

Para ejecutar las pruebas del módulo de tareas se utilizó el comando *python manage.py test tarea_app*, el cual permite validar automáticamente el correcto funcionamiento de sus *endpoints* y lógica interna. En la salida de la consola también se puede observar la ejecución de los *signals* asociados al sistema de notificaciones, evidenciando que cada vez que se crea o actualiza una tarea se dispara el evento *post_save*, lo que confirma que el mecanismo de notificaciones se integra correctamente al flujo normal de la aplicación.

```
(practicaDjango) C:\1A Mis Archivos\USM\Octavo Semestre\Proyecto de titulo\Backend_EverPlanApp\Backend_EverPlanApp>python manage.py test tarea_app
Found 8 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Signal ejecutado para tarea 1
Signal ejecutado para tarea 1
.Signal ejecutado para tarea 2
Signal ejecutado para tarea 3
.Signal ejecutado para tarea 4
.Signal ejecutado para tarea 5
.Signal ejecutado para tarea 6
Signal ejecutado para tarea 7
.Signal ejecutado para tarea 8
Signal ejecutado para tarea 8
.Signal ejecutado para tarea 9
.Signal ejecutado para tarea 10
.
-----
Ran 8 tests in 5.926s

OK
Destroying test database for alias 'default'...
```

Figura 42: Pruebas unitarias del módulo *tarea_app*.

Fuente: Elaboración propia.

4.3 Validación del módulo *rutinas_app*

I. Test: Creación de rutina

Método: *test_crear_rutina*

Objetivo: Verificar que un usuario autenticado pueda crear correctamente una rutina.

Condiciones de éxito:

- El usuario está autenticado mediante *JWT*.
- Se envían fechas de inicio y fin válidas (fin posterior al inicio).
- Se proporciona un título para la rutina.

Condiciones de fallo:

- El *token* de autenticación es inválido o no se envía.
- La fecha de término es anterior o igual a la fecha de inicio.

Resultado esperado: El servidor responde con código *201 CREATED* y retorna los datos de la rutina creada asociada al usuario autenticado.

II. Test: Listado de rutinas

Método: *test_listar_rutinas*

Objetivo: Verificar que el usuario autenticado pueda obtener la lista de sus rutinas.

Condiciones de éxito:

- El usuario está autenticado.
- Existen rutinas asociadas al usuario.

Condiciones de fallo:

- No se envía token o el token es inválido.

Resultado esperado: El servidor responde con *200 OK* y retorna únicamente las rutinas pertenecientes al usuario autenticado.

III. Test: Detalle de rutina

Método: *test_detalle_rutina*

Objetivo: Verificar que el usuario pueda consultar los detalles de una rutina específica.

Condiciones de éxito:

- El usuario está autenticado.
- La rutina pertenece al usuario.

Condiciones de fallo:

- La rutina no existe.
- La rutina pertenece a otro usuario.
- El *token* es inválido.

Resultado esperado: El servidor responde con *200 OK* y entrega los datos completos de la rutina solicitada.

IV. Test: Actualización de rutina

Método: *test_actualizar_rutina*

Objetivo: Verificar que una rutina pueda ser modificada por su propietario.

Condiciones de éxito:

- El usuario está autenticado.
- La rutina pertenece al usuario.
- Los nuevos datos cumplen las validaciones.

Condiciones de fallo:

- El *token* es inválido.
- Se intenta actualizar una rutina que no pertenece al usuario.
- La fecha de fin es menor o igual a la fecha de inicio.

Resultado esperado: El servidor responde con *200 OK* y refleja los datos actualizados de la rutina.

V. Test: Eliminación de rutina

Método: *test_eliminar_rutina*

Objetivo: Verificar que una rutina pueda ser eliminada por su dueño.

Condiciones de éxito:

- El usuario está autenticado.
- La rutina pertenece al usuario.

Condiciones de fallo:

- El token es inválido.
- La rutina no existe o no pertenece al usuario.

Resultado esperado: El servidor responde con *204 NO CONTENT*, indicando que la rutina fue eliminada correctamente.

VI. Test: Eliminación de todas las rutinas

Método: *test_eliminar_todas_las_rutinas*

Objetivo: Verificar que el usuario pueda eliminar todas sus rutinas en una sola operación.

Condiciones de éxito:

- El usuario está autenticado.
- Existen rutinas asociadas al usuario.

Condiciones de fallo:

- El *token* no es válido o no se envía.

Resultado esperado: El servidor responde con *200 OK* e indica cuántas rutinas fueron eliminadas.

Test: Listado de ocurrencias

Método: *test_listar_ocurrencias*

Objetivo: Verificar que el sistema genere correctamente las ocurrencias de las rutinas usando reglas de recurrencia (*RRULE*).

Condiciones de éxito:

- El usuario está autenticado.
- Existen rutinas con fechas y recurrencia válida.

Condiciones de fallo:

- El *token* es inválido.
- No existen rutinas registradas.

Resultado esperado: El servidor responde con *200 OK* y retorna una lista de ocurrencias calculadas dinámicamente a partir de las rutinas del usuario.

VII. Test: Validación de fechas

Método: *test_no_permitir_fin_antes_de_inicio*

Objetivo: Verificar que no se permita crear una rutina cuya fecha de término sea anterior o igual a la fecha de inicio.

Condiciones de éxito:

- Se envían fechas válidas (*fin > inicio*).

Condiciones de fallo:

- La fecha de término es menor o igual a la fecha de inicio.

Resultado esperado: El servidor responde con *400 BAD REQUEST*, indicando que la rutina no cumple las reglas de validación definidas en el modelo y el *serializer*.

4.3.1 Ejecución de las pruebas unitarias de *rutinas_app*

```
test_actualizar_rutina (rutinas_app.tests.RutinaAPITest.test_actualizar_rutina) ... ok
test_crear_rutina (rutinas_app.tests.RutinaAPITest.test_crear_rutina) ... ok
test_detalle_rutina (rutinas_app.tests.RutinaAPITest.test_detalle_rutina) ... ok
test_eliminar_rutina (rutinas_app.tests.RutinaAPITest.test_eliminar_rutina) ... ok
test_eliminar_todas_las_rutinas (rutinas_app.tests.RutinaAPITest.test_eliminar_todas_las_rutinas) ... ok
test_listar_ocurrencias (rutinas_app.tests.RutinaAPITest.test_listar_ocurrencias) ... ok
test_listar_rutinas (rutinas_app.tests.RutinaAPITest.test_listar_rutinas) ... ok
test_no_permitir_fin_antes_de_inicio (rutinas_app.tests.RutinaAPITest.test_no_permitir_fin_antes_de_inicio) ... ok

-----
Ran 8 tests in 5.836s

OK
Destroying test database for alias 'default' ('test_everplanappdb')...
```

Figura 43: Pruebas unitarias del módulo *rutinas_app*.

Fuente: Elaboración propia.

4.4 Validación del módulo horarioPersonal_app

I. Test: Creación de evento

Método: *test_crear_evento*

Objetivo: Verificar que un usuario autenticado pueda crear un evento en su horario personal.

Condiciones de éxito:

- El usuario está autenticado mediante *JWT*.
- La hora de fin es posterior a la hora de inicio.
- Se envían los datos obligatorios del evento.

Condiciones de fallo:

- El *token* no es válido o no se envía.
- La hora de fin es menor o igual a la hora de inicio.

Resultado esperado: El servidor responde con *201 CREATED* y registra el evento asociado al usuario autenticado.

II. Test: Listado de eventos

Método: *test_listar_eventos*

Objetivo: Verificar que el usuario pueda visualizar los eventos de su horario.

Condiciones de éxito:

- El usuario está autenticado.
- Existen eventos asociados al usuario.

Condiciones de fallo:

- El *token* es inválido o no se envía.

Resultado esperado: El servidor responde con *200 OK* y devuelve únicamente los eventos del usuario autenticado.

III. Test: Detalle de evento

Método: *test_detalle_evento*

Objetivo: Verificar que el usuario pueda consultar el detalle de un evento específico.

Condiciones de éxito:

- El usuario está autenticado.
- El evento pertenece al usuario.

Condiciones de fallo:

- El evento no existe.
- El evento pertenece a otro usuario.
- El *token* es inválido.

Resultado esperado: El servidor responde con *200 OK* y retorna la información completa del evento.

IV. Test: Actualización de evento

Método: *test_actualizar_evento*

Objetivo: Verificar que un evento pueda ser modificado por su propietario.

Condiciones de éxito:

- El usuario está autenticado.
- El evento pertenece al usuario.

Condiciones de fallo:

- El *token* es inválido.
- Se intenta modificar un evento que no pertenece al usuario.

Resultado esperado: El servidor responde con *200 OK* y refleja los datos actualizados del evento.

V. Test: Eliminación de evento

Método: *test_eliminar_evento*

Objetivo: Verificar que un evento pueda ser eliminado.

Condiciones de éxito:

- El usuario está autenticado.
- El evento pertenece al usuario.

Condiciones de fallo:

- El *token* es inválido.
- El evento no existe o no pertenece al usuario.

Resultado esperado: El servidor responde con *204 NO CONTENT*, confirmando que el evento fue eliminado.

VI. Test: Eliminación de todos los eventos

Método: *test_eliminar_todos_los_eventos*

Objetivo: Verificar que el usuario pueda eliminar todos los eventos de su horario.

Condiciones de éxito:

- El usuario está autenticado.
- Existen eventos registrados.

Condiciones de fallo:

- El token es inválido.

Resultado esperado: El servidor responde con *200 OK* indicando cuántos eventos fueron eliminados.

VII. Test: Validación de horas

Método: *test_no_permitir_hora_fin_menor*

Objetivo: Verificar que el sistema no permita registrar eventos con una hora de término inválida.

Condiciones de éxito:

- La hora de fin es mayor a la hora de inicio.

Condiciones de fallo:

- La hora de fin es menor o igual a la hora de inicio.

Resultado esperado: El servidor responde con *400 BAD REQUEST*, indicando que los datos no cumplen las reglas de validación.

4.4.1 Ejecución de las pruebas unitarias de `horarioPersonal_app`

```
test_actualizar_evento (horarioPersonal_app.tests.HorarioAPITest.test_actualizar_evento)
Verifica que un evento pueda ser actualizado. ... ok
test_crear_evento (horarioPersonal_app.tests.HorarioAPITest.test_crear_evento)
Verifica que un usuario pueda crear un evento de horario. ... ok
test_detalle_evento (horarioPersonal_app.tests.HorarioAPITest.test_detalle_evento)
Verifica que se pueda obtener el detalle de un evento. ... ok
test_eliminar_evento (horarioPersonal_app.tests.HorarioAPITest.test_eliminar_evento)
Verifica que un evento pueda ser eliminado. ... ok
test_eliminar_todos_los_eventos (horarioPersonal_app.tests.HorarioAPITest.test_eliminar_todos_los_eventos)
Verifica que se puedan eliminar todos los eventos del usuario. ... ok
test_listar_eventos (horarioPersonal_app.tests.HorarioAPITest.test_listar_eventos)
Verifica que se puedan listar los eventos del usuario. ... ok
test_no_permitir_hora_fin_menor (horarioPersonal_app.tests.HorarioAPITest.test_no_permitir_hora_fin_menor)
Verifica que no se permita crear un evento con hora de fin menor o igual a la de inicio. ... ok

-----
Ran 7 tests in 5.328s

OK
Destroying test database for alias 'default' ('test_everplanappdb')...
```

Figura 44: Pruebas unitarias del módulo `horarioPersonal_app`.

Fuente: Elaboración propia.

4.5 Validación del módulo *notificaciones_app*

La validación del módulo *notificaciones_app* se realizó verificando el funcionamiento integrado de las señales automáticas del sistema y el flujo de ejecución periódica de notificaciones mostrado en el diagrama. Este módulo no depende de llamadas directas desde los controladores, sino que se activa de forma reactiva y automática cuando cambian los datos de tareas o rutinas.

Cuando una tarea o una rutina es creada o modificada, los *signals post_save* asociados a los modelos Tarea y Rutina se ejecutan automáticamente. Estos *signals* se encargan de crear, actualizar o eliminar los registros en la tabla *Notificacion*, según el estado de la entidad. En el caso de las tareas, se valida que la notificación esté activa y que la fecha aún no haya pasado. Para las rutinas, se calcula la próxima ocurrencia válida mediante reglas de recurrencia y se programa la notificación correspondiente.

De esta forma, la base de datos de notificaciones siempre refleja exactamente qué eventos deben ser notificados y en qué momento, sin intervención manual del *backend* ni del *frontend*.

El envío de notificaciones se valida mediante el flujo representado en el diagrama. Un proceso programado (*cron*) ejecuta periódicamente un comando de administración de Django, el cual consulta la base de datos en busca de notificaciones pendientes, es decir, aquellas cuyo campo *scheduled_at* es menor o igual a la hora actual y que aún no han sido marcadas como enviadas. Cuando existen notificaciones pendientes, estas son enviadas al canal *WebSocket* correspondiente al usuario a través de *Django Channels*, permitiendo que el *frontend* las reciba y muestre en tiempo real. Luego, la notificación es marcada como enviada para evitar duplicaciones.

Este mecanismo valida que el sistema de notificaciones funciona de extremo a extremo: desde la generación automática de notificaciones mediante *signals*, pasando por su almacenamiento en la base de datos, hasta su entrega en tiempo real al usuario. Esto garantiza coherencia, sincronización y confiabilidad en la entrega de recordatorios y alertas dentro de la plataforma.

4.5.1 Limitaciones actuales y mejoras futuras del módulo *notificaciones_app*

Si bien el módulo de notificaciones cumple correctamente su función y fue validado mediante *signals* y ejecución periódica, su implementación actual presenta algunas limitaciones importantes a nivel de escalabilidad y eficiencia.

Actualmente, el sistema utiliza un proceso *cron* que se ejecuta cada 2 segundos para consultar la base de datos y verificar si existen notificaciones pendientes (*scheduled_at* <= *now*). Este enfoque, aunque simple y funcional, implica que la base de datos es consultada de forma muy frecuente incluso cuando no existen notificaciones por enviar. A medida

que el número de usuarios y notificaciones crece, este patrón puede generar carga innecesaria, aumento de latencia y un mayor consumo de recursos del servidor.

Como mejora futura, una alternativa sería el uso de sistemas de colas y tareas asíncronas, como *Celery* con *Redis* o *RabbitMQ*. En este enfoque, cada notificación se programaría como una tarea diferida que se ejecuta exactamente en el momento requerido, evitando la necesidad de hacer consultas periódicas a la base de datos. Sin embargo, esta solución introduce mayor complejidad operativa, ya que requiere mantener servicios adicionales, monitorear colas y manejar fallos de *workers*.

Otra opción es utilizar servicios externos de *scheduling*, como *AWS EventBridge*, *Google Cloud Scheduler* o servicios de notificaciones *push*. Estos permiten delegar la responsabilidad del temporizado a una infraestructura especializada, reduciendo la carga del *backend*. La desventaja es la dependencia de servicios de terceros, costos asociados y una menor capacidad de personalización del comportamiento interno del sistema.

También es posible optimizar el diseño actual utilizando funcionalidades del motor de base de datos, como tareas programadas o extensiones de *PostgreSQL* (por ejemplo, *pg_cron*) para ejecutar consultas directamente en la base de datos. Esto reduciría el tráfico entre el *backend* y la base de datos, pero aumenta el acoplamiento con un motor específico y dificulta la portabilidad del sistema.

En conclusión, aunque la implementación actual es adecuada para un entorno académico y de carga moderada, en un escenario productivo con muchos usuarios sería recomendable migrar hacia un sistema de ejecución basado en eventos o colas, con el fin de mejorar la eficiencia, la escalabilidad y la robustez del módulo de notificaciones.

4.6 Apoyo de herramientas en el proceso de validación

Además de las pruebas unitarias automatizadas, el proceso de validación del *backend* se complementó mediante el uso de herramientas que permitieron inspeccionar directamente el comportamiento del sistema y los datos generados. En este contexto, *Postman* y el panel de administración de *Django* (*Django Admin*) jugaron un rol clave.

Por un lado, *Postman* fue utilizado para realizar pruebas manuales de los *endpoints* de la *API*, permitiendo enviar solicitudes *HTTP* (*GET*, *POST*, *PUT*, *DELETE*) y analizar de forma clara las respuestas del servidor en formato JSON. Esto facilitó verificar que los datos enviados desde el *frontend* serían interpretados correctamente por el *backend*, validar estructuras de respuesta, códigos de estado *HTTP* y también probar casos de error. Además, *Postman* permitió exportar y compartir colecciones de *endpoints*, lo que facilitó que el equipo de *frontend* pudiera ejecutar las mismas pruebas en su entorno local, asegurando una integración más fluida entre ambas partes del sistema.

Por otro lado, *Django Admin* permitió validar el estado interno de la aplicación desde una perspectiva administrativa. A través de esta interfaz fue posible comprobar rápidamente si

los registros se estaban creando, actualizando o eliminando correctamente en la base de datos, incluyendo usuarios, tareas, rutinas, eventos de horario y notificaciones. Además, el panel administrativo permitió crear, modificar y eliminar registros de forma directa y visual, sin necesidad de ejecutar consultas *SQL* ni realizar llamados a los *endpoints* de la *API*, lo que agilizó considerablemente la verificación de reglas de negocio, señales (*signals*) y relaciones entre modelos.

En conjunto, estas herramientas permitieron una verificación más rápida, visual y confiable del sistema, complementando las pruebas unitarias con una validación práctica del flujo real de datos entre el *backend*, la base de datos y el *frontend*.

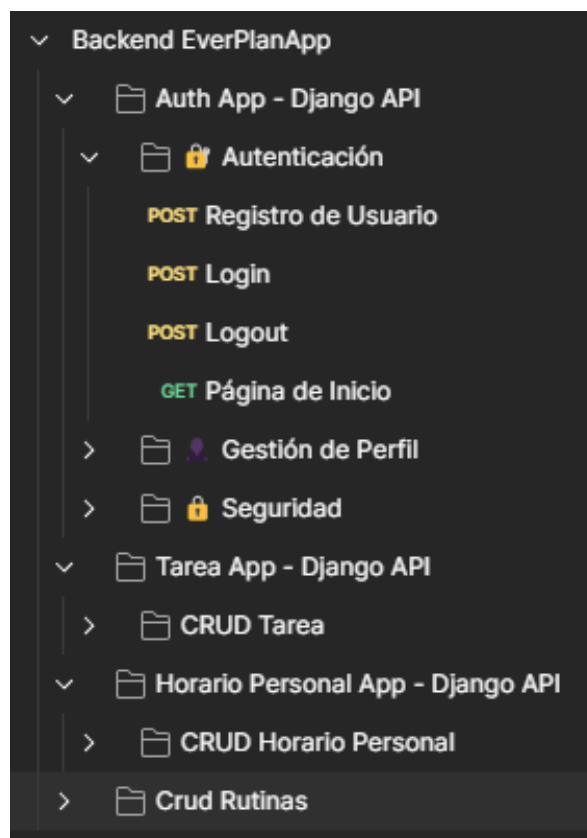


Figura 45: Endpoints de EverPlanAPP en Postman.
Fuente: Elaboración propia.

DISEÑO E IMPLEMENTACIÓN DEL BACKEND DE EVERPLANAPP: APLICACIÓN DE ORGANIZACIÓN CON ENFOQUE INCLUSIVO

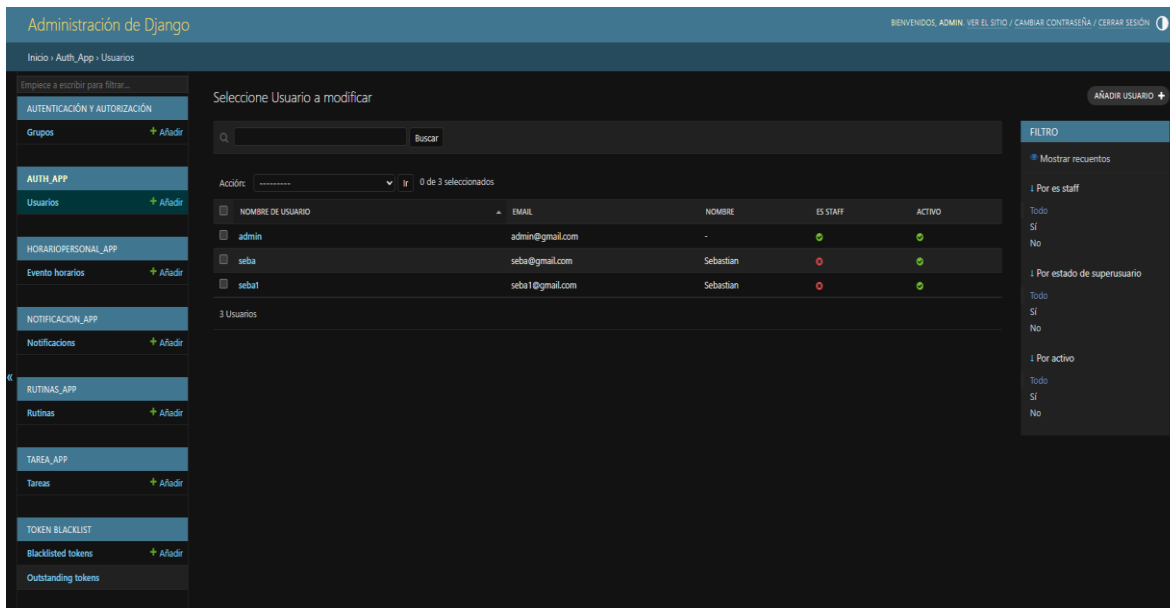


Figura 48: Ejemplo administrador de Django consultar tabla de usuarios.

Fuente: Elaboración propia.

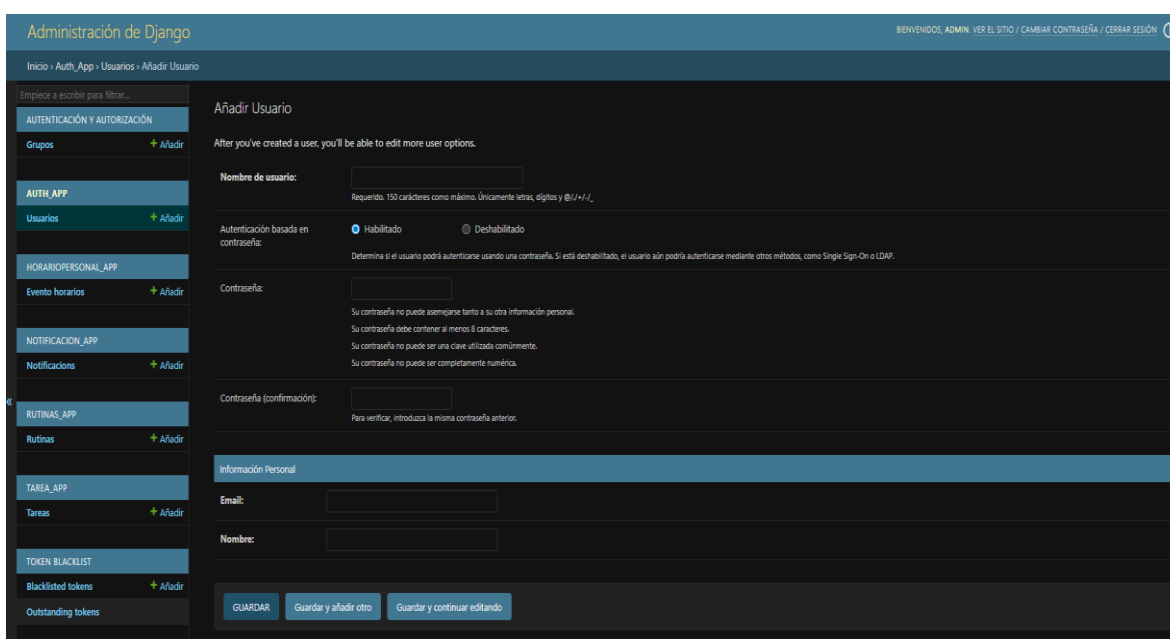


Figura 49: Ejemplo administrador de Django añadir nuevo usuario.

Fuente: Elaboración propia.

CAPÍTULO 5: CONCLUSIONES

El desarrollo del backend de EverPlanApp fue una experiencia que permitió aplicar, consolidar y profundizar muchos de los conocimientos adquiridos durante la carrera. A través de este proyecto no solo se logró construir una solución funcional para la gestión de planificación personal, sino que también se comprendió de forma práctica cómo se diseña, estructura y mantiene un sistema backend real, pensado para ser utilizado por usuarios finales y no solo como un ejercicio académico.

Uno de los aprendizajes más importantes fue comprender la importancia de una arquitectura bien organizada. Al separar el sistema en módulos como autenticación, tareas, rutinas, horario personal y notificaciones, fue posible trabajar cada parte de forma independiente, lo que facilitó tanto el desarrollo como las pruebas. Esto permitió entender que una buena estructura no solo mejora el orden del código, sino que también hace que el sistema sea más fácil de mantener, extender y depurar a futuro.

También se aprendió el valor de la automatización a través de mecanismos como los signals de Django. Implementar notificaciones que se crean o actualizan automáticamente cuando una tarea o rutina cambia permitió comprender cómo los eventos del sistema pueden manejarse de forma elegante y sin duplicar lógica en múltiples partes del código. Esta parte fue especialmente importante, ya que permitió que el sistema se mantuviera coherente incluso cuando los datos se modificaban desde distintos puntos, como la API o el panel de administración.

Las pruebas unitarias fueron otro aspecto clave del aprendizaje. Antes de este proyecto, las pruebas solían quedar en segundo plano, pero al implementarlas con APIClient se entendió lo útiles que son para asegurar que el sistema realmente haga lo que se espera. Poder ejecutar todos los tests y confirmar que la creación, actualización y eliminación de datos funcionaban correctamente entregó mucha más confianza en la estabilidad del backend.

Herramientas como Postman y Django Admin también jugaron un rol fundamental durante el desarrollo. Postman permitió probar rápidamente los endpoints y visualizar el formato real de los datos que el frontend recibiría, mientras que Django Admin facilitó la creación, modificación y eliminación de registros sin necesidad de escribir consultas SQL ni crear llamadas a la API para todo. Esto hizo que el proceso de validación y depuración fuera mucho más rápido y eficiente.

Sin embargo, el proyecto también permitió identificar limitaciones y desafíos reales. Por ejemplo, el uso de un proceso cron que se ejecuta cada pocos segundos para revisar

notificaciones funciona correctamente, pero no representa la solución más eficiente para un sistema a gran escala. Esto llevó a comprender que muchas veces una solución técnica puede ser válida para un prototipo o una aplicación de tamaño medio, pero debe evolucionar si el sistema crece. En ese sentido, este proyecto abrió la puerta a conceptos más avanzados como colas de mensajes, servicios de scheduling y arquitecturas orientadas a eventos.

En conclusión, EverPlanApp no solo cumple con los objetivos planteados a nivel funcional, sino que también representó una experiencia de aprendizaje muy completa. Permitió enfrentar problemas reales, tomar decisiones de diseño, implementar validaciones, automatizar procesos y asegurar la calidad del sistema mediante pruebas. Todo esto representa una base sólida para la formación como desarrollador backend y deja una visión mucho más clara de cómo se construyen y mantienen aplicaciones modernas en un entorno profesional.

REFERENCIAS BIBLIOGRÁFICAS

- Ministerio de Educación de Chile. (2023). *Guía de apoyo para la elaboración de un Plan de Equidad en la Educación Superior (PEES)*. Recuperado de https://acceso.mineduc.cl/wp-content/uploads/2024/11/guia-PESD_compressed.pdf
- Red de Salud UC Christus. (s.f.). *Neurodiversidad: una forma diferente de pensar y aprender*. Recuperado el 14 de mayo de 2025, de https://www.redclinica.cl/especiales-de-salud/especial_neurodiversidad.aspx
- Vásquez, M., & Tapia, C. (2021). *Revisión sobre el impacto del TDAH en el desempeño académico universitario*. Revista Chilena de Neuropsicología, 13(1), 45-56. Recuperado de https://www.scielo.cl/scielo.php?pid=S0718-07052021000100091&script=sci_arttext
- González, P., & Muñoz, V. (2021). *Prevalencia de Trastorno del Espectro Autista en población escolar chilena*. Revista Chilena de Pediatría, 92(4), 519–527. Recuperado de https://www.scielo.cl/scielo.php?pid=S2452-60532021000400519&script=sci_arttext
- Child Mind Institute. (s.f.). *¿Cómo pueden las escuelas apoyar a estudiantes neurodivergentes?* Recuperado el 14 de mayo de 2025, de <https://childmind.org/es/articulo/como-pueden-las-escuelas-apoyar-a-estudiantes-neurodiversos>
- Fundación Descúbreme. (2023). *Tecnología y neurodiversidad: herramientas inclusivas para el aprendizaje*. Recuperado de <https://www.descubreme.cl/tecnologia-y-neurodiversidad>