

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Diseño e implementación de un sistema modular de reservas y recordatorios para peluquerías

Federico Triviño García-Villaraco

Julio, 2026

Diseño e implementación de un sistema modular de reservas y recordatorios para peluquerías



UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Tecnologías y Sistemas de Información

GRADO EN INGENIERÍA INFORMÁTICA

Diseño e implementación de un sistema modular de reservas y recordatorios para peluquerías

Autor: Federico Triviño García-Villaraco

Director: David Vallejo Fernández

Co-director: Santiago Sánchez Sobrino

Julio, 2026

Federico Triviño García-Villaraco

Ciudad Real – Spain

E-mail: Federico.Trivino@alu.uclm.es
Telephone: 686 18 52 80

© 2026 Federico Triviño García-Villaraco

This work is licensed under CC BY-SA 4.0. To view a copy of this license, visit
`https://creativecommons.org/licenses/by-sa/4.0/`

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL	SECRETARIO
Fdo.:	Fdo.:	Fdo.:

Resumen

Este Trabajo Fin de Grado propone el diseño y desarrollo de un sistema de gestión automatizada de citas para pequeños negocios de peluquería, basado en un *backend* propio, una base de datos estructurada, un módulo de recordatorios y un canal de comunicación conversacional mediante *Telegram*. La motivación surge de una problemática habitual en estos establecimientos, donde las llamadas constantes para solicitar cita interrumpen el trabajo del profesional y obligan a llevar la planificación en papel o en herramientas poco integradas, con el riesgo añadido de errores, pérdidas de información o solapamientos.

La solución planteada se basa en una arquitectura ligera y modular que centraliza la configuración del negocio (servicios, duraciones, horarios y reglas de reserva) y reduce al mínimo la intervención humana en la operativa diaria. El *backend* se encarga de la lógica de negocio, el control del flujo conversacional, la validación de la información del cliente y la generación de recordatorios programados. La base de datos almacena perfiles, catálogo de servicios e historial de visitas, permitiendo aplicar reglas simples de recomendación (por ejemplo, detectar periodicidades habituales y sugerir de forma proactiva una cita) sin depender de modelos complejos de aprendizaje automático.

La gestión de la agenda no se implementa localmente, sino que se delega a *Google Calendar*, aprovechando su API para crear, modificar o cancelar reservas en tiempo real, manteniendo además un registro propio de citas para trazabilidad (estado, identificadores de evento y recordatorios enviados). El cliente interactúa únicamente a través de chat, sin necesidad de instalar aplicaciones adicionales ni acceder a páginas web, para solicitar un servicio, de forma que el sistema consulte disponibilidad, proponga franjas compatibles y confirme la reserva con notificación inmediata. Se pretende que el resultado final reduzca las interrupciones que sufre el dueño del negocio durante la jornada y mejore la experiencia tanto del profesional como del cliente.

Abstract

This Final Degree Project proposes the design and development of an automated appointment management system for small hairdressing businesses, based on a custom *backend*, a structured database, a reminder module and a conversational communication channel through *Telegram*. The motivation comes from a common problem in these establishments, where constant phone calls to request an appointment interrupt the professional's work and force them to keep the planning on paper or in poorly integrated tools, with the added risk of mistakes, loss of information or overlapping bookings.

The proposed solution is based on a light and modular architecture that centralises the business configuration (services, durations, schedules and booking rules) and reduces human intervention in the daily operation to a minimum. The *backend* is in charge of the business logic, the control of the conversational flow, the validation of the customer information and the generation of scheduled reminders. The database stores profiles, the service catalogue and the visit history, which makes it possible to apply simple recommendation rules (for example, detecting usual patterns and suggesting an appointment in a proactive way) without depending on complex machine learning models.

The calendar is not managed locally, but is delegated to *Google Calendar* or an equivalent service, using its API to create, modify or cancel bookings in real time, while also keeping an own record of appointments for traceability (status, event identifiers and reminders sent). The customer interacts only through chat, without the need to install extra applications or visit web pages, in order to request a service, so that the system checks availability, proposes compatible time slots and confirms the booking with an immediate notification. The goal is that the final result reduces the interruptions that the business owner suffers during the working day and improves the experience for both the professional and the customer.

Agradecimientos

Con la finalización de este TFG se cierra una etapa de mi vida que quedó en *standby* durante mucho más tiempo del que debería, pero como se suele decir, *más vale tarde que nunca*.

En primer lugar, quiero agradecer a mis abuelos y a mis padres todo el esfuerzo realizado a lo largo de sus vidas. Gracias a ese esfuerzo, los que vinimos después hemos tenido la oportunidad de completar una formación académica y comenzar una carrera profesional en un sector que nos apasiona y, en definitiva, tener una vida mejor. Algo que hoy en día parece tan habitual, para ellos no fue tan sencillo. Gracias de corazón.

En segundo lugar, me gustaría mencionar a esas personas que empezaron siendo compañeros de viaje y que decidieron quedarse para ser compañeros de vida y a aquellos que, por las circunstancias que sean, ya no están tan presentes pero lo estuvieron antes. Juntos hemos disfrutado de muy buenos momentos y también hemos superado momentos más complicados, apoyándonos y trabajando juntos, estos momentos al final son los que más unen, gracias a ellos hemos llegado hasta aquí. A todos vosotros, muchas gracias.

También me gustaría tener unas palabras de agradecimiento para todas aquellas personas que, de una forma u otra, han estado presente a lo largo de mi educación: maestros, profesores y personal de administración y servicios. En especial a aquellos que sobrepasaron los aspectos más puramente profesionales y tuvieron una implicación más personal, demostrando confianza en mí, ayudándome en los momentos en los que pude estar más disperso, inculcándome pasión por el aprendizaje y buscando la motivación para impulsar mi crecimiento tanto personal como académico. Habéis sido un ejemplo y la mejor forma de agradecéroslo es seguir vuestro ejemplo día a día.

Por último, un agradecimiento especial para David y Santiago y a todo el personal de la Escuela Superior de Informática de Ciudad Real por haber confiado en mí y haberme dado la oportunidad de *sacarme la espinita* y, por fin, poder cerrar como merece una de las etapas más importantes de mi vida. Muchísimas gracias.

Fede ;)

A los que siempre estuvieron

Índice general

Resumen V

Abstract VII

Agradecimientos IX

Índice general XIII

Índice de tablas XVII

Índice de figuras XIX

Lista de acrónimos XXI

1. Introducción 1

1.1. Visión general 1

1.2. Problemática 2

1.3. Propuesta de solución 3

2. Objetivos 7

2.1. Objetivo principal 7

2.2. Objetivos específicos 7

3. Estado del arte 9

3.1. Sistemas integrales de gestión de citas en el sector peluquería 9

3.2. Tecnologías de desarrollo web 11

3.2.1. *Backend*: Java y Spring Boot 11

3.2.2. Persistencia: PostgreSQL, JPA/Hibernate y Flyway 13

3.2.3. Seguridad: Spring Security y JWT 14

3.2.4. Arquitectura hexagonal 16

3.3. Agentes de IA y LLMs 18

3.3.1. Google Calendar 21
3.3.2. Bot conversacional: Python y Telegram Bot API 21
3.3.3. Frontend: TypeScript y React 21

4. Metodología **23**
4.1. Metodología de desarrollo iterativo-incremental 23
4.1.1. Ventajas 23
4.1.2. Desventajas 24
4.1.3. Aplicación en el proyecto 24
4.2. Marco tecnológico 27

5. Arquitectura **29**
5.1. Visión general 29
5.1.1. Organización en capas 31
5.2. Módulo de persistencia 33
5.3. Modulo Backoffice - *Backend* 37
5.3.1. El filtro de seguridad (JWT) 37
5.3.2. El controlador REST (adaptador de entrada) 38
5.3.3. El mapper 38
5.3.4. El caso de uso 39
5.3.5. El puerto de salida y su adaptador 39
5.3.6. Viaje de vuelta y el manejo de errores 39
5.4. Módulo Backoffice - Frontend 40
5.4.1. Estructura general y punto de entrada 40
5.4.2. Páginas y formularios 40
5.4.3. Gestión de autenticación 42
5.4.4. Enrutado (React Router) 42
5.4.5. Roles y permisos 43
5.4.6. Visualización del calendario 43
5.5. Módulo Google Calendar API 44
5.5.1. Autenticación con Google mediante OAuth 2.0 46
5.6. Módulo IA 47
5.6.1. Spring AI 48
5.6.2. Configuración de Spring AI 48
5.6.3. Herramientas de consulta para LLM (MCP) 49
5.6.4. Caso de uso: orquestación del flujo de chat 50

5.6.5. Síntesis de componentes Spring AI 50
5.6.6. Prompt 51
5.7. Módulo ChatBot 53
5.7.1. Funciones auxiliares 54
5.7.2. El handler 54
5.7.3. Inicialización 55
5.8. Construcción y despliegue 55
5.8.1. Construcción 55
5.8.2. Despliegue 56

6. Resultados **59**
6.1. Código de la aplicación 59
6.2. Historias de usuario 60
6.2.1. Usuario Backoffice 60
6.2.2. Cliente 72
6.3. Costes de desarrollo 76

7. Conclusiones **77**
7.1. Objetivos alcanzados 77
7.2. Trabajo futuro y mejoras propuestas 79

A. Apéndice A - Historias de Usuario **83**
A.1. Historias de usuario 83

B. Apéndice B - Diagramas **89**
B.1. Flujo Login JWT 90
B.2. Flujo Login Frontend 91
B.3. Flujo Backoffice 92
B.4. Flujo Google Calendar 93
B.5. Flujo OAuth2 94
B.6. Flujo SpringAI 95

C. Apéndice C - Técnicas de Ingeniería de Prompts **97**
C.1. Técnicas de Ingeniería de Prompts 97

D. Apéndice D - Herramientas MCP **99**

References **101**

Índice de tablas

3.1. Comparativa de la solución propuesta frente a las principales alternativas comerciales 10

4.1. Recursos Hardware 27
4.2. Recursos Software - Backend, BBDD y Build 27
4.3. Software - Frontend 28
4.4. Otros Recursos Software 28

6.1. Costes de desarrollo 76

A.1. HU01 – Login Usuario Backoffice 83
A.2. HU02 – Logout Usuario Backoffice 83
A.3. HU03 – Gestionar empresas 84
A.4. HU04 – Gestionar usuarios 84
A.5. HU05 – Visualizar trabajadores 84
A.6. HU06 – Gestionar servicios 85
A.7. HU07 – Gestionar horario 85
A.8. HU08 – Gestionar días inhábiles 85
A.9. HU09 – Visualizar calendario empresa 85
A.10. HU10 – Visualizar calendario trabajador 86
A.11. HU11 – Gestionar periodos inhábiles 86
A.12. HU12 – Autenticar cuenta de Google 86
A.13. HU13 – Consultar servicios empresa 86
A.14. HU14 – Consultar horario empresa 87
A.15. HU15 – Agendar cita 87
A.16. HU16 – Cancelar cita 87
A.17. HU17 – Modificar cita 87
A.18. HU18 – Agregar cita a calendario personal 88
A.19. HU19 – Recibir recordatorio de cita agendada 88

D.1. Herramientas (*tools*) expuestas por McpToolService al modelo 100

Índice de figuras

3.1. Arquitectura hexagonal 17

4.1. Estimación del plan de trabajo para el desarrollo 26

5.1. Visión general 31
5.2. Esquema de Arquitectura Hexagonal en el proyecto 33
5.3. Modelo de datos de Company 35
5.4. Modelo de datos de Users 35
5.5. Modelo de datos de Appointment 36

6.1. Pantalla Login Usuario 60
6.2. Pantalla Menú Principal ADMIN 61
6.3. Pantalla Empresas 62
6.4. Pantalla Formulario empresas 62
6.5. Pantalla Usuarios 63
6.6. Pantalla Formulario Usuarios 63
6.7. Pantalla Menú Principal MANAGER 64
6.8. Pantalla Trabajadores 64
6.9. Pantalla Servicios 65
6.10. Pantalla Formulario Servicios 65
6.11. Pantalla Horarios 66
6.12. Pantalla Formulario Horarios 66
6.13. Pantalla Días Inhábiles 67
6.14. Pantalla Formulario Días Inhábiles 67
6.15. Pantalla Calendario vista mes MANAGER 68
6.16. Pantalla Calendario vista semana MANAGER 68
6.17. Pantalla Calendario vista día MANAGER 69
6.18. Pantalla Menú Principal WORKER 69
6.19. Pantalla Calendario vista mes WORKER 70

6.20. Pantalla Calendario vista semana WORKER 70
6.21. Pantalla Calendario vista día WORKER 71
6.22. Pantalla Periodo Inhábil 71
6.23. Pantalla Formulario Periodo Inhábil 72

B.1. Flujo login JWT 90
B.2. Flujo login frontend 91
B.3. Flujo backoffice 92
B.4. Flujo Google Calendar 93
B.5. Flujo OAuth2 94
B.6. Flujo chat 95

Lista de acrónimos

AES	Advanced Encryption Standard
API	Application Programming Interface
ACID	Atomicity, Consistency, Isolation, Durability
B2B	Business to Business
BBDD	Base de Datos
BPE	Byte Pair Encoding
CI	Integración Continua
CLEAR	Concise, Logical, Explicit, Adaptive, Reflective
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
DIP	Dependency Inversion Principle
DOM	Document Object Model
DTO	Data Transfer Object
E2E	End-to-End
ES	Entrada/Salida
GPT	Generative Pre-trained Transformer
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HU	Historia de Usuario
IA	Inteligencia Artificial
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JPA	Jakarta Persistence API
JSON	JavaScript Object Notation
JSX	JavaScript Syntax Extension
JVM	Java Virtual Machine
JWT	JSON Web Token

KISS	Keep It Simple, Stupid
LLM	Large Language Model
LSP	Language Server Protocol
MCP	Model Context Protocol
MVC	Model-View-Controller
OAuth	Open Authorization
ORM	Object-Relational Mapping
PYME	Pequeña y Mediana Empresa
RAG	Retrieval-Augmented Generation
RASCEF	Role, Action, Steps, Context, Examples, Format
REST	Representational State Transfer
RSA	Rivest–Shamir–Adleman
RUP	Rational Unified Process
SAL	Salt
SDK	Software Development Kit
SGBD	Sistema Gestor de Bases de Datos
SOLID	Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion
SPA	Single-Page Application
SQL	Structured Query Language
TFG	Trabajo Fin de Grado
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience
XSS	Cross-Site Scripting
YAGNI	You Aren't Gonna Need It

Capítulo 1

Introducción

En este capítulo se incluye el contexto general y la problemática sobre la que nace la idea del proyecto, así como la motivación y visión general para la solución planteada.

1.1 Visión general

Hoy en día vivimos en un mundo globalizado y, cada vez más, los modelos de negocio en auge de las grandes compañías se anteponen a los pequeños negocios que tiempo atrás gozaban de mayor vitalidad y que, a día de hoy, cada vez les cuesta más subsistir. Las grandes tiendas online, grandes almacenes o franquicias pasan por encima de los comercios o restaurantes de barrio que, en muchas ocasiones, se heredaban de generación en generación dando un toque de personalidad a nuestras calles y barrios, ofreciendo un servicio mucho más cercano y familiar en el que los clientes gozaban de total confianza con el dependiente y eran asesorados con la mejor de las atenciones. Estas características, que dotaban de ese ambiente especial a estos comercios, se han diluido en la vorágine del día a día y la rutina en la que todos estamos sumidos, donde cada segundo cuenta y es *más fácil* ir a lo seguro, comprar online y que te lo lleven a casa sin tener que perder tiempo en desplazamientos o ir a las grandes superficies a comprar algo que tienes claro porque te lo han recomendado o porque te han avasallado a spam.

Quizás, una de las grandes brechas que han sufrido en los últimos tiempos algunos de estos negocios es la de no haber podido adaptarse a las nuevas tecnologías por falta de recursos y no han sido capaces de subirse a un tren en marcha que viaja a una velocidad de vértigo. Aunque, dentro de todo este panorama, hay un tipo de negocio que parece resistir como lo hacía una pequeña aldea de comic poblada de irreductibles Galos contra el imperio Romano, y no solo eso, sino que da la impresión de que está en un periodo de auge y proliferación. Los negocios a los que se hace referencia son las peluquerías y barberías. Es cierto que es un modelo de negocio muy difícil de sustituir por la tecnología y actualmente no parece cercano el día en que te pueda cortar el pelo, peinarte o afeitarte un robot o máquina con la seguridad y confianza que trasmite una persona, no porque sea algo imposible a nivel tecnológico pero tal vez el retorno que podría reportar no es suficiente para compensar la inversión que conllevaría, demos un poco más de tiempo a ese tren del que hemos hablado

anteriormente.

Asumiendo que en el corto plazo esto no va a ser posible, el objetivo principal de este Trabajo Fin de Grado (TFG) es aportar un granito de arena en la modernización de este tipo de negocios, ofreciendo un sistema totalmente automático de solicitud y reserva de citas, reduciendo al mínimo la intervención humana para que tanto clientes como trabajadores puedan dedicar su valioso tiempo a las actividades que lo requieren de una forma más necesaria.

1.2 Problemática

Como se ha comentado en el punto anterior, nos encontramos con un tipo de negocio en el que, por su propia naturaleza, es difícil incluir servicios tecnológicos que realmente generen un valor añadido para el comercio y los clientes.

Uno de los puntos más débiles de las peluquerías/barberías viene a la hora de la reserva de citas ya que en la mayoría actualmente se agendan de forma manual utilizando diferentes canales, de forma presencial en el propio establecimiento, por teléfono, Whatsapp y en muchos casos anotándolo todo en agendas de papel a mano. Esta forma de trabajo se encuentra con algunos inconvenientes como los siguientes:

- Fragmentación entre canales de interacción con el cliente.
- Centralizar en un único registro todas las solicitudes/reservas recibidas por los diferentes canales.
- Falta de una visión unificada de la ocupación de negocio a la hora de gestionar varios trabajadores.
- Indisponibilidad de atención al cliente 24/7.

La solución más inmediata que se ofrece actualmente es la de contratar un servicio de terceros para la gestión de reservas en las que el cliente puede seleccionar un hueco disponible y reservarlo, como ofrecen prácticamente todos los restaurantes que actualmente ofrecen servicio. Pero esta solución aplicada a un pequeño negocio puede traer otros inconvenientes:

- Alto coste para cuando se contratan soluciones de terceros que un pequeño negocio no se puede permitir con modelos de pago como las comisiones por reserva o suscripciones mensuales.
- En este tipo de plataformas de terceros es mucho más complicado personalizar la experiencia conversacional con el cliente y la experiencia de usuario.

Antes de explicar con más detalle la solución que se propone en este TFG, se enumeran brevemente algunas de las alternativas que se ofrecen a día de hoy y que podrían ayudar a

mejorar el día a día de un negocio como el que se está tratando en este documento. Más tarde profundizaremos un poco más en estas soluciones, sus ventajas e inconvenientes, pero por el momento, el objetivo es únicamente crear una vista rápida e intentar sacar una conclusión de forma más general.

Estableciendo el foco en el sector de belleza o peluquería y estética, encontramos tres alternativas que pueden ayudar bastante a agilizar y automatizar la gestión de estos negocios en su operativa diaria, ayudando tanto a gerentes y empleados como a los propios clientes: **Fresha**, **Booksy** y **Calendly**.

Estas tres opciones están enfocadas a los negocios del sector peluquería y estética y, aunque podrían ser muy buenas candidatas para un gerente de este tipo de comercios, se observan algunas desventajas comunes presentes en todas ellas como elevados costes de cuotas o comisiones que pueden no ser sostenibles para pequeños negocios o la necesidad de interacción por parte de los propios gestores del negocio como de los clientes (configuraciones más complejas, flujos de interacción más estrictos). Además, desde un punto de vista puro de gerencia del negocio y marketing, estas aplicaciones dan soporte a muchas otras empresas, no es una aplicación *en propiedad* del negocio por lo que un cliente puede ver diferentes ofertas del mismo tipo de servicios en la competencia y que conlleve un gasto extra el poder aparecer en los primeros puestos en el resultado de una búsqueda de cliente.

Otras alternativas para la gestión de citas podrían ser: **Calendly**, **Acuity Scheduling**, **SimplyBook.me** y **Reservio**.

Estas opciones son buenas en el ámbito de la gestión de citas, pero no están pensadas exclusivamente para negocios de belleza, ni tan siquiera para negocios en los que se requiera una presencialidad física por parte del cliente, hacen muy buen trabajo con la gestión de citas pero no ofrecen los valores añadidos de la especificación del sector como la gestión de los servicios, trabajadores. En adición a esto, todas presentan los inconvenientes que se citaron para las soluciones específicas del sector en cuanto a costes, complejidad y falta de personalización.

Atendiendo a la problemática descrita y con intención de solventar los inconvenientes mencionados, surge la idea de diseñar un sistema que pueda dar una solución adaptada al pequeño negocio de peluquerías y barberías que sea eficiente y funcional, ayudando a mejorar la gestión del día a día sin perder la esencia de cercanía y familiaridad de tu establecimiento de confianza.

1.3 Propuesta de solución

Sin duda alguna, el principal tema tecnológico que nos viene a la mente al escuchar esa palabra y que está presente en todos los debates a día de hoy son los modelos de Inteligencia Artificial junto con todo el contexto funcional y técnico en el que operan. Son el principal

avance tecnológico de los últimos años y uno de los más complejos de comprender para la gente que no está muy familiarizada con el mundo de la informática en particular. Estos modelos ofrecen resultados tanto en calidad como en tiempos de procesamiento inimaginables e inabordables hace tan solo unos años atrás. Todo esto, en la fugacidad y la necesidad por la brevedad en que vivimos estos días, lo resumimos con dos letras que están presentes en la mente de todos, grandes y pequeños y que han llegado para quedarse de una forma apabullante, la *IA (Inteligencia Artificial)*.

Cualquier solución tecnológica que se empiece a diseñar en esta época obligatoriamente tiene que tener en cuenta la IA ya que abre un abanico completamente nuevo e infinito de posibilidades en las que se puede apoyar un nuevo diseño optimizando cualquier parámetro clave a tener en cuenta (tiempos, costes, complejidad) a la hora de llevarlo a cabo.

La irrupción de los modelos de lenguaje a gran escala (LLMs) durante los años 2022–2025 ha modificado de manera estructural el modo en que las PYMEs del sector servicios conciben la atención al cliente [Eur25]. Mientras la primera generación de plataformas de booking online se centró en sustituir la llamada telefónica por un formulario web, la generación actual debe responder a un cliente que espera interactuar mediante lenguaje natural en el mismo canal que utiliza para comunicarse con sus contactos personales y a cualquier hora del día.

Como cabía esperar por los párrafos anteriores, en la solución propuesta la Inteligencia Artificial va a tener un papel clave, siendo uno de los componentes principales que utilizaremos para dar forma al resultado final. En concreto, la responsabilidad que delegaremos en la IA será toda la parte relacionada con la interacción con el usuario, una parte que actualmente todos los modelos tienen bastante entrenada y en la que son brillantes, además de que es la forma en la que todo el mundo está habituado a interactuar con los diferentes modelos que existen hoy en día.

La solución se basa en un sistema de gestión para estos negocios en el que se podrá registrar toda la información relativa al negocio, como servicios, horarios, trabajadores, días festivos, citas o clientes. Hasta aquí nada novedoso, pero ¿Cómo podríamos ayudarnos de la *magia* de la IA para que esta solución alcanzase el valor añadido del que hablábamos antes y de paso nos aliviara costes en el desarrollo?

Ahora mismo la característica principal y más entrenada que tienen cualquiera a de los modelos de IA que hoy se ofertan, es la de interpretar el lenguaje natural y ofrecer al usuario respuestas cómo si realmente fuese una persona humana el que está al otro lado de la pantalla. Si incorporamos en la solución esta funcionalidad que nos ofrece un modelo de IA, podemos conseguir delegar la compleja tarea de interpretación de los mensajes y necesidades del cliente y que el sistema de gestión diseñado dote al modelo de una serie de herramientas que le ayuden a resolver estas necesidades de la mejor forma posible, guiando al cliente durante todo el proceso de reserva y ofreciéndole toda la información que pudiera requerir para

conseguir su misión.

De esta forma el sistema, además de servir para las gestiones internas que realizarán los trabajadores y gestores del negocio, actuará de servidor Model Context Protocol (MCP) para el modelo, ampliando su rango de acción y nutriéndolo de información real del negocio en la que el modelo se basará a la hora de tomar decisiones y enviar una respuesta o propuesta al cliente. Adicionalmente a esto, como todos ya deberíamos saber, hay una implicación directa entre el detalle y la elaboración de las instrucciones que se le da a un modelo con el resultado que se recibe de él, estas instrucciones se conocen comúnmente como *prompt*, será una parte muy importante del sistema elaborar un *prompt* bien definido para que se ajuste bien a las necesidades del cliente, interprete correctamente la información del sistema, se le pongan restricciones y ayude a acotar las posibles ramificaciones que pueda generar el modelo y que se salgan del contexto del negocio, ya que los modelos son muy genéricos y no son algo diseñado específicamente para gestionar negocios de peluquería y estética.

Hasta ahora, la solución tiene un modelo de IA con el que el cliente puede hablar para que le ayude a conocer los horarios, servicios o a reservar una cita en su establecimiento de confianza, pero se necesita abstraer un poco más al usuario de todo lo que hay detrás. Para ello se le ofrece la posibilidad de interactuar con el modelo a través de alguna de las aplicaciones de mensajería instantánea que utiliza habitualmente para comunicarse con sus conocidos, dándole un toque más *familiar* a la solución. En la propuesta, se incluye la implementación de un bot de *Telegram* que haga las veces de interfaz de usuario para el cliente con el que podrá chatear cómo si lo hiciese directamente con la persona que posteriormente le cortará el pelo, ahorrándole la necesidad de operar con una interfaz convencional con varios clicks, formularios o un proceso de registro en el sistema tedioso y del que muchas personas huyen hoy en día.

Otra parte muy importante de la solución es el sistema de gestión de negocio (a partir de ahora *backend*). Este sistema se diseña siguiendo el estándar de arquitectura hexagonal, debe dar soporte a múltiples empresas y ser capaz de gestionar la información de cada una de forma separada. Expone una API-REST securizada que utilizan los gestores y trabajadores de negocios para poder gestionar la información del negocio en sí, como los servicios que se ofrecen con su duración y precio, horarios, días festivos, vacaciones y consultar el calendario de trabajo de cada trabajador.

Cuenta con una Base de Datos (BBDD) relacional bien estructurada que garantiza la correcta persistencia de toda la información que se va a registrar en todos los posibles flujos y casos de uso que ofrecerá el sistema. También es el encargado de orquestar el flujo entre el bot de *Telegram* que interactúa con el usuario y el modelo de IA que toma las decisiones y genera las respuestas en lenguaje natural para el cliente. Asimismo, incluye un conjunto de herramientas para que el modelo pueda consultar la información necesaria en su interacción con el cliente y hacer las gestiones oportunas con respecto a las citas, tales como agendar,

modificar o cancelar a modo de servidor MCP.

Para gestionar todo lo relacionado con citas, calendarios y notificaciones a cliente, en lugar de crear una solución de cero para este cometido, se utilizará otro de los estándares más extendidos en este ámbito, *Google Calendar*. Esta integración dotará al sistema de un sistema de eventos, calendarios y notificaciones completamente testeado y asentado del que se ayudará en algunas acciones que pudieran resultar más complejas y que además conseguirá que el cliente pueda integrar sus citas en su calendario personal dando, una vez más, al cliente otra dosis de *familiaridad* y confianza sin tener que registrarse, instalar o aprender a manejar otras aplicaciones para un único fin e integrándolo en su contexto tecnológico particular.

Por último, se deberá añadir una forma sencilla para los trabajadores y gestores del negocio de incluir en el sistema toda la información que el negocio requiera para su correcto funcionamiento. La solución contará con una aplicación web que actuará de *backoffice*, haciendo de interfaz de usuario para el API-REST que expone el sistema de *backend* para la gestión de información de negocio y visualización de calendarios de trabajo. Al ser una aplicación web, se podrá utilizar desde cualquier navegador y no habrá restricciones de plataforma consiguiendo flexibilidad para los gestores y trabajadores, además la interfaz deberá ser sencilla de manejar, amigable e intuitiva e incluirá un sistema de autenticación y roles para que se puedan restringir las acciones que cada tipo de usuario podrá realizar.

Concluyendo con todo lo anterior, la solución propuesta ofrecerá estas ventajas principales tanto para el negocio como para los clientes:

- Reducción de llamadas telefónicas o consultas presenciales, formularios, registros, carga administrativa para el negocio y, en definitiva, reducción de burocracia y tiempos en la interacción humana para el funcionamiento del día a día.
- Mejora de la experiencia del cliente con disponibilidad 24/7 y con un ecosistema tecnológico completamente establecido y con el cliente está muy familiarizado.
- Ahorro de costes económicos: ahorro de comisiones a marketplaces, suscripciones o apps de terceros.

Capítulo 2

Objetivos

En este capítulo se define el objetivo principal del TFG y una lista de objetivos específicos, tanto funcionales como no funcionales, que estarán presentes en el resultado final.

2.1 Objetivo principal

El objetivo principal de este TFG es automatizar la gestión de citas de pequeños comercios de peluquería/barbería, reduciendo al mínimo la intervención humana en el día a día del negocio. Se busca que los clientes puedan reservar, consultar o cancelar sus citas mediante una conversación en lenguaje natural a través de un canal de mensajería de uso cotidiano, y que los gerentes y trabajadores dispongan de una herramienta centralizada para administrar su negocio (servicios, horarios, personal y agenda) sin tareas manuales repetitivas.

2.2 Objetivos específicos

Basándonos en la definición del objetivo general, se listan ahora una serie de objetivos específicos para conformar la solución final recogiendo aspectos funcionales, no funcionales, metodológicos y técnicos que se seguirán para conseguir el resultado esperado.

1. **Disponer de un núcleo de servicio que centralice y orqueste toda la lógica del sistema:** debe actuar como punto único que coordine la gestión de negocio, la agenda, la comunicación con el modelo conversacional y la seguridad, de modo que el resto de componentes dependan de un único responsable de la lógica.
2. **Permitir que gerentes y trabajadores gestionen de forma autónoma toda la información del negocio:** la aplicación debe ser amigable e intuitiva y debe incluir un sistema de autenticación y roles diferenciados para poder restringir el alcance de un usuario y ofrecer diferentes opciones según el rol que corresponde. Las acciones a las que debe dar soporte son:
    - Gestión de Trabajadores de la empresa.
    - Gestión de Servicios ofrecidos por la empresa.
    - Definir horario de apertura y días inhábiles del establecimiento.
    - Definir periodos inhábiles para trabajadores.

- Visualización de eventos agendados para los trabajadores de una empresa.

3. **Ofrecer al usuario un canal de comunicación cómodo y familiar:** este módulo es el único punto de interacción de un cliente con el sistema y se le permite hacerlo en lenguaje natural, sobre una plataforma de mensajería de amplia adopción, de modo que reservar o consultar una cita sea tan sencillo como enviar un mensaje.
4. **Garantizar la persistencia íntegra, consistente y segura de toda la información del sistema:** evitando redundancias y asegurando la precisión de los datos. La información a conservar incluye:
   - Información de Negocio (Servicios, horarios, trabajadores, días festivos...)
   - Información de Clientes (Contacto, Citas, Conversaciones...)
   - Información de *Google* (Cuentas de *Google*, credenciales, *tokens*...)
   - Información de Usuarios y Roles para la aplicación web.
5. **Conseguir que el asistente conversacional genere respuestas naturales, fiables y seguras:** debe responder de forma orgánica y eficiente, mantener el tono adecuado, no revelar información privada o de sistema, no inventar datos, pedir confirmación ante acciones irreversibles, ser proactivo sugiriendo huecos disponibles y adaptarse dinámicamente al contexto de cada empresa.
6. **La solución final debe poder dar soporte a varias empresas:** desde un único sistema se pueden gestionar varias empresas de manera independiente, sin visibilidad entre ellas. El sistema debe ser capaz de gestionar diferentes cuentas de empresa, así como sus credenciales e información de negocio, clientes, eventos...
7. **Aplicar un ciclo de desarrollo basado en metodología ágil:** que sostenga un desarrollo iterativo-incremental con hitos verificables y que permita la validación de cada módulo por separado.
8. **A lo largo de todo el desarrollo se deben aplicar buenas prácticas de diseño:** implementación, Integración Continua (CI), así como el uso de patrones de diseño y principios de diseño software como SOLID, KISS o YAGNI, que permitan que el sistema resultante sea un sistema robusto, eficiente y escalable.

Capítulo 3

Estado del arte

En el presente capítulo vamos a recopilar y detallar el estado actual de una serie de plataformas comerciales alternativas que están actualmente en el mercado, las cuales comentamos brevemente en la introducción de este documento, y que abordan, total o parcialmente, el mismo problema que la solución propuesta. Adicionalmente, se incluirá una revisión del estado actual de las diferentes herramientas o conceptos tecnológicos que son susceptibles de componer la solución final. El objetivo del capítulo, por tanto, es doble: por un lado, describir el estado tecnológico actual para la implementación de la solución y por otro lado, ubicar la propuesta de este TFG dentro del panorama actual de plataformas/aplicaciones de reserva de citas, identificando y resaltando la aportación diferencial que ofrece la solución propuesta.

3.1 Sistemas integrales de gestión de citas en el sector peluquería

A continuación, se analizan tres plataformas comerciales que compiten en el mismo espacio, de una forma u otra, para dar solución al mismo problema que el que se aborda en este TFG.

Fresha

Esta plataforma está orientada de forma muy específica al sector de la belleza y bienestar: peluquerías, barberías, centros de estética, etc. [Fre26] Su modelo de negocio es freemium: el Marketplace y el software de gestión son gratuitos pero la monetización se realiza vía comisiones por reservas hechas a través de su marketplace y por el procesamiento de pagos. Destaca por ofrecer soporte para calendario multi-profesional con asignación de servicios y franjas. Además, dispone de herramientas de marketing automatizado tales como recordatorios, campañas o programas de fidelización.

Booksy

Otra de las plataformas orientadas al sector belleza/estética más extendidas a nivel global para reserva de citas [Boo26]. Combina un sistema de gestión interna para el negocio con una app independiente para el cliente.

Su modelo de negocio es de suscripción mensual por profesional dado de alta en la plataforma. Ofrece la posibilidad de reservas online con pago previo junto con un programa de reseñas verificadas y fidelización de clientes.

Calendly

Es uno de los softwares líderes a nivel mundial en la automatización de agendas y reuniones para profesionales o corporaciones [Cal26]. Está completamente orientada al ámbito profesional y Business to Business (B2B). Su modelo de negocio se basa en una suscripción por usuario y con un elevado nivel de personalización y configuración. Uno de sus puntos más fuertes es la integración nativa con servicios externos de calendario como *Google Calendar, Microsoft 365 y Apple iCloud* y servicios de videoconferencia como *Zoom, Google Meet o Microsoft Teams.* Funcionalidades Principales:

Comparativa

En la tabla 3.1 se puede ver una comparativa de las principales características entre la solución propuesta y las alternativas descritas.

Comparativa de la solución propuesta frente a alternativas comerciales				
Característica / Plataforma	**Solución TFG**	**Fresha**	**Booksy**	**Calendly**
Asistente conversacional con IA	✓	✗	✗	✗
Interacción en lenguaje natural	✓	✗	✗	✗
Sin app ni registro para el cliente	✓	✗	✗	✗
Modelo de negocio	Gratuito	Freemium + comisión	Suscripción	Suscripción
Auto-alojado / control de datos	✓	✗	✗	✗
Sincronización con *Google Calendar*	✓	✓	✓	✓
Pagos integrados	●	✓	✓	✓
Marketplace de captación	✗	✓	✓	✗
✓ Disponible ● Planificado (*roadmap*) ✗ No disponible				

Tabla 3.1: Comparativa de la solución propuesta frente a las principales alternativas comerciales

Como se puede deducir tras el repaso al estado actual del mercado, existen varias opciones que podrían cubrir la necesidad de un establecimiento para gestionar las reservas de su día a día. De hecho, se trata de plataformas maduras y consolidadas que ofrecen funcionalidades de las que la solución propuesta aún carece, como los pagos integrados (contemplados como línea de trabajo futuro) o un *marketplace* propio de captación de clientes. Sin embargo, ninguna de ellas termina de culminar el objetivo principal que se marcó al comenzar este TFG: ayudar a los negocios del sector peluquería/estética a automatizar su gestión de una forma sencilla, económica y reduciendo al máximo la interacción humana. La propuesta se diferencia precisamente en aquello que constituye su razón de ser, pues es la única que ofrece una interfaz basada en un agente conversacional con Inteligencia Artificial (IA) que interactúa en lenguaje natural (sin conversaciones de bot con mensajes predefinidos ni flujos de respuestas guiados), integrada directamente en una aplicación de mensajería que el cliente ya utiliza a diario, sin necesidad de instalar aplicaciones ni registrarse y sin coste ni comisiones para el negocio. Se consigue así una opción mínimamente intrusiva, integrada en su día a día de forma orgánica y con disponibilidad las 24 horas los 365 días del año, en la que el asistente responde como si se tratase del propio gestor del negocio.

3.2 Tecnologías de desarrollo web

En este apartado se hace un repaso detallado de las principales tecnologías de desarrollo web que existen actualmente.

En cada uno de los apartados se procurará dar una breve descripción técnica breve de la tecnología en cuestión, su rol y principales responsabilidades dentro del sistema y algunas de las principales alternativas existentes que se podrían haber utilizado.

3.2.1 *Backend*: Java y Spring Boot

El servicio *backend* es el núcleo del sistema y el componente con más responsabilidades, encargado de la orquestación de los flujos entre el resto de componentes. A continuación se repasan las principales tecnologías sobre las que puede construirse un servicio de este tipo.

Java

Es un lenguaje de programación orientado a objetos, fuertemente tipado y multiplataforma, actualmente propiedad de Oracle [Ora23]. Aunque ha dominado el mundo del desarrollo software durante décadas, el panorama actual muestra que otros lenguajes lo han superado en cuanto al volumen de uso. Cuenta más de 25 años de evolución y ha sido utilizado y probado en innumerables infraestructuras críticas. Gracias a su madurez, estabilidad y a la gran comunidad que tiene detrás, proporciona una experiencia de desarrollo muy productiva, con gran número de herramientas y soporte para realizar prácticamente cualquier tarea.

Spring Boot

Es un framework de *Java* construido sobre el ecosistema *Spring*, que se ha consolidado como el framework de referencia para aplicaciones empresariales basadas en API-REST con *Java* [Spr24b, Spr24d]. Sus puntos fuertes son su modelo de auto-configuración y la disponibilidad de starters específicos, entre los que podríamos destacar *spring-boot-starter-web* (para API-REST), *spring-boot-starter-data-jpa* (para gestión de la persistencia) o *spring-boot-starter-security* (para aspectos de seguridad). Los *starters* son conjuntos de dependencias agrupadas por funcionalidad que posibilitan tener en unos segundos un conjunto muy completo de diferentes herramientas software listas para desarrollar y evitando conflictos entre versiones que permiten reducir notablemente el tiempo de configuración inicial frente a otras alternativas. Es un ecosistema sin parangón en el mundo Java Virtual Machine (JVM), robusto, fiable y con una comunidad de desarrollo inmensa.

Alternativas

Quarkus / Micronaut

Son frameworks de *Java/Kotlin* más modernos y orientados a entornos cloud-native y serverless. Ambos resuelven la inyección de dependencias y la configuración en tiempo de compilación (en lugar de en tiempo de ejecución como *Spring*), lo que reduce el consumo de memoria y permite tiempos de arranque más cortos. Podrían haber sido otra buena alternativa para el desarrollo de la solución, sin embargo, su ecosistema es significativamente menor que el de *Spring*, con menos librerías de terceros, menos integraciones de primera clase y una comunidad más reducida. Además, una de las piezas claves de la solución, cómo veremos un poco más adelante, es *Spring AI* y la integración con este componente no es nativa y requeriría adaptaciones manuales.

Node.js / NestJS

Node.js es un runtime de *JavaScript* basado en un modelo de Entrada/Salida (ES) no bloqueante, muy eficiente para aplicaciones cuyo rendimiento dependen de la velocidad con la que interactúa con otros dispositivos/sistemas externos. NestJS es un framework basado en Node.js y claramente inspirado en *Angular y Spring*, que aporta una arquitectura modular, inyección de dependencias y soporte de primera clase para TypeScript. No obstante, aunque *TypeScript* añade tipado estático en tiempo de compilación, este se elimina al transpilar a *JavaScript*, por lo que en runtime el tipado sigue siendo dinámico lo que dificulta la detección de errores hasta el tiempo de ejecución, mientras que el sistema de tipos de *Java* facilita su detección en tiempo de compilación. Adicionalmente, la gestión general (a nivel tecnológico) del dominio de la solución (transacciones, integración con LLMs, sincronizaciones con *Google Calendar*...) se beneficia notablemente del modelo de starters/dependencias de *Spring*, mientras que en *Node.j*s requiere bibliotecas externas con un soporte menos maduro.

3.2.2 Persistencia: PostgreSQL, JPA/Hibernate y Flyway

La capa de persistencia es la responsable de almacenar y recuperar de forma fiable toda la información que maneja el sistema. A continuación se describen las principales tecnologías que la componen —el SGBD, la capa de acceso a datos y la herramienta de migraciones— así como las alternativas relacionales y no relacionales más habituales.

PostgreSQL

PostgreSQL es el sistema de gestión de BBDDs relacional más completo del ecosistema de código abierto. Desde el punto de vista funcional, ofrece todas las alternativas presentes en un SGBD relacional convencional: almacenar, consultar y manipular datos estructurados garantizando la integridad referencial mediante claves primarias y foráneas, restricciones, vistas, triggers, procedimientos almacenados... Soporta además tipos de datos avanzados como pueden ser JavaScript Object Notation (JSON)/JSONB o ARRAY. Todo esto y otras muchas características lo posiciona como una BBDD robusta, escalable y madura, adecuada para entornos críticos y proyectos con requisitos exigentes [The24b].

Spring Data JPA

Spring Data JPA es un módulo del ecosistema *Spring Data* que actúa como capa de abstracción sobre Jakarta Persistence API (JPA), simplificando drásticamente la implementación de la capa de acceso a datos en aplicaciones *Java*. Desde una perspectiva funcional, permite definir repositorios como interfaces que extienden de jerarquías estándar (*CrudRepository, JpaRepository*), de modo que *Spring* genera automáticamente la implementación en tiempo de ejecución, sin necesidad de escribir código repetitivo para las operaciones Create, Read, Update, Delete (CRUD) habituales. Adicionalmente, integra un sistema que facilita algunas tareas como la generación automática de consultas o la paginación y ordenación de los datos [Spr24c].

Hibernate

Es el framework de mapeo objeto-relacional (ORM) más extendido del ecosistema *Java*. Es una implementación completa del estándar JPA y el proveedor que *Spring Boot* utiliza por defecto [Red24a]. Desde el punto de vista funcional, permite mapear clases *Java* (entidades) a tablas relacionales mediante anotaciones gestionando de forma automática la conversión entre objetos y filas de la BBDD. Soporta relaciones , estrategias de herencia, validaciones, cascading de operaciones y carga de asociaciones.

Flyway

Flyway es una herramienta open source de gestión de migraciones de BBDDs basada en el principio de versionado del esquema. Su misión es resolver uno de los problemas más recurrentes del desarrollo *backend*: mantener el esquema de la BBDD sincronizado, consistente

y reproducible entre todos los entornos y a lo largo del tiempo. Desde el punto de vista funcional, *Flyway* gestiona las migraciones como scripts SQL planos almacenados en el propio repositorio del proyecto, siguiendo una convención de nomenclatura estricta. Cada script se corresponde con una versión y se ejecuta una única vez, en orden estricto de versión, lo que permite reconstruir el esquema completo desde cero o evolucionarlo de forma incremental de manera totalmente determinista [Red24b].

Algunas alternativas que existen para la implementación de la persistencia son:

MySQL / MariaDB

MySQL y *MariaDB* son dos de los SGBD relacionales de código abierto más utilizados del mercado, especialmente extendidos en el ámbito del desarrollo web. Se caracterizan por su facilidad de gestión, su buen rendimiento y un ecosistema muy maduro de herramientas y soporte, lo que los convierte en opciones ampliamente probadas para aplicaciones de propósito general con un modelo de datos relacional.

Oracle / SQL Server

Otras dos opciones bastante fiables y muy profesionales como SGBD relacional, pero muy orientadas al mundo empresarial a gran escala y con un alto coste de licencia, inviables para un proyecto como el que se propone en este TFG en su versión inicial.

MongoDB

El líder indiscutible en BBDDs no relacionales. Aportaría mucha flexibilidad al esquema y un escalado horizontal sencillo, pero el modelo documental que emplea no termina de encajar con algunos tipos de relaciones con las que se abordan en la solución, el modelo del dominio es fuertemente relacional.

3.2.3 Seguridad: Spring Security y JWT

La seguridad es un aspecto transversal en toda aplicación *backend* expuesta a través de una API-REST. En este apartado se repasan las principales tecnologías y estándares empleados para resolver la autenticación, la autorización y la protección de las comunicaciones, en particular *Spring Security* y el estándar JWT.

Spring Security

Es el Framework de seguridad que se incluye dentro del ecosistema Spring, considerado el estándar de facto para implementar autenticación, autorización y protección frente a ataques comunes en aplicaciones Java [Spr24e]. Al igual que *Spring*, es altamente configurable, extensible y modular. Se integra de forma nativa con *Spring Boot* mediante auto-configuración, lo que permite incorporar seguridad robusta a la aplicación de una forma muy orgánica y sin reinventar o desarrollar nuevos componentes.

Su arquitectura se basa en una **cadena de filtros** que intercepta todas las peticiones HTTP antes de que lleguen a los controladores, aplicando de forma ordenada las reglas de seguridad definidas para cada uno.

Spring Security da solución a los problemas de seguridad más habituales en este tipo de aplicaciones. En primer lugar, se encarga de la **autenticación**, es decir, de verificar la identidad del usuario, habitualmente mediante la validación de credenciales o de *tokens* de sesión. En segundo lugar, gestiona la **autorización**, controlando a qué recursos puede acceder cada usuario en función de sus **roles**, configurables de forma declarativa o mediante anotaciones. Asimismo, incorpora **protección frente a ataques comunes**, con defensas integradas contra numerosos tipos de amenazas que pueden ajustarse según las necesidades de cada aplicación. Por último, ofrece mecanismos de **cifrado de contraseñas** a través de la abstracción *PasswordEncoder*, con implementaciones modernas como *BCryptPasswordEncoder*. Además de estas capacidades, *Spring Security* goza de las principales ventajas que aporta el uso de *Spring*: madurez, integración fluida con el resto del ecosistema, flexibilidad para adaptarse a esquemas de autenticación muy variados y un amplio respaldo comunitario.

JWT (*JSON Web Token*)

Es un estándar abierto definido en la RFC 7519 [JBS15] que describe un formato compacto y autocontenido para transmitir información de forma segura entre dos partes en forma de objeto JSON. Su uso más extendido es la autenticación y autorización en aplicaciones web modernas, especialmente en arquitecturas stateless basadas en APIs REST.

Un *token* JWT está compuesto por tres partes separadas por puntos (header.payload.signature), codificadas en Base64URL:

- Header: indica el tipo de *token* (JWT) y el algoritmo de firma utilizado (RS256 o HS256) [Jon15].
- Payload : contiene los *claims*, es decir, la información transmitida sobre el usuario y el propio *token*.
- Signature: garantiza la autenticidad e integridad del *token*.

Sus características principales son:

- **Autocontenido**: el propio *token* incluye toda la información necesaria para identificar al usuario, evitando consultas adicionales a la BBDD en cada petición.
- **Stateless**: el servidor no necesita almacenar sesiones; basta con verificar la firma del *token* entrante.
- **Verificable**: cualquier modificación del payload invalida la firma, impidiendo la manipulación.

- **Caducidad controlada**: el *claim* exp limita la validez temporal de la información, acotando el impacto en caso de una eventual fuga.

3.2.4 Arquitectura hexagonal

La *arquitectura hexagonal* (también conocida como *Ports and Adapters*) [Coc05] es un patrón arquitectónico cuyo objetivo principal es aislar la lógica de negocio (dominio) de los detalles tecnológicos externos, como la BBDD, frameworks web, las APIs externas o colas y sistemas de mensajería y eventos.

La idea central es que el núcleo de la aplicación (el dominio) no debe depender de ninguna tecnología concreta, sino que son las tecnologías externas las que deben adaptarse al dominio mediante interfaces bien definidas. Para lograrlo, se establecen dos elementos fundamentales:

- **Puertos (*ports*)**: interfaces que definen *qué necesita o qué ofrece* el dominio, sin especificar cómo. Se distinguen dos tipos:
  - **Puertos de entrada (*driving ports*)**: definen los casos de uso que el dominio expone al exterior (por ejemplo, "registrar cita" o "enviar mensaje").
  - **Puertos de salida (*driven ports*)**: definen lo que el dominio necesita del exterior (por ejemplo, "guardar usuario" o "crear evento en Google").
- **Adaptadores (*adapters*)**: implementaciones concretas de los puertos que conectan el dominio con tecnologías específicas (controladores REST, repositorios JPA, clientes HTTP, etc.).

Visualmente se representa como un hexágono con el dominio en el centro, los puertos en sus lados y los adaptadores rodeándolo, lo que enfatiza que todas las tecnologías son intercambiables sin afectar al núcleo.

La arquitectura hexagonal aporta, en primer lugar, **independencia tecnológica**, ya que el dominio no depende de frameworks o herramientas concretas como *Spring*, JPA o PostgreSQL, lo que facilita migrar o actualizar tecnologías sin reescribir la lógica de negocio. A esto se suma una mayor **mantenibilidad y claridad**, pues la separación estricta de responsabilidades hace que el código sea más legible, predecible y fácil de evolucionar, así como una notable **flexibilidad**, al permitir añadir nuevos adaptadores sin tocar el dominio. Además, este enfoque favorece el **cumplimiento de principios SOLID**, especialmente el Dependency Inversion Principle (DIP), según el cual el dominio define interfaces y la infraestructura las implementa, nunca al revés [Mar17]. Por último, todo ello se traduce en una **reducción del acoplamiento**, de modo que los cambios en la BBDD, el framework web o los servicios externos no se propagan al núcleo.

No obstante, esta arquitectura también presenta inconvenientes. El primero es una **mayor complejidad inicial**, ya que requiere más clases, paquetes e interfaces, lo que puede resultar

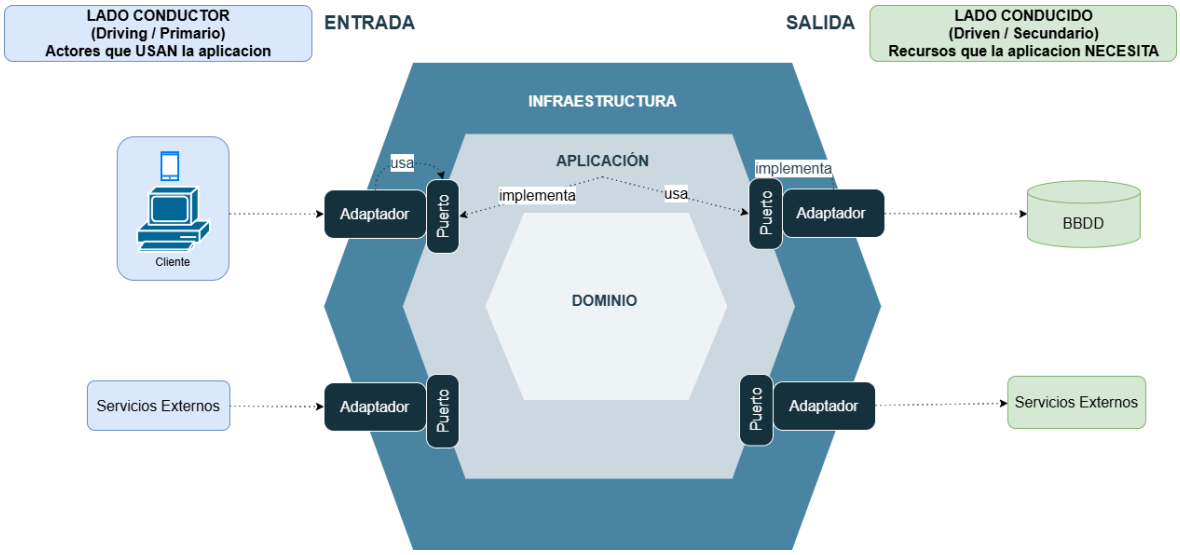


Figura 3.1: Arquitectura hexagonal

excesivo en proyectos muy pequeños. A ello se añade una **curva de aprendizaje** considerable, pues comprender la separación entre dominio, aplicación e infraestructura exige cierta madurez técnica por parte del equipo de desarrollo.

Alternativas arquitectónicas consideradas

Modelo-Vista-Controlador (MVC)

MVC es un patrón de capa de presentación cuya principal característica es que separa la interfaz de usuario en tres componentes: *Modelo* (datos y estado), *Vista* (representación visual) y *Controlador* (gestión de la entrada del usuario). Es el patrón que utiliza internamente *Spring MVC* para mapear peticiones HTTP a controladores y generar respuestas. No obstante, MVC opera únicamente en la capa de presentación y no define cómo organizar la lógica de negocio, la persistencia ni las integraciones externas, por lo que resulta insuficiente como arquitectura global para un proyecto con un dominio complejo como el que se propone.

Arquitectura por capas tradicional (Controller → Service → Repository)

La arquitectura por capas es el patrón más extendido en aplicaciones *Spring Boot*, organizando el código en capas horizontales apiladas (presentación, servicio, persistencia) donde las dependencias fluyen siempre hacia abajo. Es sencilla, intuitiva y suficiente para proyectos pequeños o medianos. Sin embargo, presenta un inconveniente estructural: el dominio queda acoplado al framework, ya que las entidades mezclan anotaciones de JPA, validaciones de *Spring* y reglas de negocio en una misma clase, y los servicios dependen directamente de interfaces del ORM (JpaRepository). Esto dificulta la sustitución de tecnologías, complica los tests unitarios y favorece que la lógica de negocio se disperse entre servicios y controladores sobrecargados, problemas que se agravan en proyectos con múltiples integraciones externas como el presente.

3.3 Agentes de IA y LLMs

Los LLMs han propiciado la aparición de una nueva generación de asistentes conversacionales capaces de interpretar el lenguaje natural y, combinados con mecanismos de integración como el protocolo MCP, de actuar como orquestadores de acciones sobre sistemas externos. En este apartado se revisan los conceptos y tecnologías fundamentales de este ámbito: qué es un LLM y cómo funciona, el protocolo MCP, el modelo GPT de *OpenAI* y el módulo *Spring AI*, así como las principales alternativas existentes.

LLM (Large Language Model)

Un *Large Language Model (*LLM*)* es un tipo de modelo de IA basado en redes neuronales profundas, entrenado sobre enormes volúmenes de texto para aprender patrones estadísticos del lenguaje natural y ser capaz de comprender, generar y razonar sobre texto de forma coherente. La gran mayoría de los LLM actuales se basan en la arquitectura *Transformer*, introducida por *Google* en 2017 [VSP+17], cuyo mecanismo de *atención (self-attention)* permite al modelo ponderar la relevancia de cada palabra dentro del contexto. Estos modelos cuentan con miles de millones de parámetros y se utilizan para tareas como traducción, resumen, clasificación, generación de código, respuesta a preguntas o asistentes conversacionales.

Un aspecto clave para entender cómo funciona un LLM son los *tokens*, las unidades mínimas de texto con las que opera internamente el modelo. Los LLM no procesan palabras enteras directamente, sino que primero descomponen el texto en *tokens* mediante un proceso llamado *tokenización*, realizado por un componente llamado *tokenizer*.

Un *token* puede corresponder a:

- Una **palabra completa** si es frecuente ("casa", "perro").
- Un **fragmento de palabra** o **subpalabra** si es menos común ("peluquer" + "o", "des" + "organizado").
- Un **carácter individual** o un **signo de puntuación** (",", "!", " ").

Como referencia práctica, en modelos como GPT, 1 *token* equivale aproximadamente a 4 caracteres en inglés o a ¾ de una palabra, y un texto en español de 1.000 palabras se traduce típicamente en unos 1.300–1.500 *tokens*.

La *tokenización* se realiza mediante algoritmos como *Byte Pair Encoding (*BPE*)*, *WordPiece* o *SentencePiece*, que aprenden durante el entrenamiento qué fragmentos de texto aparecen con mayor frecuencia y los codifican como un único *token*, optimizando así la representación del lenguaje. Cada *token* se convierte después en un identificador numérico que el modelo transforma en un vector denso (*embedding*) que captura su significado semántico en un espacio de cientos o miles de dimensiones.

Cómo genera texto un LLM

Una vez *tokenizada* la entrada, el LLM la procesa a través de sus capas de atención y, fundamentalmente, predice el siguiente *token* más probable dado el contexto anterior. Este *token* se añade a la secuencia y se repite el proceso de forma autoregresiva, generando el texto *token a token*, hasta alcanzar un criterio de parada (fin de frase, longitud máxima, etc.). Por eso, cuando se observa a un asistente conversacional "escribiendo" la respuesta letra a letra, en realidad está produciendo un *token* tras otro en tiempo real.

Importancia práctica de los tokens

Los *tokens* son cruciales por varios motivos:

- **Facturación**: las APIs comerciales (OpenAI, Anthropic) cobran por número de *tokens* procesados (tanto de entrada como de salida), por lo que optimizar los *prompts* tiene impacto económico directo.
- **Ventana de contexto**: cada modelo tiene un límite máximo de *tokens* que puede procesar en una única, lo que condiciona la cantidad de información que puede manejarse en una conversación.
- **Rendimiento**: cuantos más *tokens*, mayor latencia y consumo computacional.

La principal limitación de un LLM es que carecen de acceso nativo a información actualizada o a sistemas externos, ya que su conocimiento queda congelado en el momento del entrenamiento; por ello, en aplicaciones reales se complementan con técnicas como *Retrieval-Augmented Generation (*RAG*)*, *fine-tuning* o, más recientemente, integraciones estandarizadas mediante MCP.

MCP (Model Context Protocol)

MCP es un protocolo abierto propuesto por *Anthropic* en noviembre de 2024 [Ant24] cuyo objetivo es estandarizar la forma en que los LLM se conectan a fuentes de datos y herramientas externas, resolviendo el problema de las integraciones *ad hoc* que cada aplicación tenía que construir por separado. Funciona bajo un modelo *cliente-servidor*: los servidores MCP exponen capacidades concretas (acceso a una BBDD, lectura de ficheros, llamadas a APIs, gestión de un calendario, etc.) mediante una interfaz estandarizada, y los clientes MCP(típicamente aplicaciones que integran un LLM) las consumen de forma uniforme, sin necesidad de implementar un *adapter* específico para cada herramienta. Su diseño se inspira en el éxito del *Language Server Protocol (*LSP*)* (que estandarizó la integración entre IDEs y lenguajes de programación) y permite que un mismo servidor MCP pueda ser reutilizado por cualquier asistente compatible. En la práctica, MCP convierte al LLM en un agente capaz de interactuar de forma estructurada y segura con el mundo exterior, habilitando casos de uso como consultar BBDDs, ejecutar acciones en sistemas empresariales o acceder a informa-

ción en tiempo real, lo que lo posiciona como un estándar emergente clave para la próxima generación de aplicaciones basadas en IA.

OpenAI - GPT

*Generative Pre-trained Transformer (*GPT*)* es una familia de LLMs desarrollada por *OpenAI* desde 2018 [RNSS18], basada en la arquitectura **Transformer**. Sus siglas reflejan sus tres pilares fundamentales: *Generative* (capacidad para generar texto coherente y novedoso), *Pre-trained* (entrenado previamente sobre enormes corpus de texto procedentes de Internet, libros y código) y *Transformer* (arquitectura basada en el mecanismo de *atención (self-attention)*, que permite ponderar la relevancia de cada palabra dentro del contexto). A lo largo de sus sucesivas versiones, ha incrementado progresivamente el número de parámetros, la longitud de la ventana de contexto y sus capacidades multimodales (texto, imagen, audio), consolidándose el LLM más utilizado y el más famoso en el público generalista. Se puede acceder a él a través de la API de *OpenAI* mediante peticiones HTTP [Ope24], lo que permite integrarlo de forma sencilla en aplicaciones de terceros sin necesidad de gestionar la infraestructura del modelo.

SpringAI

Spring AI es un módulo oficial del ecosistema *Spring*, cuyo objetivo es integrar capacidades de IA generativa en aplicaciones empresariales *Java* siguiendo los mismos principios de diseño que el resto del framework: inyección de dependencias, auto-configuración, abstracciones portables y convención sobre configuración [Spr24a]. Proporciona una API *unificada* para consumir distintos proveedores de modelos de lenguaje a través de interfaces comunes como ChatClient, EmbeddingModel o ImageModel, de modo que el código de negocio no queda acoplado a un proveedor concreto. Incluye además soporte nativo para patrones modernos del desarrollo con LLMs: *prompt engineering* mediante plantillas, *function calling* para que el modelo invoque métodos del *backend*, RAG con integración a BBDDs vectoriales, memoria conversacional persistente y *structured output* para mapear respuestas del LLM a objetos *Java* tipados.

Alternativas a *Spring AI*

Para la integración del LLM dentro del sistema se podrían haber utilizado otras opciones:

- **LangChain4j:** es una librería de código abierto nativa para *Java* que simplifica la integración de LLM en proyectos *Java*. Ofrece una API unificada y tipada para conectar con múltiples proveedores de IA, gestionar memoria y estructurar flujos de datos con sistemas externos.
- ***OpenAI* API:** *OpenAI* dispone de un API nativo en *Java* para la comunicación con sus

modelos mediante HTTP. Esta opción acopla demasiado el diseño del proyecto a un modelo concreto.

3.3.1 Google Calendar

Google Calendar es el servicio de gestión de calendarios de *Google*, una herramienta mundialmente conocida e integrada en el stack tecnológico de la mayoría de usuarios particulares y PYMEs. Permite gestionar eventos y recordatorios y sincronizarlos entre distintos dispositivos y cuentas, y ofrece un SDK *Java* oficial que cubre la autenticación Open Authorization (OAUTH) y las operaciones CRUD sobre eventos [Goo24b, Goo24a]. Estas capacidades lo convierten en una de las opciones de referencia para integrar la gestión de calendarios en aplicaciones de terceros.

Alternativas a *Google Calendar*

- **Microsoft Graph (Outlook Calendar)**: técnicamente excelente pero su uso entre PYMEs del sector estético en España es mucho menor. Forzar al profesional a usar Outlook supondría una barrera de entrada que se puede evitar.
- **Calendarios propios sin integración externa**: esta opción se descarta principalmente porque rompería el principio de *familiaridad* para el usuario que tanto se ha perseguido en los objetivos marcados para el proyecto y obligaría al usuario a mantener dos agendas distintas. Además supondría un esfuerzo mucho mayor en el proceso de desarrollo, restando tiempo y recursos para que se pueden destinar a la solución de otros problemas.

3.3.2 Bot conversacional: Python y Telegram Bot API

La interfaz conversacional dirigida al cliente final se apoya en una plataforma de mensajería instantánea. Las dos plataformas líderes del sector son *WhatsApp* y *Telegram*. *Telegram* destaca por ofrecer una API abierta, gratuita y sin proceso de aprobación [Tel24], mientras que el desarrollo sobre *WhatsApp* requiere una cuenta de *WhatsApp Business* asociada a un número de teléfono real y puede conllevar costes por conversación. Para construir bots de *Telegram* existen distintas librerías, como *python-telegram-bot* en *Python* [pyt24] —que ofrece soporte para llamadas HTTP asíncronas— o *TelegramBots* en *Java*, esta última con una comunidad más reducida y actualizaciones habitualmente más lentas.

3.3.3 Frontend: TypeScript y React

El desarrollo de interfaces web modernas se apoya habitualmente en un lenguaje con tipado estático y en una librería o framework de componentes. En este apartado se describen *TypeScript* y *React*, dos de las tecnologías más extendidas para este fin, junto con sus principales alternativas.

TypeScript

TypeScript es lo que se conoce como un *superconjunto* de *JavaScript* donde su principal valor añadido es que lo expande con un sistema de tipos estático opcional que ayuda a prevenir errores antes de ejecutar el programa. Se compila (o mejor dicho, transpila) a *JavaScript* puro mediante el compilador tsc [Mic24], lo que permite ejecutarlo en cualquier navegador o runtime sin modificaciones. Además del tipado estático, incorpora otras características como interfaces, tipos genéricos, enums o inferencia de tipos, ofreciendo una experiencia de desarrollo muy próxima a lenguajes orientados a objetos conservando la flexibilidad de *JavaScript*. Es cierto que todo el envoltorio de *TypeScript* sobre *JavaScript* presenta algunas desventajas como una curva de aprendizaje inicial más pronunciada o un paso más en el proceso de compilación.

React

React es una librería de *JavaScript* de código abierto diseñada para construir interfaces de usuario declarativas, componentizadas y reactivas, especialmente en aplicaciones de una sola página (SPA) [Met24]. Su modelo se basa en el concepto de componentes reutilizables que son piezas independientes donde cada uno encapsula su propia estructura (JavaScript Syntax Extension (JSX)), comportamiento (lógica *JavaScript*) y estado. *React* introduce otros conceptos clave como hooks que son funciones especiales para gestionar estado, efecto y otras capacidades del framework dentro de componentes funcionales o el concepto de Virtual DOM para optimizar el rendimiento, actualizando de forma eficiente solo las partes de la página que cambian minimizando las actualizaciones reales del Document Object Model (DOM) del navegador. Además, ofrece un ecosistema gigantesco de librerías complementarias (*React Router, React Query*).

En definitiva, por sus propias características, *React* cuenta con varias ventajas como la componentización y reutilización, rendimiento excelente gracias al virtual DOM, una comunidad muy activa, un ecosistema enorme y documentación oficial muy madura. Por contra, presenta algunos inconvenientes como que solo cubre la capa de User Interface (UI), para acciones como enrutado o peticiones HTTP se requiere el uso de librerías externas. Frente a *React*, existen otras opciones ampliamente utilizadas como *Angular*, un framework más completo pero también más pesado y con una mayor curva de aprendizaje, o *Vue*, que cuenta con una comunidad y un soporte más reducidos.

Capítulo 4

Metodología

EL objetivo principal de este capítulo es el de describir el enfoque metodológico aplicado para la planificación y desarrollo de la solución que se propone en este TFG basándose en el modelo iterativo-incremental. Se incluirá una justificación de por qué se ha elegido esta metodología para el ciclo de desarrollo y el marco tecnológico con el que se ha realizado.

4.1 Metodología de desarrollo iterativo-incremental

La metodología iterativo-incremental es un enfoque de desarrollo de software que combina dos principios complementarios: el desarrollo iterativo, basado en realizar ciclos repetidos de planificación, diseño, implementación y evaluación que refinan progresivamente el producto, y el desarrollo incremental, que consiste en entregar el sistema en partes funcionales sucesivas (incrementos), de modo que cada entrega añade nueva funcionalidad sobre la anterior [Som16]. Sus raíces se remontan a los trabajos de Tom Gilb (1981) y Barry Boehm con el modelo en espiral (1988) [Boe88], y posteriormente fue formalizada en metodologías como Rational Unified Process (RUP) y, en su versión más moderna, en los marcos ágiles (*Scrum, Kanban*) que dominan el desarrollo profesional actual [LB03].

Frente al modelo en cascada clásico [Roy70], donde cada fase (análisis, diseño, implementación, pruebas, despliegue) se ejecuta de forma secuencial y completa antes de pasar a la siguiente, el enfoque iterativo-incremental divide el proyecto en ciclos cortos (iteraciones de días o semanas) que recorren todas las fases de desarrollo y producen, al final de cada ciclo, un incremento del producto funcional, probado e integrable. Esto permite validar continuamente las decisiones técnicas y funcionales, incorporar feedback temprano y adaptar el rumbo del proyecto sin acumular riesgos hasta una entrega final tardía.

4.1.1 Ventajas

Esta metodología permite reducir el riesgo de errores, ya que permite detectar problemas técnicos o funcionales en cada iteración en lugar de hacerlo al final del proyecto, lo que también lo hace más flexible ante posibles cambios que pueda surgir si los requisitos evolucionan entre iteraciones, evitando penalizar el avance global del proyecto. Otra ventaja es que se obtiene software funcional desde fases muy tempranas para poder validar con usuarios o partes

interesadas. Este progreso visible ayuda a mantener un ritmo sostenible y motivador, especialmente útil en proyectos individuales como es el caso. En cada iteración se incorporan lecciones aprendidas de las iteraciones previas, con lo que se puede conseguir refinar arquitectura, calidad o procesos y ayuda a medir la velocidad real de desarrollo y prever fechas con mayor precisión.

4.1.2 Desventajas

Por contra, un desarrollo basado en una metodología iterativo-incremental requiere una mayor disciplina organizativa para poder planificar, priorizar y cerrar iteraciones de forma rigurosa. La flexibilidad del modelo, al construirse de forma incremental, hace que puedan surgir necesidades de refactorización a medida que el sistema crece y derivar en que sea más difícil fijar un alcance cerrado desde el principio.

4.1.3 Aplicación en el proyecto

En este TFG, la metodología iterativo-incremental se ha aplicado dividiendo el desarrollo en iteraciones temáticas de duración acotada, cada una centrada en un incremento funcional concreto y entregable:

Iteraciones

1. **Análisis**: análisis del dominio, selección de tecnologías y configuración del entorno de desarrollo. **Incremento/Entregable:** análisis de Requisitos v1, Listado de HUs (se pueden consultar en el Apéndice A) y de tecnologías a utilizar.

2. **Diseño:** definición del modelo de dominio y diseño de arquitectura hexagonal y modelo de datos. **Incremento/Entregable:** diseño Funcional, de arquitectura hexagonal y del modelo de datos.

3. **Backend**: puesta en marcha del proyecto, arquitectura hexagonal y lógica de negocio relativa a la gestión de usuarios, empresas, trabajadores, servicios, horarios y períodos no laborables, con sus correspondientes endpoints y tests unitarios. En esta iteración también se han abordado otras tareas que sirven para asentar la base de desarrollo del que se beneficiarán el resto de iteraciones (sistema de manejo de errores, testing unitario, servicios, controladores, etc.) **Incremento/Entregable:** módulo software *Backend* operativo con API-REST para gestión de lógica de negocio.

4. **Seguridad**: esta iteración se solapa con la anterior ya que hay tareas que se comienzan a abordar junto con las fases finales de la iteración anterior. Engloba la autenticación con JWT firmado para el API-REST, gestión de contraseñas de usuario y el mecanismo de securización mediante API key que utilizará más adelante el endpoint del bot. **Incremento/Entregable:** componentes software de seguridad: autenticación JWT para API-REST, autenticación API-key para bot y gestión de contraseñas de Usuario.

5. **Google Calendar:** también en esta se produce un solapamiento con la anterior, ya que se incluye toda la parte de almacenamiento seguro de los *tokens* de *Google*. Además se llevan a cabo las tareas de integración del API de *Google Calendar*, configuración de cuenta y gestión de refresco de *tokens*. **Incremento/Entregable:** componentes software para integración *Google Calendar*: gestión de cuentas de *Google* con almacenamiento y refresco de *tokens* y gestión de calendarios y eventos.

6. **Frontend**: construcción del panel de administración web para la gestión de negocio y la visualización de calendarios. **Incremento/Entregable:** módulo software *Frontend* funcional integrado con *backend* para la operativa de *backoffice*.

7. **Spring AI + GPT:** integración dentro del proyecto de *Spring AI* con la comunicación con el modelo GPT y desarrollo de todas las herramientas de apoyo al modelo a través del protocolo MCP. **Incremento/Entregable:** componentes software de IA: integración de *Spring AI* en proyecto *backend*, comunicación con LLM y herramientas MCP para consulta del modelo.

8. **Bot conversacional**: desarrollo del componente *Python* que dará *vida* al bot y el endpoint para la conexión con el *backend* mediante API key. **Incremento/Entregable:** módulo *Python* para la operativa del bot conversacional integrado con el *backend*.

9. **Pruebas integradas + RR:** durante esta iteración se realizan pruebas integradas del software completo y se llevan a cabo tareas de refinamiento y refactorización que puedan haber surgido durante el resto de las iteraciones. **Incremento/Entregable:** plan de pruebas realizado y componentes softwares refactorizados.

10. **Memoria:** iteración destinada a generar toda la documentación asociada al proyecto y realización del presente documento como memoria entregable para el TFG. **Incremento/Entregable:** documento de memoria TFG final.

Cada iteración que conlleva entregables de software ha seguido un ciclo completo de análisis, diseño, implementación, pruebas y validación, cerrándose con un incremento funcional integrado en la solución. Esta dinámica ha permitido detectar problemas en fases avanzadas del desarrollo y adaptar el alcance sin comprometer la entrega final.

En la Figura 4.1 se puede ver el Diagrama de Gantt que representa el ciclo de desarrollo seguido para este proyecto.

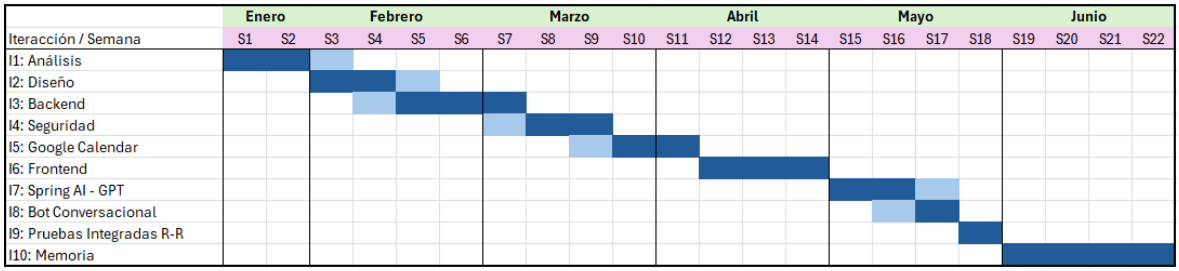


Figura 4.1: Estimación del plan de trabajo para el desarrollo

Conclusión

La metodología iterativo-incremental ha permitido gestionar la complejidad y la incertidumbre tecnológica del proyecto sin las rigideces del modelo en cascada ni el sobrecoste organizativo de marcos ágiles completos como *Scrum*. Su combinación de iteraciones cortas, entregas funcionales frecuentes y adaptación continua ha resultado especialmente adecuada para un proyecto que integra múltiples tecnologías emergentes y que se beneficia de una validación incremental tanto técnica como funcional a lo largo de todo el desarrollo.

4.2 Marco tecnológico

Recursos Hardware		
Recurso	**Modelo / especificaciones**	**Función**
Equipo de desarrollo principal	PC sobremesa CPU AMD Ryzen 7 9800, 32GB RAM	Investigación, Desarrollo, Pruebas y Documentación.
Dispositivo móvil de pruebas	OnePlus Nord 12GB RAM	Pruebas del bot de *Telegram* en cliente real.
Conexión a Internet	Fibra simétrica 600 Mbps	Investigación, Acceso a APIs externas y *Telegram*.

Tabla 4.1: Recursos Hardware

Recursos Software - Backend, BBDD y Build	
Recurso	**Función**
Java	Lenguaje programación backend
Spring Boot	Framework principal de aplicación
Spring Security	Autenticación y autorización
Spring AI	Integración con LLMs y MCP
Hibernate / Spring Data JPA	ORM y acceso a datos
PostgreSQL	SGBD relacional
Flyway	Gestión de Esquema BBDD
Google API Client	Cliente HTTP genérico de *Google*
Google Calendar API	Integración con *Google Calendar*
Google OAuth Client	Flujo OAuth 2.0 para *Google*
JJWT	Generación y verificación de *tokens* JWT
MapStruct	Mapeo entre DTOs y entidades
Lombok	Reducción de boilerplate
Maven	Build y gestión de dependencias
JUnit	Testing unitario backend
Mockito	Mocks para tests unitarios backend
Easy Random	Generación de datos aleatorios para tests
OpenAI GPT API	LLM para comunicación con usuario

Tabla 4.2: Recursos Software - Backend, BBDD y Build

Software - Frontend	
Recurso	**Función**
TypeScript	Lenguaje de programación frontend
Node.js	Entorno de ejecución frontend
React	Framework de desarrollo principal de frontend
Vite	Bundler y servidor local para desarrollo frontend
TailwindCSS	Gestión de estilos *CSS*
FullCalendar	Visualización de calendarios
React Router DOM	Enrutado en *frontend*
React Icons	Iconos en *frontend*
Axios	Cliente HTTP para *React*
Vitest	Testing *frontend*

Tabla 4.3: Software - Frontend

Otros Recursos Software	
Recurso	**Función**
Python	Lenguaje de programación para bot
python-telegram-bot	Librería *Python* para *Telegram* Bot API
aiohttp	Librería *Python* para HTTP
Docker	Ejecución de contenedores
Docker Compose	Orquestación de build y despliegue
IntelliJ	IDE backend para desarrollo *Java* y *Python*
Visual Studio Code	IDE frontend para desarrollo TypeScript
DBeaver	Cliente para BBDD *PostgreSQL*
Postman	Cliente HTTP para pruebas
Git	Control de versiones
ChatGPT, Claude, Gemini	LLMs para investigación y apoyo en desarrollo
Overleaf	Creación de documentación LaTeX
Microsoft Excel, Word	Otra documentación
Draw.io	Generación de Diagramas

Tabla 4.4: Otros Recursos Software

Capítulo 5

Arquitectura

En este capítulo se expondrán los aspectos técnicos de todos los módulos que componen la solución propuesta para la problemática abordada en este TFG. Primero se proporcionará una visión global de la solución y después se profundizará en cada uno de los módulos que se han desarrollado a lo largo del ciclo de vida del proyecto, destacando las partes más importantes de cada uno de ellos, con referencias directas al código y diagramas que ayudarán a facilitar su comprensión.

5.1 Visión general

En esta sección se aportan una visión general de la estructura de la solución en la que se muestran la forma en la que los usuarios pueden interactuar con el sistema y como los módulos internos del propio sistema interactúan entre ellos o con otros servicios externos. El sistema aceptará cuatro tipos de usuarios diferentes, por un lado, tendremos el usuario CLIENTE, que interactuará con el sistema a través de la *App* de *Telegram* para conversar con el bot y gestionar sus citas y por otro existirán tres roles de usuarios diferentes que interactuarán a través del backoffice:

- **ADMIN**: administrador, gestiona los usuarios y empresas dadas de alta en el sistema.
- **MANAGER**: gerente, gestiona toda la información de negocio relativa a una empresa, servicios, horarios, días festivos. También puede consultar los calendarios de todos los trabajadores de su empresa.
- **WORKER**: trabajador, gestiona sus periodos inhábiles y puede visualizar su propio calendario de trabajo.

El sistema necesita de tres servicios externos fundamentales para su funcionamiento:

- **Telegram Bot API:** es el conector entre la aplicación de Telegram y el Sistema para la comunicación del cliente con el bot.
- **LLM OpenAI GPT:** el modelo LLM que se encarga de la interpretación de los mensajes de cliente y generación de las acciones y respuestas correspondientes.
- **Google Calendar API:** API de *Google* con la que el sistema trabaja para gestionar los diferentes calendarios y eventos.

El Sistema se conforma de estos módulos principales:

- **Aplicación Backoffice:** proporciona a los usuarios de *backoffice* un panel de administración a través del cual realizar las gestiones propias de su rol.
- **Bot:** punto de entrada para el cliente en el sistema, se encarga de gestionar los mensajes que el usuario envía al sistema y viceversa.
- **API-REST:** es el punto de entrada para la comunicación de la aplicación *backoffice* y el bot con el núcleo del sistema.
- **Lógica de Negocio:** orquesta los flujos necesarios para todas las operaciones de negocio.
- **Módulo IA:** responsable de la comunicación con el LLM.
- **Módulo Google Calendar:** implementa la comunicación con *Google* para la gestión de calendarios, eventos y los flujos de autenticación OAUTH2.
- **Módulo de persistencia:** se encarga de la comunicación con BBDD para almacenar y recuperar toda la información necesaria para el funcionamiento del sistema.

En las siguientes secciones de este capítulo se profundizará más en cada uno de los módulos, por el momento se definen de una forma muy genérica y reducida, los principales flujos de interacción con el usuario:

- El cliente chateará con el bot conversacional a través de la app de *Telegram*, en ese momento se ejecutará el código del bot en *Python* para enviar los mensajes del cliente al sistema a través del endpoint api/v1/chat autenticado con API-key. El *backend* recibirá el mensaje del cliente se lo enviará al LLM el cual tendrá que interpretarlo y hacer uso de cualquier de las herramientas que ofrece el MCP, una vez haya completado todas las acciones requeridas, generará un mensaje en lenguaje natural a modo de respuesta para el cliente, el backend lo devolverá al bot como respuesta en la petición al endpoint de chat y éste se encargará de mostrar el mensaje al cliente.
- Un usuario de *backoffice* entrará en el panel de administración autenticándose mediante usuario y contraseña, lo cual generará un *token* JWT y el *backoffice* mostrará las acciones que pueda realizar según el rol asignado al usuario. Todas las acciones que realice el usuario a través del *backoffice* envían una petición al API-REST expuesta por el *backend* para obtener la información necesaria para mostrar o completar la acción requerida.

La figura 5.1 muestra un esquema general de los módulos que componen el sistema, y cómo interactúan entre ellos y con los módulos externos:

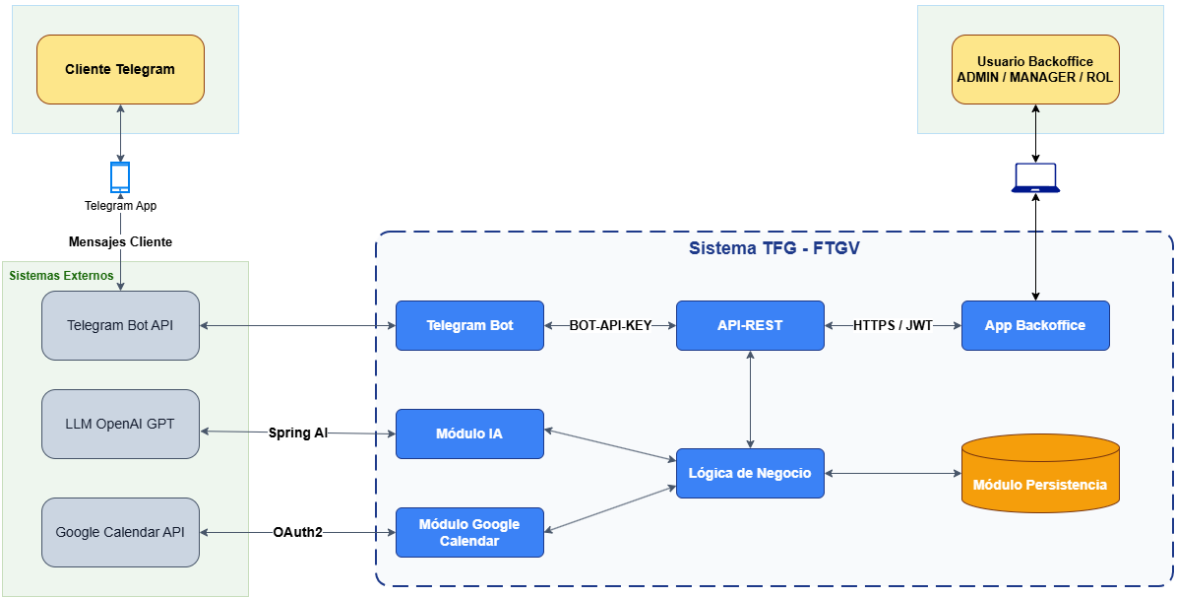


Figura 5.1: Visión general

5.1.1 Organización en capas

Para el desarrollo de este proyecto se ha seguido un diseño basado en *arquitectura hexagonal*. Este diseño aporta al proyecto una separación clara entre la lógica de negocio y los detalles técnicos, garantizando independencia tecnológica y facilitando el mantenimiento a largo plazo. Aunque introduce cierta complejidad estructural inicial que en las primeras fases del sistema pueden suponer un esfuerzo extra, esta inversión queda plenamente justificada por ofrecer el mejor equilibrio entre rigor arquitectónico y complejidad asumible para el contexto del proyecto: aporta las ventajas clave que MVC y la arquitectura por capas no ofrecen (dominio puro, inversión de dependencias, independencia tecnológica), sin incurrir en la sobreingeniería de *Clean Architecture*. Además, su estructura clara basada en *puertos y adaptadores* encaja de forma natural con un proyecto que integra múltiples sistemas, permitiendo gestionar cada integración como un adaptador independiente sin contaminar el núcleo y construye una excelente base para la escalabilidad y mantenimiento del sistema. No obstante, esto no significa que el patrón MVC quede excluido: de hecho está presente en el proyecto de forma interna, a través de los *@RestController* de *Spring*, pero integrado dentro de la arquitectura hexagonal, que le aporta el marco organizativo global del que por sí solo carece.

El código se divide en tres grandes capas, las cuales se implementan a través de los paquetes principales de *Java*:

1 - Dominio

Es el corazón de la aplicación y no depende de ninguna tecnología externa. Contiene:

- Modelos de dominio (domain.model): clases puras que representan los conceptos del negocio (Appointment, Company, Customer, etc). No llevan anotaciones de persisten-

cia ni de serialización, solo expresan datos y comportamiento de negocio (por ejemplo, el método Appointment.cancel()).

- Puertos de entrada (domain.ports.in): interfaces que describen los casos de uso que la aplicación ofrece al exterior (p.ej. SaveAppointmentUseCase o GetAvailableSlotsUseCase). Definen *qué* se puede hacer, no *cómo* hay que hacerlo.
- Puertos de salida (domain.ports.out): interfaces que describen lo que el dominio *necesita* del exterior para funcionar (p.ej. *AppointmentPort* o *CalendarEventPort)*. El dominio invoca estas interfaces sin saber si detrás hay una BBDD PostgreSQL, la API de *Google Calendar* o cualquier otro servicio externo al dominio.

2 - Aplicación

Contiene la implementación de los casos de uso (*application.usecase*), es decir, las clases que orquestan la lógica de negocio. Cada clase "*...UseCaseImpl*" implementa un puerto de entrada y se apoya exclusivamente en puertos de salida para acceder a recursos externos. Vamos a ver un ejemplo real del proyecto a través de SaveAppointmentUseCaseImpl: Para registrar una cita recupera los datos a través de CustomerPort, ServicePort, WorkerPort y CompanyPort, valida la disponibilidad, crea el evento mediante CalendarEventPort y finalmente persiste con AppointmentPort. En ningún momento aparece una referencia a JPA, a una entidad de BBDD o al cliente HTTP de *Google*, todo se hace contra interfaces del dominio.

3 - Infraestructura

Aquí viven los adaptadores, las piezas que conectan el núcleo con el mundo real. Se distinguen dos lados:

- Adaptadores de entrada(infrastructure.in): son los que *invocan* los casos de uso.
  - Controladores REST (infrastructure.in.controller): exponen la API HTTP (p.ej. PostCreateCompanyController). Reciben un Data Transfer Object (DTO), lo traducen al modelo de dominio y llaman al puerto de entrada del que obtienen la respuesta que devolverán al componente que lo haya invocado.
  - Servidor MCP (infrastructure.in.ai): *McpToolService* publica los casos de uso como *herramientas* consumibles por el modelo mediante *Spring AI* y el protocolo MCP. Es un segundo punto de entrada al dominio, completamente distinto de los Controllers pero que reutiliza exactamente los mismos puertos de entrada.
- Adaptadores de salida (infrastructure.out): implementan los puertos de salida.
  - Persistencia (infrastructure.out.repository y out.adapter): por ejemplo, AppointmentAdapter es la implementación de AppointmentPort, el cual sirve como interfaz de conexión con la BBDD, traduce entre el modelo de dominio y las entidades

JPA y delega en un JpaRepository sobre *PostgreSQL* la operativa de acceso a los datos.

- Integraciones externas: se generan otro tipo de adaptadores para servicios externos como por ejemplo con *Google Calendar* para la gestión de OAuth2.
- Configuración (infrastructure.config): clases de configuración propias del sistema para seguridad, clientes externos, *schedulers*, etc.

El siguiente diagrama (figura 5.2) representa la arquitectura hexagonal tal y cómo se ha implementado en el proyecto, en el que podemos ver la correspondencia de cada capa con el paquete *Java* que la implementa y las interacciones más relevantes del sistema con el exterior:

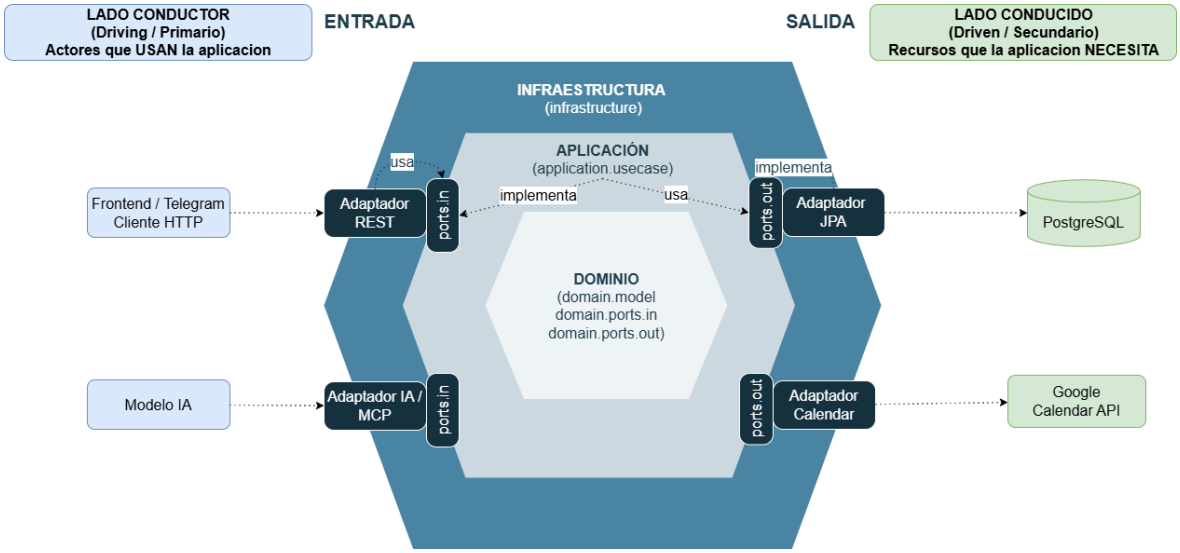


Figura 5.2: Esquema de Arquitectura Hexagonal en el proyecto

5.2 Módulo de persistencia

El diseño del modelo de datos es una de las partes más importantes en cualquier proyecto. Es imprescindible que el modelo esté bien diseñado y sea robusto, evitando redundancia y garantizando la integridad de los datos. En este proyecto se ha optado por un modelo relacional debido a la necesidad de garantizar la integridad, consistencia y robustez de la información mediante el cumplimiento estricto de las propiedades ACID. Este enfoque es fundamental para estructuras de datos bien definidas y con relaciones complejas e integraciones con sistemas externos, como las que presenta el dominio de negocio de este proyecto. Frente a soluciones no relacionales, este modelo previene eficazmente la duplicidad de datos y asegura una consistencia inmediata, proporcionando un entorno predecible y altamente estructurado.

Tecnológicamente, esta capa se compone de un SGBD relacional, *PostgreSQL*, al que el servicio *backend* accede a través de *Spring Data JPA* junto con *Hibernate* como proveedor ORM. La gestión del esquema y sus migraciones se realiza con *Flyway*, lo que permite

versionar la BBDD de forma alineada al código fuente. La elección de *PostgreSQL* frente a otros SGBD relacionales aporta además un valor añadido de cara a futuras mejoras relacionadas con la IA, gracias a su potente soporte JSON/JSONB y a la extensibilidad *pgvector*, que permitiría evolucionar hacia una BBDD vectorial si el proyecto lo requiriese.

En el modelo resultante, cada tabla representa a una entidad presente en el sistema. A continuación, se muestra el esquema de datos dividido en tres partes para facilitar su visualización y comprensión. El primer esquema (figura 5.3) representa las tablas y relaciones relacionadas con los datos de una empresa:

- La tabla 'company' almacena toda la información relativa a la empresa.
- La tabla 'google_account' almacena la información de la cuenta de *Google* asociada a una empresa, como los *tokens* de autenticación y el tiempo de expiración de los mismos.
- Para la persistencia de los datos correspondientes a los servicios, días inhábiles y horarios de una empresa se crean las tablas 'service', 'non_working_day' y 'service_hours' respectivamente.

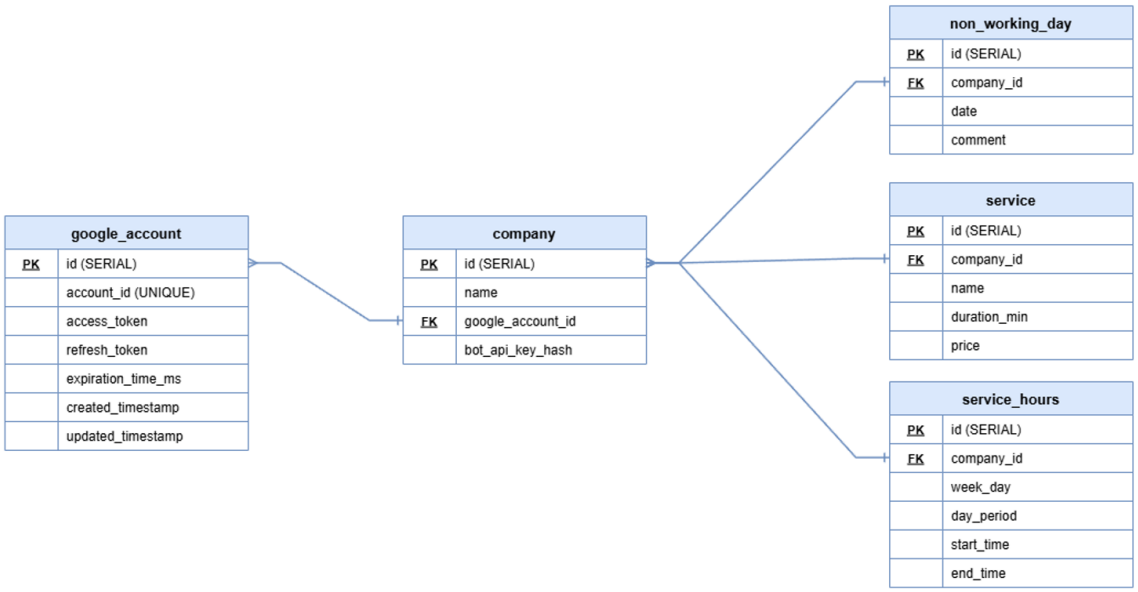


Figura 5.3: Modelo de datos de Company

El siguiente esquema (figura 5.4) representa el modelo de datos para la gestión de usuarios:

- La tabla 'users' es la encargada de registrar toda la información relativa a un usuario.
- La tabla 'user_role' guarda la relación entre el usuario y el rol asignado a cada uno.
- La tabla 'worker' extiende la información de un usuario de tipo WORKER añadiendo el id del calendario.
- La tabla 'non_working_period' se utiliza para almacenar los periodos inhábiles de un trabajador.

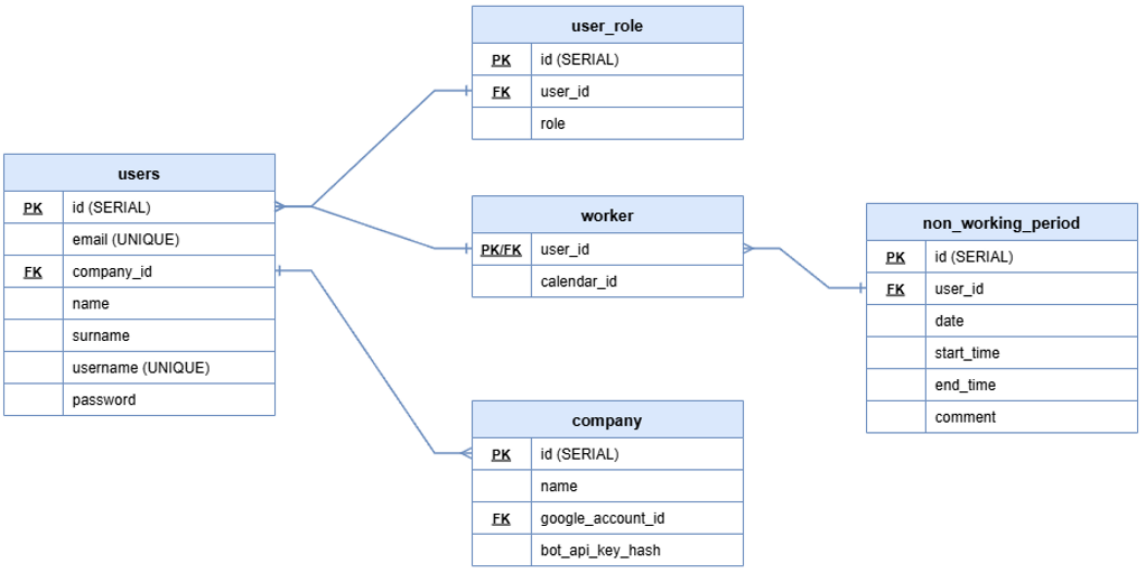


Figura 5.4: Modelo de datos de Users

En el último esquema (figura 5.5) podemos ver las tablas y relaciones correspondientes a la gestión de citas y clientes:

- La tabla ‘appointment’ almacena toda la información de las diferentes citas registradas en el sistema.
- La tabla ‘customer’ es la encargada de persistir la información de los clientes.
- La tabla ‘message’ guarda los mensajes que los clientes envíen al bot conversacional así como las respuestas que reciban de él.

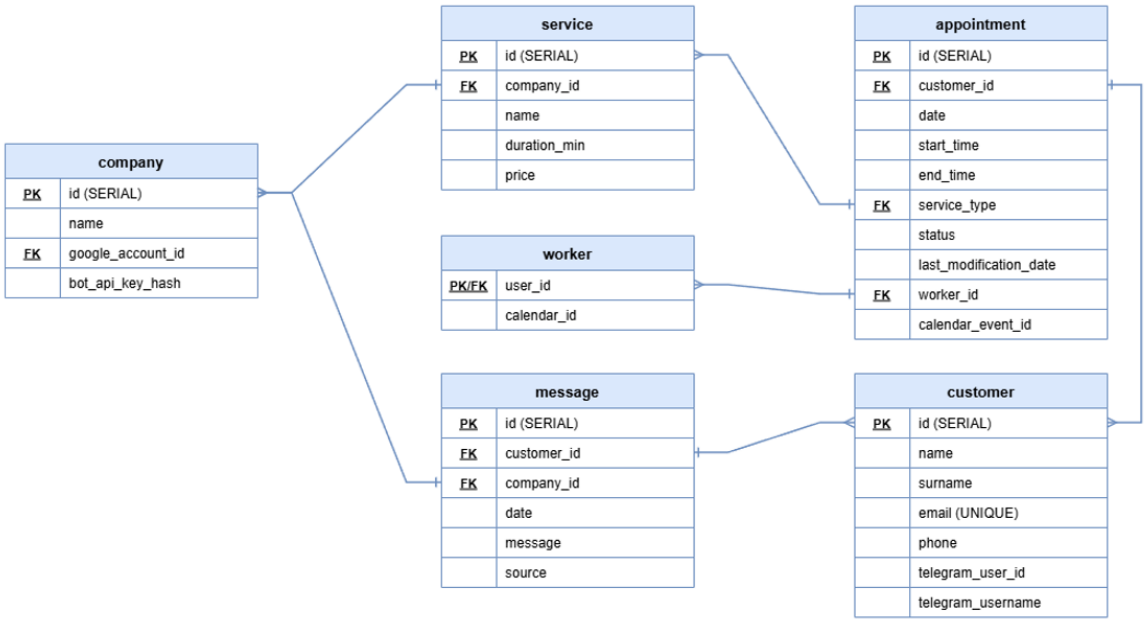


Figura 5.5: Modelo de datos de Appointment

5.3 Modulo Backoffice - *Backend*

El servicio *backend* constituye el núcleo del sistema y el componente con más responsabilidades, encargado de orquestar los flujos entre el resto de componentes. Se ha desarrollado sobre un tándem tecnológico consolidado como uno de los estándares de facto para el desarrollo de aplicaciones API-REST: *Java* 21 como lenguaje y *Spring Boot* 3.4 como framework de aplicación, que se apoya internamente sobre *Spring Framework* 6 [Spr24b, Spr24d]. La elección de la versión 21 de *Java* permite aprovechar las características modernas del lenguaje y asegura soporte y actualizaciones a largo plazo [Ora26]. Además, el dominio previo del autor sobre *Java* y *Spring Boot* ha permitido concentrar el esfuerzo de desarrollo en los componentes diferenciales del TFG, en lugar de invertirlo en el aprendizaje de un lenguaje o framework nuevo.

El panel de administración (*backoffice*) del proyecto permite gestionar varias entidades del negocio (**Service**, **Company** o **NonWorkingDay**, entre otras). Aunque cada una de ellas representa un concepto distinto, todas comparten un mismo flujo técnico de forma genérica: las operaciones CRUD (crear, leer, actualizar y eliminar) viajan desde el navegador a través de los endpoints del API-REST hasta la BBDD atravesando siempre las mismas capas y respetando la arquitectura hexagonal del *backend*. Por ese motivo basta con describir un caso representativo para entender todos los demás. En este punto se toma como ejemplo la gestión de servicios, y en concreto la operación de creación/guardado, para definir de forma genérica el flujo que seguirán el resto de operaciones similares. Otras gestiones como las de Company, User o NonWorkingDay son idénticas salvo por el nombre de la entidad, el endpoint y los campos del formulario. El punto fuerte de este diseño es que todas las operaciones siguen un mismo patrón, por tanto, para implementar cualquier operación bastaría con replicar la misma estructura cambiando únicamente la lógica de negocio correspondiente. Esta uniformidad hace el sistema predecible, mantenible y fácil de extender. El recorrido de una petición desde el momento que llega a cualquiera de los endpoints que implementan el API-REST de *backoffice* se puede resumir de la siguiente forma:

Un controlador REST recibe la petición, la traduce al dominio, ejecuta la lógica de negocio en el caso de uso correspondiente y accede a la BBDD para persistir o recuperar información a través de un puerto y su adaptador.

En las siguientes secciones, se detallan todas las piezas que intervienen en estos flujos.

5.3.1 El filtro de seguridad (JWT)

La seguridad del *backend* se articula en su mayor parte en torno a *Spring Security* y se apoya en dos mecanismos de autenticación diferenciados según el canal de comunicación: una autenticación *stateless* basada en *tokens* JWT firmados con criptografía asimétrica RSA para las peticiones del *frontend* del *backoffice*, y un esquema de API-key único por empresa para proteger la comunicación con el bot, que se detalla más adelante en el Módulo ChatBot. A

ello se suma la gestión segura de las contraseñas de usuario, almacenadas mediante *hashing* criptográfico unidireccional con *BCrypt* + SAL a través de la implementación *BCryptPasswordEncoder*. De este modo, *Spring Security* centraliza tanto la protección de los endpoints autenticados con JWT como la del endpoint */chat* mediante API-key, además del *hashing* de contraseñas y la política CORS frente al *frontend*.

A continuación se detalla cómo se materializa este esquema de autenticación en el flujo del API-REST que expone el *backend*. Antes de nada, el usuario deberá registrarse en el sistema, para eso se expone el *endpoint /api/v1/auth/login*. Este endpoint tiene la responsabilidad de validar las credenciales de usuario que recibe y generar el *token JWT* de autenticación que devolverá en la petición para utilizar en peticiones posteriores Al recibir una petición HTTP, lo primero que actúa es *JwtAuthenticationFilter*, un filtro que se ejecuta una vez por petición, extrae el *token* de la cabecera *Authorization*, lo valida, recupera el usuario y sus roles, y los coloca en el contexto de seguridad de *Spring*. Si el *token* es inválido, corta la petición con un 401 como respuesta. Si el *token* es correcto, vuelve a entrar en juego *la magia* de *Spring*, ya que previamente se ha establecido una lista de restricciones de acceso por rol para cada endpoint en la clase *SecurityConfig*. Si el rol del usuario es el correcto para la ruta que se ha invocado el flujo pasará al Controller, en caso contrario se devolverá un error 403 Forbidden.

En el Apéndice B, sección B.1, se puede consultar un diagrama representando el proceso de emisión del *token* JWT.

5.3.2 El controlador REST (adaptador de entrada)

Cada operación tiene su propio controlador, siguiendo el principio de responsabilidad única. Para el caso del ejemplo de alta de servicios es *PostCreateServiceController*. Su responsabilidad es deliberadamente mínima:

1. Recibe el DTO (ServiceDto) del cuerpo de la petición.
2. Lo traduce al modelo de dominio mediante el mapper.
3. Invoca el caso de uso (*puerto de entrada*).
4. Cuando recibe la respuesta, vuelve a traducir el resultado al DTO de respuesta y responde con 200 y los datos correspondientes.

El controlador no contiene lógica de negocio: solo orquesta la traducción y la llamada.

5.3.3 El mapper

ServiceControllerMapper es una interfaz anotada con *MapStruct* que genera automáticamente, en tiempo de compilación, el código de conversión DTO - modelo de dominio. Gracias a esto, el modelo de dominio (*Service*) nunca se *contamina* con anotaciones de serialización JSON, manteniéndose limpio e independiente de la capa web.

5.3.4 El caso de uso

SaveServiceUseCaseImpl implementa el puerto de entrada SaveServiceUseCase y orquesta la lógica de negocio. Es el punto donde se concentra el valor de la arquitectura hexagonal: el caso de uso solo conoce interfaces (puertos de salida), nunca detalles de BBDD. Este es el punto en que más varía de una operación a otra, en el caso del alta de un servicio:

1. Recupera la empresa a través de *companyPort.findById(companyId)* si no existe, lanza *ResourceNotFoundException*.
2. Asocia el servicio a esa empresa (*service.assignToCompany(company)*).
3. Lo persiste con *servicePort.createService(service)*.

5.3.5 El puerto de salida y su adaptador

ServicePort es una interfaz que declara *qué* necesita el dominio (crear, actualizar, borrar, listar, buscar). Su implementación concreta es el adaptador de *ServiceAdapter*, que:

- Traduce el modelo de dominio a entidad JPA(ServiceEntity) con su propio *mapper* (ServiceAdapterMapper).
- Delega la persistencia en *ServiceRepository*, que extiende JpaRepository de *Spring Data JPA* sobre *PostgreSQL*.
- Devuelve de nuevo un modelo de dominio, ocultando por completo la tecnología de persistencia.

Es *Spring* quien, en tiempo de ejecución, conecta la interfaz ServicePort con su implementación ServiceAdapter mediante inyección de dependencias gracias a la anotación @Component y la inyección por constructor. Esta es la inversión de dependencias que mantiene el dominio aislado.

5.3.6 Viaje de vuelta y el manejo de errores

El resultado devuelto por el repositorio al adaptador se traduce de nuevo al modelo del dominio (Service) y asciende por las mismas capas: el adaptador lo devuelve al caso de uso, este al controlador, el controlador lo convierte a ServiceDto y lo devuelve como respuesta de la petición HTTP (200 OK). Si a lo largo del flujo de la petición se lanza alguna excepción, ésta es recogida por *CommonExceptionHandler* que se encarga de traducir las excepciones a respuestas HTTP homogéneas: por ejemplo ResourceNotFoundException $\rightarrow$ 404 o cualquier otro error $\rightarrow$ 500, siempre con un cuerpo ErrorResponse uniforme.

En el Apéndice B, sección B.3,se muestra un diagrama de secuencia abarcando el flujo completo.

5.4 Módulo Backoffice - Frontend

El panel de administración del *backoffice* es una aplicación de página única (*Single Page Application*, SPA) construida con React/TypeScript y Tailwind CSS para la maquetación y estilos que proporciona un menú principal con diferentes opciones según el rol del usuario autenticado (ADMIN, MANAGER, WORKER).

La elección de este stack responde a criterios de calidad y productividad. *TypeScript* añade sobre *JavaScript* un sistema de tipos estático que permite detectar errores en tiempo de compilación y produce un código más legible y autodocumentado, lo que se traduce en un producto de mayor calidad pese a la ligera curva de aprendizaje adicional que introduce. *React*, por su parte, ofrece un buen equilibrio entre rendimiento, flexibilidad y ecosistema frente a alternativas como *Angular* o *Vue*, y aunque solo cubre la capa de UI —requiriendo librerías externas para el enrutado o las peticiones HTTP—, estas limitaciones quedan perfectamente resueltas para las necesidades del proyecto. El stack se completa con *Vite* como herramienta de compilación, transpilación y empaquetado, *TailwindCSS* como sistema de maquetación y utilidades CSS, y *FullCalendar* para la visualización de calendarios.

5.4.1 Estructura general y punto de entrada

El proyecto se organiza en carpetas de forma que se separan responsabilidades de forma convencional:

- *pages/:* vistas asociadas a una ruta (Login, Dashboard, CalendarPage...).
- *components/:* componentes reutilizables (Sidebar, ProtectedRoute, modales).
- *context/:* estado global (UserContext).
- *config/:* configuración declarativa (permissions.ts).
- *api/:* cliente HTTP (apiClient) y utilidades de *token* (*auth*).
- *interfaces/:* tipos *TypeScript* que reflejan los DTO del *backend*.

El punto de entrada a la aplicación se concentra en el archivo *main.tsx*, en él se monta el árbol de componentes y se envuelve en un proveedor de contexto de *React UserProvider.* De esta forma se consigue que cualquier componente del proyecto pueda consultar el estado del usuario sin necesidad de pasar información manualmente entre componentes.

5.4.2 Páginas y formularios

La mayoría de las secciones del *backoffice* (usuarios, empresas, servicios, trabajadores, horarios, días y periodos inhábiles) siguen un mismo patrón repetible de tipo CRUD (*Create, Read, Update, Delete*). Ese patrón reparte las responsabilidades entre tres piezas que conviene distinguir con claridad, porque cada una juega un papel diferente: la página, la interfaz y el modal.

Página

La página (*pages/*) es el componente *contenedor* que orquesta la sección completa. Sus responsabilidades son:

- Encierra todo el código de maquetación para la visualización de la página y los datos.
- Orquesta el flujo de interacción, gestiona la modal del formulario y los posibles errores.
- Realiza las llamadas al *backend* para las operaciones de lectura y escritura de la información de negocio.

Interfaz

La interfaz (*interfaces/*) son tipos de *TypeScript* que describen la forma de los datos que viajan entre el *frontend* y el *backend*, es decir, son los DTOs que maneja el proyecto front.

- No contienen lógica ni se renderizan.
- Garantizan que la página y el modal manejan exactamente los campos esperados.
- Separan la entidad de lectura de la entidad de escritura, ya que pueden ser distintas.

Modal

El modal (*components/...Modal.tsx*) es el componente *reutilizable* que contiene el formulario para cada entidad. La misma pieza sirve para insertar nuevos registros y para editar registros ya existentes.

Sus responsabilidades son:

- Contiene el código HTML/CSS para la maquetación de los formularios.
- Precarga el formulario cuando se abre en modo edición, o limpiarlo en modo alta.
- Carga datos auxiliares que necesite el formulario (p. ej., la lista de empresas para un desplegable).
- Realiza validaciones definidas sobre los campos.
- Construye el *payload* y delega el guardado a la página.

El flujo completo que siguen todas las secciones es:

1. La página consulta la API y renderiza la información (generalmente en forma de tabla).
2. Cuando el usuario pulsa *Nuevo* o *Editar* abre el modal, indicándole con una propiedad si es alta o edición.
3. La modal muestra el formulario y recoge los datos en un objeto que respeta la interfaz correspondiente y los devuelve a la página.
4. La página realiza la llamada a la API y refresca la vista.

5.4.3 Gestión de autenticación

La autenticación que se utiliza en el proyecto front es sin estado (*stateless*): el *backend* emite un JWT en el login y el *frontend* lo guarda y lo adjunta en cada petición. No hay cookies de sesión ni estado de sesión en el servidor.

Inicio de sesión

El formulario de LoginPage envía las credenciales al endpoint POST /auth/login. Si la respuesta es correcta, guarda el *token* en *localStorage*, refresca el estado global y navega al *dashboard*.

El manejo de errores contempla un caso especial del dominio: si la empresa del usuario aún no ha autorizado el acceso a *Google Calendar*, el *backend* responde 403 con una consentUrl. En ese caso, el *frontend* muestra entonces una modal mostrando al usuario la URL de consentimiento (*ConsentModal*) en lugar de un error genérico.

Cliente HTTP e interceptores

Las llamadas al API de todos los componentes se centralizan en *apiClient*, una instancia de Axios [Axi24] con dos interceptores que automatizan la seguridad y evitan repetir lógica en cada llamada.

El primer interceptor para las peticiones se encarga de recuperar el *token* del *localStorage* y adjuntarlo en la cabecera *Authorization Bearer.*

El segundo interceptor actúa en la respuesta a la petición, si el código de respuesta es del tipo 401 porque el *token* haya caducado o sea inválido, limpia la sesión y redirige al formulario de *login.*

Gracias a este diseño, ningún componente necesita preocuparse de adjuntar el *token* ni de detectar la expiración de la sesión, para ellos ocurre de forma transparente.

Información de usuario

Una vez se completa el login, entra en juego la pieza *UserContext* que se ejecuta una única vez y: se encarga de obtener la información del usuario desde el *backend* y exponerla a toda la aplicación mediante el *hook useUser()* que da acceso al objeto completo de usuario para poder consultar nombre de usuario, empresa o roles desde cualquier componente. *UserContext* gestiona un estado propio *loading* que es clave que *ProtectedRoute* no tome decisiones de enrutado antes de saber si hay sesión válida.

5.4.4 Enrutado (React Router)

Las rutas de toda la aplicación se declaran en *App.tsx* que utiliza el componente de *React BrowserRouter* [Rem24] para delegar toda la parte del enrutamiento en el cliente y utiliza la API nativa *History* para sincronizar la interfaz de usuario con la barra de direcciones, per-

mitiendo una navegación fluida sin recargar la página completa. El patrón distingue rutas públicas (como el login) de rutas privadas, que se envuelven en el componente *ProtectedRoute*. Cada ruta privada declara además la clave de permiso (*requiredKey*) asociada a un rol que el usuario debe poseer para acceder. También incluye un mecanismo para redirigir cualquier ruta inexistente a la que se intente acceder mostrando un 404.

ProtectedRoute es el componente que materializa el control de acceso del cliente, comprobando si hay sesión de usuario cargada y si la sesión cargada es válida, en caso contrario redirige al formulario de login. En el caso de que el usuario tenga sesión iniciada correctamente redirige al menú principal (*dashboard).*

5.4.5 Roles y permisos

El control de acceso del *frontend* se centraliza en *permissions.ts*. En este archivo se define un mapa de permisos por rol que asocia un conjunto de permisos a cada uno de los tres roles existentes. Para cada rol se definen tres conjuntos de permisos que permiten granular las restricciones que se establecen para cada rol.

Por una parte, los componentes *dashboard* y *sidebar* definen para cada rol qué secciones son visibles en cada uno de estos componentes para poder acceder. Por otra parte, el conjunto *usersTable* define los campos de la tabla de usuarios que puede ver cada rol. De esta forma podemos hacer que la columna de acciones solo sea visible por el rol ADMIN para poder editar o eliminar mientras que un MANAGER solo podrá visualizar los datos de sus trabajadores sin realizar acciones sobre ellos.

En el mismo archivo permissions.ts se define una función *getPermissions()* la cual es el único punto de consulta del mapa de permisos, y es invocada por el *UserContext* cada vez que éste carga un nuevo usuario para almacenar los conjuntos de roles que corresponden al usuario autenticado. Desde este momento los permisos quedan accesibles para todo el árbol de componentes a través de la función *useUser().permissions*. De esta forma se consigue que haya una única fuente de verdad sobre los permisos y que la problemática sea resuelta de una forma coherente y robusta. El resto de componentes como el *dashboard, sidebar, protectedRoute*, *UserPage...* invocan esta función y añaden las validaciones necesarias para que se visualicen o sean accesibles solo los componentes que se correspondan con los roles del usuario.

5.4.6 Visualización del calendario

Para la visualización de calendarios desde el *backoffice* se integra la librería FullCalendar dentro del proyecto *frontend* para representar los eventos que se obtienen desde el API procedentes de *Google Calendar*.

La aplicación muestra dos versiones diferentes de calendarios según el rol del usuario:

- Un MANAGER ve el calendario de todos los trabajadores de la empresa asignando a

cada uno un color diferente para que se pueda diferenciar a simple vista.

- Un WORKER únicamente visualiza los eventos que se incluyen en su propio calendario.

La información recibida desde el *backend* es diferente en cada uno de los dos casos de visualización.

- Para el calendario de un trabajador el objeto de respuesta (*WorkerCalendarDto),* incluye el nombre del trabajador, el id del calendario y una lista de eventos a mostrar modelados en *CalendarEventDto.*

- Para el calendario de una empresa recibe una lista de objetos *WorkerCalendarDto.*

FullCalendar no entiende este formato de evento, espera objetos de tipo *EventInput* por lo que es necesario adaptar la información recibida del *backend* a las necesidades de *FullCalendar* para renderizarla. Este *mapping* de los dos modelos se aplica dentro de la función *fetchCalendar* de *CalendarPage*. Una vez los eventos están adaptados, se le pasan al componente que se encarga de generar la vista de calendario.

5.5 Módulo Google Calendar API

Para resolver la gestión de calendarios, eventos, recordatorios y la sincronización bidireccional de citas entre la agenda de la empresa o el trabajador y la del cliente, se ha optado por integrar *Google Calendar*. Esta sincronización con un calendario externo no es un mero añadido: el profesional necesita visualizar sus citas de forma clara y organizada para tener una foto nítida de su jornada, y el cliente poder integrarlas allí donde ya gestiona el resto de su día. Forzarles a utilizar una interfaz nueva para consultar sus agendas podría comprometer la adopción del sistema.

La elección de *Google Calendar* se justifica por tres motivos principales. En primer lugar, su penetración dominante: la mayoría de usuarios y PYMEs españolas del sector servicios ya utiliza *Gmail/Google Workspace* o cuentas *Google* personales, lo que reduce la fricción de *onboarding* a prácticamente cero. En segundo lugar, la disponibilidad de un SDK *Java* oficial y mantenido, que recibe actualizaciones regulares y cubre todo el flujo OAUTH y el CRUD de eventos sin necesidad de código a medida [Goo24b, Goo24a]. Y en tercer lugar, su compatibilidad multidispositivo, que permite al cliente o al profesional consultar su agenda desde web, Android o iOS sin esfuerzo adicional de desarrollo. A diferencia de otras herramientas del proyecto, la integración con la API de *Google Calendar* no resulta tan orgánica, si bien con algunas adaptaciones de código queda perfectamente unificada dentro del sistema como un módulo más.

Para modelar el sistema de trabajadores y citas dentro de *Google Calendar*, el sistema asocia un calendario propio en la cuenta de *Google* de su empresa a cada trabajador y cada cita se materializa como un evento dentro de ese calendario.

La comunicación se apoya en la librería oficial *Google API Client for* Java y se organiza siguiendo la arquitectura hexagonal del *backend*: el dominio define unos puertos (*CalendarPort*, *CalendarEventPort*) que describen *qué* operaciones de calendario existen, y la capa de infraestructura aporta los adaptadores (*GoogleCalendarAdapter*, *GoogleCalendarEventAdapter*) que traducen esas operaciones a llamadas concretas contra la API de *Google*. De esta forma conseguimos desacoplar toda la lógica de la API de *Google*.

Las clases principales que se han implementado para la integración dentro del Sistema son:

- **GoogleCalendarConfig**: crea los beans necesarios para la comunicación y autenticación con *Google* y los añade al contexto de *Spring*.
- **GoogleCalendarClientProvider**: construye y devuelve un cliente *Calendar* ya autenticado para una cuenta de empresa gestionando el refresco del *token* en caso de que sea necesario.
- **CalendarPort**: define las operaciones sobre calendarios como crear, eliminar, consultar eventos o periodos ocupados. Se implementa en **GoogleCalendarAdapter.**
- **CalendarEventPort**: define las operaciones sobre eventos. Se implementa en **GoogleCalendarEventAdapter.**

Antes de cualquier operación, el adaptador necesita un objeto Calendar de *Google* asociado a la cuenta correcta. *GoogleCalendarClientProvider.getCompanyCalendar(accountId)* se encarga de proveer este objeto, carga la credencial cifrada, la refresca si está a punto de caducar y construye el cliente. Este método es el único punto de entrada a *Google Calendar*. Tanto los adaptadores de calendario como los de eventos lo invocan al inicio de cada operación, de modo que la gestión de *tokens* (carga, descifrado y refresco) queda centralizada en un solo lugar. Las diferentes operaciones para la gestión de calendarios y eventos como un caso de uso más del dominio de forma que puedan ser reutilizados para dar soporte tanto al API de *backoffice* como al modelo MCP para operaciones comunes como la consulta de eventos.

Operaciones sobre calendarios y eventos.

Los puertos *CalendarPort* y *CalendarEventPort* declaran las operaciones disponibles a nivel de calendario y sus implementaciones (*Adapters*), siguen siempre el mismo patrón: obtener el cliente autenticado → invocar a la API de *Google* → mapear la respuesta de *Google* a un modelo de dominio. Estas son las operaciones que se pueden realizar:

- **Crear un calendario:** cuando se da de alta un empleado nuevo, *SaveWorkerUseCaseImpl* crea automáticamente su calendario y guarda el calendarId devuelto en BBDD asociado al registro del trabajador.
- **Consultar un calendario:** *getCalendarById()* devuelve la lista de eventos de un calendario comprendida en el intervalo temporal solicitado.

- **Consultar disponibilidad:** la operación más interesante a efectos de negocio es *getBusyPeriods()*, que usa el método *FreeBusy* de *Google*. En lugar de descargar todos los eventos, pregunta a *Google* qué intervalos están ocupados en varios calendarios de una sola vez. Esto es lo que permite al sistema calcular huecos libres para nuevas citas de forma eficiente.
- **Crear/Actualizar un evento:** el caso de uso construye un *EventModel* con los datos de la cita, lo envía a través del puerto a la API de *Google* para que quede guardado y registra en BBDD una nueva cita asociándole el eventId devuelto por *Google*.
- **Eliminar un evento:** cuando se cancela una cita, el caso de uso *CancelAppointmentUseCaseImpl* elimina el evento del calendario de *Google* y marca la cita en BBDD con el estado Cancelled.
- **Listar eventos:** *getEventList()* consulta a *Google* los eventos comprendidos en una venta temporal ordenados por fecha de inicio y los devuelve en su respuesta.

En el Apéndice B, sección B.4, se muestra un diagrama en el que se resumen los flujos para la gestión de calendarios y eventos en la integración de *Google Calendar*.

5.5.1 Autenticación con Google mediante OAuth 2.0

La parte más compleja de la integración con *Google Calendar* es la gestión de la autenticación de las cuentas de *Google* de las diferentes empresas, necesaria para poder operar sobre los calendarios en nombre de las mismas.

El sistema delega la autenticación frente a *Google* en el protocolo OAUTH 2.0 [Har12]. La integración se apoya en la librería oficial *Google API Client for Java* (*GoogleAuthorizationCodeFlow*), sobre la cual se ha construido una capa de persistencia propia que almacena los *tokens* cifrados en reposo dentro de la BBDD relacional del proyecto, empleando cifrado simétrico AES a través del módulo *spring-security-crypto*.. Un planificador de tareas (*scheduler*) complementa el flujo refrescando de forma proactiva los *tokens* próximos a expirar, garantizando que las operaciones de calendario no se interrumpan. Para disponer de esta funcionalidad es necesario registrar la aplicación con una cuenta de *Google* en *Google Cloud Console* para obtener las credenciales OAUTH y configurar algunas propiedades de nuestra aplicación.

El sistema necesita proporcionar un flujo de consentimiento para que el propio usuario pueda autorizar a la aplicación a modificar sus calendarios y eventos. Cuando un usuario (MANAGER o WORKER) hace login en el panel de administración del backend, si la cuenta de la empresa a la que pertenecen necesita aceptar el consentimiento, en lugar de redirigirlo al panel de administración el sistema devuelve una Uniform Resource Locator (URL) a la que tendrán que acceder para autorizar la aplicación.

Se puede consultar un diagrama que representa el flujo de autorización OAUTH2 en el

Apéndice B, sección B.5.

Refresco activo de tokens

Cuando un caso de uso necesita operar sobre el calendario, invoca *GoogleCalendarClientProvider.getCompanyCalendar(accountId)* el cual encierra el siguiente flujo:

1. Recupera y descifra la credencial de la cuenta.
2. Si el *access token* está próximo a caducar, se llama a *credential.refreshToken()*, que usa el *refresh_token* para pedir a *Google* un nuevo *access token* y, de nuevo, lo almacena cifrado sustituyendo al anterior.
3. Si el *refresh_token* ha sido revocado o ha expirado, se revocan los *tokens* locales y se lanza una *OAuthConsentRequiredException*. El *CommonExceptionHandler* la traduce a un HTTP 403 que incluye la URL de consentimiento, indicando al *frontend* que debe reiniciar el flujo de autorización.
4. En el caso de que las credenciales estén correctas, devolverá el objeto Calendar para que el caso de uso pueda operar sobre él.

Para evitar que el refresco dependa exclusivamente del tráfico de los usuarios (lo que podría dejar *tokens* caducados sin renovar durante largos periodos de tiempo), el proyecto incorpora un planificador proactivo: *TokenRefreshScheduler* que se encarga de refrescar automáticamente los *tokens* de las cuentas de las empresas registradas en el sistema. En cada ejecución calcula un umbral igual a *ahora + 10 minutos* y consulta las cuentas cuyos credenciales están próximos a caducar. Por cada cuenta, invoca *getCompanyCalendar(accountId)*, reutilizando exactamente la misma lógica de refresco descrita anteriormente. De este modo se reutiliza esa parte del código y el *scheduler* solo se encarga de seleccionar las cuentas y disparar el mecanismo de refresco ya existente. El método captura y registra (log) de los posibles errores por cuenta de forma aislada, de modo que el fallo de una credencial no interrumpe el procesamiento del resto.

5.6 Módulo IA

A lo largo de este documento se ha reiterado en la idea de que uno de los componentes diferenciales del proyecto era la integración con un LLM, en el que se delega gran parte de la interacción con el cliente, actuando a la vez como asistente conversacional y como orquestador de acciones de dominio. De este modo, el modelo funciona como una capa de razonamiento natural sobre la lógica de negocio existente, sin reemplazarla: asume la interpretación del lenguaje natural mientras el *backend* mantiene la integridad transaccional. Esta integración se realiza a través de *Spring AI* y en esta sección se describen los aspectos más relevantes para llevarla a cabo.

5.6.1 Spring AI

La integración con el LLM se realiza a través de *Spring AI*. Su uso desacopla el sistema por completo del modelo utilizado, permitiendo reutilizar la misma lógica de negocio independientemente del proveedor de LLM y facilita la implementación de las funcionalidades MCP que permiten la comunicación con el sistema. Como valor añadido, al formar parte del ecosistema *Spring* queda totalmente integrado en el proyecto de una forma orgánica y sencilla. La elección de este módulo, dota al proyecto de una mayor flexibilidad y escalabilidad de cara a líneas de trabajo futuras. Gracias a esta abstracción de proveedores, el mismo código puede funcionar con *OpenAI*, *Claude*, *Gemini* o modelos locales aplicando cambios mínimos de configuración. Como contrapartida, *Spring AI* presenta los inconvenientes propios de una herramienta joven: una comunidad y una documentación todavía más reducidas que las de otros módulos consolidados del ecosistema *Spring*.

Spring AI aporta tres piezas clave que el proyecto utiliza:

1. ***ChatClient*:** un cliente genérico e independiente del modelo para enviarle prompts y recibir su respuesta, sin tener que gestionar peticiones HTTP, serializaciones ni formatos acoplados al LLM.
2. ***ChatMemory:*** implementa la *memoria de la conversación,* para que el modelo *recuerde* los mensajes anteriores de cada cliente dentro de la misma conversación.
3. ***Tool calling:*** implementa la parte MCP. Proporciona al modelo un conjunto de herramientas para invocar métodos *Java* del sistema y de esta forma interactuar con él para realizar acciones o recuperar información que le permitan razonar y generar una respuesta basada en los datos reales del sistema.

5.6.2 Configuración de Spring AI

Lo primero que *Spring AI* necesita para poder operar es saber el modelo con el que va a trabajar y un API-key de conexión con el mismo. Estos valores se configuran en el proyecto a nivel de properties en las siguientes:

spring.ai.openai.api-key: OPENAI_API_KEY

spring.ai.openai.chat.options.model: OPENAI_MODEL

En el proyecto los valores se almacenarán como variables de entorno por seguridad.

Además de estas propiedades, se declaran dos clases que encapsulan toda la configuración relativa a la integración de *Spring AI* en el proyecto: *PromptConfig* y *SpringAiConfig.*

PromptConfig: el *prompt* de sistema no está incrustado en el código sino que se guarda en un fichero de recursos src/main/resources/prompt/prompt.st. *PromptConfig* simplemente carga ese fichero como un bean en el contexto para que esté disponible en el resto de componentes.

SpringAIConfig: en esta clase se produce el ensamblaje de las piezas necesarias dentro de un único *ChatClient* reutilizable listo para ser inyectado en cualquier caso de uso con *prompt*, memoria y herramientas para consulta del modelo ya configurados. Las partes más destacables de esta clase son:

- *defaultSystem(systemPrompt):* fija el *prompt* de sistema cargado por *PromptConfig* como instrucción base de toda conversación.
- *defaultAdvisors(MessageChatMemoryAdvisor...):* incluye un interceptor que, antes de cada llamada, recupera el historial de la conversación y lo añade al prompt.
- defaultTools(mcpToolService): registra el objeto que expone las herramientas que el modelo podrá invocar para interactuar con el sistema.

El bot puede atender a muchos clientes a la vez, cada uno con su propia conversación. Por esta razón la memoria no puede ser global, debe estar segmentada por conversación. *Spring AI* resuelve esto con dos conceptos:

- *ChatMemory*: la abstracción que almacena los mensajes.
- *CONVERSATION_ID* : un identificador que distingue cada conversación dentro de la memoria.

Para intentar optimizar el tamaño del contexto, la memoria conserva solo los últimos 10 mensajes de cada conversación, descartando los más antiguos. Así el contexto enviado al modelo se mantiene acotado (controlando coste y longitud) sin perder el hilo reciente.

5.6.3 Herramientas de consulta para LLM (MCP)

Un LLM por sí solo no conoce los servicios o los horarios del establecimiento. Para dar este soporte al LLM, el proyecto expone un conjunto de herramientas (implementadas como métodos *Java*) que el modelo puede decidir invocar durante su razonamiento. Es el patrón *tool calling*.

En *Spring AI* basta con anotar métodos con @Tool y añadir una descripción en lenguaje natural que ayude al modelo a identificar cómo y cuándo utilizar ese método. Esa descripción es la que *lee* el modelo para decidir. En el proyecto todas estas herramientas se encuentran en la clase *McpToolService*.

Algunos aspectos destacables del diseño:

- *Spring AI* genera el esquema de cada herramienta automáticamente a partir de la firma del método y de la descripción. Ese esquema es lo que se envía al modelo para que sepa qué herramientas tiene y cómo llamarlas.
- Las herramientas son adaptadores de entrada hacia el dominio. Cada método delega en un *caso de uso* (*SaveAppointmentUseCase*, *GetAvailableSlotsUseCase*...) y no contiene lógica de negocio.

- El *tool calling* es lo que hace *inteligente* al asistente. El modelo puede, dentro de un mismo turno, encadenar varias antes de redactar su respuesta, y todo ello lo gestiona *Spring AI* de forma transparente para el caso de uso, que solo ve la entrada (mensaje del cliente) y la salida (respuesta del LLM).
- Distinción entre lecturas y escrituras. Las consultas usan el método *runTool* y devuelven directamente la información solicitada. Las operaciones que modifican el estado del sistema como reservas o cancelaciones de citas utilizan el método *runWriteTool*, que envuelve el resultado en un objeto de tipo *ToolResult* que devuelve códigos de retorno categorizados para operaciones exitosas o las posibles excepciones que se hayan podido capturar en la ejecución del método para informar al modelo del resultado de la acción y poder reaccionar en caso de que se produzca algún error.

En el Apéndice D se puede consultar una lista completa de las herramientas que el sistema ofrece para la interacción con el LLM

5.6.4 Caso de uso: orquestación del flujo de chat

El corazón del flujo es *SendChatMessageUseCaseImpl*. Recibe el mensaje del cliente y la información del usuario y orquesta todo el flujo hasta que devuelve la respuesta del modelo al cliente.

1. **Resolución de contexto**: se localiza la empresa por su companyId (obtenido durante la autenticación por API key) y al cliente. Si el cliente no existe aún, se registra automáticamente en el sistema mediante *getCustomerOrCreate*().
2. **Persistencia**: tanto el mensaje del cliente como la respuesta del modelo se guardan vía *MessagePort*.
3. **Incorporación del *prompt* de sistema:** el método *.system()* inyecta los valores correspondientes en *placeholders* de la plantilla del *prompt*. Con esto se consigue un *prompt* personalizado en cada llamada.
4. El mensaje del cliente se añade con *.user(customerMessage)*.
5. La conversación se identifica con *CONVERSATION_ID*, lo que conecta esta llamada con la memoria del cliente correspondiente.
6. *.call().content()* ejecuta la llamada al LLM de forma síncrona y extrae el texto final de la respuesta que se devolverá al cliente.

5.6.5 Síntesis de componentes Spring AI

- *spring-ai-starter-model-openai + spring-ai-bom:* dependencias que habilita *Spring AI* y autoconfiguran *ChatClient*.
- *application.yml (spring.ai.openai):* propiedades de configuración donde se definen el modelo a utilizar y el API-KEY necesario para la conexión.

- *PromptConfig:* clase de configuración que carga el *system prompt* como **bean**.
- *prompt.st:* plantilla que encierra todo el contenido del *system prompt*.
- *SpringAIConfig:* clase de configuración que enlaza *ChatClient* con *system prompt*, memoria y las herramientas MCP.
- *ChatMemory: bean* que gestiona la memoria de conversación.
- *McpToolService (@Tool):* Anotación que expone los métodos *Java* que el modelo puede invocar.

El LLM elegido para el desarrollo de este proyecto ha sido *OpenAI - GPT*. Para el contexto en el que se desarrolla esta solución, cualquiera de los LLMs más importantes actualmente podrían haber obtenido un resultado bastante bueno ya que no requiere de tareas muy complejas o específicas cómo el Deep research de documentación, generación de imágenes, etc.

La elección de GPT frente a otros modelos igualmente potentes como *Claude (Anthropic)* o *Gemini (Google)* se apoya en su carácter generalista y en varias características que lo hacían especialmente idóneo para el proyecto. En primer lugar, ofrece una calidad de generación superior, pues está considerado uno de los modelos más capaces del mercado en comprensión semántica, razonamiento contextual y generación de texto coherente en español. A ello se suma la madurez y estabilidad de su API, una de las más consolidadas del sector, con documentación completa, SDKs oficiales y alta disponibilidad. Por último, su soporte para *function calling* y herramientas facilita la construcción de agentes capaces de invocar funciones del *backend* de forma estructurada, algo esencial para escenarios como la gestión de citas. Aunque tanto *Claude* como *Gemini* ofrecen una calidad comparable, GPT cuenta además con una integración de primera clase con *Spring AI* y un ecosistema de herramientas más amplio. Por otra parte, *OpenAI* fue el pionero en estandarizar el patrón de *function-calling*, ampliamente adoptado por la industria y clave en el desarrollo que nos ocupa, y permite configurar parámetros como la *temperatura* y el *esfuerzo de razonamiento* del modelo, muy útiles para ajustar el comportamiento esperado en sus respuestas.

5.6.6 Prompt

El núcleo del comportamiento del asistente conversacional reside en el *prompt* que se envía al LLM. En el proyecto se utiliza un *prompt* bien estructurado modelado como una plantilla parametrizada que *Spring AI* rellena en tiempo de ejecución con el contexto de cada conversación. Este *prompt* se ha diseñado teniendo en cuenta los conceptos CLEAR y RASCEF, dos de los marcos de trabajo y buenas prácticas más utilizados actualmente en la *ingeniería de prompts* para optimizar la comunicación con LLMs. La convergencia de ambos es deliberada, mientras RASCEF (Role – Action – Steps – Context – Examples – Format) describe la estructura del *prompt* (qué piezas debe contener) [Viv25], CLEAR (Concise –

Logical – Explicit – Adaptive – Reflective) define los *criterios de calidad* que cada pieza debe cumplir [Lo23]. Se puede ver de forma más detallada la aplicación de ambos conceptos en el *system prompt* en el Apéndice C.

Siguiendo estos conceptos, el *prompt* generado se articula en bloques delimitados con encabezados Markdown, lo que facilita su lectura tanto por el modelo como por quien lo mantiene:

1. ROLE: define la identidad y el propósito del agente.
2. CORE RULES: políticas transversales de tono, idioma, privacidad y veracidad.
3. INITIALIZATION: acciones silenciosas en la primera interacción.
4. DATE & TIME HANDLING: reglas deterministas para resolver expresiones relativas de tiempo.
5. APPOINTMENT WORKFLOWS: procedimientos numerados paso a paso para reservar, modificar y cancelar citas.
6. ERROR HANDLING: reacción estructurada según el código de error devuelto por cada herramienta.
7. HARD CONSTRAINTS: restricciones no negociables para evitar conversaciones que se salgan del dominio de negocio, *alucinaciones* del modelo o mostrar información propia del sistema al cliente.
8. RUNTIME CONTEXT: definición de las variables dinámicas para el contexto.

Fortalezas del diseño

- Anclaje del contexto en un único bloque (*single source of truth*). Todas las variables dinámicas se concentran en RUNTIME CONTEXT, al final del prompt, y el modelo recibe la orden explícita de no inventarlas ni inferirlas. Esto reduce las alucinaciones sobre datos sensibles del negocio.
- Prevención activa de alucinaciones. Reglas redactadas en negativo y reforzadas (Never affirm… before the relevant tool has been called) obligan a que toda afirmación sobre disponibilidad, precios o confirmaciones esté respaldada por una llamada real a una herramienta cuyo resultado se haya leído.
- Orientación a la acción y eficiencia conversacional. El flujo de reserva está diseñado para proponer, no interrogar, minimizando las interacciones del cliente.
- Manejo de errores determinista. Cada código tiene una respuesta predefinida, lo que hace el comportamiento predecible y auditable.
- Separación entre representación interna y presentación. El *prompt* prohíbe exponer IDs, formatos ISO o mecánicas del sistema, garantizando una experiencia de usuario natural.

- Modularidad y mantenibilidad. La segmentación en secciones permite modificar una política sin afectar al resto.

5.7 Módulo ChatBot

Uno de los *leitmotivs* del proyecto es ofrecer al cliente una interfaz completamente integrada en su cotidianidad, y la mejor forma de lograrlo es incorporarla a una aplicación de mensajería instantánea que ya utilice a diario. Entre las dos plataformas líderes del sector, *WhatsApp* y *Telegram*, se ha optado por *Telegram* como punto de partida por la flexibilidad y las facilidades que ofrece: su API es abierta, gratuita y no requiere un proceso de aprobación [Tel24], lo que la hace mucho más viable para un primer despliegue real y las fases tempranas del sistema. Un desarrollo sobre *WhatsApp*, en cambio, exigiría crear una cuenta de *WhatsApp Business*, asociarla a un número de teléfono real e incluso podría conllevar costes por conversación.

Para la implementación se ha empleado un lenguaje distinto al del resto del *backend*, *Python*, apoyándose en las librerías de bots del propio *Telegram*, con las que se obtiene un resultado muy competente con muy pocas líneas de código y que incluyen los mecanismos necesarios para las llamadas HTTP asíncronas, idóneas para comunicarse con el *backend* [pyt24]. Existe también la posibilidad de implementar bots de *Telegram* en *Java* mediante *TelegramBots*, lo que mantendría la homogeneidad de lenguaje con el resto del desarrollo; sin embargo, su comunidad es notablemente menor y sus actualizaciones suelen ir por detrás de los cambios de *Telegram*, una penalización en velocidad de iteración que no compensa la ventaja de unificar el lenguaje, máxime cuando *Python* es de muy fácil adaptación.

Gracias al desacoplamiento entre la interfaz de mensajería y la lógica de negocio, el sistema podría incorporar otros canales de comunicación (como un bot de *WhatsApp*) e incluso mantener varios de forma simultánea, dejando que sea el propio cliente quien elija, sin impactar en las gestiones de negocio ni requerir cambios en el *backend*.

El bot de *Telegram* se ha implementado en *Python* que actúa como *pasarela* entre los usuarios de la aplicación de mensajería (clientes) y el backend del proyecto, de esta forma queda totalmente desacoplada la interfaz de *Telegram* de la lógica de negocio. Su responsabilidad es sencilla y bien delimitada: recibir los mensajes de texto que los usuarios envían por *Telegram*, reenviarlos al API-REST del backend a través del endpoint de chat y devolver al usuario la respuesta que recibe del API.

Para la autenticación del bot en el sistema, se utiliza un esquema diferente al de la autenticación de usuarios, más simple y propio de la comunicación máquina a máquina: la autenticación mediante API-KEY. Cuando el administrador del sistema da de alta una empresa, el formulario incluye un campo para establecer el API-KEY que corresponde al bot de dicha empresa. De esta forma se consigue securizar la llamada del bot y asociarlo de forma

única e independiente a una empresa. En cada petición al endpoint de chat, el bot envía el API-KEY y, de forma similar a como se gestiona la autenticación de usuario, existe un filtro que actúa de interceptor (*BotApiKeyFilter*), registrado en la cadena de *Spring Security* que valida el API-KEY.

Los puntos principales de diseño del bot son la asincronía (puede atender a varios clientes sin quedarse a la espera de una respuesta por parte del API), la seguridad (credenciales en variables de entorno y autenticación por clave de API) y la separación de responsabilidades (cada función tiene un cometido único y bien definido).

El script se apoya en dos librerías principales:

- **python-telegram-bot:** gestiona toda la comunicación con la plataforma de *Telegram* (recepción de mensajes, envío de respuestas).
- **aiohttp**: cliente HTTP asíncrono que permite comunicarse con la API interna sin bloquear el programa [aio24].

El archivo está organizado en cuatro bloques claramente diferenciados: *configuración*, *helpers* (funciones auxiliares), *handler* (manejador de mensajes) e *inicialización*.

Configuración:

Se define un sistema de *logging* para el registro de eventos que registra cada acción relevante que realiza el script.

También se configuran tres variables que se utilizarán en el resto del archivo:

- **BOT_TOKEN:** *token* del bot registrado en *Telegram*.
- **BOT_API_KET:** API-key de autenticación para las llamadas al *backend*.
- **API_URL:** URL del endpoint de chat en el *backend*.

5.7.1 Funciones auxiliares

build_user_info: extrae los datos del usuario de *Telegram* y los empaqueta en conjunto clave-valor para enviarla en las llamadas al API-REST.

post_to_api: es la función que gestionar el flujo comunicación con el *backend*. Realiza una petición HTTP *POST* de forma asíncrona, incluyendo la clave de API en una cabecera de autenticación (*X-Bot-Api-Key*), y procesa la respuesta.

5.7.2 El handler

La función *chat_response* es el flujo principal del bot. Se ejecuta automáticamente cada vez que llega un mensaje de texto siguiendo estos pasos:

1. Muestra el indicador *"escribiendo..."* (typing) para mejorar la experiencia de usuario mientras se procesa la petición.

2. Construye el *payload* (carga útil) con el texto del mensaje y la información del usuario.
3. Envía ese *payload* a la API y espera la respuesta.
4. Reenvía la respuesta al usuario en *Telegram*.

5.7.3 Inicialización

La función *main* ensambla y arranca la aplicación. Registra un *MessageHandler* con un filtro que indica que solo deben procesarse mensajes de texto que *no* sean comandos. Finalmente, arranca el bot en modo polling, en el que el bot consulta periódicamente al servidor de *Telegram* si hay mensajes nuevos.

5.8 Construcción y despliegue

5.8.1 Construcción

Igual que en todo el proyecto el *backend* actúa como core y orquestador del resto de componentes, juego el mismo papel para la fase de construcción del artefacto final centralizando las construcciones y ejecuciones de todos los módulos del sistema. Obviamente, es el responsable de generar el artefacto del sistema *backend* pero en lugar de compilar el *frontend* y el *backend* por separado, con un único comando de *Maven* [The24a] (*mvn clean package*) se consigue generar el producto final completo. Esto es posible gracias a la combinación de tres plugins configurados en el *pom.xml* del *backend*, que se ejecutan de forma encadenada durante el ciclo de vida de *Maven*.

La *compilación* del proyecto *frontend* se delega a *Maven* mediante el *frontend-maven-plugin*, que ejecuta tres pasos automáticos:

1. *install-node-and-npm*: descarga e instala una versión concreta y aislada de *Node.js* y *npm*, de modo que el build no depende de que la máquina tenga Node instalado.
2. *npm install*: instala las dependencias del proyecto *frontend*.
3. *npm run build*: compila la interfaz con Vite, generando los archivos estáticos optimizados.

Con *maven-resources-plugin* se copia el resultado del *frontend* dentro de la carpeta de recursos estáticos del *backend* (/static). De esta forma la interfaz queda embebida dentro de la propia aplicación *Java* y el servidor expone ambas aplicaciones.

Finalmente, *Maven* empaqueta el *backend* compilado junto con el *frontend* ya incrustado en un único archivo ejecutable .jar

Para la ejecución del bot de *Telegram*, se hace uso de la clase *PythonBotService*, la cual ejecuta el método *startPythonBotScript()* al arrancar la aplicación creando un nuevo proceso del sistema operativo que ejecuta el script como un proceso hijo en segundo plano que finalizará también al detener la aplicación *backend*.

5.8.2 Despliegue

Para la operativa de despliegue de la aplicación se ha optado por una de las prácticas más habituales hoy en día, la *contenedorización*. Consiste en empaquetar una aplicación junto con todas sus dependencias en una unidad aislada y portable llamada contenedor [Doc24]. Esto garantiza que el proyecto se ejecute de forma idéntica en cualquier máquina. En este proyecto este proceso se realiza gracias a dos piezas complementarias:

- **Dockerfile**: define *cómo se construye* la imagen de la aplicación.
- **docker-compose.yml**: define cómo se orquestan y comunican los distintos servicios necesarios para la ejecución de la aplicación.

DockerFile

El Dockerfile utiliza una técnica llamada construcción *multi-etapa*, que separa el proceso en dos fases con objetivos distintos.

- **Etapa 1 - Compilación:** parte de una imagen con *Maven y Java 21* y ejecuta la orden *mvn clean package* para generar el artefacto .jar que incluirá tanto la aplicación *backend* como *frontend*.
- **Etapa 2 - imagen final (ejecución)**: parte de una imagen mínima de *Java*, mucho más ligera porque no incluye *Maven* ni las herramientas de compilación. En ella se realiza:
  1. La instalación de *Python* y las dependencias del script necesarias para la ejecución del bot.
  2. La copia del .jar y de los scripts generados en la etapa de compilación.
  3. Expone la aplicación en el puerto 8080 y la define como *entrypoint* la orden de arranque que ejecuta la aplicación *Java*.

Docker-compose

Se encarga de la orquestación que coordina el conjunto del sistema. En él se definen dos servicios (los contenedores, uno para la BBDD y otro para la aplicación), una red (cómo se comunican entre ellos) y un volumen (dónde se guardan los datos).

Al ejecutar la orden *Docker compose up* se realizan secuencialmente una serie de acciones que concluirán en una aplicación en ejecución completamente funcional:

1. ***networks***: crea una red privada virtual a la que se conectarán ambos servicios para comunicarse entre sí.
2. ***volumes:*** crea el volumen para la persistencia de BBDD. Este volumen es independiente del contenedor y de su ciclo de vida.
3. Servicio ***db***: descarga una imagen ya generada de *PostgreSQL*, la conecta con el volumen para la persistencia de datos e inyecta las credenciales y el nombre de la BBDD

para la inicialización del primer arranque.

4. Servicio ***app***: construye su propia imagen con el Dockerfile, espera a que db esté lista, inyecta todas las variables de entorno definidas en el archivo y expone al exterior la aplicación en el puerto 8080.

Capítulo 6

Resultados

En este capítulo se recogen los resultados obtenidos durante el proceso de desarrollo del TFG y que son fruto de la aplicación de todo el contenido expuesto en los capítulos anteriores.

6.1 Código de la aplicación

El código generado a lo largo de todo el desarrollo para implementar la solución propuesta se encuentra disponible al completo para su consulta en el siguiente repositorio: https://github.com/FedericoTgv/ftgv_tfg2026

Contiene la implementación de todos los módulos que se han definido a lo largo de todo este documento junto con una batería completa de test unitarios. El código de los componentes principales se encuentra en las siguientes rutas:

Backend: *./tfg_backend*

Frontend: *./tfg_frontend*

Bot: *./tfg_backend/src/main/resources/scripts/bot.py*

Prompt: *./tfg_backend/src/main/resources/prompt/prompt.st*

6.2 Historias de usuario

A continuación, se muestran los flujos resultantes de las principales historias de usuario definidas para el sistema:

6.2.1 Usuario Backoffice

Login Usuario Backoffice (HU01)

El usuario entra al panel de administración de *backoffice* y accede al formulario de *Login* (Figura 6.1).

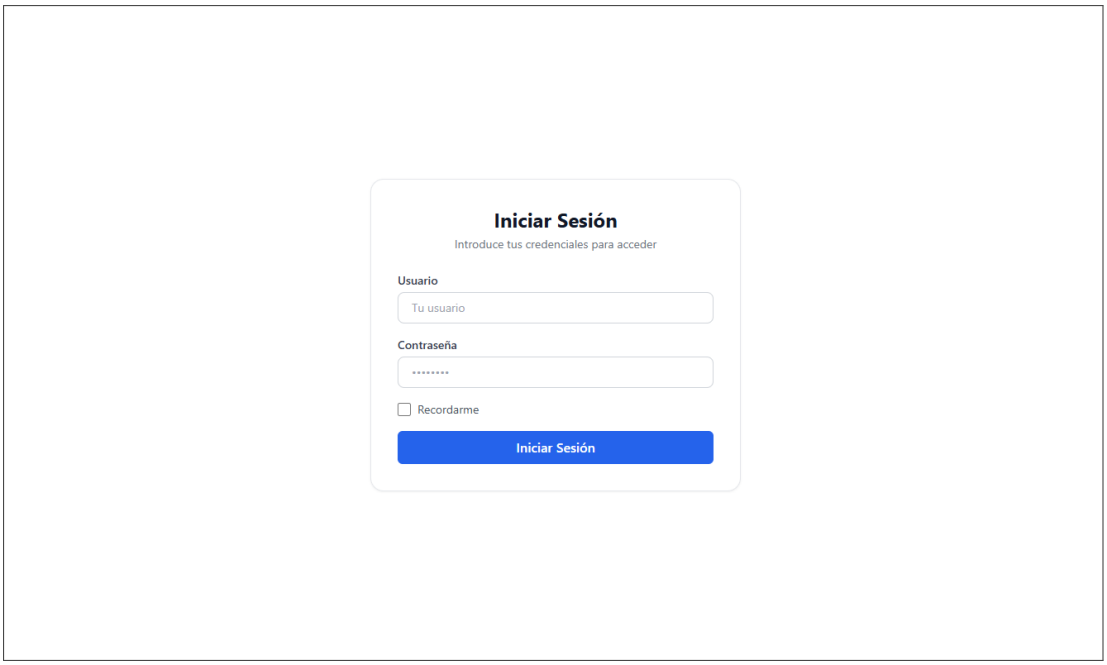


Figura 6.1: Pantalla Login Usuario

Gestionar empresas (HU03)

Si el usuario autenticado tiene el rol de ADMIN, la primera pantalla que encontrará será el menú principal (Figura 6.2) con las opciones correspondientes a su rol, *Usuarios* y *Empresas*.

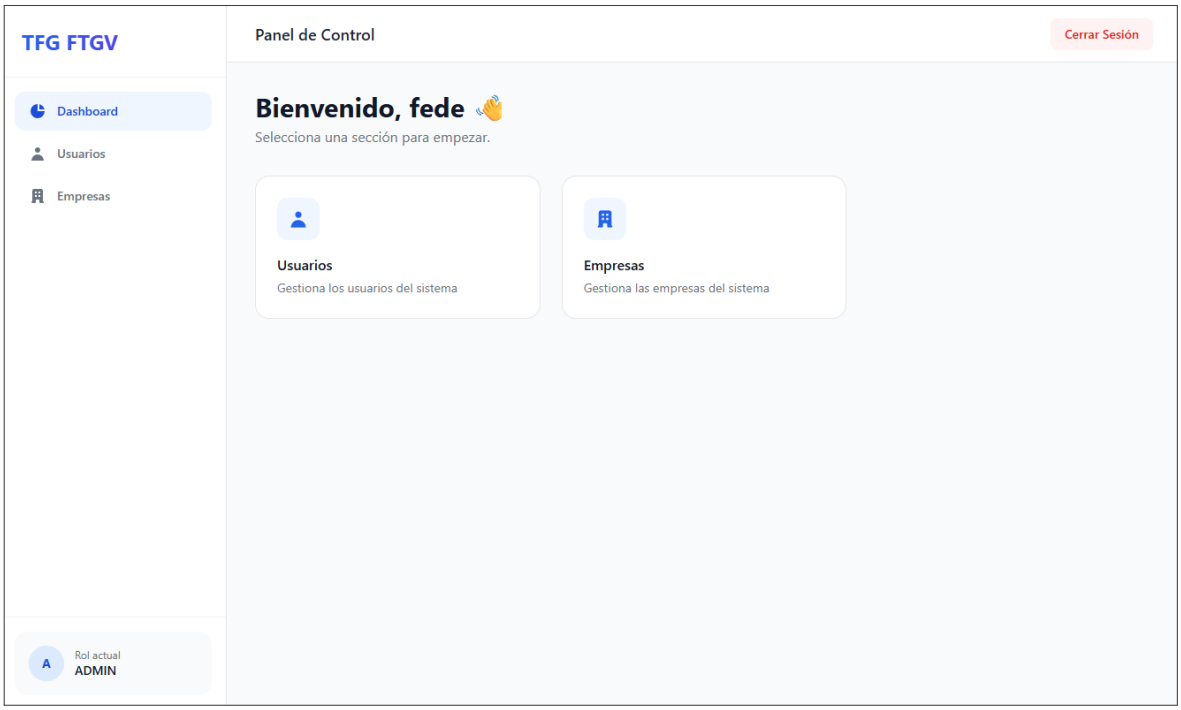


Figura 6.2: Pantalla Menú Principal ADMIN

Al acceder a la página de *Empresas* podrá ver un listado completo de todas las empresas registradas en el sistema junto con una opción de *añadir nueva empresa* o *editar/eliminar* usuarios existentes, como se muestra en la Figura 6.3. Pinchando en el botón de *Nueva Empresa* o en el de *editar* de alguna de las empresas listadas, se accede al formulario de *Empresas* (Figura 6.4).

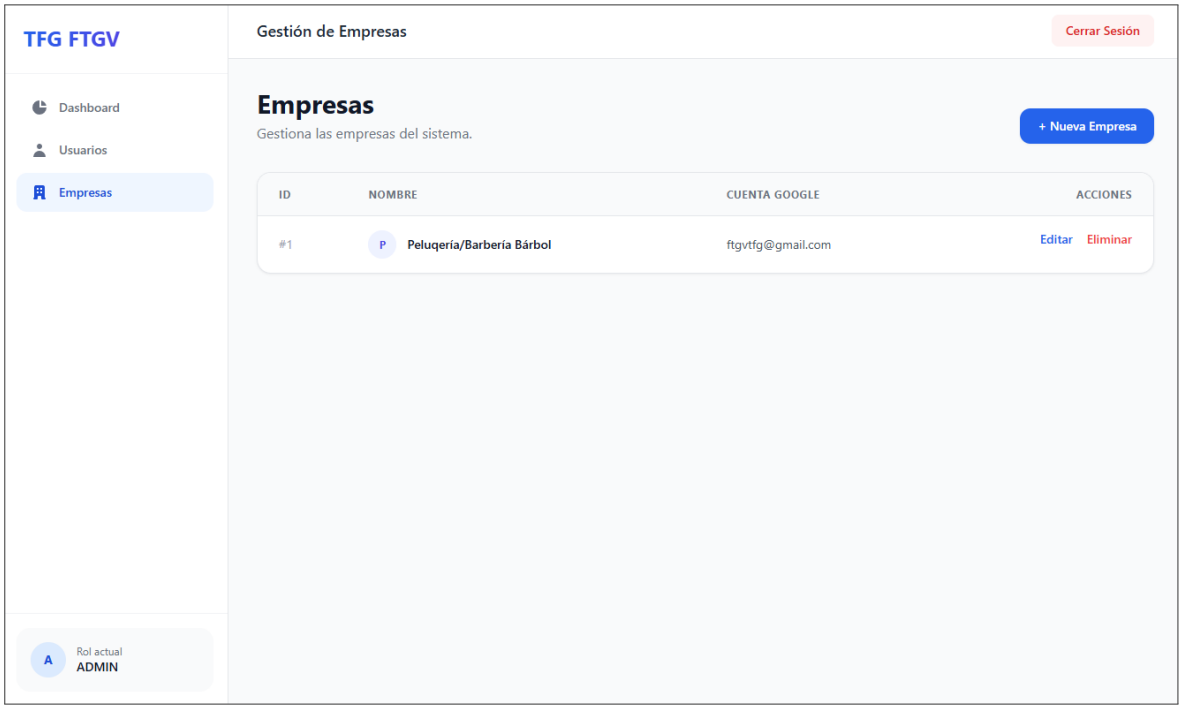


Figura 6.3: Pantalla Empresas

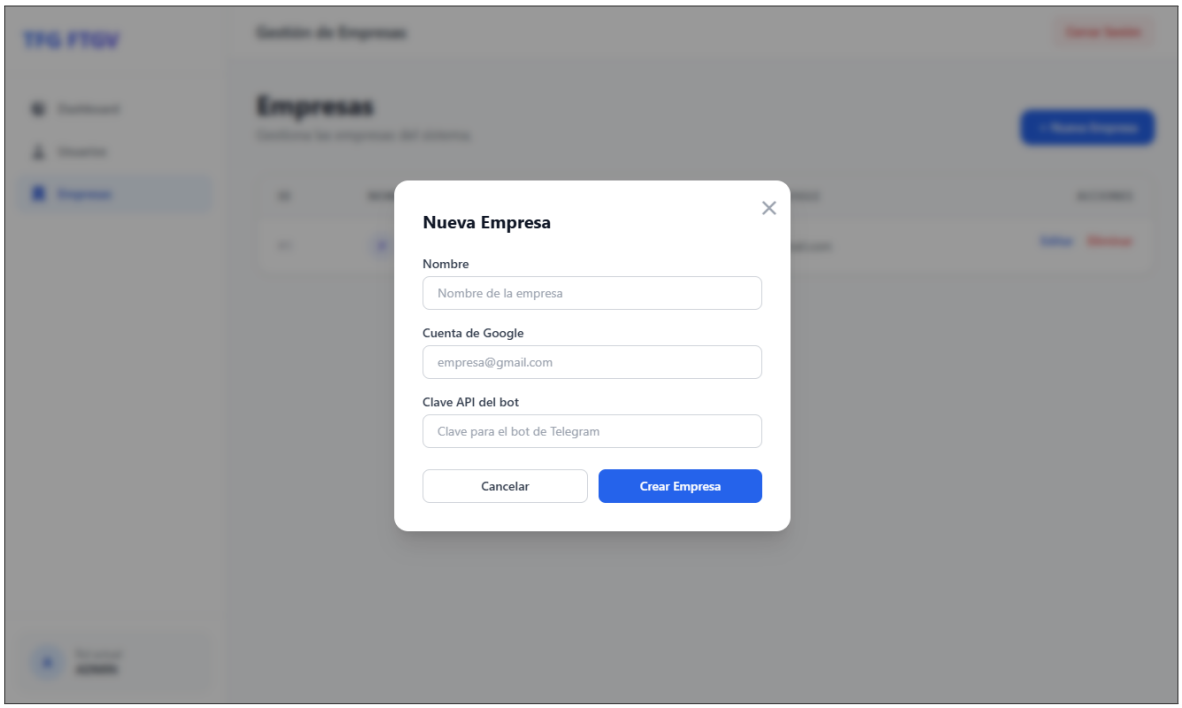


Figura 6.4: Pantalla Formulario empresas

Gestionar usuarios (HU04)

Al acceder a la página de *Usuarios*, se visualiza un listado de todos los usuarios registrados en el sistema junto con las opciones de acción, en la Figura 6.5) se puede ver el resultado de esta página. Pinchando en el botón de *Nuevo Usuario* o en el de *editar* de alguno de los usuarios listados, podrá acceder al formulario de *Usuarios* (Figura 6.6).

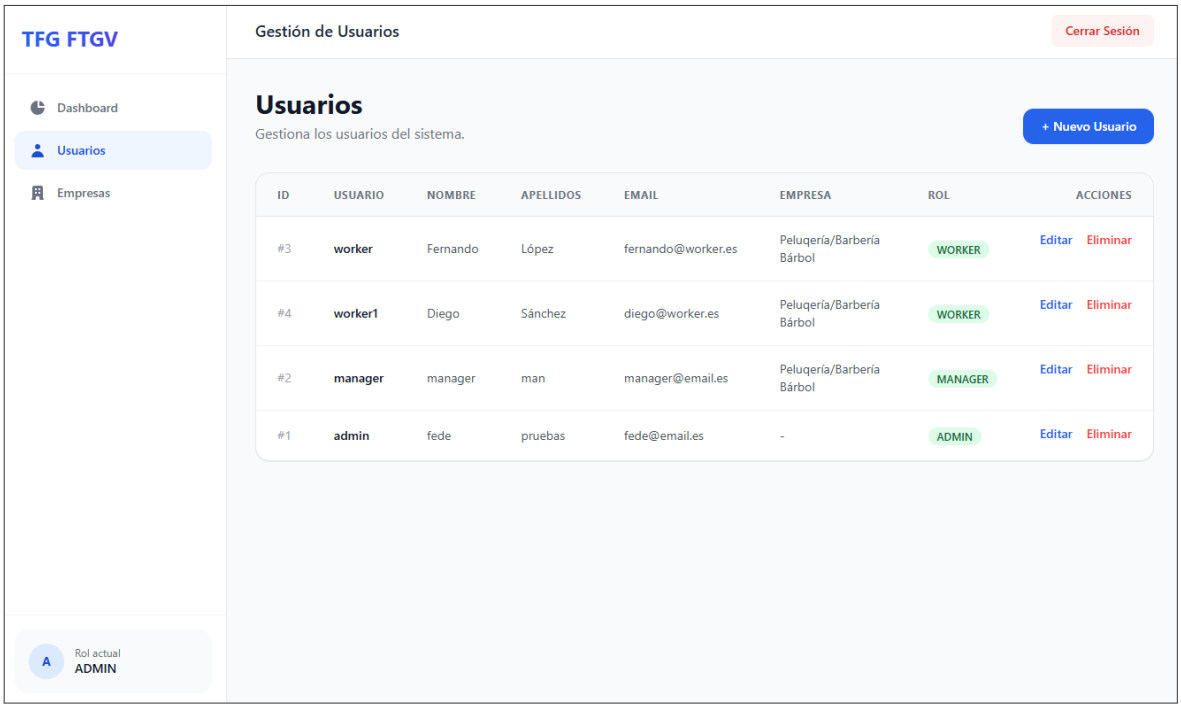


Figura 6.5: Pantalla Usuarios

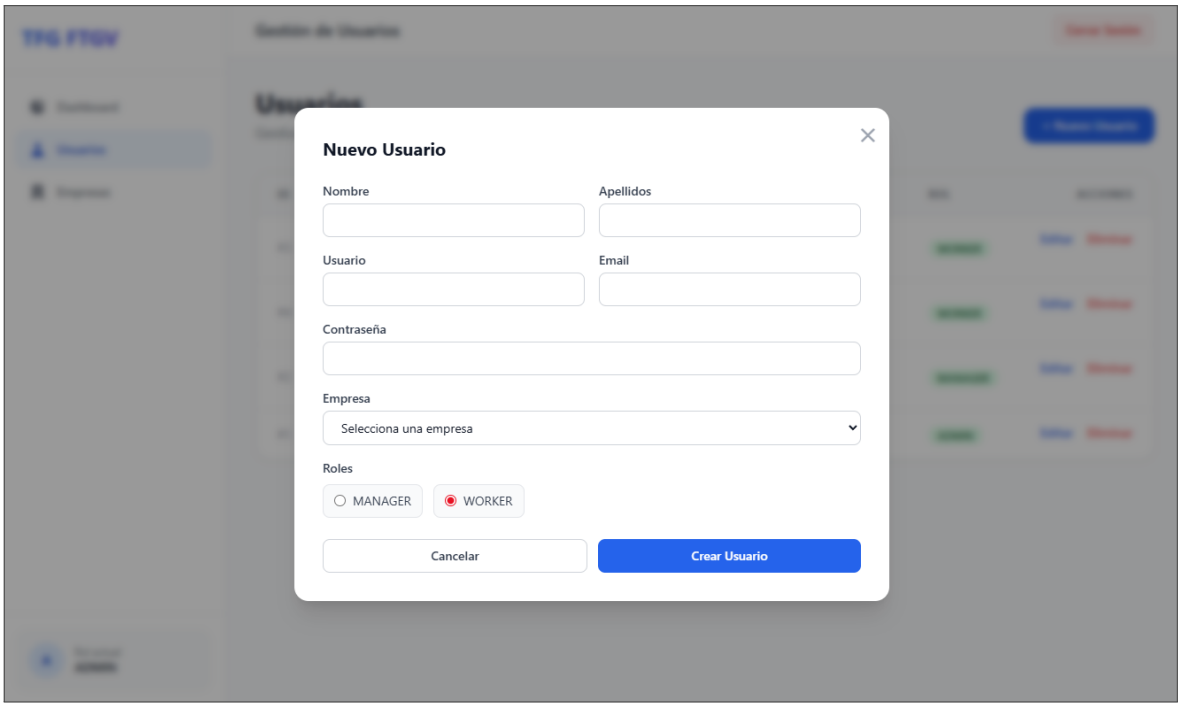


Figura 6.6: Pantalla Formulario Usuarios

Visualizar trabajadores (HU05)

Si el usuario autenticado tiene el rol de MANAGER, la primera pantalla que encontrará será el menú principal con las opciones correspondientes a su rol (Figura 6.7). La opción de *Trabajadores*, muestra la tabla de trabajadores asociados a su empresa, pero sin poder realizar ninguna acción sobre ellos (Figura 6.8).

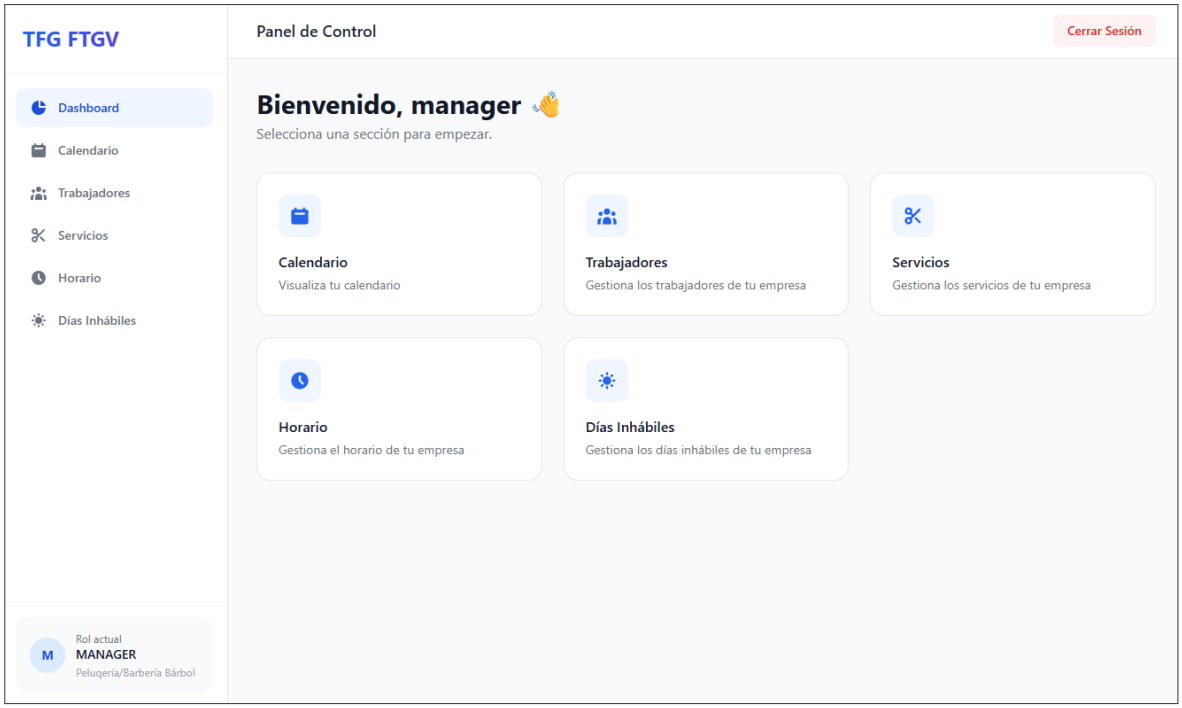


Figura 6.7: Pantalla Menú Principal MANAGER

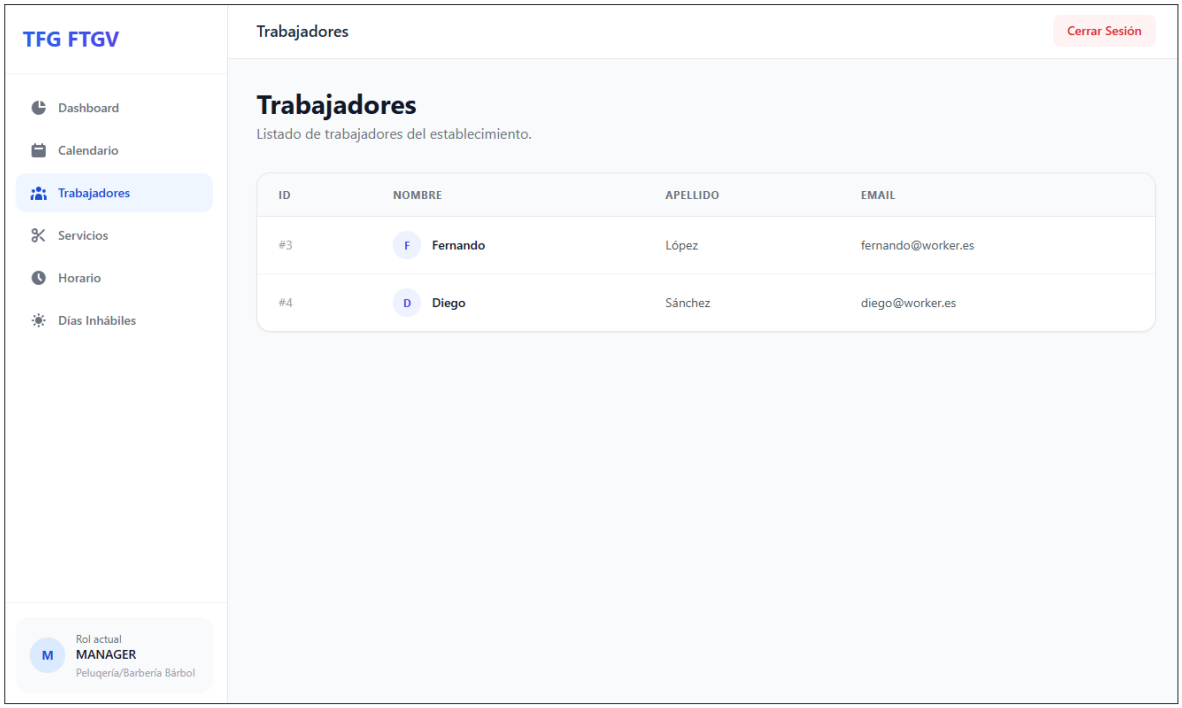


Figura 6.8: Pantalla Trabajadores

Gestionar servicios (HU06)

Otra opción que muestra el menú principal del panel de administración para MANAGER es la de *Servicios,* en la que gestionar los servicios que ofrece su empresa. La Figura 6.9 muestra la pantalla de visualización de servicios mientras que la Figura 6.10 muestra el formulario de creación y edición de servicios.

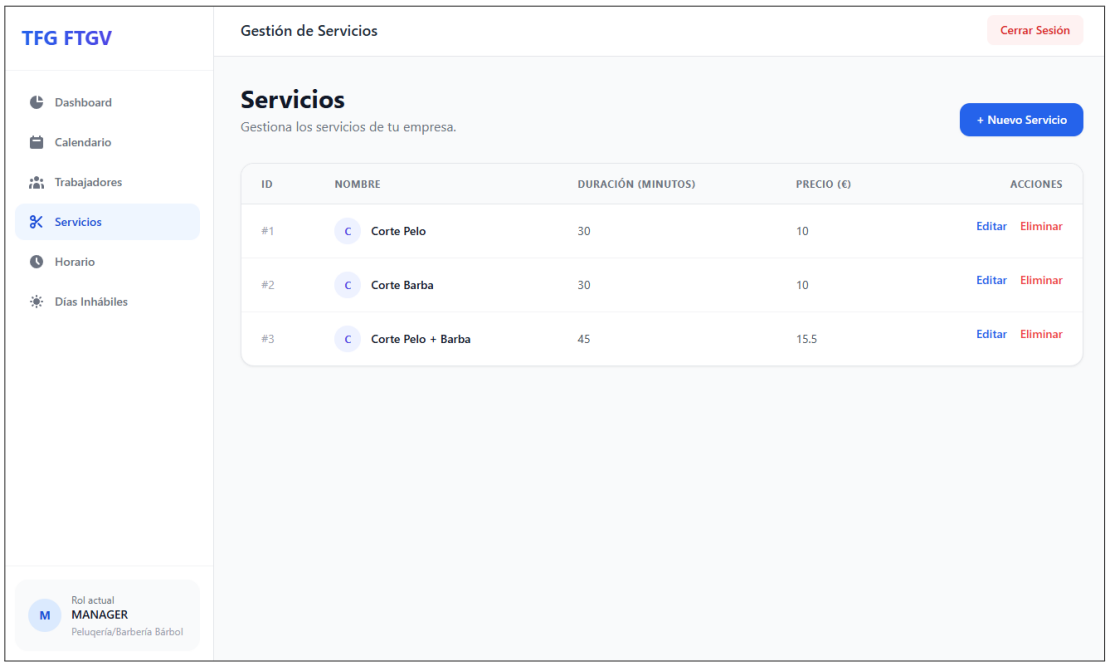


Figura 6.9: Pantalla Servicios

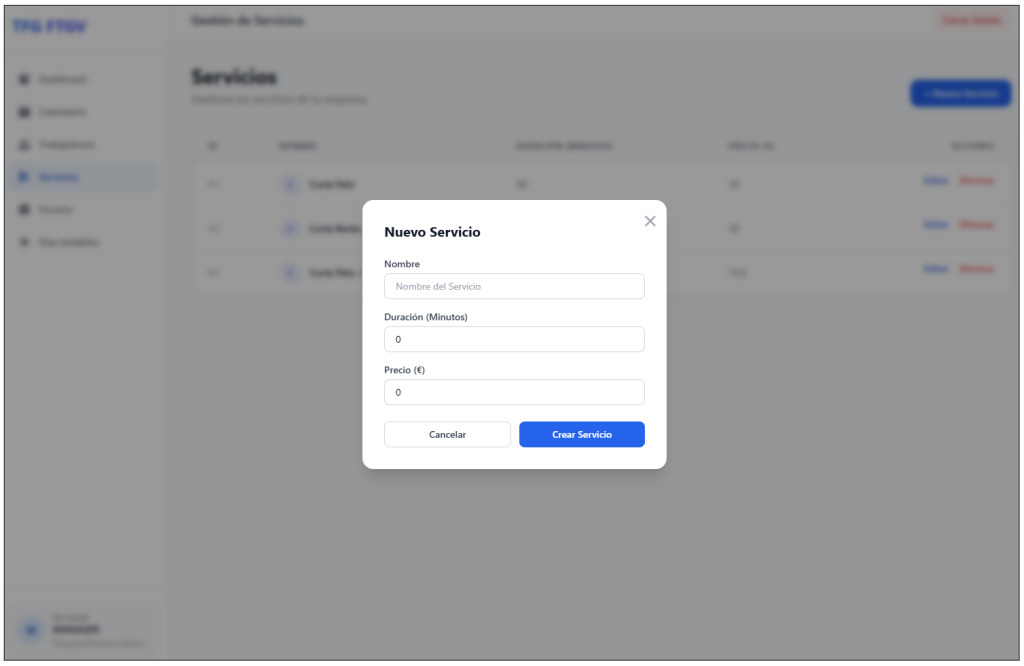


Figura 6.10: Pantalla Formulario Servicios

Gestionar horarios (HU07)

Otra de las características del negocio que un MANAGER puede gestionar son los *Horarios* de su empresa. Las Figuras 6.11 y 6.12 muestran la pantalla de Visualización y el formulario de creación/edición de horarios respectivamente.

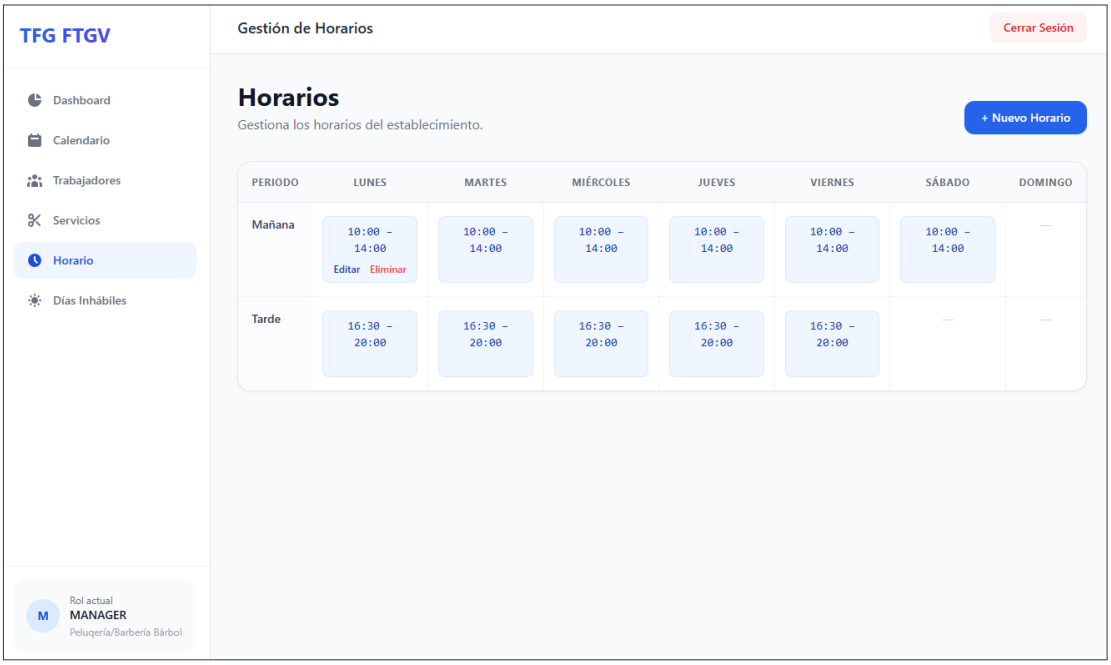


Figura 6.11: Pantalla Horarios

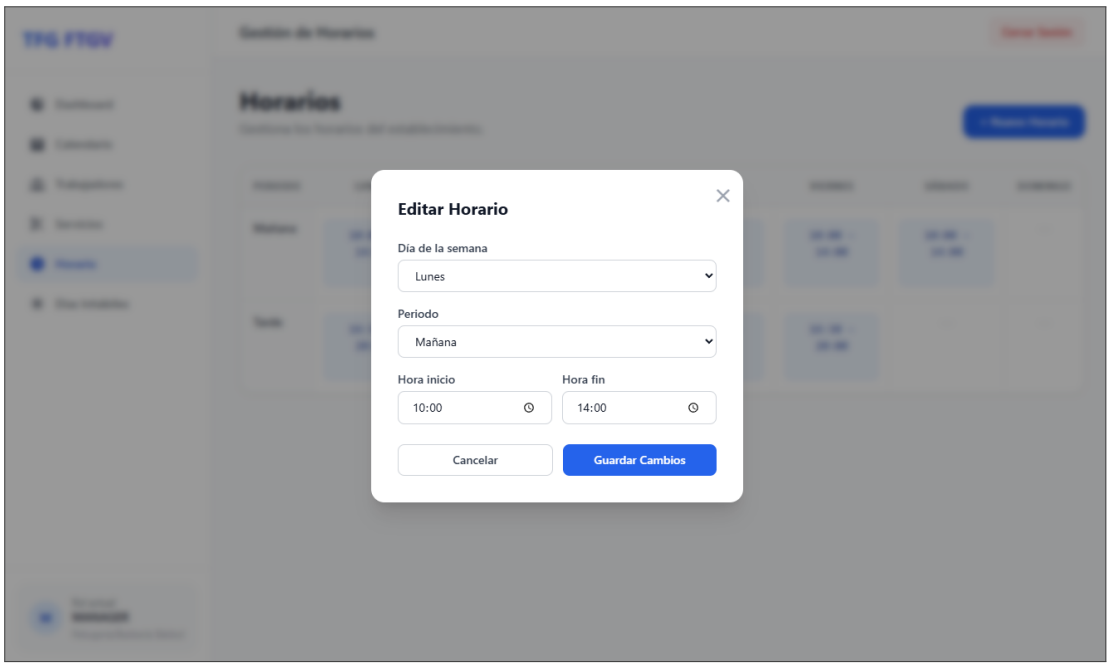


Figura 6.12: Pantalla Formulario Horarios

Gestionar días inhábiles (HU08)

El usuario MANAGER también tiene la opción de gestionar los *Días Inhábiles* de su empresa. Esta pantalla se puede consultar en la Figura 6.13 junto con su formulario en la Figura 6.14.

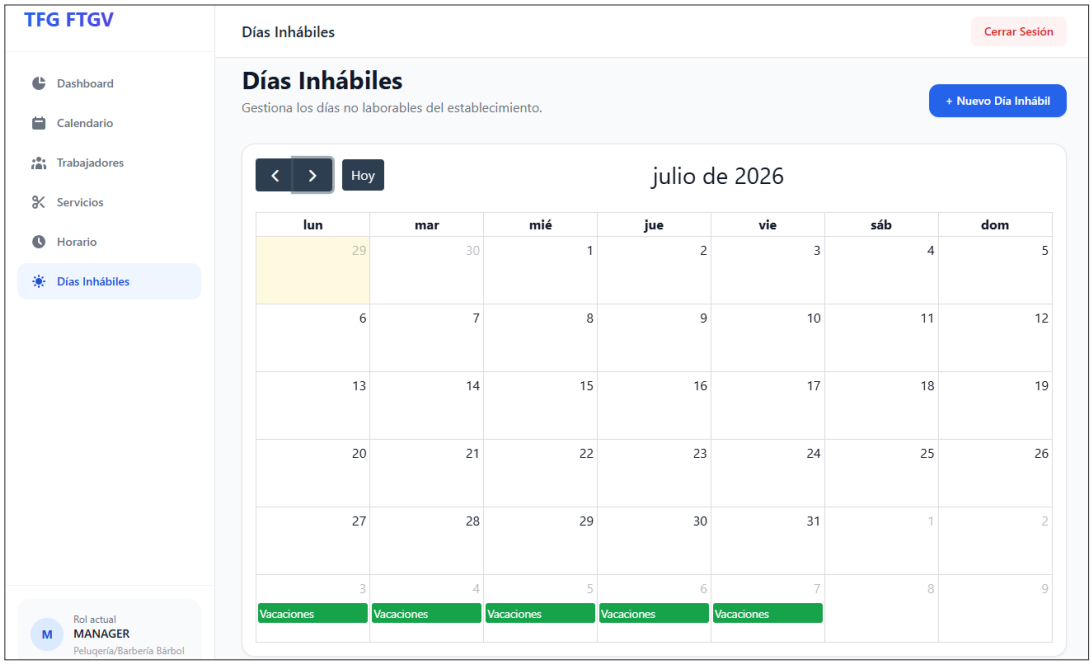


Figura 6.13: Pantalla Días Inhábiles

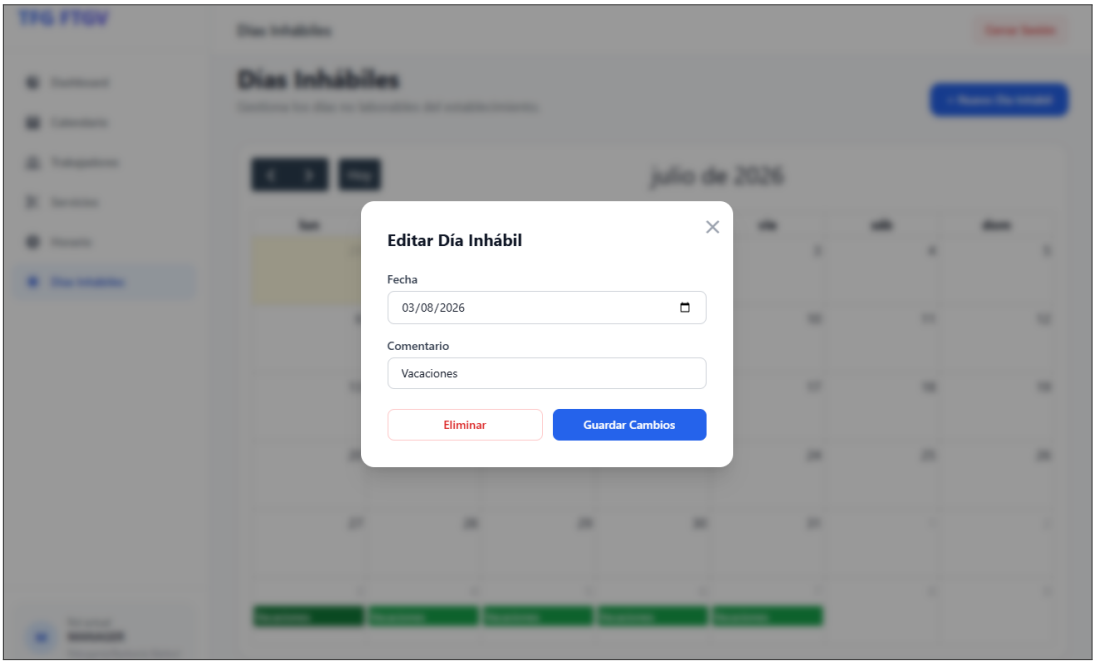


Figura 6.14: Pantalla Formulario Días Inhábiles

Visualizar calendario empresa (HU09)

La opción de *Calendario* mostrará a un usuario MANAGER todos los calendarios de los trabajadores asociados a su empresa diferenciados por colores. La vista por defecto cargará el calendario de todo el mes, pero se podrá navegar entre vistas por semana y día. Las Figura 6.15, 6.16 y 6.17 muestran las vistas de calendario de empresa por mes, semana y día respectivamente.

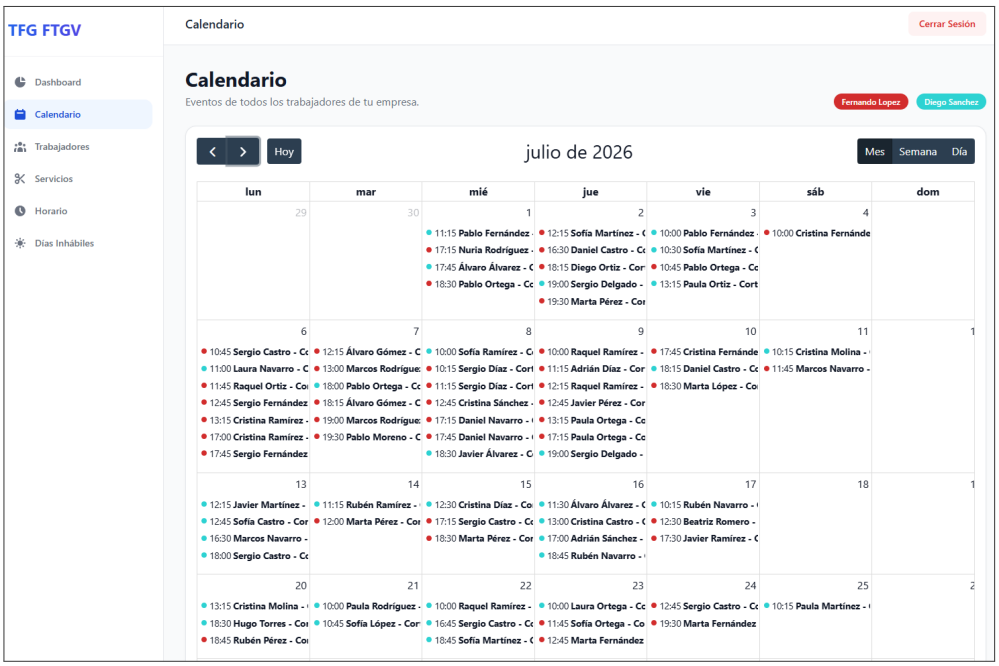


Figura 6.15: Pantalla Calendario vista mes MANAGER

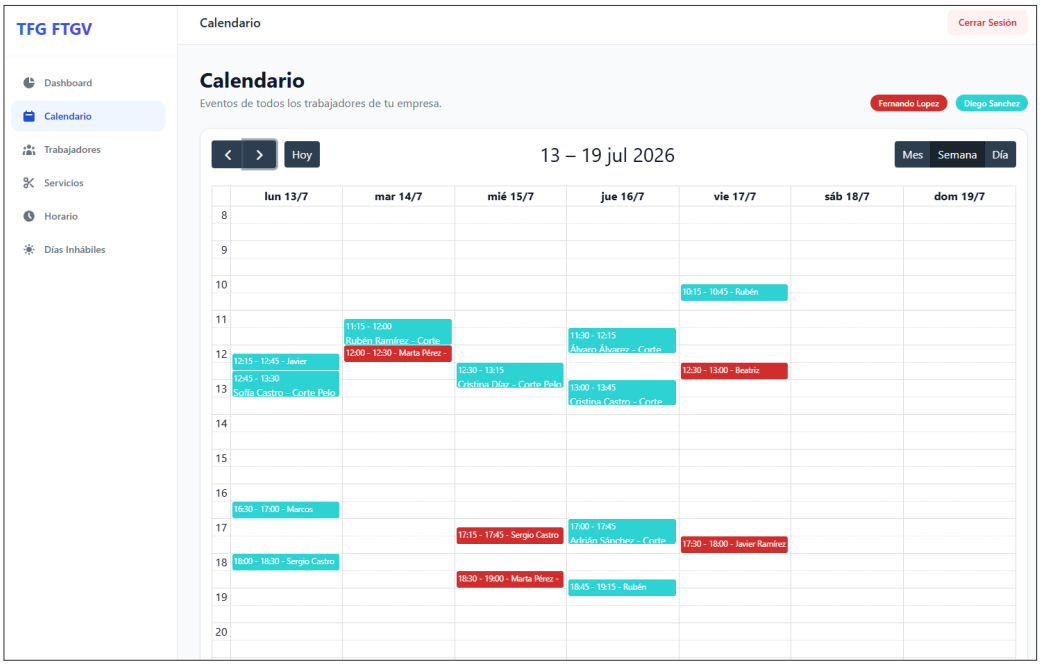


Figura 6.16: Pantalla Calendario vista semana MANAGER

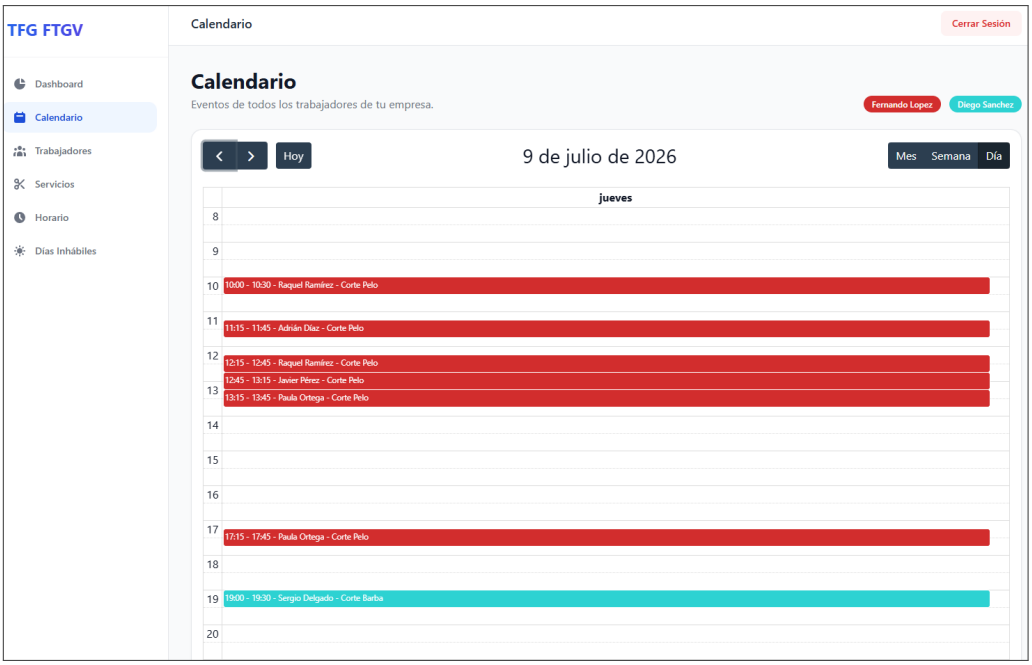


Figura 6.17: Pantalla Calendario vista día MANAGER

Visualizar calendario trabajador (HU10)

Si el usuario autenticado tiene el rol de WORKER, la primera pantalla que encontrará será el menú principal con las opciones correspondientes a su rol, *Calendario y Periodos Inhábiles* (Figura 6.18).

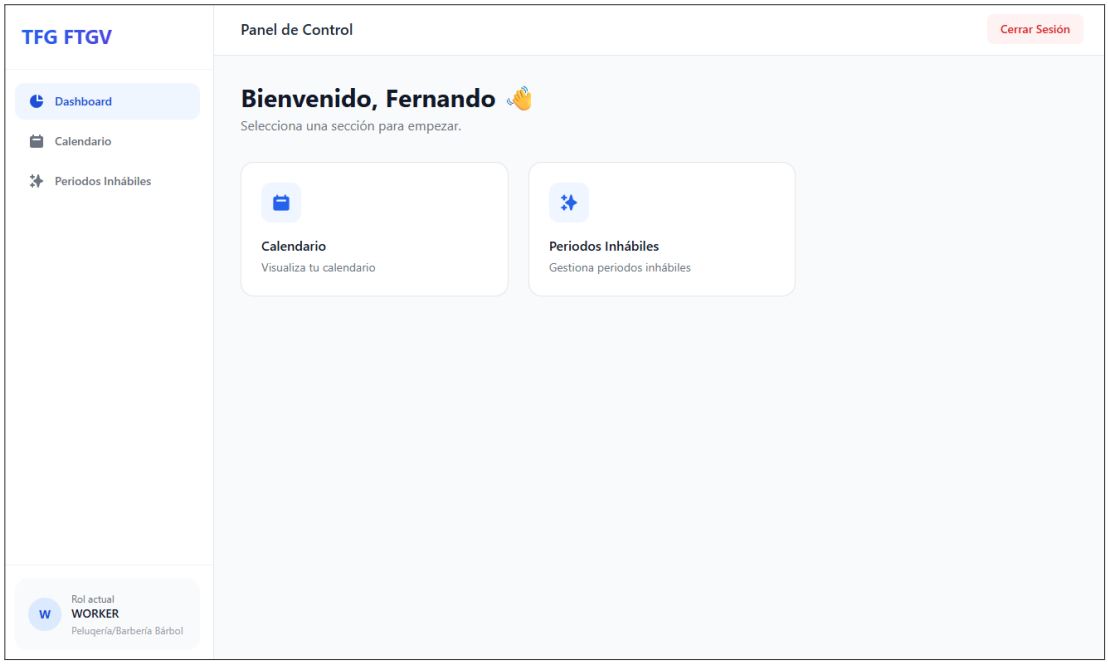


Figura 6.18: Pantalla Menú Principal WORKER

La opción de *Calendario* mostrará a un usuario WORKER los eventos de su propio calendario. Al igual que en la vista de MANAGER, por defecto cargará el calendario de todo

el mes, pero se podrá navegar entre vistas por semana y día. Las Figura 6.19, 6.20 y 6.21 muestran las vistas de calendario de trabajador por mes, semana y día respectivamente.

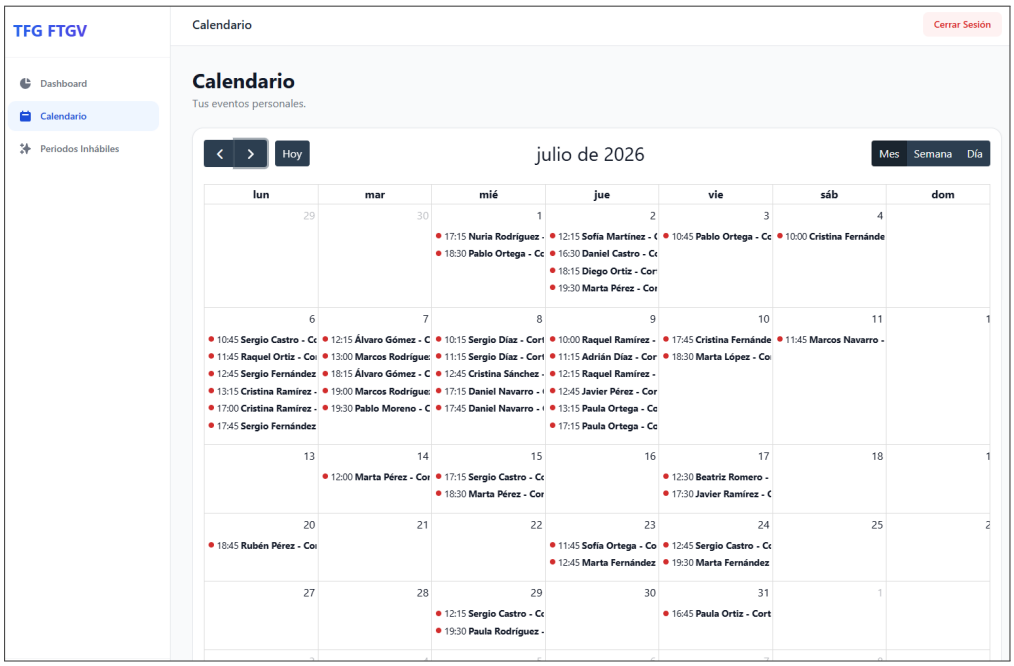


Figura 6.19: Pantalla Calendario vista mes WORKER

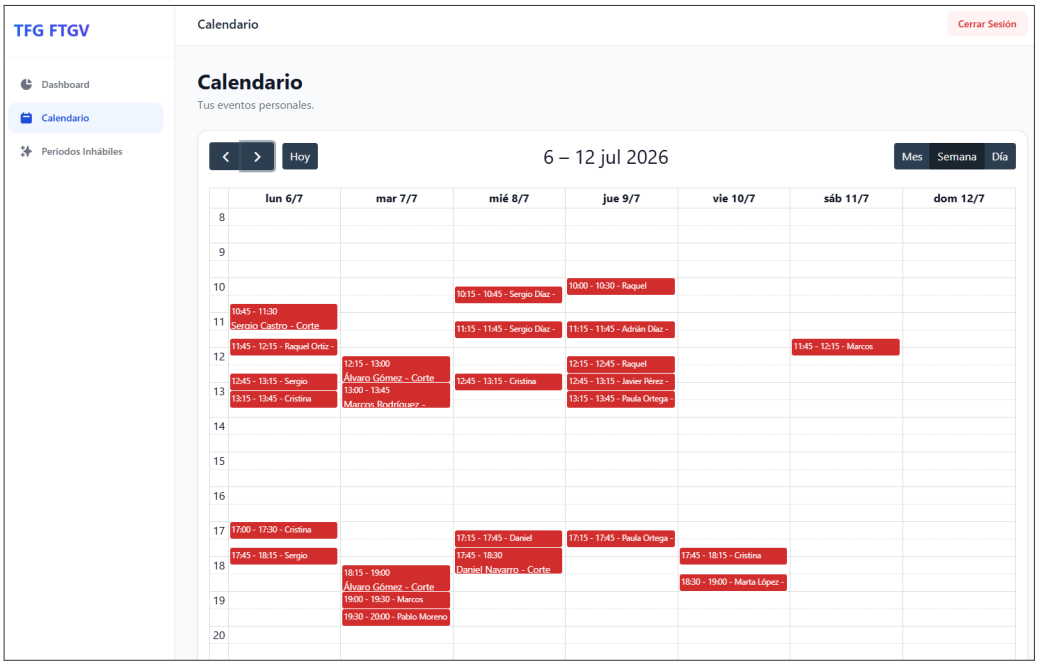


Figura 6.20: Pantalla Calendario vista semana WORKER

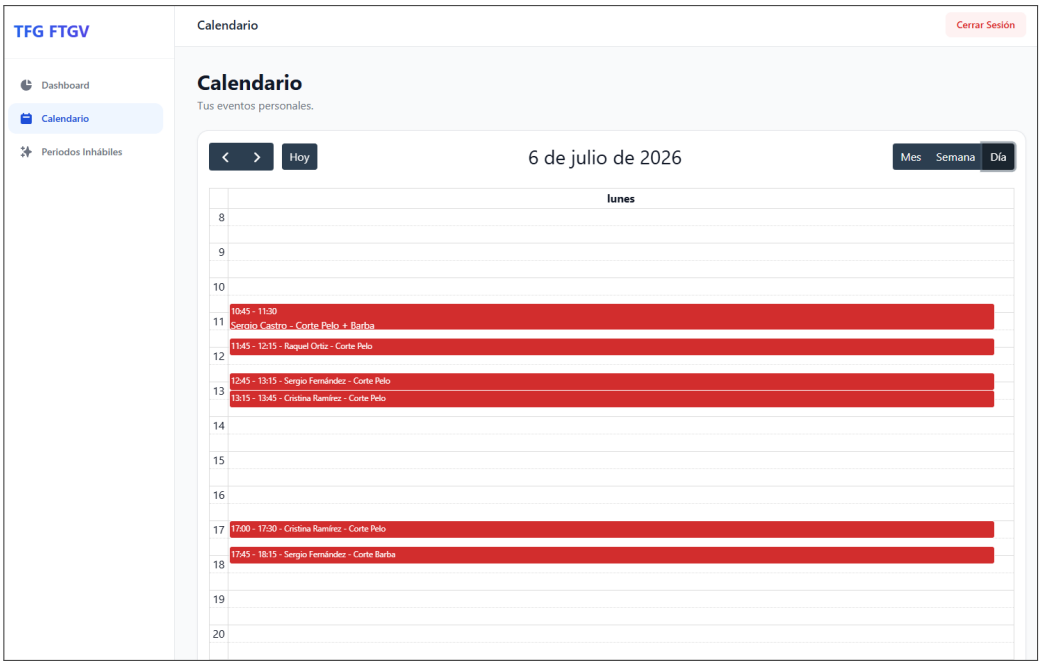


Figura 6.21: Pantalla Calendario vista día WORKER

Gestionar periodos inhábiles (HU11)

La opción *Periodos Inhábiles* de la que disponen los usuarios de tipo WORKER, mostrará un listado de todos los periodos inhábiles que el usuario haya registrado en el sistema (Figura 6.22), una vez más, con las opciones de añadir, editar o eliminar (Figura 6.23).

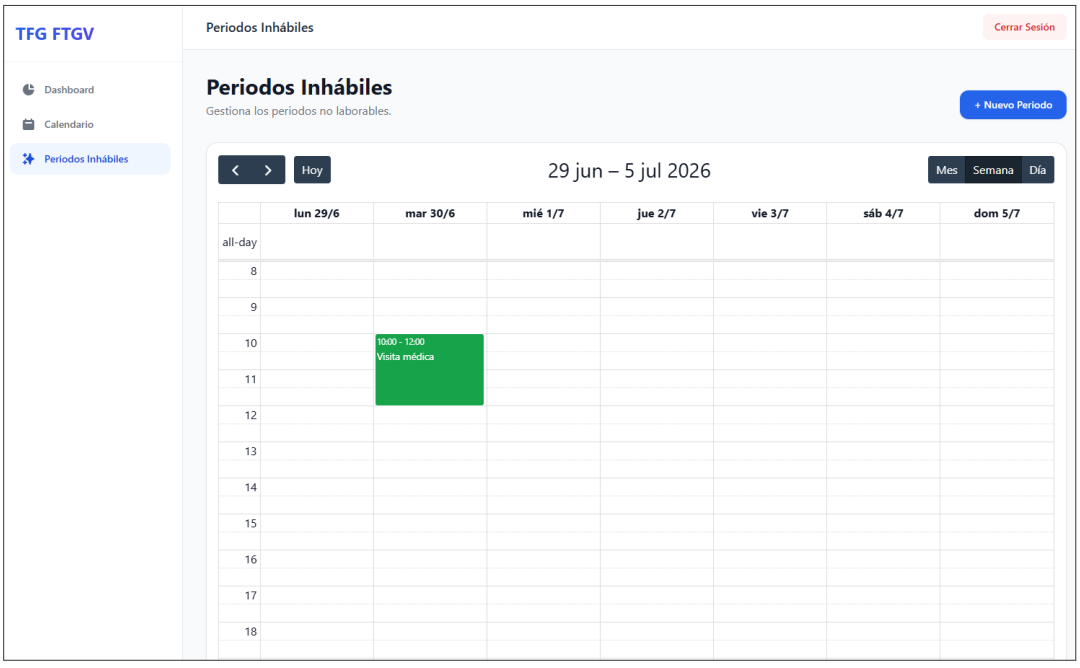


Figura 6.22: Pantalla Periodo Inhábil

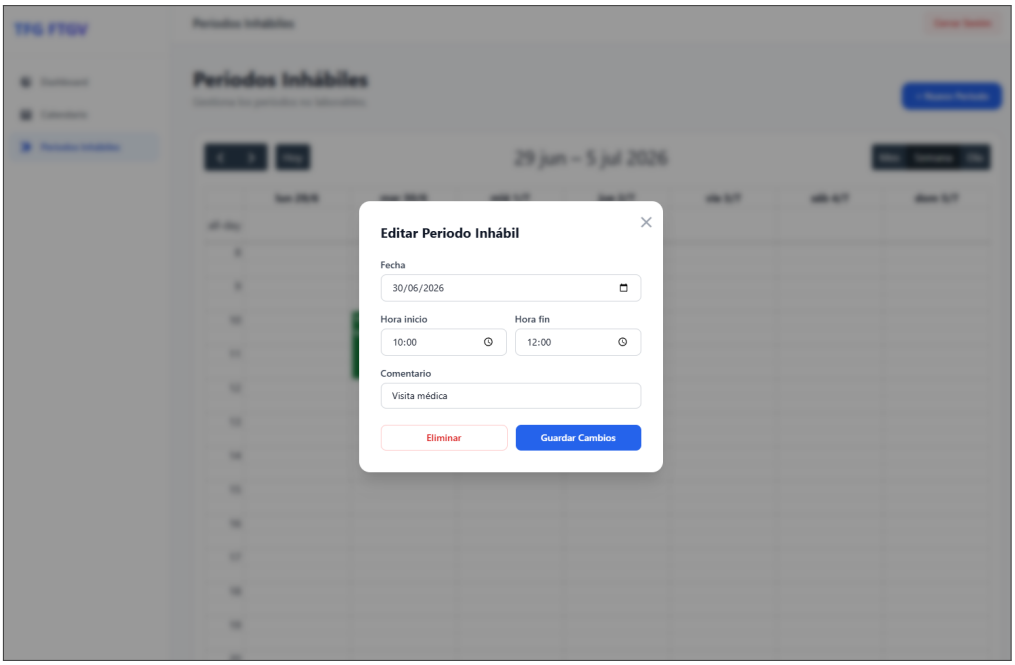


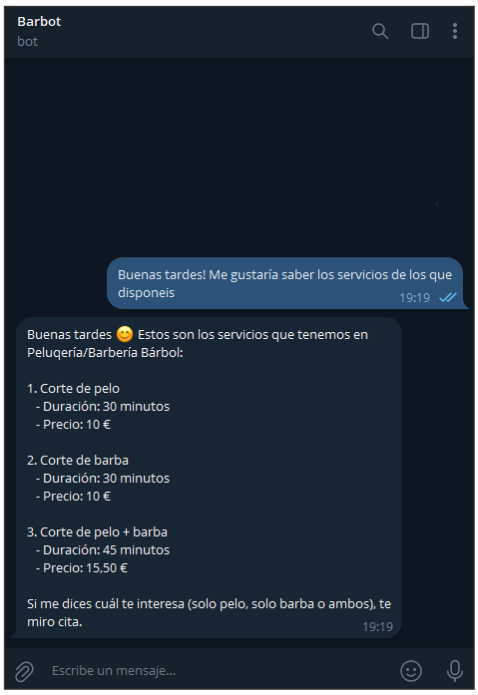
Figura 6.23: Pantalla Formulario Periodo Inhábil

6.2.2 Cliente

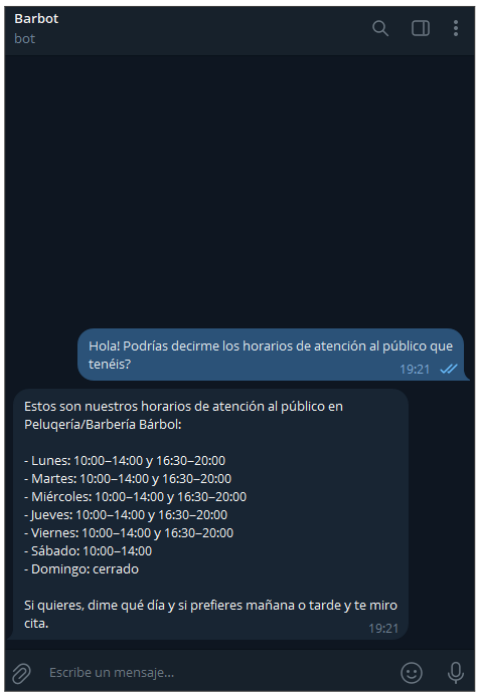
Consultas sobre el establecimiento (HU13 y HU14)

El único canal de interacción que tiene el cliente con el sistema es la aplicación oficial de *Telegram*. Deberá abrir una conversación con el bot *@Barbot* e iniciar la conversación.

El cliente puede plantear al agente otras cuestiones sobre el establecimiento como los servicios ofrecidos con su precio y duración (Figura 6.24a) u horarios de apertura (Figura 6.24b).



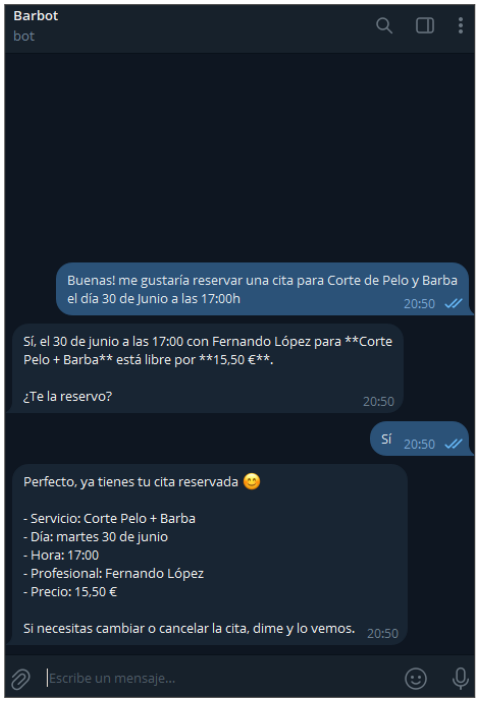
(a) Consulta de servicios



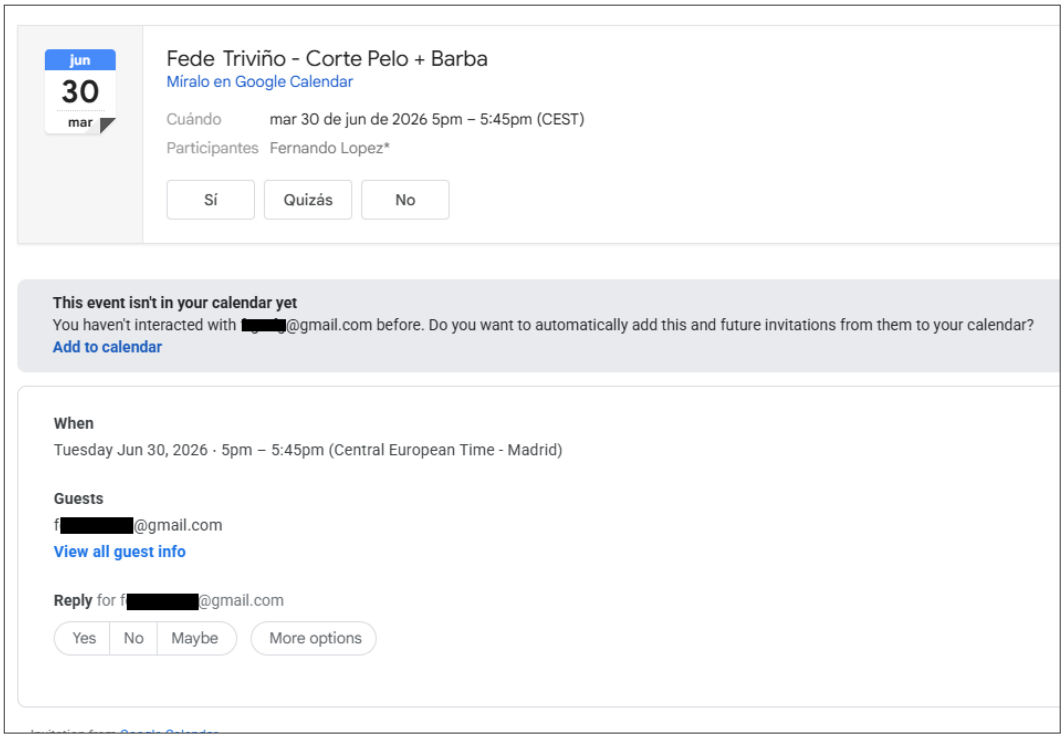
(b) Consulta de horarios

Reservar una cita (HU15)

El usuario podrá gestionar sus citas en lenguaje natural con el agente conversacional, pudiendo reservar, actualizar o cancelar una cita. La Figura 6.25a muestra una conversación con el agente conversacional para reservar una cita. Una vez reservada el cliente recibirá un email de confirmación junto con la invitación para añadir el evento a su calendario personal como muestra la Figura 6.25b.



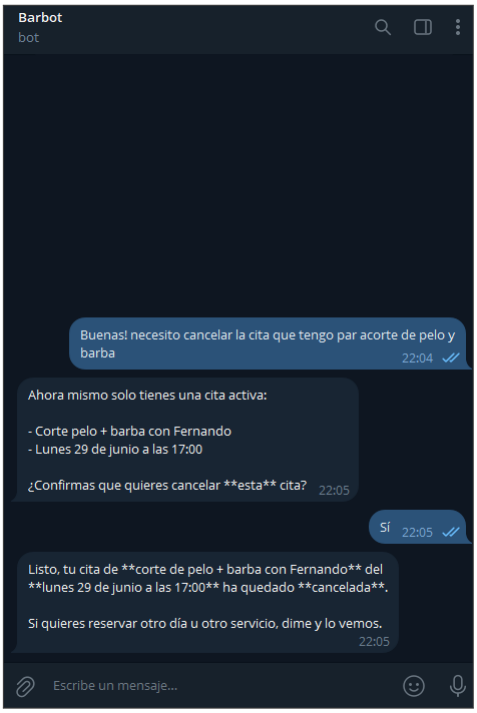
(a) Bot - Reservar cita



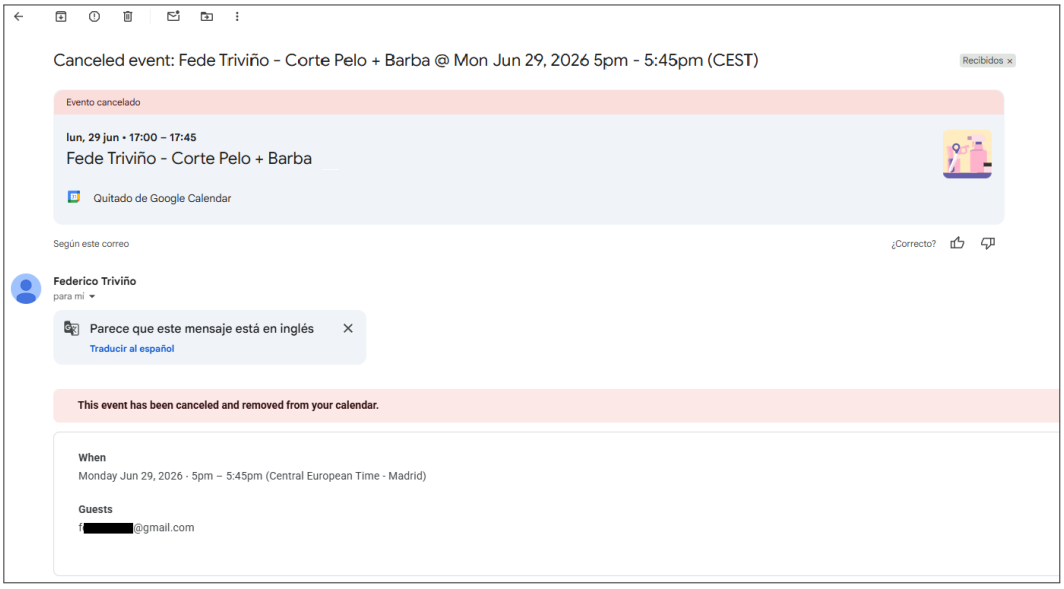
(b) Email reserva de cita

Cancelar una cita (HU16)

En la Figura 6.26a se puede ver un ejemplo de conversación para cancelar una cita previamente asignada y en la Figura 6.26b la recepción del email de cancelación de la misma.



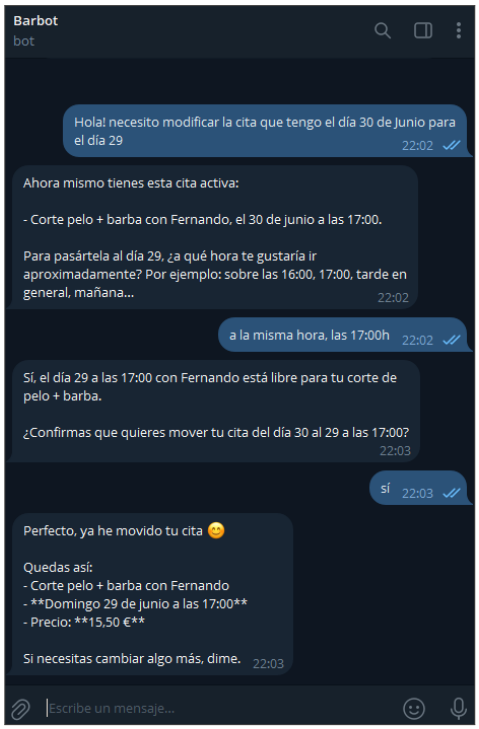
(a) Bot - Cancelar cita



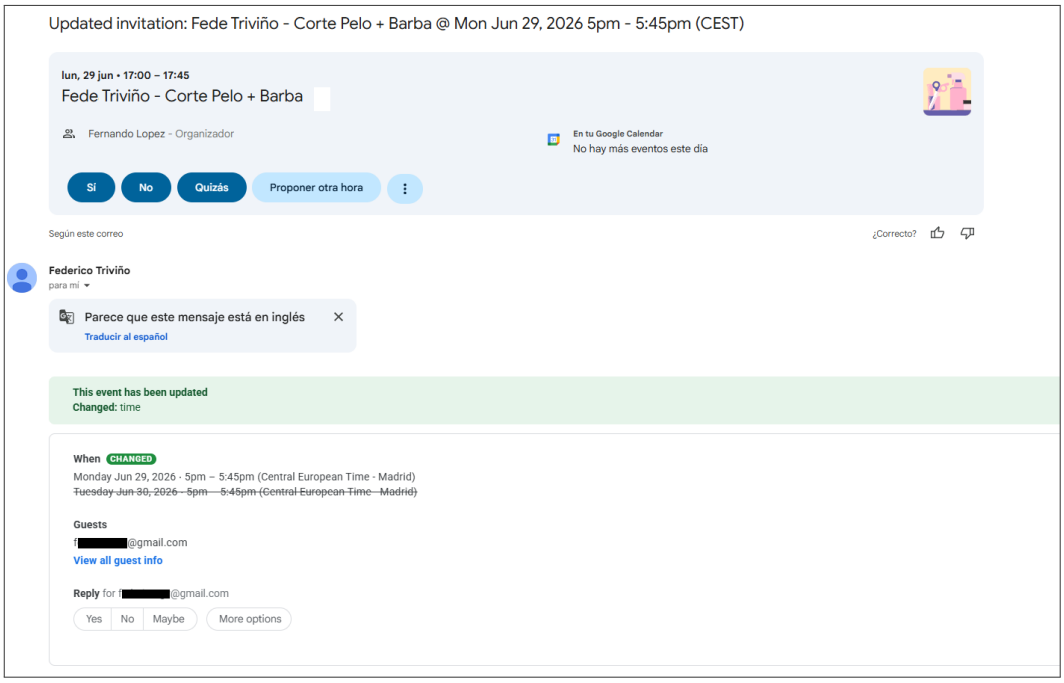
(b) Email cancelar cita

Modificar una cita (HU17)

Para modificar una cita, el cliente podrá consultar al agente como muestra la Figura 6.27a. Además, recibirá el email con la actualización del evento como se puede ver en la Figura 6.27b.



(a) Bot - Modificar cita



(b) Email modificar cita

6.3 Costes de desarrollo

La Tabla 6.1 refleja el desglose de los costes de desarrollo del proyecto atendiendo a los recursos de personal, hardware y software empleados.

Presupuesto estimado del proyecto		
Concepto	**Detalle**	**Importe**
Personal		
Análisis del dominio y requisitos	Analista, 60 h × 30,00 €/h	1.800,00 €
Diseño técnico y modelo de datos	Analista, 60 h × 30,00 €/h	1.800,00 €
Desarrollo Backend y Seguridad	Desarrollador junior, 150 h × 20,00 €/h	3.000,00 €
Integración con *Google Calendar*	Desarrollador junior, 60 h × 20,00 €/h	1.200,00 €
Desarrollo Frontend	Desarrollador junior, 105 h × 20,00 €/h	2.100,00 €
Integración IA (*Spring AI* / GPT)	Desarrollador junior, 45 h × 20,00 €/h	900,00 €
Bot conversacional (*Telegram*)	Desarrollador junior, 60 h × 20,00 €/h	1.200,00 €
Pruebas integradas	Desarrollador junior, 30 h × 20,00 €/h	600,00 €
Subtotal personal	570 h	**12.600,00 €**
Hardware (amortización)		
PC de desarrollo	1.300,00 €× 6/48 meses	162,50 €
Monitor y periféricos	250,00 €× 6/60 meses	25,00 €
Subtotal hardware		**187,50 €**
Software y servicios		
OpenAI GPT API	Consumo en desarrollo y pruebas	120,00 €
Stack *open source*	*Java*, *Spring*, *PostgreSQL*, *React*…	0,00 €
Google Cloud / Calendar API	*Free tier*	0,00 €
Subtotal software		**120,00 €**
TOTAL		**12.907,50 €**

Tabla 6.1: Costes de desarrollo

Capítulo 7

Conclusiones

En este capítulo se revisan los objetivos planteados al comienzo del proyecto, exponiendo cómo han sido resueltos a lo largo del desarrollo de la solución. Por último, se presentan una serie de propuestas que sirven de guía para definir posibles líneas de trabajo futuro para el proyecto.

7.1 Objetivos alcanzados

Para el presente TFG se planteó como meta principal el diseño y desarrollo de un sistema modular de gestión automatizada de citas para pequeños comercios del sector de la peluquería y la barbería, con la intención de reducir al máximo la intervención humana en las tareas rutinarias del negocio. Una vez finalizado el desarrollo, puede afirmarse que dicho objetivo principal se ha alcanzado de forma satisfactoria construyendo una solución completa y funcional que integra un servicio de *backend* central, una BBDD relacional, una aplicación web de gestión y un canal conversacional para los clientes a través de Telegram. A continuación se repasan, uno a uno, los objetivos específicos que se marcaron al inicio del proyecto y se explica de qué manera han sido resueltos.

- **Núcleo del sistema para centralizar y orquestar toda la lógica**: este objetivo constituía el núcleo del proyecto y se ha cumplido íntegramente. El servicio se ha desarrollado siguiendo una arquitectura hexagonal, con una separación clara entre la capa de dominio, la capa de aplicación y la capa de infraestructura, lo que facilita el mantenimiento y las pruebas de cada componente de forma aislada. El *backend* expone una API-REST bien formada que consume la aplicación web, actúa como intermediario entre el cliente y el modelo conversacional, ofrece al modelo un conjunto de herramientas mediante un servidor MCP local, gestiona los calendarios y la comunicación con *Google*, y centraliza todos los aspectos de seguridad del sistema. De este modo, se confirma que el *backend* da soporte efectivo al resto de módulos de la solución.
- **Interfaz para gerentes y trabajadores**: se ha desarrollado una aplicación web que permite a gerentes y trabajadores administrar la información del negocio. La aplicación incorpora un sistema de autenticación basado en *tokens* JWT y una distinción de roles que restringe el alcance de cada usuario y adapta las opciones disponibles según su

perfil. A través de ella es posible gestionar los trabajadores de la empresa, los servicios ofrecidos, el horario de apertura y los días inhábiles del establecimiento, los periodos inhábiles de cada trabajador y la visualización de los eventos agendados. Se cubren, por tanto, todas las funcionalidades de gestión previstas.

- **Canal de comunicación cómodo y familiar para el cliente**: se ha implementado un bot de *Telegram* que constituye el único punto de interacción del cliente con el sistema. Aprovechando la difusión y la familiaridad de esta plataforma de mensajería, el módulo ofrece al usuario un entorno cómodo y cercano. Su responsabilidad se limita a recoger los mensajes y los datos del cliente y trasladarlos al *backend*, que devuelve una respuesta en lenguaje natural lista para mostrarse.
- **Persistencia íntegra, consistente y segura de toda la información**: se ha definido un modelo de datos relacional consistente, sin redundancias y orientado a garantizar la integridad y precisión de la información. El esquema contempla la información de negocio, clientes, información asociada a *Google* y la información de usuarios de la aplicación web. El diseño obtenido proporciona una base de persistencia sólida para el conjunto del sistema.
- **Asistente conversacional con respuestas naturales, fiables y seguras:** se ha elaborado un *prompt* claro y conciso que guía al modelo para generar respuestas naturales y eficientes. En él se establecen restricciones explícitas y se dota al modelo de un comportamiento proactivo. Además, el *prompt* admite variables de configuración que permiten inyectar información dinámica, lo que lo hace reutilizable para distintos negocios.
- **Soporte multiempresa**: el sistema se ha diseñado para gestionar varias empresas de manera independiente desde una única instancia, sin visibilidad entre ellas. Cada empresa dispone de su propia configuración, credenciales, clientes, eventos e información de negocio, garantizándose el aislamiento de los datos entre los distintos negocios. Con ello se cumple el carácter multiempresa exigido.
- **Ciclo de desarrollo basado en metodología ágil**: el proyecto se ha llevado a cabo mediante un desarrollo iterativo e incremental, con hitos verificables que han permitido validar cada módulo de forma independiente antes de integrarlo en el conjunto.
- **Aplicación de buenas prácticas**: a lo largo de todo el desarrollo se han aplicado buenas prácticas de ingeniería del software: CI, uso de patrones de diseño y respeto por principios como SOLID, KISS y YAGNI. El resultado es un sistema modular, mantenible y preparado para crecer, en el que la separación de responsabilidades favorece tanto la robustez como la escalabilidad.

En conjunto, puede concluirse que la totalidad de los objetivos planteados se han satisfecho. El proyecto no solo demuestra la viabilidad técnica de automatizar la gestión de citas

en pequeños comercios mediante un modelo conversacional, sino que también ha servido como ejercicio de aplicación práctica de los conocimientos adquiridos a lo largo del ciclo académico, integrando arquitectura de *software*, diseño de BBDDs, desarrollo, seguridad e IA en una solución coherente y completa.

7.2 Trabajo futuro y mejoras propuestas

Aunque la solución desarrollada cumple los objetivos marcados, durante el proyecto se han identificado diversas líneas de evolución que permitirían mejorar su rendimiento, su seguridad y su alcance funcional. Se recogen a continuación las más relevantes.

- **Seguridad y gestión de credenciales**: actualmente el *token* JWT se almacena en el localStorage del navegador y deja la aplicación expuesta a ataques de tipo XSS. Como mejora prioritaria se propone migrar a un esquema de *cookie httpOnly* acompañada de un *token* CSRF, que constituye la práctica profesional recomendada. En esta misma línea, conviene completar el ciclo de vida de las cuentas de la aplicación web incorporando los flujos de restablecimiento y recuperación de contraseña para los usuarios. Asimismo, se plantea evolucionar el modelo de verificación de *tokens* hacia un esquema en el que cualquier componente que necesite verificar pueda hacerlo con la clave pública. Por último, se considera necesario aplicar limitación de peticiones (*rate-limiting*) sobre el *endpoint* /api/v1/chat, tanto para proteger el sistema frente a usos abusivos y posibles ataques de denegación de servicio como para contener el coste asociado al modelo conversacional.
- **Evolución arquitectónica**: a medida que el sistema crezca, sería conveniente extraer el servidor MCP como componente independiente y avanzar hacia una división en microservicios, separando los grandes dominios del sistema. Esta evolución mejoraría la escalabilidad y permitiría desplegar y mantener cada parte de forma autónoma. La arquitectura hexagonal adoptada facilita notablemente esta transición, al mantener el dominio desacoplado de la infraestructura.
- **Evolución del modelo conversacional**: el *prompt* diseñado no debe entenderse como un artefacto cerrado, sino como un elemento vivo que conviene ajustar de forma dinámica y continua en función del uso real del sistema. El análisis de las conversaciones mantenidas con los clientes permitirá detectar respuestas poco naturales, malentendidos recurrentes o restricciones que deban reforzarse, e iterar sobre el *prompt* para mejorar progresivamente la calidad de la interacción. En esta misma línea, se propone estudiar el comportamiento del sistema con otros LLMs, comparando distintas alternativas en términos de calidad de las respuestas, latencia y coste, con el fin de seleccionar en cada momento el modelo que mejor se adapte a las necesidades del proyecto y evitar una dependencia excesiva de un único proveedor.
- **Rendimiento y robustez**: se propone introducir una capa de caché para las consultas

más frecuentes, reduciendo la carga sobre la BBDD y mejorando los tiempos de respuesta del sistema tanto para el modelo como para las peticiones de *backoffice*. De la misma forma, se propone implementar una memoria caché específica para el modelo en la que pueda almacenar diferente información como huecos disponibles para un día que pueda ser reutilizada entre conversaciones y aporten fluidez a la interacción del cliente.

- **Mejoras visuales y de experiencia de usuario en el backoffice**: el panel de administración de *backoffice* cumple su función, pero presenta un amplio margen de mejora en el plano visual y de experiencia de usuario (UX). Se propone refinar la interfaz hacia un diseño más cuidado y coherente que refuerce la sensación de producto profesional. Entre las mejoras concretas destacan: garantizar un diseño plenamente *responsive* que permita una gestión cómoda desde diferentes dispositivos, mejorar los aspectos de usabilidad, accesibilidad y retroalimentación al usuario. Estas mejoras, aunque no afectan a la lógica de negocio, incrementarían notablemente la experiencia de usuario.
- **Nuevas funcionalidades de negocio**: en el plano funcional se proponen varias ampliaciones de valor para los comercios: la gestión de programas de fidelización de clientes, la incorporación de métodos de pago que permitan abonar o reservar servicios desde el propio canal, y la posibilidad de asociar la cuenta de *Google* de cada trabajador para que las citas se reflejen directamente en su calendario personal.
- **Otras mejoras recomendadas**: adicionalmente, se considera oportuno estudiar la incorporación de canales de comunicación alternativos (por ejemplo, WhatsApp), dado que la arquitectura actual permitiría añadir nuevos canales con un impacto muy reducido. También sería interesante ampliar la observabilidad del sistema para facilitar su explotación en producción y reforzar la cobertura de pruebas automáticas de integración, End-to-End (E2E) y de calidad del LLM. Por ejemplo, automatizando un LLM externo que interactúe con el sistema asumiendo el rol de cliente y evalúe la calidad de interacción con el bot conversacional.

En definitiva, estas líneas de trabajo futuro confirman que el proyecto, además de cumplir con los objetivos propuestos, queda preparado para seguir creciendo y adaptándose a necesidades reales de explotación, consolidándose como una base sólida sobre la que continuar trabajando.

Apéndices

Anexo A

Apéndice A - Historias de Usuario

A.1 Historias de usuario

Durante la fase de análisis, se han modelado como historias de usuario las funcionalidades más importantes que recogen los objetivos principales del proyecto y que han sido objeto de validación durante las fases de desarrollo y pruebas de la solución.

HU01 – Login Usuario Backoffice	
Rol	Como Usuario Backoffice
Funcionalidad	Quiero autenticarme en el sistema
Utilidad	Para acceder al panel de administración del Backoffice
Criterio de Aceptación	El usuario accede al sistema y visualiza el panel de administración con las opciones disponibles para su rol.

Tabla A.1: HU01 – Login Usuario Backoffice

HU02 – Logout Usuario Backoffice	
Rol	Como Usuario Backoffice
Funcionalidad	Quiero cerrar sesión en el sistema
Utilidad	Para abandonar el panel de administración del Backoffice y volver al formulario de login
Criterio de Aceptación	El usuario cierra su actual sesión en el sistema y regresa al formulario de login.

Tabla A.2: HU02 – Logout Usuario Backoffice

HU03 – Gestionar empresas	
Rol	Como Usuario Backoffice ADMIN
Funcionalidad	Quiero gestionar las empresas del sistema
Utilidad	Para añadir, editar o eliminar empresas registradas en el sistema
Criterio de Aceptación	El usuario ADMIN accede al panel de empresas de *backoffice* y es capaz de añadir nuevas o visualizar, editar y eliminar cualquiera de las existentes.

Tabla A.3: HU03 – Gestionar empresas

HU04 – Gestionar usuarios	
Rol	Como Usuario Backoffice ADMIN
Funcionalidad	Quiero gestionar los usuarios del sistema
Utilidad	Para añadir, editar o eliminar usuarios registrados en el sistema
Criterio de Aceptación	El usuario ADMIN accede al panel de usuarios de *backoffice* y es capaz de añadir nuevos o visualizar, editar y eliminar cualquiera de los existentes.

Tabla A.4: HU04 – Gestionar usuarios

HU05 – Visualizar trabajadores	
Rol	Como Usuario Backoffice MANAGER
Funcionalidad	Quiero visualizar el listado de trabajadores asociados a mi empresa
Utilidad	Para consultar los trabajadores de mi empresa y sus datos
Criterio de Aceptación	El usuario MANAGER accede al panel de trabajadores de *backoffice* y visualiza el listado de trabajadores asociados a su empresa y los datos de cada trabajador.

Tabla A.5: HU05 – Visualizar trabajadores

HU06 – Gestionar servicios	
Rol	Como Usuario Backoffice MANAGER
Funcionalidad	Quiero gestionar los servicios de mi empresa
Utilidad	Para añadir, editar o eliminar servicios de mi empresa
Criterio de Aceptación	El usuario MANAGER accede al panel de servicios de *backoffice* y es capaz de añadir nuevos o visualizar, editar y eliminar cualquiera de los existentes.

Tabla A.6: HU06 – Gestionar servicios

HU07 – Gestionar horario	
Rol	Como Usuario Backoffice MANAGER
Funcionalidad	Quiero gestionar los horarios de mi empresa
Utilidad	Para modificar el horario de mi empresa
Criterio de Aceptación	El usuario MANAGER accede al panel de horario de *backoffice* y es capaz de modificar el horario.

Tabla A.7: HU07 – Gestionar horario

HU08 – Gestionar días inhábiles	
Rol	Como Usuario Backoffice MANAGER
Funcionalidad	Quiero gestionar los días inhábiles de mi empresa
Utilidad	Para añadir, editar o eliminar los días inhábiles de mi empresa
Criterio de Aceptación	El usuario MANAGER accede al panel de días inhábiles de *backoffice* y es capaz de añadir nuevos o visualizar, editar y eliminar cualquiera de los existentes.

Tabla A.8: HU08 – Gestionar días inhábiles

HU09 – Visualizar calendario empresa	
Rol	Como Usuario Backoffice MANAGER
Funcionalidad	Quiero visualizar el calendario de mi empresa
Utilidad	Para consultar el calendario de todos los trabajadores de mi empresa
Criterio de Aceptación	El usuario MANAGER accede al panel de calendario y es capaz de visualizar los calendarios de todos los trabajadores de su empresa.

Tabla A.9: HU09 – Visualizar calendario empresa

HU10 – Visualizar calendario trabajador	
Rol	Como Usuario Backoffice WORKER
Funcionalidad	Quiero visualizar mi calendario de trabajo
Utilidad	Para consultar mi calendario y ver las citas que tengo asignadas
Criterio de Aceptación	El usuario WORKER accede al panel de calendario y es capaz de visualizar su calendario de trabajo y las citas asignadas.

Tabla A.10: HU10 – Visualizar calendario trabajador

HU11 – Gestionar periodos inhábiles	
Rol	Como Usuario Backoffice WORKER
Funcionalidad	Quiero gestionar mis periodos inhábiles
Utilidad	Para añadir, editar o eliminar mis periodos inhábiles de trabajo
Criterio de Aceptación	El usuario WORKER accede al panel de periodos inhábiles y es capaz de añadir uno nuevo o visualizar, editar o eliminar cualquiera de los existentes.

Tabla A.11: HU11 – Gestionar periodos inhábiles

HU12 – Autenticar cuenta de Google	
Rol	Como usuario Backoffice MANAGER/WORKER
Funcionalidad	Quiero obtener enlace
Utilidad	Para volver a autorizar cuenta de *Google* si el *token* ha caducado
Criterio de Aceptación	El usuario MANAGER/WORKER al hacer login en el *backoffice*, si el *token* de la cuenta de *Google* de su empresa ha expirado, recibe un enlace para volver a autorizar la cuenta

Tabla A.12: HU12 – Autenticar cuenta de Google

HU13 – Consultar servicios empresa	
Rol	Como Cliente
Funcionalidad	Quiero hablar con el bot de *Telegram*
Utilidad	Para consultar los servicios disponibles de la empresa
Criterio de Aceptación	El cliente consulta al bot de *Telegram* por los servicios disponibles de la empresa y este le responde con el listado de servicios junto con su duración y precio.

Tabla A.13: HU13 – Consultar servicios empresa

HU14 – Consultar horario empresa	
Rol	Como Cliente
Funcionalidad	Quiero hablar con el bot de *Telegram*
Utilidad	Para consultar el horario de apertura de la empresa
Criterio de Aceptación	El cliente consulta al bot de *Telegram* por el horario de apertura de la empresa y este le responde con el horario habitual de la empresa.

Tabla A.14: HU14 – Consultar horario empresa

HU15 – Agendar cita	
Rol	Como Cliente
Funcionalidad	Quiero hablar con el bot de *Telegram*
Utilidad	Para agendar una cita en el establecimiento
Criterio de Aceptación	El cliente solicita al bot de *Telegram* agendar una cita y este registra la cita en el sistema y le devuelve un mensaje de confirmación.

Tabla A.15: HU15 – Agendar cita

HU16 – Cancelar cita	
Rol	Como Cliente
Funcionalidad	Quiero hablar con el bot de *Telegram*
Utilidad	Para cancelar una cita previamente agendada
Criterio de Aceptación	El cliente solicita al bot de *Telegram* cancelar una cita que tuviese agendada y este elimina la cita del sistema y le devuelve un mensaje de confirmación.

Tabla A.16: HU16 – Cancelar cita

HU17 – Modificar cita	
Rol	Como Cliente
Funcionalidad	Quiero hablar con el bot de *Telegram*
Utilidad	Para modificar una cita previamente agendada
Criterio de Aceptación	El cliente solicita al bot de *Telegram* modificar una cita que tuviese agendada y este registra la modificación en el sistema y le devuelve un mensaje de confirmación.

Tabla A.17: HU17 – Modificar cita

HU18 – Agregar cita a calendario personal	
Rol	Como Cliente
Funcionalidad	Quiero recibir notificación
Utilidad	Para agregar una cita agendada a mi calendario personal.
Criterio de Aceptación	El cliente agenda una cita en el sistema y recibe un email con la invitación al evento de *Google Calendar*, la acepta y se agrega a su calendario personal.

Tabla A.18: HU18 – Agregar cita a calendario personal

HU19 – Recibir recordatorio de cita agendada	
Rol	Como Cliente
Funcionalidad	Quiero recibir notificación
Utilidad	Para recordar que tengo una cita agendada.
Criterio de Aceptación	El cliente agenda una cita en el sistema y la añade a su calendario personal. El sistema de *Google Calendar* le envía un aviso de recordatorio según el tiempo configurado.

Tabla A.19: HU19 – Recibir recordatorio de cita agendada

Anexo B

Apéndice B - Diagramas

En este apéndice se recogen los diagramas que representan algunos de los flujos más importantes del sistema que se han descrito a lo largo del documento.

- La Figura B.1 representa el flujo de login en el *backend* incluyendo la emisión del *token* JWT.

- La Figura B.2 representa el flujo de la operación de login desde el punto de vista del *frontend*, incluyendo la recuperación y carga de usuarios y permisos.

- La Figura B.3 representa un flujo genérico entre los diferentes componentes del sistema para cualquier petición de operaciones de *backoffice*.

- La Figura B.4 representa el flujo de creación de eventos de *Google Calendar* que invoca el modelo para registrar una cita en el sistema.

- La Figura B.5 representa el flujo de autenticación OAUTH2 para las cuentas de *Google* de las empresas.

- La Figura B.6 representa el flujo de la comunicación con el LLM en una petición de chat de cliente.

B.1 Flujo Login JWT

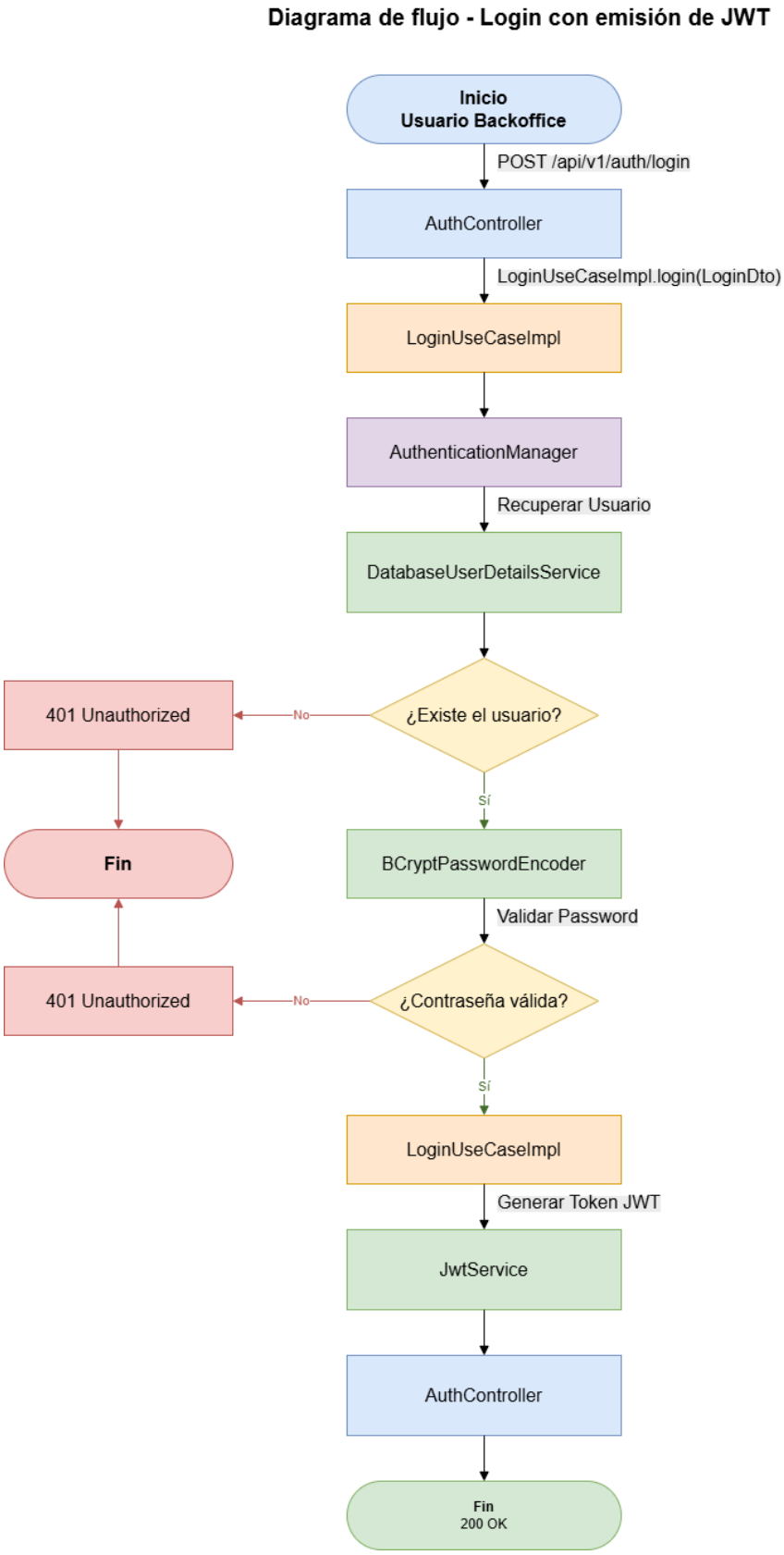


Figura B.1: Flujo login JWT

B.2 Flujo Login Frontend

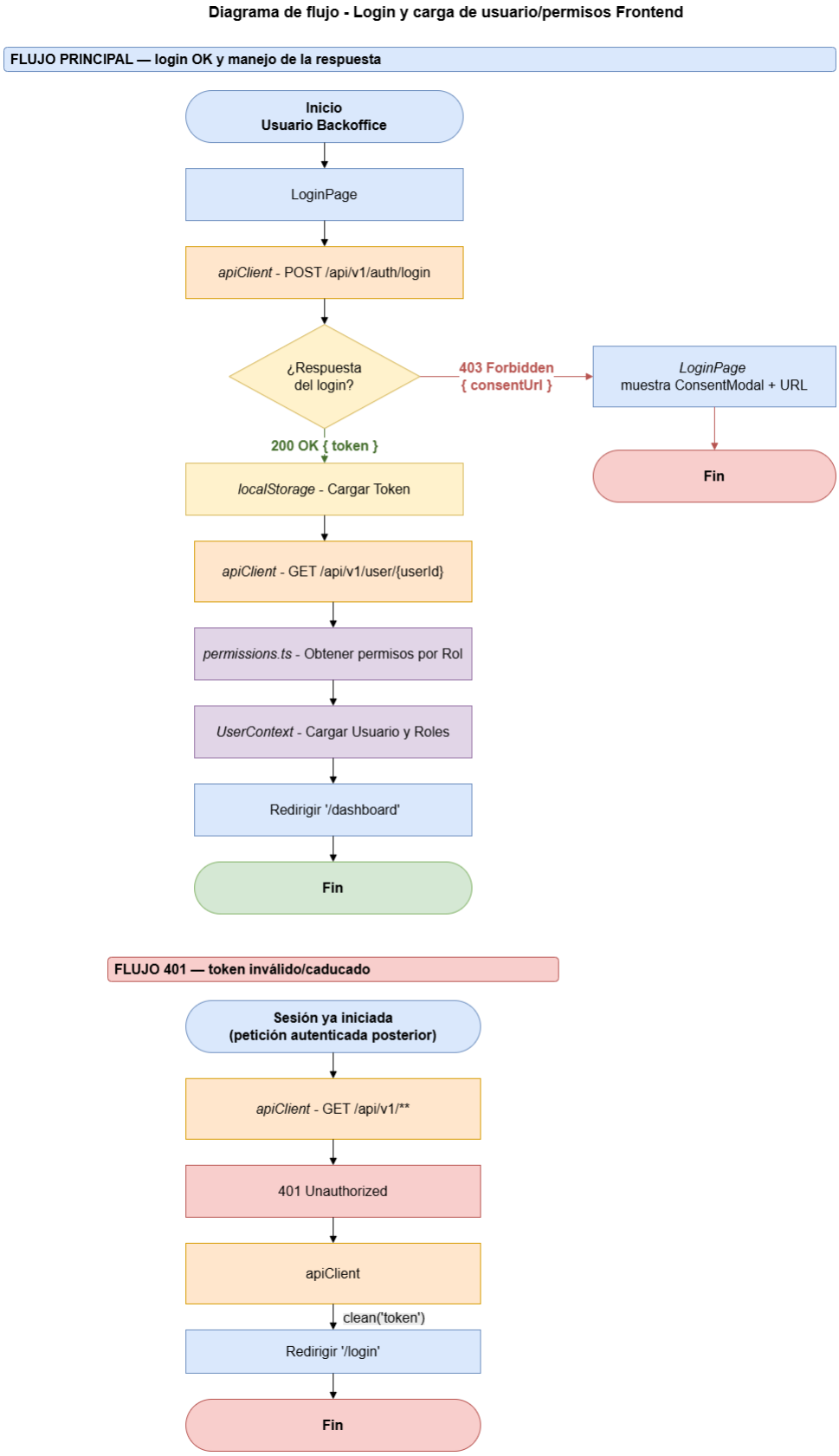


Figura B.2: Flujo login frontend

B.3 Flujo Backoffice

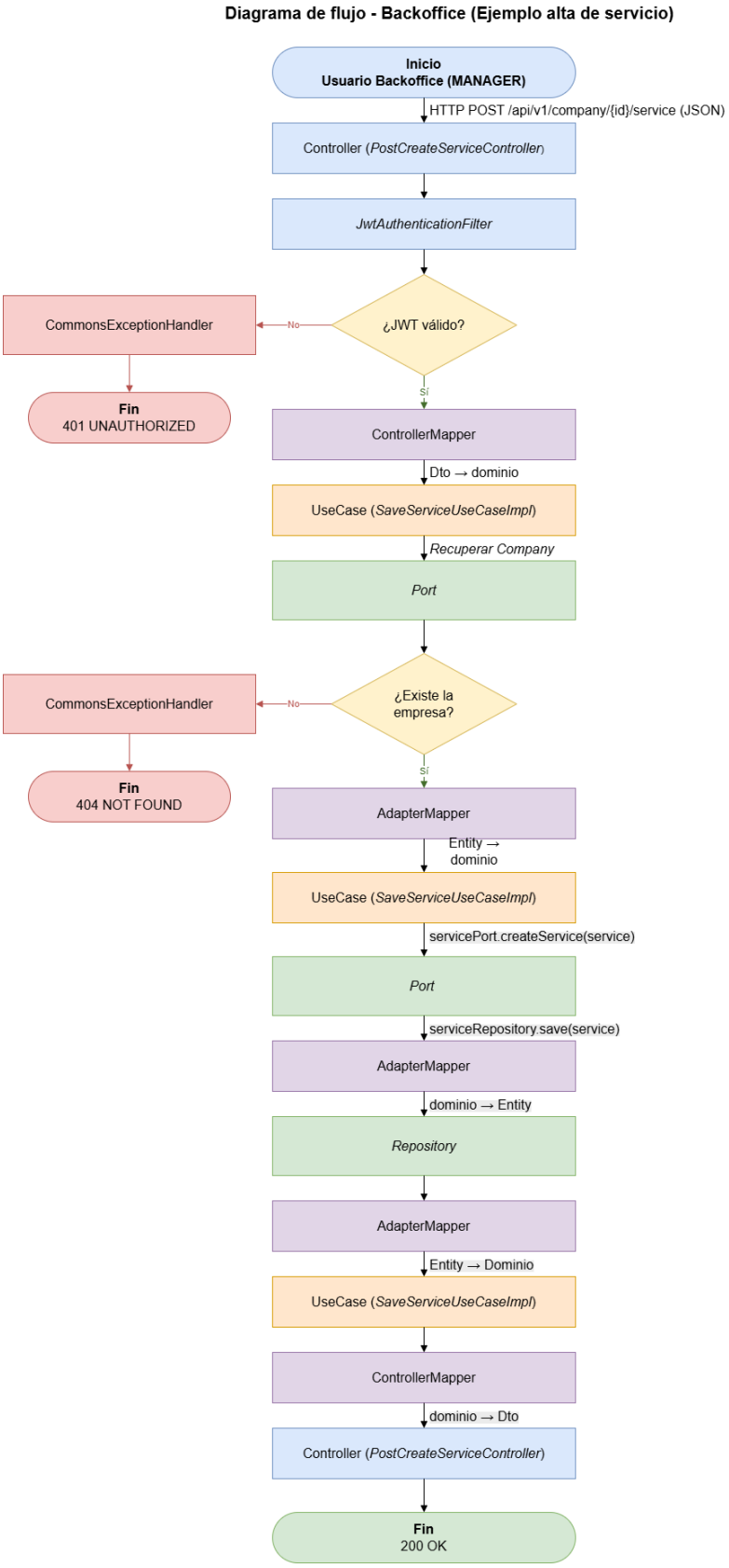


Figura B.3: Flujo backoffice

B.4 Flujo Google Calendar

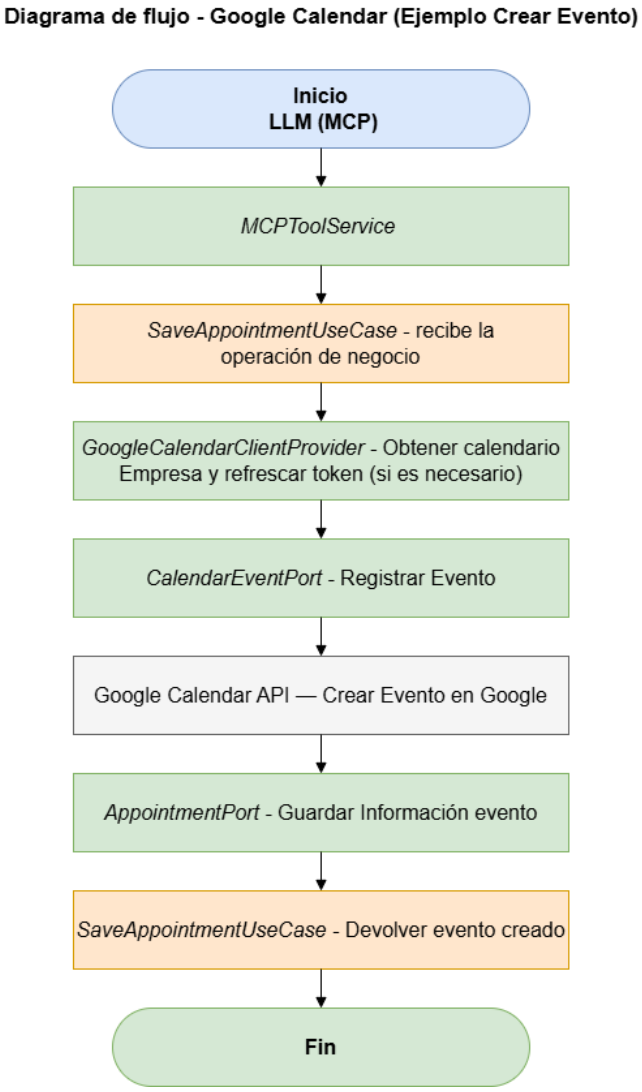


Figura B.4: Flujo Google Calendar

B.5 Flujo OAuth2

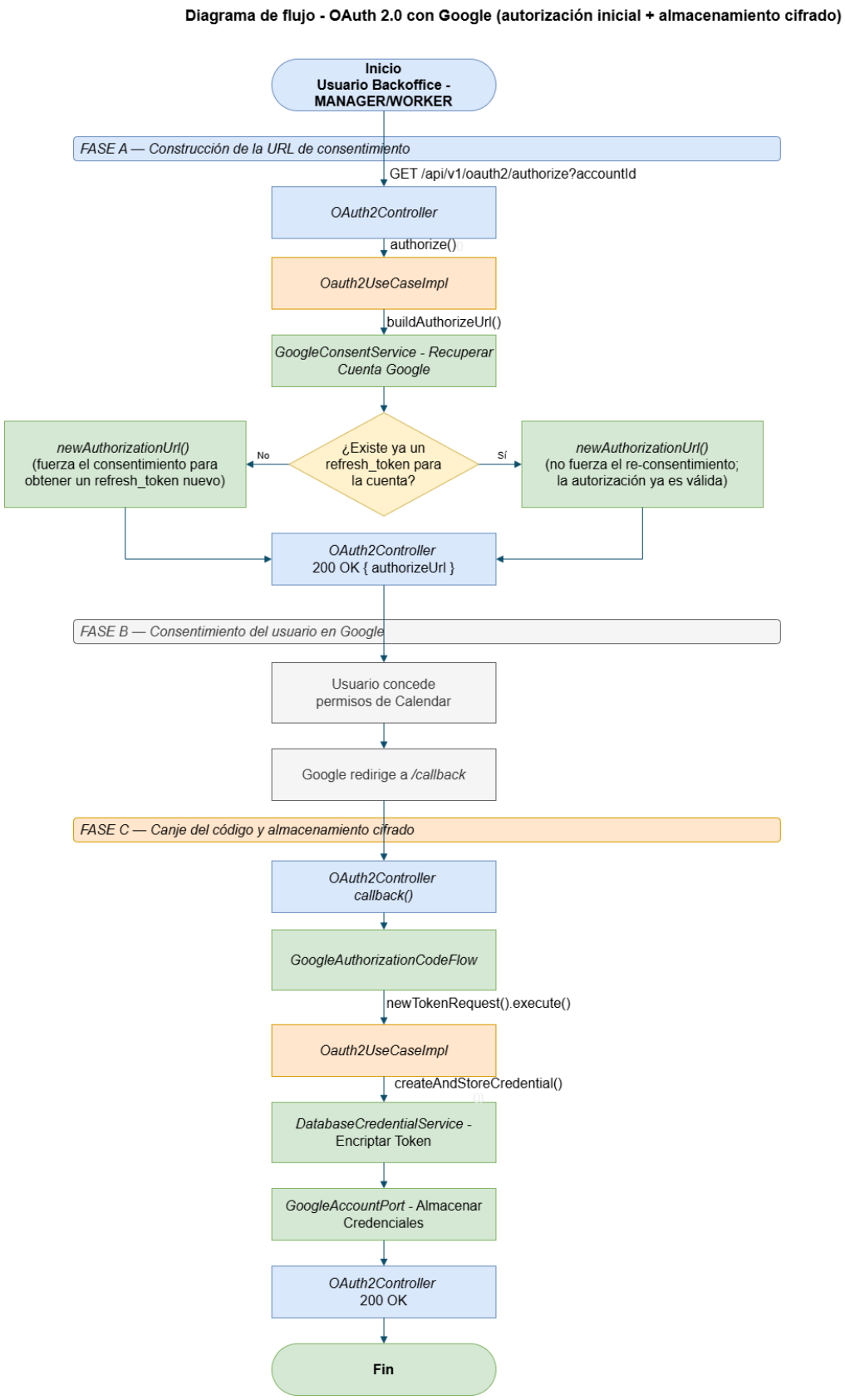


Figura B.5: Flujo OAuth2

B.6 Flujo SpringAI

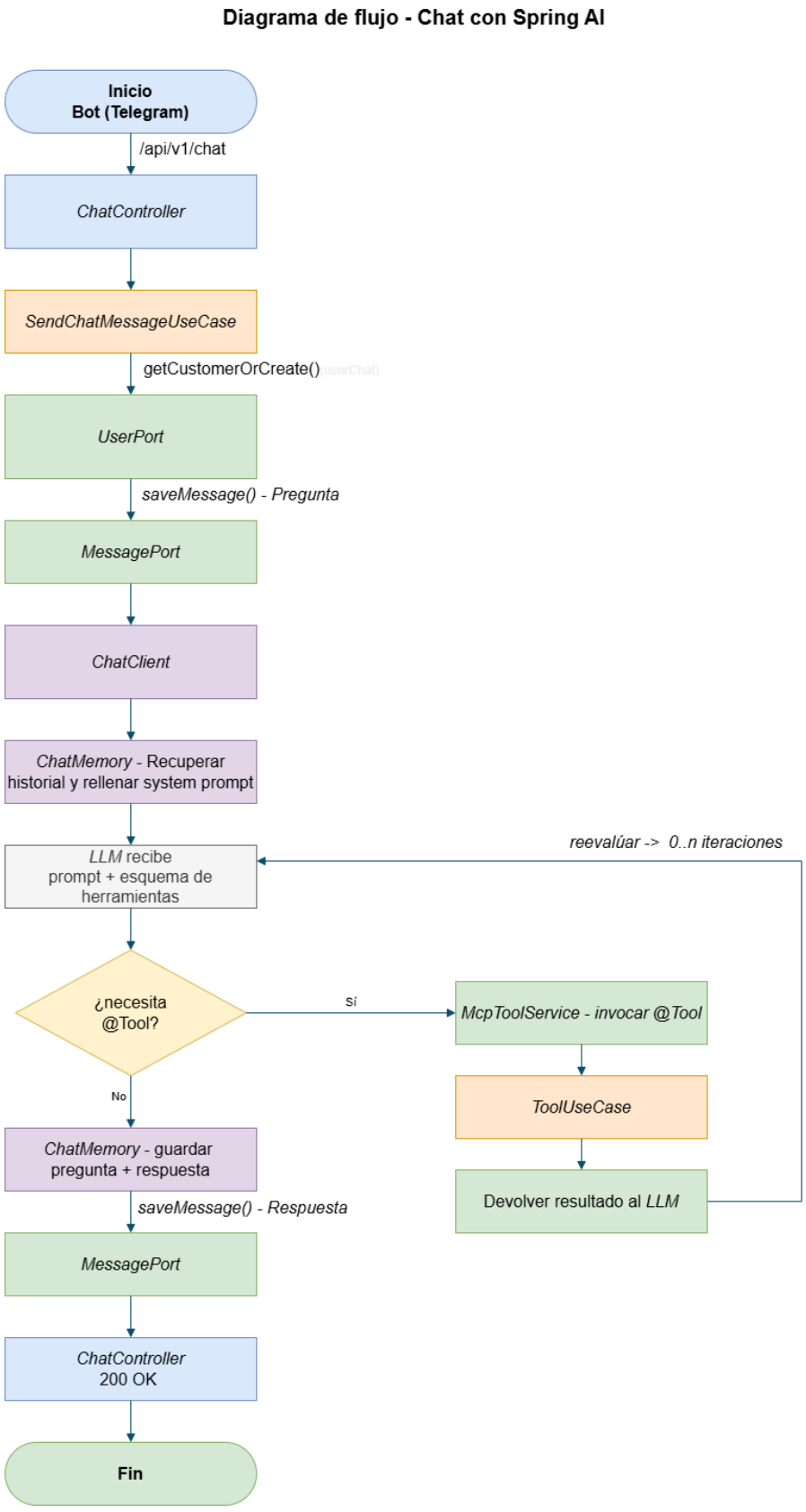


Figura B.6: Flujo chat

Anexo C

Apéndice C - Técnicas de Ingeniería de Prompts

C.1 Técnicas de Ingeniería de Prompts

Este anexo amplía el detalle sobre las técnicas de ingeniería del Prompt RASCEF y CLEAR que han sido utilizadas para la generación del *system prompt* apoyándose en una versión muy reducida del mismo.

RASCEF (Role – Action – Steps – Context – Examples – Format):

Propone una estructura bien definida para un *prompt* en seis componentes: *Role, Action, Steps, Context, Examples, Format*.

- **Role:** Definir la personalidad o experiencia del modelo (*Quién* es).
- **Action:** Establecer el objetivo principal o la meta global del *prompt* (*Qué* debe lograr).
- **Steps:** Dividir el trabajo pesado en instrucciones u órdenes secuenciales (Procedimiento ordenado).
- **Context:** Aportar la información de fondo esencial (Información situacional).
- **Examples:** Proporcionar una muestra del estilo o del formato esperado para guiar los resultados del modelo.
- **Format:** Especificar el formato en el que se desea la respuesta del modelo.

CLEAR (Concise – Logical – Explicit – Adaptive – Reflective):

Evalúa la calidad de un *prompt* según cinco propiedades: **Concise, Logical, Explicit, Adaptive y Reflective**.

- **Concise:** Evitar texto de relleno, lenguaje preciso (sin *ruido*).
- **Logical:** Ordenar las peticiones en una secuencia natural, coherente y fácil de seguir.
- **Explicit:** Definir claramente qué debe de hacer el modelo, no suponer nada.
- **Adaptive:** El *prompt* debe ser adaptativo y susceptible de modificarse en base a los resultados del modelo.
- **Reflective:** Evaluar críticamente el resultado devuelto por el modelo (información exacta, *alucinaciones*, sesgos)

Se muestra un ejemplo de cómo aplicarían cada una de las propiedades en una versión muy simplificada del *prompt* del proyecto.

```
## ROLE          #[RASCEF: Role]
You are Barbot, the virtual assistant of the salon in RUNTIME CONTEXT.

## GOAL         #[RASCEF: Action]
Help customers book, modify and cancel appointments.

## RULES
#[CLEAR: Adaptive]
- Always respond in the customer's language.
#[CLEAR: Concise]
- Be friendly and concise; never expose IDs or raw tool output.
#[RASCEF: Format]
- Present every value in natural language (e.g. "el viernes a las
    17:00").
#[CLEAR: Reflective]
- Never confirm a booking unless the write tool returned 'success' =
    true.

## BOOK AN APPOINTMENT        #[RASCEF: Steps]
1. Identify the service the customer wants.
#[CLEAR: Explicit]
2. Call get_available_slots(companyId, serviceId, fromDate, toDate)
    before stating anything about availability.
#[CLEAR: Logical]
3. Propose 3 concrete slots, then wait for explicit confirmation
    before calling save_appointment.

Example reply:       #[RASCEF: Examples]
"Te puedo ofrecer el viernes a las 10:30 con Ana o el sabado a las
    11:00"

## RUNTIME CONTEXT        #[RASCEF: Context]
- companyName: {companyName}
- companyId: {companyId}
```

Code listing C.1: Prompt (RASCEF/CLEAR)

Anexo D

Apéndice D - Herramientas MCP

Herramientas expuestas por McpToolService (`@Tool`)		
Herramienta	**Tipo**	**Propósito**
get_company_info	Lectura	Recupera los datos generales de la empresa (nombre, dirección, contacto)
get_company_services	Lectura	Devuelve el catálogo de servicios con precios y duración
get_company_schedule	Lectura	Devuelve los horarios de apertura por día y franja (mañana/tarde)
get_company_workers	Lectura	Lista los trabajadores (nombre e ID); se invoca al inicio de la conversación
get_customer_info	Lectura	Verifica si el cliente está registrado y recupera sus datos
update_customer_info	Escritura	Registra un nuevo cliente o actualiza sus datos; devuelve `ToolResult`
save_appointment	Escritura	Crea una nueva cita tras confirmación; devuelve `ToolResult`
get_customer_appointments	Lectura	Lista todas las citas del cliente en la empresa (pasadas y futuras)
update_appointment	Escritura	Modifica una cita existente (fecha, hora, trabajador o servicio); devuelve `ToolResult`
cancel_appointment	Escritura	Marca la cita como CANCELLED tras confirmación; devuelve `ToolResult`
get_open_windows	Lectura	Devuelve las ventanas horarias en que el salón está abierto en un rango
get_available_slots	Lectura	Única fuente de huecos reservables ya filtrados (horarios, ausencias, reservas)
get_customer_messages	Lectura	Recupera el historial de mensajes del cliente en la empresa

Tabla D.1: Herramientas (*tools*) expuestas por McpToolService al modelo

References

[aio24] aiohttp Developers. *aiohttp Documentation*, 2024. url: https://docs.aiohttp.org/.

[Ant24] Anthropic. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, November 2024.

[Axi24] Axios. *Axios Documentation*, 2024. url: https://axios-http.com/docs/intro.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, 1988.

[Boo26] Booksy. Booksy — Sistema de reservas online. https://booksy.com, 2026.

[Cal26] Calendly. Calendly — Automated scheduling. https://calendly.com, 2026.

[Coc05] Alistair Cockburn. Hexagonal Architecture (Ports and Adapters). https://alistair.cockburn.us/hexagonal-architecture/, 2005.

[Doc24] Docker, Inc. *Docker Documentation*, 2024. url: https://docs.docker.com/.

[Eur25] Eurostat. Use of Artificial Intelligence in Enterprises. Statistics Explained, European Commission, 2025. url: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Use_of_artificial_intelligence_in_enterprises.

[Fre26] Fresha. Fresha — Software de reservas para belleza y bienestar. https://www.fresha.com, 2026.

[Goo24a] Google. Google API Client Library for Java (google-api-java-client). https://github.com/googleapis/google-api-java-client, 2024.

[Goo24b] Google. *Google Calendar API Documentation*, 2024. url: https://developers.google.com/calendar/api/guides/overview.

[Har12] Dick Hardt. RFC 6749: The OAuth 2.0 Authorization Framework. Internet Engineering Task Force (IETF), October 2012. url: https://www.rfc-editor.org/rfc/rfc6749.

[JBS15] Michael B. Jones, John Bradley, y Nat Sakimura. RFC 7519: JSON Web Token (JWT). Internet Engineering Task Force (IETF), May 2015. url: https://www.rfc-editor.org/rfc/rfc7519.

[Jon15] Michael B. Jones. RFC 7518: JSON Web Algorithms (JWA). Internet Engineering Task Force (IETF), May 2015. url: https://www.rfc-editor.org/rfc/rfc7518.

[LB03] Craig Larman y Victor R. Basili. Iterative and Incremental Development: A Brief History. *Computer*, 36(6):47–56, 2003.

[Lo23] Leo S. Lo. The CLEAR path: A framework for enhancing information literacy through prompt engineering. *The Journal of Academic Librarianship*, 49(4):102720, 2023.

[Mar17] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Upper Saddle River, NJ, USA, edición 1st, 2017.

[Met24] Meta Open Source. *React Documentation*, 2024. url: https://react.dev/.

[Mic24] Microsoft. *TypeScript Documentation*, 2024. url: https://www.typescriptlang.org/docs/.

[Ope24] OpenAI. OpenAI API Documentation. https://platform.openai.com/docs/, 2024.

[Ora23] Oracle. *Java Platform, Standard Edition Documentation, Version 21*, 2023. url: https://docs.oracle.com/en/java/javase/21/.

[Ora26] Oracle. Oracle Java SE Support Roadmap. https://www.oracle.com/java/technologies/java-se-support-roadmap.html, 2026.

[pyt24] python-telegram-bot Developers. python-telegram-bot. https://python-telegram-bot.org, 2024.

[Red24a] Red Hat. *Hibernate ORM Documentation*, 2024. url: https://hibernate.org/orm/documentation/.

[Red24b] Redgate Software. *Flyway Documentation*, 2024. url: https://documentation.red-gate.com/flyway.

[Rem24] Remix Software. *React Router Documentation*, 2024. url: https://reactrouter.com/.

[RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, y Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. Technical report, OpenAI, San Francisco, CA, USA, 2018.

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems. En *Proceedings of IEEE WESCON*, páginas 1–9, Los Angeles, CA, USA, 1970. IEEE.

[Som16] Ian Sommerville. *Software Engineering*. Pearson Education Limited, Harlow, England, edición 10th, 2016.

[Spr24a] Spring Team. *Spring AI Reference Documentation*, 2024. url: https://docs.spring.io/spring-ai/reference/.

[Spr24b] Spring Team. *Spring Boot Reference Documentation*, 2024. url: https://docs.spring.io/spring-boot/.

[Spr24c] Spring Team. *Spring Data JPA Reference Documentation*, 2024. url: https://docs.spring.io/spring-data/jpa/reference/.

[Spr24d] Spring Team. *Spring Framework Documentation*, 2024. url: https://docs.spring.io/spring-framework/reference/.

[Spr24e] Spring Team. *Spring Security Reference Documentation*, 2024. url: https://docs.spring.io/spring-security/reference/.

[Tel24] Telegram. *Telegram Bot API*, 2024. url: https://core.telegram.org/bots/api.

[The24a] The Apache Software Foundation. *Apache Maven Documentation*, 2024. url: https://maven.apache.org/guides/.

[The24b] The PostgreSQL Global Development Group. *PostgreSQL Documentation*, 2024. url: https://www.postgresql.org/docs/.

[Viv25] Fabio Vivas. RASCEF Framework: Role, Action, Steps, Context, Examples, Format. https://fvivas.com/en/rascef-framework-prompts-llm/, 2025.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, y Illia Polosukhin. Attention Is All You Need. En *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, páginas 5998–6008, Long Beach, CA, USA, 2017. Curran Associates, Inc.

Este documento fue editado y tipografiado con LaTeX empleando
la clase **esi-tfg** (versión 0.20181017) que se puede encontrar en:
`https://bitbucket.org/esi_atc/esi-tfg`