HealthTrack: Sistema Integrado de Gestión Hospitalaria y Monitorización Remota

Román Ardanuy Hamill

El presente trabajo de fin de grado consiste en el desarrollo de HealthTrack, una plataforma digital orientada a mejorar la comunicación de los profesionales sanitarios y pacientes, tanto para clínicas pequeñas, medianas o hospitales.

El sistema incluye una aplicación web para doctores y una aplicación móvil para pacientes, ambas conectadas con un backend común. HealthTrack permite gestionar citas médicas, intercambiar mensajes en tiempo real, autenticar usuarios con un control de acceso por roles y mantener una experiencia de usuario cuidada y moderna. El proyecto se ha desarrollado siguiendo una metodología iterativa, utilizando tecnologías actuales como React Native, Next.js, Prisma, TypeScript y PostgreSQL.

El resultado final es una solución funcional, segura y escalable, con posibilidad real de evolución y uso en entornos productivos. Además, también se ha documentado todo el proceso para facilitar su validación académica y su continuidad fuera.

Paraules clau — Comunicación médico-paciente, mensajería en tiempo real, gestión de citas, generar informes, React native, Software sanitario

The present Final Degree Project presents the development of HealthTrack, a digital platform designed to enhance communication between healthcare professionals and patients, especially in small to medium-sized private clinics.

The system includes a web application for doctors and a mobile app for patients, both connected through a shared backend. HealthTrack enables appointment management, real-time messaging, role-based authentication, and a clean, user-friendly interface. The project followed an iterative methodology and was built using modern technologies such as React Native, Next.js, Prisma, and PostgreSQL.

The result is a functional, secure, and scalable solution with real potential for production use and future evolution. In addition, the development has also been documented to aid with academic validation and its future outdoors.

Index Terms — Healthcare communication, real-time messaging, appointment management, report generation, React Native, medical software

Aquest treball de fi de grau consisteix en el desenvolupament de HealthTrack, una plataforma digital orientada a millorar la comunicació entre professionals sanitaris i pacients, tant en clíniques petites i mitjanes com en hospitals.

El sistema inclou una aplicació web per metges i una aplicació mòbil per a pacients, ambdues connectades a un back-end comú. HealthTrack permet gestionar cites mèdiques, intercanviar missatges en temps real, autenticar usuaris amb control d'accés per rols i mantenir una experiència d'usuari cuidada i moderna. El projecte s'ha desenvolupat seguint una metodologia iterativa i utilitzant tecnologies actuals com React Native, React, Next.js, TypeScript i Prisma.

El resultat final és una solució funcional, segura i escalable, amb possibilitat real d'evolució i ús en entorns productius. A més, també s'ha documentat tot el procés per facilitar la seva validació acadèmica i la seva continuïtat en el futur.

Paraules clau — Comunicació metge-pacient, missatgeria en temps real, gestió de cites, informes, React Native, programari sanitari.

1 Introducción - Contexto del trabajo

L

En mi caso personal, he podido vivir de cerca esta problemática acompañando a mi madre, paciente crónica por su Esclerosis Múltiple, en diversas citas médicas. En múltiples ocasiones nos hemos enfrentado a esperas innecesarias, citas desorganizadas y dificultades para comunicarnos con los profesionales responsables. Es más, mostraban problemas para comunicarse entre ellos y entre departamentos.

HealthTrack nace como una respuesta a esta problemática, con el objetivo de ofrecer una solución tecnológica moderna que mejore la gestión de la comunicación, el seguimiento del paciente y las citas médicas. El proyecto combina elementos clave de mi formación académica: la experiencia adquirida durante mis prácticas en SensingTex, una empresa de software sanitario, y las asignaturas de Laboratorio integrado de software (LIS) y Diseño de software (DS), en la que desarrollé mi primera aplicación y aprendí las claves de la gestión y el desarrollo de software.

A partir de esta motivación, se ha desarrollado una plataforma compuesta por una aplicación web para profesionales sanitarios y una aplicación móvil para pacientes, conectados mediante un Backend común. El objetivo es permitir una gestión más ágil de citas, un canal de comunicación directa entre pacientes y profesionales, y una experiencia de uso optimizada y accesible para ambos perfiles.

Este documento se estructura de la siguiente forma: en la sección 2 se presentan los objetivos del proyecto; en la sección 3, el estado del arte y los sistemas existentes; en la sección 4, la metodología de desarrollo; en la sección 5, los resultados obtenidos; y en la sección 6, las conclusiones y propuestas de mejora futura.
Finalmente, se incluyen los agradecimientos y la bibliografía consultada

2 Objetivos

2.1 Objetivo general

El objetivo principal de este proyecto es desarrollar una plataforma digital que mejore la comunicación entre pacientes y profesionales sanitarios, especialmente en el ámbito de clínicas privadas pequeñas y medianas, ya que muestran el mejor prospecto de negocio.

La solución debe facilitar la gestión de citas, el seguimiento de pacientes crónicos y la comunicación directa mediante una aplicación web para profesionales y una aplicación móvil para pacientes, sobre una arquitectura técnica segura, escalable y moderna.

2.1 Objetivos específicos

Para alcanzar este objetivo general, se definen los siguientes objetivos específicos, priorizados según su relevancia funcional y el orden lógico de desarrollo:

1. Diseñar una arquitectura modular y escalable basada en un monorepositorio, que permita compartir código entre la aplicación web y la móvil.

2. Implementar una aplicación web para profesionales sanitarios que permita gestionar pacientes, citas y mensajes de forma centralizada.

3. Desarrollar una aplicación móvil para pacientes, accesible y fácil de usar, desde la que puedan consultar sus citas y comunicarse con los profesionales.

4. Incorporar un sistema de mensajería en tiempo real entre paciente y profesional.

5. Aplicar principios de desarrollo centrado en el usuario para garantizar una experiencia de uso óptima, especialmente en el contexto sanitario.

6. Desarrollar un sistema de autenticación con control de acceso basado en roles diferenciados (paciente y profesional).

7. Documentar de forma estructurada la solución desarrollada: arquitectura, base de datos, interfaz y guías

Actualmente existen diversas plataformas de software orientadas a la gestión de clínicas médicas, especialmente en el sector privado. Entra le las más destacadas se encuentran Clinic Cloud, DriCloud y Dasi Clinic, todas ellas con funcionalidades avanzadas como la gestión de historiales clínicos, programación de citas, emisión de facturas o generación de informes. Estas soluciones están pensadas para cubrir una amplia gama de necesidades clínicas, especialmente en centros con cierto grado de digitalización previa.

Además, estas plataformas ofrecen módulos configurables y servicios en la nube, lo que permite cierto grado de flexibilidad. No obstante, su enfoque suele estar centrado en la administración interna de la clínica, y muchas veces relegan la experiencia del paciente a funcionalidades mínimas como recordatorios por correo o SMS, sin ofrecer una interacción directa ni fluida entre ambas partes.

3.2 Limitaciones detectadas

A pesar e su madurez técnica, las soluciones existentes presentan varios puntos débiles, especialmente cuando se analizan desde el punto de vista del paciente o de pequeñas clínicas con recursos limitados.

- **Falta de foco en la experiencia del paciente:** No todas las plataformas cuentan con aplicaciones móviles que permitan una interacción ágil
- **Escasa comunicación en tiempo real:** En la mayoría de los casos el paciente y el doctor se limitan a llamadas telefónicas o recordatorios.

- **Curva de aprendizaje elevada:** Muchas herramientas requieren formación especial o personal dedicado al a gestión de software.
- **Licencias costosas:** Los planes de pago están más orientados a clínicas medianas o grandes, dificultando su adopción por parte de centros pequeños.

3.3 Propuesta diferenciadora de HealthTrack

Frente a este contexto, HealthTrack propone una solución centrada exclusivamente resolver problemas de forma ágil, como la relación entre paciente y profesional y la gestión de ellos. El sistema ha sido concebido desde el inicio con los siguientes criterios diferenciadores:

- Una **aplicación web para profesionales sanitarios**,

Aunque no se ha seguido una metodología formal como Scrum o Kanban, el desarrollo del proyecto se ha llevado a cabo de manera estructurada. Se priorizó una construcción iterativa, por bloques de funcionalidades, en base a un flujo lógico de implementación que garantizara coherencia entre la web, la aplicación móvil y el backend.

4.2 Estructura del monorepositorio

La primera fase técnica consistió en configurar un monorepostorio con Turborepo, permitiendo un entorno unificado de trabajo para todas las partes del sistema. Este se dividió en:

- **Apps**: contiene la aplicación web construida con React, Typescript y Next.js y la aplicación móvil, con React Native, Typescript y Expo.
- **Packages**: incluye módulos reutilizables como api (servicios compartidos), types (tipado de typescript) y auth (lógica de autenticación).
- **Backend**: Contiene el servidor de la aplicación, desarrollado con Node.js y el framework Express, escrito completamente en Typescript

El backend implementa una API RESTful organizada en capas modulares: controladores, rutas, middlewares, validadores y servicios.

Para la persistencia de datos se ha utilizado una base de datos PostgreSQL, gestionada mediante el ORM Prisma, que permite definir modelos en un esquema declarativo y generar consultas tipadas.
En cuanto a la seguridad, el backend incorpora: jsonwebtoken para la gestión de autenticación mediante tokens JWT. bcrypt para el cifrado seguro de contraseñas de usuario. dotenv para la configuración del entorno y protección de credenciales. cors para permitir el acceso controlado desde la aplicación web y móvil durante el desarrollo y pruebas.

4.3 Tecnologías y herramientas utilizadas

4.3.2 Aplicación móvil

- **React Native**: framework para desarrollo de aplicaciones móviles nativas utilizando JavaScript y componentes declarativos.
- **Expo**: plataforma que simplifica el desarrollo, prueba y despliegue de apps React Native.
- **Expo Router**: sistema de navegación basado en el sistema de archivos, inspirado en el enrutado de Next.js.
- **React Native Paper**: biblioteca de componentes UI siguiendo las directrices de Material Design.
- **Expo SecureStore**: solución para almacenamiento seguro de credenciales en dispositivos móviles.

4.3.3 Backend y base de datos

- **Node.js**: entorno de ejecución para JavaScript del lado del servidor.
- **Express.js**: framework web minimalista utilizado para implementar la API RESTful del sistema.
- **PostgreSQL**: sistema de gestión de base de datos relacional, robusto y escalable.
- **Prisma**: ORM moderno que permite la definición de esquemas tipados y el acceso seguro a la base de datos.
- **Socket.io**: biblioteca que permite la comunicación bidireccional en tiempo real entre cliente y servidor.

4.3.4 Seguridad y autenticación

- **JSON Web Tokens (JWT)**: sistema de autentica-
- **helmet**: middleware de seguridad que protege la aplicación Express configurando cabeceras HTTP adecuadas.

4.4 Requisitos del sistema

4.4.1 Requsitos funcionales

El desarrollo se llevó a cabo por funcionalidades, trabajando en paralelo la aplicación web y la móvil, para garantizar que el código se compartiera lo máximo posible. Las principales funcionalidades implementadas en esta etapa fueron:

- **Autenticación de usuarios**: registro y login mediante correo electrónico y contraseña, con gestión de sesiones mediante tokens JWT.
- **Gestión de citas**: creación, visualización y actualización de citas médicas por parte del profesional; los pacientes pueden consultar el calendario actualizado en su aplicación.
- **Mensajería en tiempo real**: canal de comunicación bidireccional entre paciente y profesional, con envío instantáneo de mensajes y mantenimiento del historial.
- **Gestión de usuarios**: perfil diferenciado para pacientes y profesionales; los médicos pueden ver y buscar su lista de pacientes asignados.
- **Interfaz dual**: plataforma accesible desde una aplicación web para profesionales y desde una aplicación móvil para pacientes.

4.4.2 Requsitos no funcionales

Además de los aspectos funcionales, el sistema ha sido diseñado con los siguientes requisitos técnicos y de calidad:

- **Rendimiento**: las respuestas de la API están optimizadas para mantenerse por debajo de los 200ms en entornos locales de prueba.
- **Seguridad**:
  - Encriptación de contraseñas con bcrypt.
  - Tokens JWT con expiración y validación.
  - Middleware de protección de rutas y validación de entrada en los endpoints.
- **Usabilidad**: tanto la app como la web han sido diseñadas con navegación clara, textos accesibles y estilos adaptables.

4.4.3 Análisis de riesgos

Durante el desarrollo y futura implementación del sofware, se han identificado diferentes riesgos potenciales:

- Seguridad de la información:
  - Riesgo: acceso no autorizado a datos médicos sensibles
  - Se ha resuelto con autenticación segura (JWT + bcrypt) y control de roles.
- Disponibilidad del sistema
  - Riesgo: Caídas del servidor backend o problemas de conexión que puedan afectar el acceso de profesionales o pacientes
  - Solución: desplegando el sistema en entornos cloud con auto-scaling y balanceo de carga
- Integridad de los datos
  - Pérdida de información médica o inconsistencia de mensajes.
  - Se puede resolver mediante una base de datos relacional robusta y backups periódicos.
- Cumplimiento legal
  - Este sería el riesgo más grande del proyecto, incumplir la normativa de protección de datos
  - Solución: Almacenar mínimamente los datos sensibles, cifrar contraseñas, control de acceso e implementación de consentimiento explícito.

4.5 Arquitectura y modelo de datos

4.5.1 Arquitectura general del sistema

El sistema se basa en una arquitectura cliente-servidor con múltiples capas.

Los dos clientes (web y móvil) se comunican con el servidor mediante peticiones HTTP a una API REST. Toda la lógica de negocio reside en el backend, desarrollado en TypeScript sobre express. La base de datos de PostgreSQL se gestiona con prisma ORM, que permite mantener un esquema estructura y realizar consultas tipadas.

4.5.2 Modelo de datos principal

Las entidades centrales del sistema son:

- **User**: contiene información común como email, nombre y rol.
- **Patient** y Professional: extensiones del perfil base con información adicional específica.
- **Appointment:** almacena información sobre las citas médicas (fecha, estado, participantes).
- **Converstion** y **message**: representan la mensajería entre usuarios, con el control de emisor, contenido y timestamp.

Estas entidades están relacionadas mediante claves foráneas, permitiendo navegación eficiente en ambos sentidos (por ejemplo, obtener todas las citas de un paciente o todos los mensajes de una conversación).

4.5.3 Diagrama de secuencia login

Este diagrama detalla cómo se ha implementado el requisito funcional de login descrito anteriormente, mostrando las interacciones técnicas entre el frontend, backend y base de datos.

Figura 2

4.6 Cumplimiento de protección de datos (GPDR)

HealthTracks gestiona información médica y de identificación personal de los usuarios, divididos en su rol. Por ello, esta sujeta al reglamento general de protección de datos (GPDR) de la Unión Europea, que clasifica los datos de salud como datos especialmente protegidos.

En el estado actual del software, se han implementado las siguientes medidas:

- Autenticación segura, cifrando las contraseñas con bcrypt.
- Control de acceso por roles (paciente y profesional).
- Almacenamiento seguro: Uso de postgreSQL con acceso restringido y variables de entorno para la configuración de credenciales.

Y luego, se han identificado las siguientes mejoras necesarias para el cumplimiento total:

- Consentimiento explícito, pantalla de aceptación de términos y política de privacidad antes de usar la plataforma.
- Logs de acceso y auditoría, que sería un registro de acceso, vista de datos personales para trazabilidad.
- Responsable de protección de datos: Tener un responsable de protección de datos.

5 RETOS TÉCNICOS Y SOLUCIONES

IMPLEMENTADAS

Como en cualquier desarrollo, se han encontrado múltiples problemas. Se nombrarán a continuación algunos ejemplos.

5.1 Arquitectura y modelo de datos

5.1.1 Problema de navegación en React Native

- **Reto**: al iniciar la aplicación móvil, se mostraba la pantalla por defecto de Expo ("Welcome to Expo") en lugar de la pantalla de login.
- **Causa**: coexistencia de dos directorios app/, uno con la estructura correcta de rutas de Expo Router y otro vacío, que generaba confusión en la resolución del punto de entrada.
- **Solución**:
    - Eliminación del directorio conflictivo /src/app/.
    - Configuración adecuada de babel.config.js con el plugin expo-router/babel.
    - Definición explícita del entry point "expo-router/entry" en package.json.
    - Implementación de redirecciones adecuadas en app/index.tsx.

5.1.2 Problema de persistencia de autenticación en la app móvil

- **Reto**: el estado de autenticación no se mantenía entre sesiones, lo que obligaba al usuario a iniciar sesión cada vez.
- **Solución**:
    - Implementación de un AuthProvider utilizando React Context.
    - Uso de expo-secure-store para guardar el token JWT de forma segura en el dispositivo.
    - Verificación automática del token al iniciar la aplicación.
    - Manejo de expiración y renovación de tokens.

5.1.3 Error 403 Forbidden tras login

- **Reto**: después de un login exitoso, las llamadas a la API devolvían errores 403.
- **Diagnóstico**:
    - Análisis de headers HTTP mediante logging.
    - Verificación de la correcta inclusión del token Bearer.
    - Revisión del middleware de autenticación en el backend.
- **Solución**:
    - Corrección en la función addAuthHeader del cliente API compartido.
    - Implementación de logging detallado para trazabilidad.
    - Manejo de errores 403 con lógica de reintento automático.

5.2 Gestión eficiente de datos

5.2.1 Mejoras de rendimiento

Para mejorar el rendimiento y reducir el tiempo de carga:

- Implementación de **lazy loading** de conversaciones y mensajes.
- **Paginación** en las listas de citas y pacientes.
- **Caché local** para datos frecuentemente accedidos.
- **Actualización optimista** en la interfaz de mensajería.

5.2.2 Manejo de errores y fallbacks

- Uso de datos ficticios (**mock data**) como fallback en caso de fallo de la API.
- Sistema de **reintentos automáticos** para peticiones que fallen temporalmente.
- Implementación de **error boundaries** en React para capturar fallos de componentes.
- Mensajes de estado informativos al usuario sobre conectividad y fallos.

Esta sección fue añadida para demostrar que el desarrollo no se limitó a implementar funcionalidades, sino que también incluyó un trabajo de depuración, refactorización y mejora continua orientado a ofrecer una experiencia sólida.

6 RESULTADOS Y DISCUSIÓN

6.1 Validación funcional del sistema

Una vez finalizado el desarrollo de la arquitectura y los módulos principales, se procedió a realizar pruebas funcionales para validar el correcto funcionamiento de cada una de las funcionalidades implementadas, además del testeo de forma continuada a medida que se programaba todo.

En primer lugar, se comprobó el flujo completo de autenticación de usuarios: registro inicio de sesión y persistencia de la sesión con token JWT tanto en la aplicación móvil como en la versión web. Posteriormente, se verificó que los usuarios, según su rol, tuvieran acceso únicamente a las vistas y acciones permitida.

Se realizaron pruebas de gestión de citas, asegurando que los médicos pudieran crear, modificar y listar citas, y que los pacientes pudieran visualizarlas correctamente en su aplicación móvil. También se validó la lógica de actualización del estado de cada cita y la correcta vinculación con los identificadores de usuario.

Finalmente, se validó la comunicación en tiempo real entre usuarios. A través de sockets, los mensajes entre paciente y profesional se transmiten instantáneamente, y se almacenan en la base de datos junto con su timestamp y metainformación. Se realizaron pruebas cruzadas entre diferentes dispositivos para asegurar la sincronización entre

plataformas.

De este modo, se pudo confirmar que el sistema era funcional y que respondía de forma estable ante los establecidos flujos principales.

6.2 Validación funcional del sistema

Durante la fase final del proyecto, se prepararon escenarios concretos que permiten demostrar el valor práctico del sistema. Entre ellos, se destacan los siguientes:

- **Flujo profesional-paciente**: un profesional inicia sesión en la plataforma web, accede a la lista de pacientes, selecciona uno y crea una cita médica para una fecha concreta. Desde esa misma vista, puede iniciar una conversación con el paciente para confirmar la cita o realizar recomendaciones previas.
- **Visualización de la cita desde la app**: tras la creación de una nueva cita desde la plataforma web, el paciente recibe automáticamente la actualización en su aplicación móvil. Desde la pestaña de citas, puede ver la nueva entrada, acceder a sus detalles y confirmar su asistencia. Esto permite validar que la sincronización entre backend y frontend móvil es efectiva y en tiempo real.
- **Interacción mediante mensajería**: el paciente abre la aplicación, accede al chat con el profesional y responde a un mensaje reciente. La respuesta es recibida instantáneamente en la plataforma web. El historial de conversación queda almacenado y accesible desde ambas plataformas.
- **Escenario completo con múltiples pacientes**: se realizaron pruebas con varios usuarios simultáneamente, asegurando que cada profesional solo puede ver a sus pacientes asignados y viceversa. Esto permite simular un entorno real de una clínica en funcionamiento, con múltiples usuarios activos.

6.3 Métricas de rendimiento

Se realizaron pruebas de rendimiento en entorno local para validar la eficiencia del sistema en operaciones comunes. Los resultados fueron los siguientes:

- **Tiempo de respuesta de la API**: inferior a 200 milisegundos para operaciones CRUD básicas (creación, lectura, actualización y eliminación).
- **Latencia en la mensajería**: inferior a 50 milisegundos en el canal de comunicación en tiempo real.
- **Tiempo de carga inicial**:
    - Versión web: inferior a 3 segundos.
    - Aplicación móvil: inferior a 5 segundos.
- **Uso de memoria**: estable durante sesiones prolongadas, sin detección de pérdidas de memoria (memory leaks).

6.3 Documentación VitePress

Durante el desarrollo del proyecto, se implementó un sistema de documentación técnica completo utilizando **VitePress**, una herramienta moderna para la generación de sitios de documentación estáticos. Esta documentación no solo cumple con los requisitos académicos del TFG, sino que también proporciona un recurso valioso para futuros desarrolladores y mantenedores del sistema.

6.3.1 Estructura de la documentación:

La documentación se organizó de manera modular, abarcando todos los aspectos técnicos y funcionales del sistema:

- **Documentación técnica**: Incluye la arquitectura del sistema, diseño de base de datos e implementación de seguridad
- **Referencia de API**: Documentación completa de todos los endpoints REST, métodos de autenticación y ejemplos de uso
- **Guías de usuario**: Manuales detallados para médicos, pacientes y administradores del sistema
- **Guías de despliegue**: Instrucciones paso a paso para instalación en desarrollo y producción, incluyendo configuración con Docker y Nginx

6.3.2 Características implementadas:

La documentación incorpora una navegación intuitiva mediante un sidebar organizado por categorías con búsqueda local integrada, facilitando el acceso rápido a cualquier sección del sistema.

Se implementó resaltado de sintaxis para múltiples lenguajes de programación (JavaScript, TypeScript, SQL, Shell), mejorando la legibilidad del código de ejemplo.

Además, se desarrollaron componentes interactivos como tablas de estado, grids de tecnologías y enlaces contextuales que enriquecen la experiencia del usuario.

7 CONCLUSIONES Y TRABAJO FUTURO

7.1 Conclusiones

Gracias al trabajo empleado, el proyecto HealthTrack ha alcanzado con éxito los objetivos definidos al inicio del trabajo.

Se ha desarrollado una plataforma funcional que permite mejorar la comunicación entre pacientes y profesionales sanitarios, facilitando la gestión de citas médicas y el seguimiento a través de mensajería en tiempo real. La implementación conjunta de una aplicación web y una aplicación móvil, conectadas mediante un backend común, ha permitido ofrecer una experiencia integrada, moderna y

accesible desde distintos dispositivos.

La arquitectura modular, el uso de tecnologías actuales como Next.js, React Native, Express y Prisma, y la organización en un monorepositorio han permitido un desarrollo ágil y coherente. Se han cubierto aspectos clave como la seguridad, la persistencia de datos, la validación de usuarios y la diferenciación de roles.

7.1 Mejoras técnicas prioritarias

En el futuro, para poder llevar al mercado real la aplicación, se realizarían estos cambios:

Notificaciones push nativas

- Implementación de **Expo Notifications** para el envío de notificaciones a la aplicación móvil.
- Sistema de notificaciones web utilizando **Service Workers**.
- Configuración de un **servidor de notificaciones escalable**, capaz de manejar múltiples usuarios y eventos en paralelo.

Optimización de rendimiento

- Uso de **Redis** como sistema de caché distribuido para acelerar respuestas del backend.
- Mejora de las **consultas a base de datos** mediante el uso de índices y optimización de relaciones.
- Aplicación de **lazy loading avanzado** y técnicas de virtualización para listas largas.

Seguridad avanzada

- Implementación de **refresh tokens** para reforzar la autenticación sin comprometer la experiencia de usuario.
- **Auditoría de accesos** y generación de logs de seguridad para trazabilidad.
- Cifrado **end-to-end** para mensajes sensibles entre paciente y profesional.

7.2 Mejoras técnicas prioritarias

En el caso de querer añadir funcionalidades nuevas, estas serían las siguientes:

Gestión de archivos médicos

- Subida y gestión de **documentos e imágenes clínicas** por parte de profesionales.
- Visualización integrada de **archivos PDF** y estudios médicos.
- Implementación de un sistema de **versionado de documentos** para mantener historial.

Telemedicina básica

- Integración de **videollamadas** para consultas médicas remotas.
- **Grabación de sesiones** bajo consentimiento, con almacenamiento seguro.
- Compatibilidad futura con **dispositivos médicos IoT** para obtener datos en tiempo real.

Analytics y reportes

- Desarrollo de un **dashboard de métricas** para profesionales sanitarios.
- Generación de **reportes de actividad y comunicación**.
- Análisis de **patrones de uso** para mejoras continuas y personalización.

8 VISIÓN DE NEGOCIO

Otra motivación importante que cabe mencionar es la posibilidad de llevar esta plataforma al mundo real. Después de razonarlo detenidamente, se ha ideado esta estrategia que se podría aplicar.

8.1 Modelo de monetización

HealthTrack se plantea como una solución SaaS (Software as a Service) adaptable a distintos tamaños de clínica. El modelo inicial propuesto es el “freemium”, en el que se ofrece una versión gratuita con funcionalidades esenciales (gestión de citas y mensajería), y una versión de pago con funcionalidades avanzadas como métricas, historial clínico ampliado, gestión documental y telemedicina.

El plan de pago estará estructurado en tiers por volumen de pacientes activos, lo que facilita la adopción por parte de clínicas pequeñas y permite escalar en clínicas medianas sin cambiar de sistema. A medio plazo, se contempla un sistema de licencias por suscripción mensual o anual, con soporte técnico incluido.

Además, se contempla la creación de un marketplace de integraciones, donde se ofrezcan módulos adicionales como conexión con farmacias, laboratorios o plataformas de receta electrónica, generando ingresos compartidos con terceros

8.2 Estrategia de crecimiento y alianzas

La fase inicial del proyecto se centrará en validar el producto con clínicas privadas pequeñas y medianas, priorizando entornos que aún no utilizan software especializado. Una vez validado el modelo, se buscarán alianzas estratégicas con redes de centros de salud, asociaciones médicas o entidades formativas.

Se contemplan colaboraciones con startups del sector

salud, proveedores de IoT médico y plataformas de servicios asistenciales, buscando un crecimiento conjunto basado en interoperabilidad y valor añadido para el paciente.

También, se puede ofrecer el desarrollo personalizado por empresa, que necesite determinada funcionalidad para resolver problemas, externa a la gestión de pacientes o las funcionalidades de HealthTrack.

9 Agradecimientos

Quisiera expresar mi agradecimiento a todas las personas que han hecho posible la realización de este Trabajo de Fin de Grado. En primer lugar, agradezco a mi tutor Miquel Trilla por su orientación, disponibilidad y aportaciones durante el desarrollo del proyecto.

También quiero agradecer el apoyo de mi familia, especialmente a mi madre, cuya experiencia personal fue una de las principales motivaciones para plantear este trabajo. Asimismo, agradezco a mis compañeros y compañeras por su colaboración, sugerencias y apoyo durante el proceso.

Este proyecto no solo ha sido una oportunidad para aplicar los conocimientos adquiridos a lo largo del grado, sino también una experiencia personal enriquecedora con la que me he demostrado que soy capaz de crear un software de cero y me ha dado fuerzas enfrentarme al mundo laboral.

10 Bibliografía

[1] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002.

[2] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, 2003.

[3] Meta Open Source, "React Native Documentation: Building Cross-Platform Mobile Apps," 2024. [Online]. Available: https://reactnative.dev

[4] Vercel, "Next.js Documentation: The React Framework for Production," 2024. [Online]. Available: https://nextjs.org

[5] PostgreSQL Global Development Group, "PostgreSQL Documentation," 2024. [Online]. Available: https://www.postgresql.org/docs/

[6] Prisma, "Modern Database Toolkit Documentation," 2024. [Online]. Available: https://www.prisma.io/docs

[7] World Health Organization, "Digital Health Strategy 2020–2025," 2019. [Online]. Available: https://www.who.int/publications/i/item/9789240020924

[8] HL7 Organization, "FHIR: Fast Healthcare Interoperability Resources Specification," 2024. [Online]. Available: https://www.hl7.org/fhir/

ANEXO

1 ESQUEMA DE BASES DE DATOS (SCHEMA.PRISMA)

Este archivo define la estructura de la base de datos utilizada en HealthTrack, incluyendo modelos, relaciones, claves foráneas y tipos de datos. Prisma facilita su definición declarativa y la generación de migraciones y consultas tipadas:

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id                   String      @id @default(uuid())
  email                String        @unique
  password             String
  name                 String
  surname              String
  role                 UserRole
  phone                String?
  profilePicture       String?
  createdAt            DateTime      @default(now())
  updatedAt            DateTime      @updatedAt
  createdAppointments      Appointment[] @relation("AppointmentCreator")
  patientAppointments      Appointment[] @relation("PatientAppointment")
  professionalAppointments Appointment[] @relation("ProfessionalAppointment")
  receivedMessages         Message[]     @relation("Receiver")
  sentMessages          Message[]    @relation("Sender")
  patientProfile           Patient?
  professionalProfile      Professional?
}

model Patient {
  id               String        @id @default(uuid())
  userId           String        @unique
  birthDate        String?
  address          String?
  emergencyContact String?
  professionalId   String
  measurements     Measurement[]
  professional     Professional  @relation(fields: [professionalId], references: [id])
  user                User            @relation(fields: [userId], references: [id], onDelete: Cascade)
}

model Professional {
  id            String    @id @default(uuid())
  userId        String    @unique
  specialty     String?
  licenseNumber String?
  patients      Patient[]
  user           User      @relation(fields: [userId], references: [id], onDelete: Cascade)
}

model Appointment {
  id             String            @id @default(uuid())
  createdById    String
  patientId      String
  professionalId String
  date           DateTime
  startTime      String
  endTime        String
  status         AppointmentStatus
  notes          String?
  reason         String?
  createdAt      DateTime          @default(now())
  updatedAt      DateTime          @updatedAt
  createdBy       User              @relation("AppointmentCreator", fields: [createdById], references: [id])
  patient        User              @relation("PatientAppointment", fields: [patientId], references: [id])
  professional    User               @relation("ProfessionalAppointment", fields: [professionalId], references: [id])
}

model Conversation {
  id             String    @id @default(uuid())
  participantIds String[]
  createdAt      DateTime  @default(now())
  updatedAt      DateTime  @updatedAt
  lastMessageId  String?   @unique
  lastMessage       Message?   @relation("LastMessage", fields: [lastMessageId], references: [id])
  messages           Message[]  @relation("ConversationMessages")
}

model Message {
  id             String        @id @default(uuid())
  senderId       String
  receiverId     String
  content        String
  read           Boolean       @default(false)
  createdAt      DateTime      @default(now())
  attachments    String[]
  conversationId String
  lastMessageFor Conversation? @relation("LastMessage")
  conversation   Conversation  @relation("ConversationMessages", fields: [conversationId], references: [id])
  receiver        User            @relation("Receiver", fields: [receiverId], references: [id])
  sender       User        @relation("Sender", fields: [senderId], references: [id])
}

model Measurement {
  id        String   @id @default(uuid())
  patientId String
  type      String
  value     Float
  unit      String
  date      DateTime @default(now())
  notes     String?
  patient   Patient  @relation(fields: [patientId], references: [id])
}

enum UserRole {
  PATIENT
  PROFESSIONAL
  ADMIN
}

enum AppointmentStatus {
  SCHEDULED
  CONFIRMED
  CANCELLED
  COMPLETED
}
```

2 CÓDIGO

A continuación, se mostrarán las partes del código más relevantes con el fin de ayudar a entender lo comentado en este informe.

2.1 Provider factory compartido

Implementación del cliente HTTP que maneja todas las comunicaciones con el backend. Utiliza el patrón Factory para crear instancias específicas de cada servicio (pacientes, citas, mensajes) manteniendo la autenticación centralizada.

```
// Factory para crear todas las APIs
export const createApi = (token?: string, baseURL?: string) => {
  return {
    auth: createAuthApi(baseURL),
    patients: token ? createPatientApi(token, baseURL) : null,
    appointments: token ? createAppointmentApi(token, baseURL) : null,
    messages: token ? createMessageApi(token, baseURL) : null,
  };
};

// Cliente de pacientes
export const createPatientApi = (token: string, baseURL?: string) => {
  const client = createApiClient(baseURL, token);

  return {
    getAll: (): Promise<Patient[]> => client.get('/patients'),
    getById: (id: string): Promise<Patient> => client.get(`/patients/${id}`),
    create: (patient: Omit<Patient, 'id' | 'createdAt' | 'updatedAt'>): Promise<Patient> =>
      client.post('/patients', patient),
    update: (id: string, patient: Partial<Patient>): Promise<Patient> =>
      client.put(`/patients/${id}`, patient),
    delete: (id: string): Promise<void> => client.delete(`/patients/${id}`),
  };
};
```

2.2 Cliente api compartido

Implementación del patrón Factory para crear providers de React que inyectan el cliente API en el contexto de la aplicación. Este patrón permite adaptar el almacenamiento de tokens según la plataforma (localStorage para web, SecureStore para móvil) manteniendo la misma interfaz

```
// Factory para crear todas las APIs
export const createApi = (token?: string, baseURL?: string) => {
  return {
    auth: createAuthApi(baseURL),
    patients: token ? createPatientApi(token, baseURL) : null,
    appointments: token ? createAppointmentApi(token, baseURL) : null,
    messages: token ? createMessageApi(token, baseURL) : null,
  };
};

// Cliente de pacientes
export const createPatientApi = (token: string, baseURL?: string) => {
  const client = createApiClient(baseURL, token);

  return {
    getAll: (): Promise<Patient[]> => client.get('/patients'),
    getById: (id: string): Promise<Patient> => client.get(`/patients/${id}`),
    create: (patient: Omit<Patient, 'id' | 'createdAt' | 'updatedAt'>): Promise<Patient> =>
      client.post('/patients', patient),
    update: (id: string, patient: Partial<Patient>): Promise<Patient> =>
      client.put(`/patients/${id}`, patient),
    delete: (id: string): Promise<void> => client.delete(`/patients/${id}`),
  };
};
```

2.3 Implementación web

Configuración específica para la aplicación web que utiliza localStorage para el almacenamiento de tokens de autenticación. Demuestra cómo el código compartido se adapta a las particularidades de la plataforma web.

```
export const ApiProvider = ({ children }: { children: React.ReactNode }) => {
  const auth = useAuth();

  const getToken = () => {
    if (typeof window !== 'undefined') {
      return localStorage.getItem('auth_token');
    }
    return null;
  };

  const ApiProviderComponent = createApiProvider(ApiContext, getToken);
  return ApiProviderComponent({ children });
};
```

2.4 Implementación móvil

Configuración específica para la aplicación móvil que utiliza Expo SecureStore para el almacenamiento seguro de tokens. Incluye lógica de refresco automático d**e tokens y manejo de estados de autenticación.**

```
export const ApiProvider = ({ children }: { children:
```

```
React.ReactNode }) => {
  const [token, setToken] = useState<string | null>(null);
  const auth = useAuth();

  useEffect(() => {
    const loadToken = async () => {
      try {
        const storedToken = await SecureStore.getItemAsync('auth_token');
        if (storedToken !== token) {
          setToken(storedToken);
        }
      } catch (error) {
        console.error('Error loading token:', error);
      }
    };

    loadToken();
    const tokenRefreshInterval = setInterval(loadToken, 5000);
    return () => clearInterval(tokenRefreshInterval);
  }, [auth.user]);

  const getToken = useCallback(() => token, [token]);
  const ExpoApiProvider = createApiProvider(ApiContext, getToken);
  return <ExpoApiProvider>{children}</ExpoApiProvider>;
};
```

3 CONFIGURACIÓN DEL PROYECTO

3.1 Estructura del monorepo

Configuración principal del proyecto que define la estructura de workspaces y scripts de automatización.

```
{
  "name": "healthtrack-system",
  "private": true,
  "workspaces": [
    "apps/*",
    "packages/*"
  ],
  "scripts": {
    "dev": "turbo run dev",
    "build": "turbo run build",
    "lint": "turbo run lint",
    "test": "turbo run test"
  },
  "devDependencies": {
    "turbo": "^1.10.0",
    "typescript": "^5.0.0"
  }
}
```

3.2 Configuración del monorepo

Configuración del sistema de build que optimiza la compilación y ejecución de tareas en el monorepo. Turbo permite ejecutar comandos en paralelo y cachear resultados, mejorando significativamente los tiempos de desarrollo y build.

```
{
  "$schema": "https://turbo.build/schema.json",
  "pipeline": {
    "build": {
      "dependsOn": ["^build"],
      "outputs": ["dist/**", ".next/**"]
    },
    "dev": {
      "cache": false,
      "persistent": true
    }
  }
}
```

4 AUTENTICACIÓN Y SEGURIDAD

4.1 Middleware de autenticación

Middleware que valida los tokens JWT en cada petición al backend. Este componente es fundamental para la seguridad del sistema, asegurando que solo usuarios autenticados puedan acceder a los recursos protegidos.

```
import jwt from 'jsonwebtoken';
import { Request, Response, NextFunction } from 'express';

export const authMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const token = req.headers.authorization?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ message: 'Token no proporcionado' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET!);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(403).json({ message: 'Token inválido' });
  }
};
```

4.2 Servicio de autenticación

Lógica de negocio para el manejo de autenticación, incluyendo verificación de credenciales, generación de tokens JWT y gestión de sesiones. Este servicio centraliza toda la lógica relacionada con la seguridad de usuarios.

```
import bcrypt from 'bcrypt';
```

```
import jwt from 'jsonwebtoken';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export class AuthService {
  async login(email: string, password: string) {
    const user = await prisma.user.findUnique({ where: {
email } });

    if   (!user   ||   !await   bcrypt.compare(password,
user.password)) {
      throw new Error('Credenciales inválidas');
    }

    const token = jwt.sign(
      { userId: user.id, role: user.role },
      process.env.JWT_SECRET!,
      { expiresIn: '24h' }
    );

    return { token, user: { ...user, password: undefined
} };
  }
}
```