



UNIVERSIDADE DA CORUÑA

Facultade de Informática

Máster Universitario en Enxeñaría Informática

TRABALLO DE FIN DE MÁSTER

**Sistema de alta seguridad basado en
Blockchain para la gestión privada de datos
de pacientes de servicios de salud digitales**

Estudiante: Ángel Paderne Cázar

Dirección: Tiago Manuel Fernández Caramés

A Coruña, febrero de 2025.

Por todos los que han hecho esto posible, gracias de corazón.

Agradecimientos

La verdad que durante el desarrollo de este proyecto me he sentido muy apoyado en muchos momentos. Trabajar mientras uno estudia puede llegar a ser algo complicado, y se me ha atragantado mucho. La verdad es que sin el apoyo de mi gente no lo hubiera conseguido, he pensado en rendirme muchas veces, pero gracias al apoyo de mis seres queridos no lo he hecho.

Gracias a Paula, a su cariño, a su apoyo diseñando Health Hub (el nombre de este proyecto). Por estar conmigo todo este tiempo, y ser la persona que ha visto más que nadie el esfuerzo que he puesto en este proyecto. Los últimos momentos han sido complicados y seguramente sin ti, me hubiera rendido. A mi familia, a mis padres, mi hermano y a polete, por estar siempre ahí con palabras de apoyo y cariño, por vuestros buenos consejos, me hubiera gustado acabar el Máster antes, pero no ha sido fácil. A mis amigos, por ser tan buenos amigos, por motivarme a acabar y por sobre todo ayudarme a evadirme.

A todos los profesores del máster que me han impartido clase, por haberme enseñado y apoyado tanto cuando estaba preparando las entrevistas para Facebook, si leéis esto, conseguí pasar las entrevistas, y haber trabajado allí me ha ayudado muchísimo a ser mejor ingeniero. A Javier Parapar, por plantearme preguntas de entrevistas en ratos muertos de clase. Gracias Tiago, por darme la oportunidad de implementar mi primer proyecto basado en blockchain en APM, ha pasado tiempo de eso, muchas idas y venidas, pero muchas gracias por tu paciencia, por siempre apostar por mí y por tu tiempo, tus palabras de ánimo también han hecho esto posible.

Resumen

En la actualidad, la gestión de datos médicos enfrenta diversos desafíos relacionados con la seguridad, interoperabilidad y accesibilidad de la información sanitaria. A pesar de los avances tecnológicos, persisten problemas como la fragmentación de datos entre distintos sistemas sanitarios, sistemas públicos y privados, entre comunidades autónomas y entre países. Estos datos suelen estar en bases de datos centralizadas, dejando un sólo punto de acceso expuestos a ciberataques y dejando hospitales inhabilitados. A mayores, los usuarios de los sistemas sanitarios no tenemos ningén control sobre nuestros datos, al no ser dueños de estos, estamos expuestos a que organizaciones vendan nuestros datos y puedan lucrarse de ello o que nuestros datos acaben en manos no deseadas. Este Trabajo de Fin de Máster propone el desarrollo de un sistema que combina la transparencia, inmutabilidad y descentralización de la tecnología blockchain, a través de la implementación de contratos inteligentes, con IPFS (Interplanetary-file system) para el almacenamiento off-chain. Mediante OrbitDB, IPFS nos permite almacenar nuestros datos siguiendo el mismo paradigma descentralizado que la tecnología blockchain, obteniendo como resultado un sistema escalable y robusto, que con distintos mecanismos criptográficos, nos permite almacenar, acceder y compartir nuestros valiosos datos médicos de forma segura. Como resultado de este trabajo, obtendremos un sistema capaz de alcanzar todos estos objetivos.

Abstract

Currently, medical data management faces various challenges related to the security, interoperability and accessibility of health information. Despite technological advances, problems persist, such as data fragmentation between different healthcare systems, public and private systems, between autonomous communities and between countries. These data are usually stored in centralized databases, leaving a single point of access exposed to cyber-attacks and leaving hospitals disabled. In addition, users of healthcare systems have no control over our data, as we are not the owners of our own data, we are exposed to organizations that sell our data and can profit from it. This Master's Thesis proposes the development of a system that combines the transparency, immutability and decentralization of blockchain technology, through the implementation of Smart Contracts, with IPFS. Through OrbitDB, IPFS allows us to store our data following the same decentralized paradigm as blockchain technology, obtaining as a result a scalable and robust system, which with different cryptographic mechanisms, allows us to store, access and share our valuable medical data securely. As a result of this work, we will obtain a system capable of achieving all these objectives.

Palabras clave:

- Blockchain
- Sistema sanitario
- App móvil
- OrbitDB
- IPFS

Keywords:

- Blockchain
- Healthcare
- Mobile App
- OrbitDB
- IPFS

Índice general

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	3
1.3	Metodología	4
1.3.1	Planificación	4
1.3.2	Análisis de requisitos	4
1.3.3	Diseño	5
1.3.4	Implementación	5
1.3.5	Pruebas	5
1.4	Estructura de la memoria	6
2	Estado del arte	7
2.1	Sistemas actuales de gestión de datos médicos	7
2.1.1	Interoperabilidad Deficiente	7
2.1.2	Seguridad y Ciberataques	8
2.1.3	Cumplimiento Regulatorio y Ética	8
2.1.4	Accesibilidad y Exclusión Digital	8
2.1.5	Falta de Confianza en la Integridad de los Datos	9
2.2	Aplicaciones de la tecnología blockchain en el sector salud	9
2.2.1	Historias Clínicas Electrónicas (EHR)	9
2.2.2	Trazabilidad y seguridad en la cadena de suministro farmacéutica	9
2.2.3	Caso de éxito: Implementación en sistema sanitario Estonia	10
2.2.4	Conclusiones estado del arte	10
3	Fundamentos tecnológicos	11
3.1	Hardware	11
3.2	Blockchain	11
3.3	Inter-planetary File System	12

3.4	OrbitDB	12
3.5	Criptografía	12
3.6	Algoritmos de criptografía simétrica	12
3.7	Algoritmos de criptografía simétrica	13
3.8	Lenguajes de programación	13
3.8.1	Solidity	13
3.8.2	Javascript	13
3.8.3	TypeScript	14
3.8.4	CSS	14
3.8.5	Node.js	14
3.8.6	Metamask	14
3.8.7	Remix	15
3.8.8	React Native	15
3.8.9	Redux Toolkit	15
3.8.10	Wagmi	15
3.8.11	Reown	15
3.8.12	Postman	16
3.8.13	Android Studio	16
3.8.14	Infura	16
3.8.15	Figma	16
4	Análisis	17
4.1	Actores del sistema	17
4.1.1	Pacientes	17
4.1.2	Doctores	18
4.1.3	Administradores	19
4.2	Entidades persistentes del sistema	20
4.3	Requisitos no funcionales	21
4.4	Requisitos funcionales	22
4.4.1	Paciente: Registro en la plataforma	22
4.4.2	Paciente: Inicio de sesión	22
4.4.3	Paciente: Visualizar sus propios datos médicos	22
4.4.4	Paciente: Visualización de los doctores autorizados	23
4.4.5	Paciente: Autorización de doctores	23
4.4.6	Paciente: Revocar autorización	23
4.4.7	Doctor: Iniciar sesión	24
4.4.8	Doctor: Acceder a los datos médicos de los pacientes	24
4.4.9	Doctor: Añadir datos médicos a los pacientes	24

4.4.10	Doctor: Visualizar los pacientes a los que tiene acceso	24
4.4.11	Doctor: Solicitar permisos a los pacientes	25
4.4.12	Administrador: Registrar doctores	25
4.4.13	Administrador: Verificación datos	25
4.4.14	Administrador: Visualización de logs	25
4.5	Casos de Uso	26
4.5.1	Autenticación	26
4.5.2	Registro de paciente	26
4.5.3	Registro de doctor	27
4.5.4	Paciente: Visualización de sus propios datos	28
4.5.5	Paciente: Añadir datos	29
4.5.6	Paciente: Visualizar permisos	29
4.5.7	Paciente: Dar permisos	29
4.5.8	Doctor: Visualización de los datos de un paciente.	29
4.5.9	Doctor: Solicitar permisos	29
4.5.10	Doctor: Visualizar pacientes	29
4.5.11	Doctor: Añadir datos paciente	30
5	Diseño	31
5.1	Arquitectura global	31
5.1.1	Diseño del Backend	31
5.1.2	OrbitDB	31
5.1.3	Contratos Inteligentes	35
5.2	Diseño del Frontend	39
5.2.1	Capa de acceso a servicios	39
5.2.2	Capa Interfaz de Usuario	39
6	Implementación	47
6.1	Estructura global	47
6.2	Implementación del Backend	47
6.2.1	Estructura del backend	47
6.2.2	OrbitDB	49
6.2.3	Rutas	52
6.2.4	Contratos inteligentes	58
6.3	Implementación del Frontend	64
6.3.1	Estructura del frontend	64
6.3.2	Capa de acceso a servicios	66
6.3.3	Capa Interfaz de Usuario	68

6.3.4	Estilos y Componentes	73
7	Pruebas	75
7.1	Backend	75
7.2	Servicio Node.js	75
7.3	Smart Contracts	76
7.4	Frontend	78
7.5	Pruebas de validación funcional	79
8	Incidencias, conclusiones y trabajo futuro	93
8.1	Conclusiones	93
8.2	Líneas futuras	95
A	Diagrama de Gantt y costes	99
A.1	Sprints	99
A.1.1	Sprint 0	99
A.1.2	Sprint 1	99
A.1.3	Sprint 2	99
A.1.4	Sprint 3	100
A.1.5	Sprint 4	100
A.1.6	Sprint 5	100
A.1.7	Sprint 6	100
A.2	Diagrama de Gantt	100
A.3	Costes del proyecto	100
	Bibliografía	103

Capítulo 1

Introducción

En este capítulo introductorio abordaremos aspectos fundamentales de este proyecto: el porqué, mediante la motivación que nos ha llevado a llevar a cabo este proyecto, el qué, mediante los objetivos que hemos perseguido, el cómo, mediante la metodología que hemos seguido para conseguir estos objetivos y, por último, explicaremos lo que se espera de este documento y de cómo está estructurado.

1.1 Motivación

La motivación inicial de este Trabajo de Fin de Máster (TFM) nace de una experiencia personal: he residido en Irlanda, y el tiempo que estuve residiendo allí, ¿qué pasaría si tuviera que ir al hospital? La realidad es que los sistemas sanitarios irlandeses no tienen forma de acceder a mis datos médicos españoles y viceversa. Esto despertó mi curiosidad sobre cómo se gestiona la información médica. Vivimos en un mundo globalizado, donde el movimiento de personas es masivo y todos estaremos de acuerdo en que nuestra salud, sobre todo cuando estamos en el extranjero, es de lo más importante. Resulta sorprendente que, pese a los grandes avances tecnológicos que estamos experimentando, la gestión de los datos médicos siga tan desactualizada. Por ejemplo, podemos pensar, en el ecosistema financiero, cómo con un clic podemos hacer transferencias bancarias entre países, incluso con divisas diferentes. Podemos ver claramente que la industria sanitaria se está quedando muy atrás respecto a otras industrias. Esta problemática no sólo se limita a la relación entre España e Irlanda: la falta de interoperabilidad es un desafío a nivel global. De hecho, incluso en nuestro país, entre Comunidades Autónomas, la gestión de nuestros datos médicos resulta en una odisea. Y esto no sólo se queda ahí: cada país cuenta con sistemas de salud privados que añaden más complejidad a esta trama, ya que tampoco existe interoperabilidad entre sistemas sanitarios públicos y privados. Imaginemos entonces la dispersión de nuestros datos médicos entre diferentes países, comunidades autónomas y entre sistemas públicos y privados.

Cómo podemos observar, la gestión de datos de pacientes de servicios de salud digitales representa un desafío en la era de la transformación tecnológica. Nuestros datos sanitarios son datos muy importantes. La seguridad, privacidad, controles de acceso, verificación de los datos e inmutabilidad son requisitos indiscutibles. Por esto, este proyecto se ha orientado al desarrollo de un sistema de gestión privada, en el que cada usuario tiene control de sus datos. En este contexto, la adopción de tecnologías emergentes como blockchain y la descentralización de los datos mediante bases de datos descentralizadas, como OrbitDB, puede ayudarnos a construir un sistema con las características previamente descritas.

Uno de los principales impulsores de este enfoque es la necesidad de garantizar la seguridad de los datos. La combinación de blockchain y bases de datos descentralizadas nos permite asegurar la integridad de la información médica mediante el uso de registros inmutables y verificables. Cada modificación en la base de datos se registra de forma que cualquier intento de alteración indebida pueda ser detectado de inmediato. Además, la criptografía protege la confidencialidad de los datos, asegurando que solo las personas autorizadas puedan acceder a información sensible.

Un aspecto diferenciador que usa este trabajo como enfoque, es el control que se le otorga a los propios pacientes sobre su información. A través de contratos inteligentes, los usuarios pueden gestionar de forma autónoma y con opciones casi infinitas los permisos de acceso a sus datos y revocar dichos permisos en cualquier momento. Este modelo de gestión descentralizada empodera al paciente y elimina la dependencia de intermediarios, promoviendo un entorno más seguro y privado. ¿Por qué una clínica u entidad privada de otro país, de la que hayamos sido clientes durante un tiempo, debe tener nuestros datos médicos, incluso cuando ya no somos clientes?

Por otro lado, la transparencia y la trazabilidad se convierten en pilares fundamentales de este sistema. Gracias a la naturaleza de blockchain, cada operación queda registrada de manera permanente, permitiendo auditar continuamente los datos sin comprometer la privacidad del paciente. Esto no solo facilita la supervisión por parte de las autoridades sanitarias, sino que también fortalece la confianza de los usuarios en la gestión de sus datos personales.

Por último, la integración de estas dos tecnologías no solo mejora la seguridad y el control de la información, sino que también optimiza la eficiencia operativa. La consolidación de datos médicos en un solo sistema facilita el trabajo de los profesionales al ofrecer una visión completa del historial del paciente, lo que puede traducirse en diagnósticos y tratamientos más acertados. Asimismo, al delegar en un sistema centralizado, pero de carácter descentralizado, al ser los usuarios los dueños de sus datos, se reduce la necesidad de desarrollar software médico independiente en cada hospital o entidad.

1.2 Objetivos

El objetivo de este proyecto ha sido poner a prueba un concepto para otorgar a los usuarios la habilidad para la gestión de sus propios datos médicos y poder compartirlos con quien deseen, de una forma segura, auditible, descentralizada, inmutable y robusta. Para poner a prueba este concepto, se ha desarrollado un sistema de gestión de datos médicos usando la tecnología blockchain, OrbitDB (base de datos descentralizada) y la criptografía. Para interactuar con este sistema, se ha desarrollado una aplicación móvil usando React Native. Con todo esto dicho, los objetivos que se han perseguido han sido:

- Despliegue y desarrollo de Smart Contract(s) en la Blockchain. Estos Smart Contract(s) son los encargados de asegurar la trazabilidad, auditoria y verificación de los datos.
- Despliegue y configuración de una base de datos descentralizada (OrbitDB) con nodos distribuidos en Inter-Planetary File System (IPFS). Encargada de almacenar los datos off-chain de manera descentralizada.
- Desarrollo de un servicio basado en Node.js para interactuar con OrbitDB y los Smart Contract(s). Servicio encargado de interactuar con OrbitDB de manera segura mediante el uso de criptografía, gestión de los permisos y la verificación de los datos interactuando con los contratos inteligentes.
- Desarrollo de una aplicación móvil con React Native para interactuar con el sistema. Esta aplicación será necesaria para establecer la conexión de las Wallets de los usuarios, interactuando con Metamask y enviar los datos de los usuarios de manera segura al servicio anteriormente mencionado mediante el uso de criptografía.

1.3 Metodología

En cuanto a la metodología utilizada durante el proyecto, se ha seguido una iterativa, en la que cada iteración se ha dirigido a desarrollar una o varias funcionalidades, dependiendo del tamaño de estas. Cada ciclo iterativo constó de las fases descritas en los siguientes subapartados.

1.3.1 Planificación

En esta fase, se realizó un estudio más en profundidad de las tecnologías. En concreto, se definió para qué se debería utilizar cada tecnología, discutiéndose temas como:

- ¿Dónde deberíamos almacenar los datos, directamente en la Blockchain o deberíamos usar una solución off-chain?
- ¿Deberíamos usar un oráculo para proporcionar a nuestros contratos inteligentes con estos datos off-chain?
- Usar la blockchain cuesta dinero, por lo que será necesario tener esto en cuenta a la hora de implementar los contratos inteligentes.
- Los datos en OrbitDB, si no son cifrados, son públicos para todo el mundo, por lo que será necesaria criptografía para asegurar la privacidad de los datos.
- Si se quiere compartir los datos, pero estos datos están cifrados, ¿cómo se pueden compartir con los demás?
- Si comparto los datos con varios usuarios, ¿cómo se puede gestionar cifrar de nuevo los datos, sin que no afecte a los demás usuarios?

1.3.2 Análisis de requisitos

Esta fase ha ido muy de la mano con la planificación, ya que se planificó mediante los requisitos que teníamos que cumplir: Descentralización, Seguridad, Privacidad, Integridad... Se han identificado todos los requisitos no funcionales y funcionales para asegurar el objetivo principal del proyecto.

1.3.3 Diseño

Con la planificación y los requisitos claros, durante esta etapa se ha realizado:

- Diseño de la arquitectura de comunicaciones de sistema.
- Diseño de los smart contracts para que no sean costosos y poco eficientes.
- Diseño de la interfaz de la aplicación móvil. Atención en la experiencia de usuario y la accesibilidad.
- También se ha tenido en cuenta el uso de las billeteras blockchain, ya que, si no se diseña bien, podemos tener una experiencia de usuario horrible teniendo que firmar cada cosa que hacemos.
- Diseño de la base de datos descentralizada, cómo estructurar los datos, qué datos son necesarios, etc.

1.3.4 Implementación

En esta fase, con la planificación, análisis y diseño, se ha realizado la implementación de los sistemas y los flujos diseñados para llevar a cabo las funcionalidades necesarias para alcanzar los objetivos. Se ha llevado un orden back-to-front, implementando y configurando la base de datos, implementando la lógica de negocio, implementando el API REST y realizando la implementación final de la aplicación móvil.

1.3.5 Pruebas

En esta fase, se han realizado las pruebas de lo implementado. Para conseguir esto, se ha llevado a cabo:

- Uso exhaustivo de la aplicación.
- Comprobados logs de la blockchain.
- Flujos revisados tanto en emulador como en dispositivo móvil.

1.4 Estructura de la memoria

- **Estado del arte:** Capítulo 2 de la memoria, en el cual se comentarán los sistemas actuales de gestión de datos médicos, los problemas de estos y de los sistemas que ya están implantados usando la tecnología blockchain en la actualidad. También se presentará un análisis de los usuarios, de si existe realmente la problemática expuesta de la segmentación de los datos.
- **Fundamentos tecnológicos:** Capítulo 3 de la memoria, en el cual se expondrá el hardware utilizado para el desarrollo, los lenguajes de programación usados, frameworks, librerías y entorno de desarrollo usados a lo largo del proyecto.
- **Análisis:** Capítulo 4 de la memoria, en el cual se hablará de los resultados del análisis realizado a partir de las historias de usuario para identificar las funcionalidades a realizar, los actores que participan en el sistema, las entidades identificadas para la realización de este módulo y los criterios de aceptación.
- **Diseño:** Capítulo 5 de la memoria, en el cual hablaremos de la arquitectura global de la plataforma, como está diseñada, de cómo participan las tecnologías en cada una de las capas y el diseño de las vistas en la Capa Interfaz de Usuario.
- **Implementación:** Capítulo 6 de la memoria, en el cual hablaremos sobre los detalles de la implementación de la plataforma pasando por las clases generales de esta. En este capítulo se muestran unas clases de demostración con aspectos básicos que nos sirven de referencia.
- **Pruebas:** Capítulo 7 de la memoria, en el cual se comentarán las pruebas realizadas. Se comentarán los tres tipos de pruebas realizadas: Pruebas Unitarias, Pruebas de Integración y Pruebas funcionales.
- **Incidencias, conclusiones y trabajo futuro:** Capítulo 8 de la memoria, en el cual se hablará de las conclusiones, lo que hemos sacado en claro de este proyecto y posibles líneas futuras del proyecto.
- **Anexo de planificación:** Anexo A de la memoria, en el cuál expondremos la planificación basada en ‘sprints’ ilustrado con un Diagrama de Gantt y los costes del proyecto.

Capítulo 2

Estado del arte

En este capítulo hablaremos de la situación actual de los sistemas de gestión de datos médicos y también de aquellos que implementan la tecnología blockchain para solventar las deficiencias de los sistemas actuales tradicionales.

2.1 Sistemas actuales de gestión de datos médicos

Los sistemas de administración de información médica han experimentado una evolución notable en años recientes, motivados por la digitalización, la interoperabilidad y la protección de los datos. No obstante, estos se encuentran con varios retos críticos. En España, este problema se intensifica debido a la fragmentación de los registros médicos entre comunidades autónomas, la susceptibilidad a ciberataques, las demandas regulatorias y los problemas de acceso en áreas rurales.

2.1.1 Interoperabilidad Deficiente

Uno de los principales desafíos de los sistemas sanitarios contemporáneos es la ausencia de interoperabilidad, así como la ausencia de normalización en la organización y distribución de los datos. En España, numerosos hospitales, clínicas y sistemas sanitarios utilizan plataformas que no son compatibles entre ellas. Cada comunidad autónoma administra su propio historial médico digital, lo que complica el acceso a la información cuando un paciente requiere ser atendido fuera de su comunidad. El sistema HCDSNS [1] (Historia Clínica Digital del Sistema Nacional de Salud) ofrece un grado de integración, sin embargo, no satisface todas las demandas de acceso en tiempo real ni permite un control directo del paciente sobre su información. Esta circunstancia hace que, por ejemplo, un enfermo de Madrid se mueva a un centro hospitalario en Cataluña tenga dificultades para que sus médicos accedan a su historial de manera inmediata.

2.1.2 Seguridad y Ciberataques

La ciberseguridad es muy importante para el sistema sanitario, cada vez los hospitales reciben más ciberataques, sobre todo ataques de ransomware. En España, un caso notable fue el ciberataque al Hospital Clinic de Barcelona [2], donde los servicios quedaron inoperativos durante varios días, no pudiendo someterse 300 pacientes a cirugía, además de 4000 análisis y más de 11000 visitas de consultas externas. Además de esto, también se produjo el robo de datos de tanto los pacientes, como de los trabajadores del centro, desde datos personales de salud como de tratamientos. Los problemas de seguridad no solo se limitan a España: en Estados Unidos, los Universal Health Services [3] fueron atacados con un ransomware que dejó a alrededor de 400 centros deshabilitados durante semanas, dejando a médicos sin acceso a historiales médicos, resultando en pérdidas de 67 millones de dólares.

2.1.3 Cumplimiento Regulatorio y Ética

Los datos médicos son datos muy sensibles que se deben manejar de forma muy cuidadosa. Existen varias normativas estrictas que establecen las restricciones sobre el almacenamiento y el uso de la información, como el Reglamento General de Protección de Datos (GDPR) y la Ley de Autonomía del Paciente. Estas normativas deben cumplirse a la hora de realizar un software que maneje datos médicos, sin embargo, tanto instituciones privadas como Google tanto públicas como la Comunidad de Madrid investigados debido a irregularidades en el cumplimiento de estas normativas [4], [5]. Google por ceder datos de terceros sin el consentimiento de los usuarios y la Comunidad de Madrid por violar la privacidad de los ciudadanos con su web del certificado COVID, mediante una brecha de seguridad que expuso los datos de los ciudadanos, no tenían la información cifrada en sus bases de datos.

2.1.4 Accesibilidad y Exclusión Digital

Hay que admitir que se ha avanzado mucho, pero aún existen barreras tecnológicas en algo que no debería de tener barreras, como es en la salud. Cada Comunidad Autónoma cuenta con sus recursos para implementar sus sistemas médicos, resultando en desigualdades entre la población española. Estos sistemas no son baratos de implementar, y no se realiza una inversión adecuada. Esto, en muchas ocasiones, resulta en exclusión para personas que residen en zonas rurales, los hospitales o centros de salud de zonas remotas de España no están bien integrados en el sistema digital del SNS, ya que varía de la infraestructura de cada comunidad autónoma.

2.1.5 Falta de Confianza en la Integridad de los Datos

Por último, la integridad de los datos médicos es muy importante. Los datos médicos pueden usarse para investigaciones científicas que, de no usar datos confiables, pueden concluir en resultados engañosos. También es necesario para mantener nuestro sistema de salud funcionando, no podemos permitir estafas a nuestro sistema nacional de salud mediante la falsificación de los datos.

Para dar un ejemplo, un caso significativo tuvo lugar en 2018 en el Hospital de La Paz de Madrid, donde se descubrió que algunos registros médicos habían sido alterados fraudulentamente para promover tratamientos en investigaciones clínicas, lo que provocó una significativa disminución de la confianza en el sistema de salud.

2.2 Aplicaciones de la tecnología blockchain en el sector salud

La tecnología blockchain ha surgido como una solución para mejorar la seguridad de varios procesos software debido a su naturaleza inmutable, esto no ha pasado desapercibido en el ámbito sanitario. A continuación, veremos diferentes formas en las que la tecnología blockchain se ha implementado en este sector. También veremos la implementación de esta tecnología en el sistema sanitario de un país, como en el caso de Estonia.

2.2.1 Historias Clínicas Electrónicas (EHR)

Cómo se ha indicado previamente, uno de los problemas más importantes es el de la dispersión de la información. Patientory [6] es una plataforma que conecta a médicos, proveedores de atención y pacientes en un solo lugar, usando la tecnología blockchain para construir un registro permanente y seguro, en el que el paciente cuenta con todos sus datos médicos. Con esto, permitimos el acceso de todos los datos médicos para el paciente y para los especialistas de diferentes sistemas permitiendo a los médicos, proveedores de atención actuar como uno al atenderlo, siempre con el permiso de este. Así, ayudando al ecosistema sanitario a mitigar las violaciones dañinas de datos para que no tengan que preocuparse por la confidencialidad del paciente.

2.2.2 Trazabilidad y seguridad en la cadena de suministro farmacéutica

La falsificación de los medicamentos [7] es un gran problema global que afecta a muchos países. Recientemente, este mercado ha ido aumentando desde el COVID, ya que se multiplicó la demanda de medicamentos de todo tipo. Con la blockchain, podemos implementar fácilmente sistemas que puedan rastrear cada paso del proceso de fabricación y distribución,

registrando cada operación de manera inmutable, así garantizando su origen y su autenticidad, reduciendo el riesgo de fraudes.

Empresas como IBM han colaborado con la industria farmacéutica para desarrollar soluciones basadas en blockchain que aseguran la integridad y autenticidad de los medicamentos en toda la cadena de suministro.

2.2.3 Caso de éxito: Implementación en sistema sanitario Estonio

Estonia [8] se ha establecido como un modelo a seguir en el uso de blockchain para salvaguardar y administrar sus registros médicos digitales. A partir de 2012, la nación ha estado integrando esta tecnología en su sistema de salud, y en 2016, la Agencia de Sistemas de Información Sanitaria (EHIF) finalizó la incorporación de blockchain en la administración de los datos sanitarios.

En el sistema estonio, se cuenta con un expediente electrónico clínico centralizado que vincula hospitales, clínicas privadas y farmacias. Esto posibilita que, con el permiso del paciente, los doctores tengan la posibilidad de consultar su historial médico tanto en contextos públicos como privados. Este sistema se sustenta en la tecnología KSI Blockchain, creada por Guardtime, que resguarda la integridad de los datos y posibilita auditorías en tiempo real.

Este sistema otorga privacidad y un control total para verificar quien puede o quien ha consultado sus datos, pudiendo comprobar irregularidades, ya que cada acceso queda registrado de forma inmutable. En este modelo, como hemos visto en otros modelos, los datos en sí no se almacenan en la blockchain, la blockchain actúa como auditor de los cambios y accesos que sufre esta información. Este sistema también se utiliza para gestionar reembolsos de seguros médicos privados, ayudando a validar reclamaciones y a reducir el fraude en la facturación.

2.2.4 Conclusiones estado del arte

Como hemos visto, en España existen propuestas como la Historia Clínica del Sistema Nacional de Salud (HCDSNS) o la Historia resumida europea (EU-Patient Summary) (EUPS), de la que los pacientes no suelen tener conocimiento y puede ser de difícil adopción para el ciudadano. Frente a ello, la solución presentada en este TFM, basada en blockchain y en bases de datos descentralizadas, ofrece una alternativa innovadora que garantiza la seguridad, transparencia, control del usuario y eficiencia, respondiendo a los desafíos actuales del sector sanitario.

Capítulo 3

Fundamentos tecnológicos

En este capítulo se detallarán las tecnologías necesarias para todo lo necesario para la realización de este proyecto. Se comenzará por el hardware utilizado, continuando con las bases tecnológicas utilizadas junto con las herramientas, frameworks y lenguajes de programación utilizados para su implementación.

3.1 Hardware

El Hardware necesario para el desarrollo ha sido un ordenador personal, cuyas especificaciones son las siguientes:

- Sistema Operativo Microsoft Windows 11 Pro
- Procesador 13th Gen Intel(R) Core(TM) i5-13420H
- Memoria RAM de 16GB
- Tarjeta Gráfica Nvidia GeForce RTX 4050 Laptop GPU

3.2 Blockchain

Posiblemente la palabra más mencionada en este documento. Es el núcleo de este TFM, ya que nos permite una capa de seguridad y descentralización rápida y automática. Una blockchain^[9], es un registro digital descentralizado de transacciones compartidas entre una red inmutable, como un libro de contabilidad compartido. Hay numerosas redes blockchain, Bitcoin es la más conocida, y Ethereum, también muy conocida, y en la que se ha desarrollado este TFM. Ethereum es una blockchain pública en la que podemos ejecutar contratos inteligentes. Los Contratos Inteligentes, o Smart Contracts, como su nombre indica, son contratos que se ejecutan automáticamente cuando se cumplen los términos de este, predeterminados en el

mismo. Estos contratos no son otra cosa que programas codificados que se almacenan en la blockchain. Usar Ethereum es costoso, por lo que se ha usado una red de prueba de Ethereum llamada Sepolia Ethereum. En esta testnet, se ha desplegado nuestros contratos inteligentes e interactuado con ellos de forma gratuita, mediante tokens obtenidos en los llamados ‘grifos’.

3.3 Inter-planetary File System

IPFS [10] es un conjunto modular de protocolos para organizar y transferir datos, diseñado desde cero con los principios de direccionamiento de contenidos y las redes entre iguales. Dado que IPFS es de código abierto, existen múltiples bases de datos que se apoyan en IPFS, como en nuestro caso OrbitDB. Aunque IPFS tiene múltiples casos de uso, el principal es la publicación de datos (archivos, directorios, sitios web, etc.) de forma descentralizada.

3.4 OrbitDB

OrbitDB [11] es una base de datos distribuida, peer-to-peer y sin servidor. OrbitDB utiliza IPFS como almacenamiento de datos y Libp2p Pubsub para sincronizar automáticamente las bases de datos con sus pares. Es una base de datos eventualmente consistente que utiliza Merkle-CRDTs para escrituras y fusiones de bases de datos libres de conflictos, haciendo de OrbitDB una excelente opción para aplicaciones p2p y descentralizadas, aplicaciones blockchain y aplicaciones web local-first. Los datos almacenados en OrbitDB, al usar IPFS, son de carácter público, por lo que es necesario el uso de la criptografía en nuestra aplicación.

3.5 Criptografía

La criptografía [12] es la práctica de desarrollar y utilizar algoritmos codificados para proteger y ocultar la información transmitida, de modo que solo pueda ser leída por quienes tengan permiso y capacidad de descifrarla. Dicho de otro modo, la criptografía oscurece las comunicaciones para que las partes no autorizadas no puedan acceder a ellas. En nuestro TFM, hemos usado la criptografía simétrica y asimétrica.

3.6 Algoritmos de criptografía simétrica

Estos algoritmos se basan en poder cifrar y descifrar datos usando la misma clave, siendo un proceso simétrico. En nuestro caso, hemos usado la criptografía simétrica para securizar nuestros datos mediante el cifrado y descifrado usando el algoritmo AES. Hemos usado este algoritmo debido a que es uno de los algoritmos más seguros y eficientes, ofreciéndonos

rapidez en el cifrado y descifrado, lo cual es importante para la experiencia de usuario. Para determinar la clave, derivamos la dirección del usuario con PBKDF2 (Password-Based Key Derivation Function 2) y una sal aleatoria para añadir entropía, a parte elegimos un keysize de 256 bits con 10000 iteraciones para incrementar la dificultad para ataques de fuerza bruta, resultando en una clave bastante robusta.

3.7 Algoritmos de criptografía simétrica

Estos algoritmos se basan en poder cifrar y descifrar datos usando distintas claves. La clave pública se usa para cifrar los datos, esta clave, como su nombre indica, es pública para todo el mundo. Por otra parte, la clave privada se usa para descifrar los datos, y sólo la conoce el usuario en cuestión, es muy importante mantenerla secreta. Para esto, se ha usado el algoritmo RSA para generar ambas claves, cifrando la clave privada para que esta sea secreta y solo el usuario con su contraseña pueda descifrarla. Usamos este algoritmo para compartir las claves simétricas que mantienen a los datos cifrados entre usuarios. Si un paciente quiere compartir su clave simétrica, debe cifrar la clave con la clave pública del doctor.

3.8 Lenguajes de programación

3.8.1 Solidity

Solidity [13] es un lenguaje de alto nivel orientado a objetos que nos permite implementar contratos inteligentes. Solidity es un lenguaje de llaves diseñado para la máquina virtual de Ethereum (EVM). Está influenciado por C++, Python y JavaScript. Este lenguaje está tipado estéticamente, soporta herencia, bibliotecas y tipos complejos definidos por los usuarios, entre otras características.

3.8.2 Javascript

JavaScript [14] es un lenguaje de programación ligero, interpretado o compilado just-in-time. Es más conocido como un lenguaje de scripting para páginas web, pero también es usado en muchos entornos fuera del navegador, como es nuestro caso, en Node.js. Es un lenguaje de programación basado en prototipos, multiparadigma, de un solo hilo, dinámico, con soporte para la programación orientada a objetos, imperativa y declarativa (por ejemplo, programación funcional). Cómo podemos ver, es un lenguaje muy versátil que se puede usar para un sinfín de aplicaciones.

3.8.3 TypeScript

TypeScript [15] es un lenguaje de programación de código abierto, desarrollado por Microsoft, que extiende JavaScript añadiendo tipado estático opcional. Está diseñado para el desarrollo de aplicaciones a gran escala, ya que con el tipado, podemos evitar muchos errores y hacer grandes repositorios de código mucho más comprensibles. TypeScript se transpila a JavaScript estándar, lo que permite su ejecución en cualquier entorno que soporte JavaScript.

3.8.4 CSS

Hojas de Estilo en Cascada (del inglés Cascading Style Sheets) o CSS [16] es un lenguaje de estilos utilizado para describir la presentación de documentos HTML o XML. CSS describe cómo debe ser renderizado el documento estructurado en pantalla.

3.8.5 Node.js

Node.js [17] es un entorno de ejecución de JavaScript multiplataforma y de código abierto. Es una herramienta muy popular para cualquier tipo de proyecto. Node.js ejecuta el motor JavaScript V8, el núcleo de Google Chrome, fuera del navegador. Esto permite que Node.js tenga un gran rendimiento.

Node.js puede usarse tanto del lado cliente como del lado servidor. Se pueden crear tanto servidores como aplicaciones web en él, siendo muy atractivo para los desarrolladores ya que pueden usar un solo lenguaje de programación. Una aplicación en Node.js se ejecuta en un único proceso, sin crear un nuevo hilo para cada solicitud. Además, proporciona un conjunto de primitivas de E/S asíncronas en su biblioteca estándar que evitan que el código de JavaScript se bloquee, y en general, sus bibliotecas están escritas utilizando paradigmas de no-bloqueo, haciendo que el comportamiento de bloqueo sea la excepción y no la norma. Esto permite que Node pueda manejar miles de conexiones concurrentes con un solo servidor sin introducir carga de gestionar la concurrencia de hilos, lo que podría ser una fuente importante de errores.

Esto que acabamos de comentar es algo importante a la hora de implementar aplicaciones. Además, Node tiene un gran ecosistema de librerías Web3, como puede ser Ethers, librerías criptográficas, etc.

3.8.6 Metamask

Metamask [18] es la billetera líder para interactuar con blockchain compatibles. Está presente tanto como extensión de navegador y aplicación móvil que permite a los usuarios manejar sus activos en el blockchain, interactuar con aplicaciones descentralizadas de forma segura

y privada. Con Metamask, los usuarios pueden gestionar sus claves públicas y privadas, necesarias para las operaciones criptográficas que se desarrollan en el blockchain.

3.8.7 Remix

Remix [19] es un IDE (Entorno de Desarrollo Integrado), que nos permite sin configuración ninguna y con una interfaz gráfica de usuarios desarrollar contratos inteligentes. La utilizan tanto expertos, como principiantes, simplificando el proceso de desarrollo, compilación y despliegue de contratos inteligentes en la red que elijamos.

3.8.8 React Native

React Native [20] es un framework de código abierto desarrollado por Meta, que permite a los desarrolladores que conocen React crear aplicaciones nativas multiplataforma, sin la necesidad de realizar funcionalidades múltiples veces por dispositivo. Para hacer más fácil la experiencia del desarrollo, se ha usado Expo, un framework de React Native. Expo proporciona un gran número de herramientas y características como enruteamiento basado en archivos, bibliotecas universales de alta calidad y poder probar nuestra aplicación tanto en emulador, como en nuestro propio dispositivo móvil.

3.8.9 Redux Toolkit

Redux Toolkit [21] es una herramienta que nos permite integrar Redux en nuestra aplicación React de manera más sencilla, gestionando el estado y reduciendo el boilerplate de original de Redux. En nuestro caso, se ha utilizado por sus funcionalidades de obtención de datos (fetching) y almacenamiento en caché mediante RTK Query.

3.8.10 Wagmi

Wagmi [22] es una librería que nos facilita el desarrollo de aplicaciones descentralizadas proporcionándonos una colección de hooks de React para interactuar con Ethereum. Con esta herramienta podemos interactuar con cuentas, carteras, contratos inteligentes de forma eficiente y con un buen rendimiento. Es muy beneficioso su uso, ofrece conectores oficiales para carteras como Metamask, Reown, etc. permitiéndonos realizar un sinfín de acciones de forma sencilla ofreciendo seguridad de tipos con TypeScript, gestión de caché y lecturas y escrituras a Ethereum.

3.8.11 Reown

Reown [23], previamente WalletConnect es una herramienta que nos permite construir dApps (aplicaciones descentralizadas) de forma que podamos lograr una mejor experiencia de

usuario. Nos permite interactuar con las diferentes wallets de forma sencilla para el usuario, eliminando barreras de entrada para las aplicaciones descentralizadas.

3.8.12 Postman

Postman [24] es una plataforma integral para el desarrollo y gestión de APIs que nos permite simplificar cada etapa del ciclo de vida de una API, facilitando la colaboración entre equipos para crear APIs de manera más eficiente. En nuestro caso, se ha usado para probar el correcto funcionamiento de los diferentes endpoints.

3.8.13 Android Studio

Android Studio [25] es un IDE para el desarrollo de aplicaciones para Android desarrollado por Google, basado en IntelliJ IDEA de JetBrains. Ofrece un conjunto completo de herramientas como: editor de texto inteligente, compilación y lo más importante para nosotros, poder usar emuladores Android.

3.8.14 Infura

Infura [26] es una plataforma que ofrece una suite de herramientas y APIs blockchain de alta disponibilidad para desarrolladores que buscan construir y escalar aplicaciones descentralizadas en diversas redes blockchain, incluyendo Ethereum e IPFS.

3.8.15 Figma

Figma [27] es una herramienta de diseño colaborativo para crear diseños de productos. Con ella hemos realizado el diseño de la interfaz de usuario, mediante wireframes y mockups. Además de planificar la navegación del usuario a través de la aplicación para asegurar una buena experiencia de usuario.

Capítulo 4

Análisis

En este capítulo comentaremos los distintos actores que participan en nuestro módulo y de tallaremos las entidades persistentes del sistema mediante un diagrama de la estructura de los datos almacenados, con una breve explicación de cada una. Para terminar, resumiremos los requisitos funcionales y no funcionales.

4.1 Actores del sistema

La aplicación cuenta con tres tipos de actores: pacientes, doctores y administradores.

4.1.1 Pacientes

Los pacientes, junto con los doctores, son los beneficiarios de esta aplicación. Cualquiera persona puede ser un paciente, lo único que necesita es una billetera blockchain, como puede ser Metamask para acceder a la aplicación. Los pacientes se identifican por la dirección de su billetera y, una vez accedido a través de Metamask, comprobamos si están registrados en la aplicación. Si ya están registrados, se inicia sesión automáticamente; en caso contrario, estos podrán registrarse indicando su información básica, y a continuación acceder a la pantalla inicial. En la pantalla inicial, el usuario puede acceder a su información personal, datos básicos de salud, sus pruebas analíticas, pruebas de imagen, vacunas, incapacidades temporales, medicaciones e informes clínicos. Inicialmente, después de registrarse, el usuario solo tendrá sus datos personales, pero puede añadir toda esta información por él mismo si lo desea. También, el usuario puede visualizar los doctores que tienen acceso a sus datos, que inicialmente, no habrá ninguno, pero podrá añadirlo manualmente. A continuación, ilustramos el rol del paciente en nuestro sistema a través de un diagrama de casos de uso:

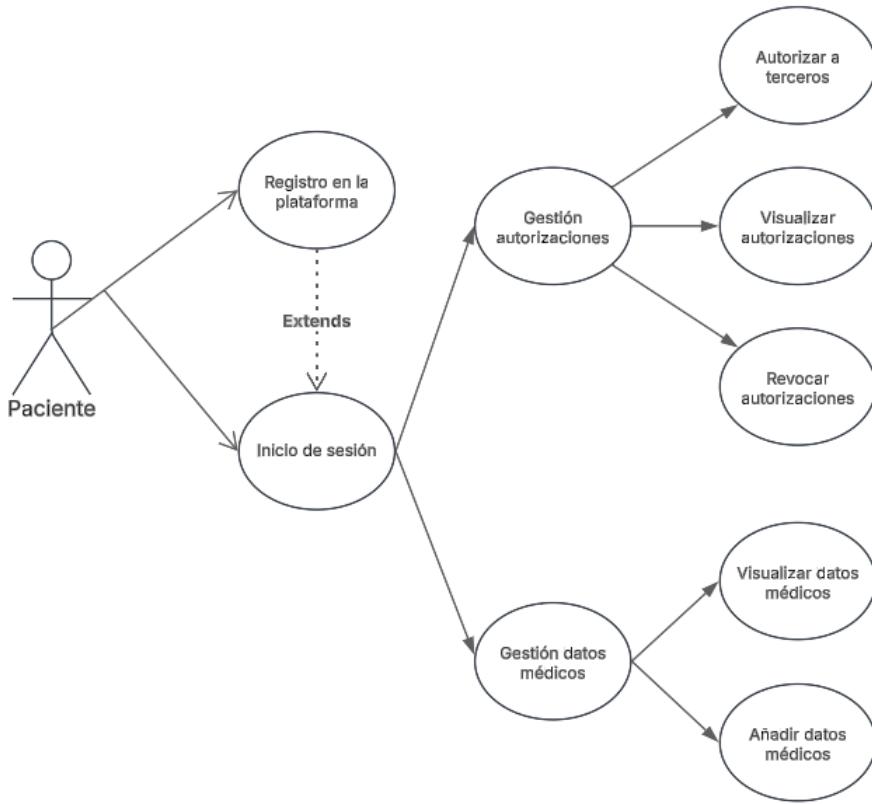


Figura 4.1: Diagrama casos de uso UML paciente.

4.1.2 Doctores

Los doctores, inicialmente, deben ser registrados por los administradores del sistema. Para ser registrados, deben acreditar que son doctores a los administradores para estos ser añadidos. Los doctores también se identifican por la dirección de su billetera, por lo que deberán iniciar sesión de la misma forma que lo hacen los pacientes. Un doctor también puede ser paciente, por lo que se le preguntará con qué perfil de usuario quiere acceder. Si elige acceder como paciente, será tratado como paciente y tendrá las mismas funcionalidades que un paciente normal y corriente, pero si decide iniciar sesión como doctor, contará con otras funcionalidades. Al iniciar sesión, un doctor podrá acceder a los usuarios a los que él tiene acceso, pudiendo entrar en el detalle de cada uno, visualizar y añadir información de estos. Igual que el paciente, puede modificar su información básica y gestionar los permisos sobre los pacientes. Podrá solicitar permisos a los pacientes manualmente introduciendo la dirección de estos. Para entender más claramente el papel del doctor, podemos ver el siguiente diagrama:

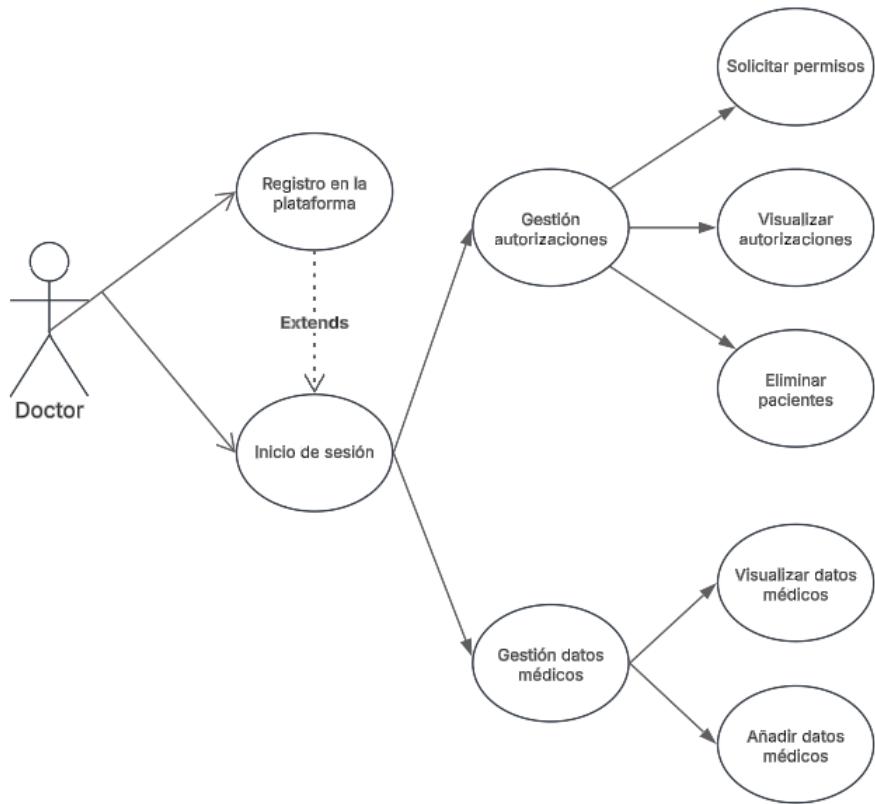


Figura 4.2: Diagrama casos de uso UML doctor.

4.1.3 Administradores

Los administradores son los encargados de desplegar los contratos inteligentes para la verificación de los datos, registrar a los doctores una vez hayan acreditado que efectivamente son doctores y hacer controles rutinarios de verificación de los datos y auditoría. Los administradores harán todas estas acciones programáticamente a través de un API REST dedicada a realizar todas estas acciones. Aquí se ilustran las acciones que puede realizar un administrador:



Figura 4.3: Diagrama casos de uso UML administrador.

4.2 Entidades persistentes del sistema

Como parte del análisis, ha sido necesario definir las siguientes entidades, las cuales podemos observar en la figura.

- **Usuarios:** Esta entidad identifica los doctores y los pacientes.
- **Medicamentos:** Esta entidad corresponde a los medicamentos de cada paciente.
- **Vacunas:** Esta entidad corresponde a las vacunas de cada paciente.
- **Documentos:** Esta entidad corresponde a los documentos de cada paciente. Estos pueden ser pruebas analíticas, pruebas de imagen, incapacidades temporales o informes clínicos.
- **Permisos:** Entidad que corresponde con las relaciones de permisos entre doctores y pacientes.

Datos médicos que puede almacenar el paciente:

- **Datos médicos básicos:** Incluyen información esencial sobre la salud del paciente, problemas salud actuales, alergias, grupo sanguíneo.
- **Pruebas analíticas:** Análisis de laboratorio, en forma de documento, de muestras biológica: sangre, orina y otros tejidos.

- **Pruebas de imagen:** Radiografías, ecografías, resonancias, en forma de documentos.
- **Vacunas:** Información básica sobre las vacunas que ha recibido el usuario.
- **Incapacidades temporales:** Documentos que reflejan incapacidades del paciente en las que no ha podido trabajar por enfermedad o lesión.
- **Medicaciones:** Información básica sobre la medicación que está tomando o ha tomado el paciente.
- **Informes clínicos:** Documentos que detallan evaluaciones médicas, diagnósticos, tratamientos, etc.

4.3 Requisitos no funcionales

Según lo aprendido en el estudio del arte, hemos definido los siguientes requisitos no funcionales para nuestro sistema:

- **Asegurar cohesión de los datos:** Los datos de los pacientes deben estar cohesionados entre ellos, independientemente del número de doctores y del sistema de salud al que pertenezcan.
- **Integridad de los datos médicos:** Debe asegurarse la integridad de los datos médicos una vez registrados mediante el uso del blockchain. La integridad de los datos debe ser verificable.
- **Seguridad:** El sistema de almacenamiento de datos debe ser seguro y privado. Por lo que se emplearán mecanismos criptográficos para almacenar la información de forma segura. Además, cada evento relativo al almacenamiento de datos u acceso a datos por parte de terceros será registrado.
- **Autenticación de los doctores:** Solo doctores verificados pueden ser registrados en la aplicación.
- **Escalabilidad y rendimiento:** El sistema debe ser escalable, pudiendo manejar un gran volumen de datos y poder acceder a estos de manera eficiente y con un buen rendimiento.
- **Eficiente con el uso de la blockchain:** El sistema debe tener en cuenta los posibles gastos del uso de la blockchain y debe ser diseñado para conseguir las funcionalidades deseadas sin costes de gas elevados.

- **Accesibilidad:** El sistema debe ser accesible para personas con dificultades en el uso de la tecnología atendiendo a la experiencia de usuario y a la realización de una interfaz sencilla de usar.

4.4 Requisitos funcionales

En este apartado hablaremos de los distintos requisitos funcionales en forma de historias de usuario.

4.4.1 Paciente: Registro en la plataforma

Como paciente, quiero poder registrarme en la plataforma, indicando información básica como: mi nombre, apellidos, fecha de nacimiento, número de teléfono y una contraseña que deberé guardar o recordar.

Descripción funcional

El paciente podrá registrarse en la aplicación haciendo uso de su billetera de MetaMask. Tras esto, la aplicación mostrará un formulario de registro de usuario donde introducir toda la información previamente descrita.

4.4.2 Paciente: Inicio de sesión

Como paciente, quiero poder iniciar sesión en la aplicación para poder acceder a mis datos y poder gestionar los permisos.

Descripción funcional

El paciente accederá a la aplicación usando su billetera de Metamask. Una vez identificado el paciente con su dirección, la aplicación comprobará en la base de datos de OrbitDB y en la blockchain conjuntamente que efectivamente el paciente existe. Una vez hecha esta comprobación, el paciente será redirigido al menú principal.

4.4.3 Paciente: Visualizar sus propios datos médicos

Como paciente, quiero poder visualizar mis propios datos médicos que haya podido introducir yo mismo o los doctores autorizados de forma clara y accesible, pudiendo acceder al detalle de cada uno de ellos.

Descripción funcional

Una vez iniciada la sesión, el paciente se encuentra en un menú en el que puede seleccionar que datos médicos acceder. El usuario puede elegir qué datos visualizar, podrá elegir entre: datos médicos básicos, pruebas analíticas, pruebas de sangre, vacunas, incapacidades temporales, medicaciones e informes clínicos.

4.4.4 Paciente: Visualización de los doctores autorizados

Como paciente, quiero poder visualizar quién tiene acceso a mis datos de forma clara y accesible.

Descripción funcional

Desde la pantalla inicial, en la barra de navegación inferior, el usuario podrá navegar hacia la pantalla de gestión de permisos, donde podrá visualizar los doctores que tienen acceso a sus datos.

4.4.5 Paciente: Autorización de doctores

Como paciente, quiero poder autorizar quién tiene acceso a mis datos de forma clara y accesible.

Descripción funcional

Desde la pantalla inicial, en la barra de navegación inferior, el usuario podrá navegar hacia la pantalla de gestión de permisos, donde podrá añadir manualmente la dirección de la billetera del doctor para otorgarle permisos.

4.4.6 Paciente: Revocar autorización

Como paciente, quiero poder autorizar quién tiene acceso a mis datos de forma clara y accesible.

Descripción funcional

Desde la pantalla inicial, en la barra de navegación inferior, el usuario podrá navegar hacia la pantalla de gestión de permisos, donde ver todos los doctores que tienen acceso a sus datos, al clicar en el doctor en cuestión, se mostrará un mensaje de confirmación y se revocará el permiso.

4.4.7 Doctor: Iniciar sesión

Como doctor, quiero poder iniciar sesión en la aplicación para poder acceder a los datos de mis pacientes y solicitar permisos.

Descripción funcional

El doctor accederá a la aplicación usando su billetera de Metamask. Una vez identificado el doctor con su dirección, se le redirigirá al menú principal.

4.4.8 Doctor: Acceder a los datos médicos de los pacientes

Como doctor, quiero poder visualizar los datos médicos de los pacientes que me han concedido permiso, de forma clara y accesible.

Descripción funcional

Una vez iniciada sesión, el doctor se encontrará en un menú en el que puede visualizar la lista de los usuarios que le han concedido permisos. Clicando en cada uno de ellos, podrá ver en detalle cada uno de sus diferentes datos médicos.

4.4.9 Doctor: Añadir datos médicos a los pacientes

Como doctor, quiero poder añadir datos médicos a los pacientes que me han concedido permiso, de forma clara y accesible.

Descripción funcional

Una vez iniciada la sesión, el doctor se encontrará en un menú en el que puede visualizar la lista de los usuarios que le han concedido permisos. Clicando en cada uno de ellos, podrá acceder al detalle del paciente, aquí, navegando a la sección de los datos, podrá añadir manualmente los datos.

4.4.10 Doctor: Visualizar los pacientes a los que tiene acceso

Como doctor, quiero poder visualizar los pacientes a los que tengo acceso, de forma clara y accesible.

Descripción funcional

Desde la pantalla inicial, en la barra de navegación inferior, el doctor podrá navegar hacia la pantalla de gestión de permisos, donde podrá visualizar los pacientes a los que tiene acceso y acceder a su detalle.

4.4.11 Doctor: Solicitar permisos a los pacientes

Como doctor, quiero poder solicitar permisos a los pacientes de forma sencilla.

Descripción funcional

Desde la pantalla inicial, en la barra de navegación inferior, el doctor podrá navegar hacia la pantalla de gestión de permisos, donde podrá añadir manualmente la dirección de la billetera del paciente para solicitarle permisos de acceso.

4.4.12 Administrador: Registrar doctores

Como administrador, quiero poder registrar doctores una vez estos pasen un proceso de verificación.

Descripción funcional

Para realizar esta tarea, se le habilitará al administrador un endpoint para que pueda realizar la petición de registro de usuario al sistema.

4.4.13 Administrador: Verificación datos

Como administrador, quiero poder contrastar los datos de OrbitDB con los registros de la blockchain.

Descripción funcional

Para realizar esta tarea, se habilitará al administrador un endpoint para que pueda realizar las consultas necesarias al sistema para verificar la información. Cada vez que se registran datos en OrbitDB, también se almacena este registro en la blockchain.

4.4.14 Administrador: Visualización de logs

Como administrador, quiero poder visualizar logs de todas las acciones realizadas en OrbitDB que se han registrado en la blockchain, y así poder realizar auditorías periódicas.

Descripción funcional

Para realizar esta tarea, se habilitará al administrador un endpoint para poder consultar todos los logs emitidos en la blockchain.

4.5 Casos de Uso

4.5.1 Autenticación

El usuario, tanto el paciente como el doctor, utilizará Metamask para acceder a la aplicación móvil. Así, con su dirección, podremos verificar si ya es un usuario registrado, por lo que iniciará sesión o necesita registrarse.

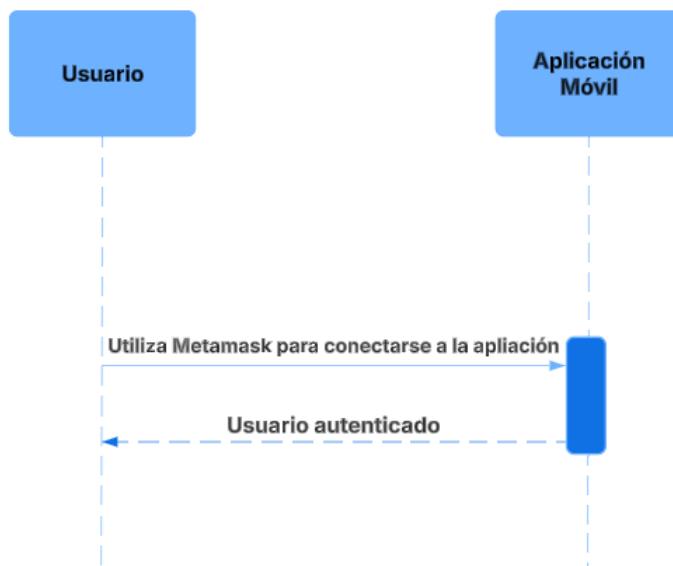


Figura 4.4: Diagrama secuencia UML autenticación.

4.5.2 Registro de paciente

Una vez identificado el paciente con su dirección, la aplicación comprobará en la base de datos de OrbitDB y en la blockchain conjuntamente que efectivamente el usuario no existe y se puede registrar. Para registrarse, en la aplicación se le habilitará un formulario donde el usuario podrá añadir todos los datos básicos mencionados anteriormente. Cuando el usuario clique en el botón de registro, la aplicación móvil generará una clave aleatoria, la cual cifrará simétricamente estos datos antes de mandarlos al servicio, esta clave aleatoria se almacenará, junto con la contraseña que escribió este, en el Secure Store del dispositivo móvil. Al recibir la petición de registro, el servicio descifrará la contraseña del usuario con la clave aleatoria. Una vez descifrada, se generarán un par de claves pública y privada usando la contraseña del usuario para cifrar la clave privada y se almacenarán junto con los datos del usuario. Una vez generamos estas claves, se cifra la clave aleatoria simétrica con la clave pública del paciente, así aseguramos que solo el con su propia clave privada pueda descifrarla, y almacenaremos la

clave simétrica cifrada junto con los datos del usuario. Este par de claves son necesarias para poder compartir la clave aleatoria que cifra y descifra simétricamente los datos del usuario. Para añadir seguridad, se registrará también el registro del usuario en el blockchain.

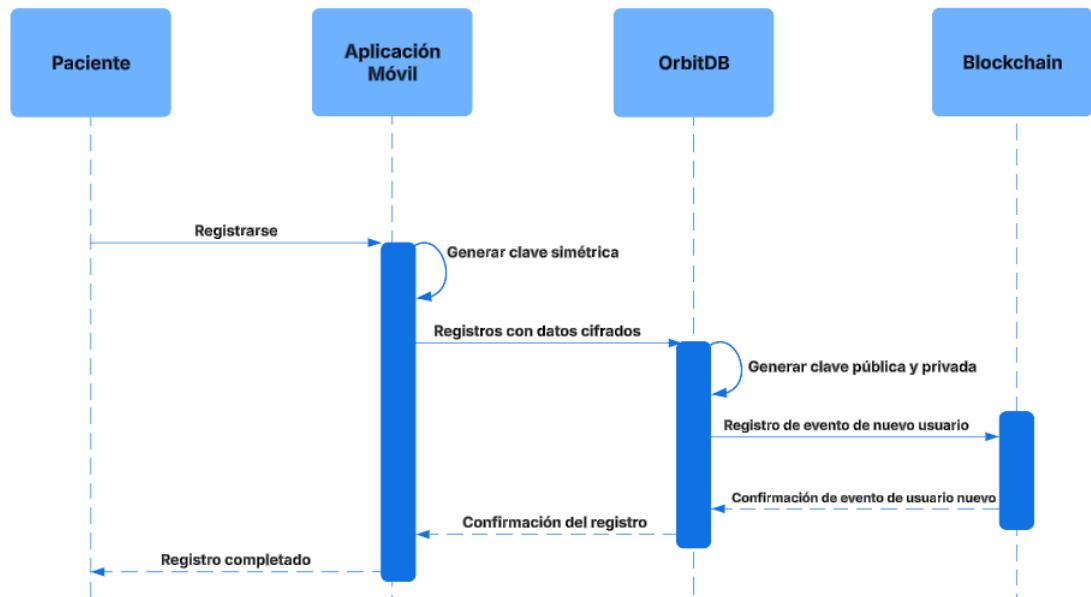


Figura 4.5: Diagrama secuencia UML registro usuarios.

4.5.3 Registro de doctor

El administrador, mediante el endpoint habilitado, podrá ejecutar el registro del doctor. Este método registrará en el blockchain que cierta dirección pertenece a un doctor verificado. Cuando el doctor quiera acceder a la aplicación, este accederá con Metamask y la aplicación comprobará que es un doctor verificado y podrá registrarse como tal. El proceso es el mismo que el paciente, con la diferencia de que los datos del doctor no serán cifrados. El servicio, al recibir la petición de registro, generará un par de claves pública y privada usando la contraseña del doctor para cifrar la clave privada y se almacenarán junto con los datos del doctor. Una vez generadas estas claves. Este par de claves son necesarias para poder otorgarle al doctor permiso de nuestros datos, ya que si el paciente cifra su propia clave simétrica con la clave pública del doctor. También se registrará el registro del doctor en el blockchain.

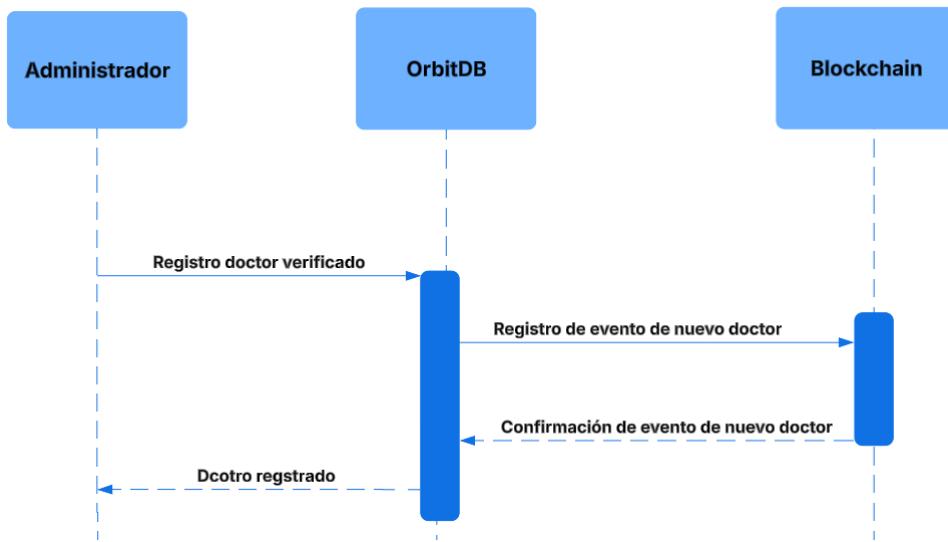


Figura 4.6: Paso 1: Diagrama secuencia UML registro doctor.

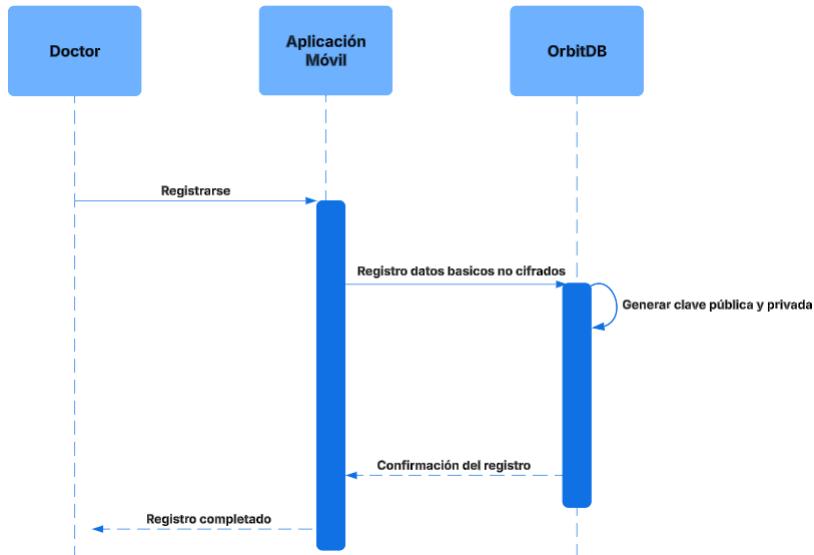


Figura 4.7: Paso 2: Diagrama secuencia UML registro usuarios.

4.5.4 Paciente: Visualización de sus propios datos

El paciente inicia sesión con Metamask. Una vez hecho esto, la aplicación con el address del paciente solicita todos sus datos. La aplicación los recibe cifrados, pero puede descifrarlos ya que tiene la contraseña simétrica almacenada en el Secure Store.

4.5.5 Paciente: Añadir datos

El paciente inicia sesión con Metamask. El paciente introduce los datos, la aplicación los cifra con la clave simétrica y los manda a OrbitDB. Los almacena en la base de datos correspondiente, dependiendo de a qué categoría pertenecen estos y guarda en el blockchain el hash de los datos junto con la fecha y quién los ha añadido.

4.5.6 Paciente: Visualizar permisos

El paciente inicia sesión con Metamask. Una vez hecho esto, la aplicación con el address del paciente solicita todos los doctores que tienen acceso a sus datos. La información de los doctores es básica y de ámbito público, por lo que no debe ser descifrada.

4.5.7 Paciente: Dar permisos

El paciente inicia sesión con Metamask. El usuario introduce manualmente la dirección del doctor. Junto con la dirección del paciente, se manda la dirección del doctor y la contraseña de este que está en el secure store. Con esto, el servicio descifra la clave privada del paciente. Con la clave privada descifrada, se descifra la clave simétrica que estaba almacenada. Se cifra la clave simétrica con la clave pública del doctor y se guarda un registro en la base de datos de los permisos en OrbitDB. Se almacena en el blockchain que el paciente ha otorgado permisos.

4.5.8 Doctor: Visualización de los datos de un paciente.

El doctor inicia sesión con Metamask. Una vez hecho esto, la aplicación con la dirección del doctor solicita los pacientes a los que el doctor tiene acceso ya descifrados. El doctor puede clicar en el paciente que desee para acceder a un menú, similar al que usa cada paciente, pudiendo acceder a cada categoría de los datos y al detalle de estos.

4.5.9 Doctor: Solicitar permisos

El doctor inicia sesión con Metamask a través de la aplicación. Una vez iniciada la sesión, el doctor navega a la pantalla de gestión de permisos. Aquí, podrá insertar manualmente la dirección que identifica a cada usuario. Una vez hecho esto, el usuario recibe una solicitud que podrá aceptar o rechazar.

4.5.10 Doctor: Visualizar pacientes

El doctor inicia sesión con Metamask a través de la aplicación. Una vez iniciada la sesión, el doctor puede ver sus pacientes directamente en su menú o navegar a la pantalla de gestión de permisos.

4.5.11 Doctor: Añadir datos paciente

El doctor inicia sesión con Metamask a través de la aplicación. Una vez iniciada la sesión, el doctor puede acceder a sus pacientes desde su menú. Desde este menú, este puede acceder a cada una de las pantallas de datos del usuario y añadir los datos. Además, se guarda en la blockchain el hash de los datos junto con la fecha y la dirección del doctor.

Capítulo 5

Diseño

En este capítulo comentaremos cómo es la arquitectura global de la plataforma. También se detallará el diseño seguido para las distintas capas de la aplicación hasta el diseño de las vistas de la interfaz de usuario.

5.1 Arquitectura global

La arquitectura global del sistema propuesto cuenta con 3 grandes pilares: una aplicación móvil con la que los usuarios interactúan, un servicio que interactúa con OrbitDB y los Contratos Inteligentes desplegados en la blockchain. A continuación, hablaremos del diseño de cada una de estas partes.

5.1.1 Diseño del Backend

El Backend de nuestro sistema es un servicio implementado con Node.js y Express.js encargado de gestionar el almacenamiento y lectura de los datos de forma segura con OrbitDB, interactuar con los Contratos Inteligentes para registrar cada acción que se realiza y gestionar la creación y gestión de las claves públicas y privadas de los usuarios, haciendo uso el algoritmo criptográfico RSA. En este servicio también se realiza la configuración de las diferentes bases de datos de OrbitDB para poder interactuar con ellas y contar con los tipos de bases de datos necesarios dependiendo del caso de uso. Por último, también se encarga de exponer todas estas operaciones a nuestra aplicación móvil proporcionando una API REST para que este la pueda consumir. Esto lo hace mediante métodos HTTP como GET, POST, DELETE y PATCH.

5.1.2 OrbitDB

OrbitDB es nuestro sistema de almacenamiento de datos basado en IPFS. OrbitDB proporciona diferentes tipos de bases de datos, las cuales se han tenido en cuenta para el diseño. Ya

que contamos con varios tipos de entidades, hemos decidido configurar varias bases de datos atendiendo a las características de estas:

UsersDB

Esta base de datos se encarga de almacenar los datos personales de los pacientes y los doctores.

Debido a que para los doctores y los pacientes almacenamos atributos ligeramente diferentes, pero ambos se identifican por la dirección de su cuenta de Ethereum, hemos decidido usar una base de datos de tipo clave-valor, en la que la clave corresponde a cada dirección, y el valor, a los datos que queramos añadir de cada una de las entradas, pudiendo ser un doctor o un paciente.

- **Campos obligatorios para los pacientes:**

Todos estos campos, menos la clave pública, se almacenan cifrados.

- **Nombre y apellidos**
- **Fecha de nacimiento**
- **Número de contacto**
- **Clave de cifrado simétrico**
- **Clave pública y privada**

A mayores, cada paciente podrá incluir de forma voluntaria:

- **Email**
- **Hospital al que está asociado**
- **Residencia**
- **DNI**

- **Campos obligatorios para los doctores:**

Todos estos campos, se almacenan descifrados, menos la clave privada, ya que no son datos sensibles y son de interés común.

- **Nombre y apellidos**
- **Número de contacto**
- **Hospital al que está asociado**
- **Tipo de doctor (especialidad)**

- **Clave pública y privada**

A mayores, cada doctor podrá incluir de forma voluntaria:

- **Email**
- **Residencia**
- **DNI**

Gracias a este tipo de base de datos, podemos añadir o modificar la estructura de datos de los pacientes y de los doctores de una manera muy flexible, ya que el valor del par clave-valor puede ser cualquiera. Además, es una base de datos de rápido acceso, con la clave de cada usuario podemos recuperar los datos de un usuario en específico de manera eficiente, sin necesidad de realizar consultas complejas. También, nos ayuda a que no tengamos usuarios duplicados en la base de datos, ya que no pueden haber dos valores con la misma clave asociada.

PermissionsDB

Esta base de datos se encarga de almacenar los permisos que están establecidos. En este caso, como aquí si tenemos una estructura clara de cómo son los permisos, y además necesitamos hacer consultas un poco más complejas, se ha optado por una base de datos de tipo ‘documents’. Esta es una base de datos de documentos, donde podemos almacenar documentos JSON de forma indexada por un id. Aquí, generaremos un id mediante la dirección del paciente y del doctor y contamos con los siguientes campos:

- **Atributos de los permisos:**

Todos estos no se almacenan cifrados, solo se almacena la clave simétrica del paciente cifrada con la clave pública del doctor para que esta la pueda usar.

- **Id generado del permiso**
- **Dirección del paciente**
- **Dirección del doctor**
- **Fecha en la que se concedió el permiso**
- **Clave simétrica del paciente cifrada con la clave pública del doctor**

Gracias a este tipo de base de datos, podemos almacenar una estructura que es más rígida, donde los atributos son siempre iguales. Y lo más importante, podemos hacer consultas más completas y flexibles sobre los datos, teniendo igualmente un gran rendimiento.

MedicationsDB

Esta base de datos se encarga de almacenar los medicamentos de cada paciente. De la misma forma, tenemos una estructura clara de cómo son los medicamentos, y además necesitamos hacer consultas un poco más complejas, por lo que se ha obtenido igualmente base de datos de tipo ‘documents’. Aquí contamos con los siguientes campos:

- **Atributos de los medicamentos:**

Todos estos datos, menos dirección del paciente y el doctor, se almacenan cifrados con la clave simétrica del paciente.

- **Id generado del medicamento**
- **Dirección del paciente**
- **Dirección del doctor que prescribió el medicamento**
- **Nombre del medicamento**
- **Dosis**
- **Frecuencia**
- **Fecha de inicio de la toma**
- **Duración**
- **Notas**
- **Fecha de creación del registro**

VaccinesDB

Esta base de datos se encarga de almacenar las vacunas de cada paciente. De la misma forma, tenemos una estructura clara de cómo son las vacunas, y además necesitamos hacer consultas un poco más complejas, también se ha obtenido por una base de datos de tipo ‘documents’. Aquí contamos con los siguientes campos:

- **Atributos de las vacunas:**

Todos estos datos, menos dirección del paciente y el doctor, se almacenan cifrados con la clave simétrica del paciente.

- **Id generado de la vacuna**
- **Dirección del paciente**
- **Dirección del doctor que administró la vacuna**
- **Nombre de la vacuna**

- **Edad del paciente cuando se administró la vacuna**
- **Fecha de aplicación**
- **Centro de salud**
- **Fecha de creación del registro en base de datos**

A continuación se muestra el Diagrama E-R simplificado:

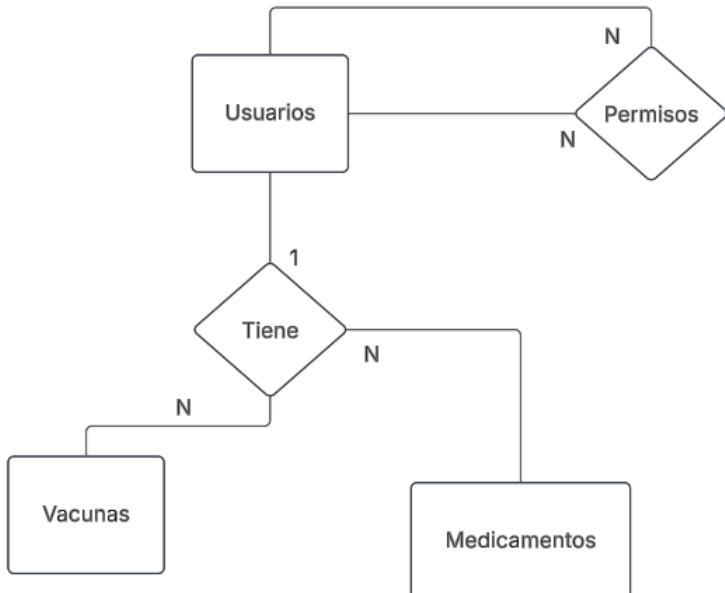


Figura 5.1: Diagrama Entidad-Relación simplificado.

5.1.3 Contratos Inteligentes

En cuanto a los contratos inteligentes, estos son desplegados en Ethereum y son los encargados de proporcionar esa capa de trazabilidad, verificación de la integridad de los datos, auditoría y transparencia. El diseño de los contratos inteligentes ha sido un proceso que ha requerido mucho esfuerzo y análisis, ya que se puede lograr la misma funcionalidad de muchas formas. Almacenar datos y ejecutar operaciones en Ethereum cuesta dinero (a no ser que uses una red de pruebas). Por lo tanto, el diseño de los contratos inteligentes es crucial para hacer una aplicación viable económicamente. Durante este proyecto, se han probado numerosos enfoques de diseño de los contratos inteligentes, como los detallados a continuación.

Almacenar datos médicos directamente en Ethereum

La primera aproximación que se tuvo en cuenta en el diseño de los contratos fue almacenar los datos médicos directamente en los contratos inteligentes. Esta opción era la más intuitiva y fácil de implementar, simplemente sería crear las estructuras de los datos en el smart contract y hacer todas las operaciones ahí, sin necesidad de tener OrbitDB. Al realizar el análisis de este enfoque, nos dimos cuenta de que era económicamente inviable almacenar datos médicos, que pueden aumentar mucho en volumen. Según el whitepaper de Ethereum [28], el coste de almacenamiento de datos en Ethereum es de 20.000 gas (medida de coste en Ethereum) por cada 256 bits, cogiendo el ejemplo de una vacuna, podemos realizar estos cálculos:

Siendo un string en Solidity 32 bytes, la ecuación para el tamaño total es:

$$\text{Tamaño total} = 8 \text{ campos} \times 32 \text{ bytes} = 256 \text{ bytes} \quad (5.1)$$

Según la documentación de Ethereum, almacenar 32 bytes (256 bits) en Ethereum, cuesta 20,000 unidades de gas, entonces:

$$\text{Gas total} = 8 \text{ campos} \times 20,000 \text{ gas} = 160,000 \text{ gas} \quad (5.2)$$

El precio del gas se mide en gwei, donde 1 gwei = 0.000000001 ETH, dependiendo de la congestión de la red, asumiremos un precio de 50 gwei:

$$\text{Coste en ETH} = 160,000 \text{ gas} \times 50 \text{ gwei} \times 10^{-9} = 0.008 \text{ ETH} \quad (5.3)$$

Entonces, tomando en cuenta que el precio de Ethereum, día de hoy, está a 2,557.52 euros:

$$\text{Coste en euros} = 0.008 \text{ ETH} \times 2,557.52 \text{ EUR} = 20.46016 \text{ EUR} \quad (5.4)$$

Nos costaría 20 euros más o menos almacenar una sola vacuna en la blockchain para 1 usuario. Imaginémonos si quisiéramos almacenar muchos otros datos para un numero considerable de datos, por ello, esta opción no es viable.

Almacenar mappings para relacionar pacientes y doctores

Esta segunda aproximación ya contemplaba el uso de un almacenamiento de datos off-chain en OrbitDB, pero quería seguir delegando cierto almacenamiento a la blockchain. En este caso, se puede plantear el almacenar un par de estructuras mapping de Solidity, mapeando direcciones para identificar a los usuarios, ya que cuestan menos bytes que un string, 20 en concreto, y con booleanos, que ocupan 1 byte, para tener el sistema de permisos. Esta idea, tampoco era muy costosa, pero no estábamos cumpliendo completamente con los requisitos, ya que no estábamos contemplando con la trazabilidad y auditoría de los datos en sí.

Solución definitiva: Emitir eventos indexados

Después de la segunda aproximación, optamos por almacenar un hash de los datos en OrbitDB. Esta solución está bastante extendida en la comunidad de desarrolladores, es la solución más estándar por así decirlo. Esta solución sigue almacenando datos permanentes en el contrato inteligente, y guarda un hash por cada registro, pudiendo también crecer mucho en coste aún. Además, sería una funcionalidad un poco limitada, ya que no tendríamos información de quién actualizó o añadió el hash, ni cuando, sin que supusiera un aumento significativo del coste. Investigando más, se descubrieron los eventos.

Los eventos son mecanismos que permiten a los contratos inteligentes registrar información en logs de la blockchain. Trabajar con estos logs, es considerablemente más barato y eficiente que trabajar con datos almacenados permanentemente en la blockchain, siendo estos igual de inmutables y permanentes. La única desventaja es que no son accesibles desde dentro del contrato, pero son fácilmente accesibles desde fuera de la blockchain, pudiendo igualmente realizar tareas de auditoría, validación de datos y comprobar la trazabilidad. Estos eventos pueden contener parámetros, los cuales pueden ser indexados, convirtiendo a los eventos en eventos indexados. Con los eventos indexados, el rendimiento de las búsquedas y filtrados es muy bueno y hacen que la búsqueda de información específica por parámetro sea muy rápida.

En nuestro caso, hemos definido eventos para:

- **Almacenamiento de hashes:**

Este evento se emite cada vez que hace una operación de creación, actualización o eliminación de datos en OrbitDB. En este evento, emitimos:

- **Hash del CID de OrbitDB**
- **Fecha**
- **Propietario de los datos**
- **Quién hizo el cambio sobre los datos en OrbitDB**

Hay un límite de cuantos atributos pueden ser indexados, siendo 3 este límite. Los tres primeros campos mencionados anteriormente fueron indexados, ya que son los más interesantes a la hora de realizar las búsquedas.

- **Cuando se añade un paciente/doctor:**

Este evento, en la práctica son dos eventos, que se emiten cuando se crea un usuario en OrbitDB. En este evento, emitimos ambos campos indexados:

- **Dirección del usuario**

- Fecha

- **Eventos de gestión de permisos:**

Para gestión de permisos, contamos con 3 eventos:

- Registrar que un paciente otorga acceso a un doctor
- Registrar que un paciente elimina acceso a un doctor
- Registrar una solicitud de un doctor a un paciente

Estos eventos siempre emiten los mismos 3 campos indexados:

- Dirección del paciente
- Dirección del doctor
- Fecha

Emitir el evento más caro, el de almacenamiento de hashes, ya que cuenta con 4 datos, 3 indexados y 1 no indexado sería de:

- **Coste estático de la operación LOG:** 375 unidades de gas.
- **Coste por tópico indexado:** 375 unidades de gas x 3 tópicos.
- **Coste datos no indexados en memoria:** 20 bytes de address x 3 de gas.
- **Coste datos almacenamiento en log:** 20 bytes de address x 8 de gas.

Entonces el coste total de gas sería de $375 + 375 \times 3 + 20 \times 3 + 20 \times 8 = 1,720$ unidades de gas.

Como se ha indicado previamente, el precio del gas se mide en gwei, donde 1 gwei = 0.000000001 ETH, dependiendo de la congestión de la red, asumiremos un precio de 50 gwei:

$$\text{Costo en ETH} = 1,720 \text{ gas} \times 50 \text{ gwei} \times 10^{-9} = 0.000086 \text{ ETH} \quad (5.5)$$

Entonces, tomando en cuenta que el precio de Ethereum, día de hoy, está a 2,557.52 euros:

$$\text{Costo en euros} = 0.000086 \text{ ETH} \times 2,557.52 \text{ EUR} = 0.22 \text{ EUR} \quad (5.6)$$

Comparando con el almacenamiento bajamos el coste de las operaciones en 100 veces.

Así, nuestro contrato inteligente solo se limita a la emisión de eventos, operación muy eficiente y barata. Estos métodos de emisión de eventos son ejecutados mediante el servicio cada vez que se hace una operación de este tipo en la base de datos de OrbitDB. Y luego, el mismo servicio puede recuperar los eventos de forma eficiente y rápida usando los campos indexados como filtros, y así hacer las verificaciones de integridad de datos, auditoría y seguridad.

5.2 Diseño del Frontend

El frontend es la parte con la que interactúa el usuario. En esta parte nos encargamos de la interfaz de usuario, la cual contiene todo lo relativo a la vista de la aplicación, los componentes que dan forma a esta vista, experiencia de usuario... En esta parte también está presente la capa acceso a los servicios, la cual se encarga de la conexión con los servicios del API REST implementada en el backend.

5.2.1 Capa de acceso a servicios

Esta capa es crucial para poder interactuar con el API REST, y no sólo esto, también es donde se realiza el cifrado y descifrado de los datos. Por temas de seguridad, nos hemos decantado por enviar los datos al Backend ya cifrados y también descifrarlos en esta capa. Aunque HTTPS sea seguro y ya cifra los datos, se ha decidido hacer esto para añadir un nivel adicional de seguridad. Esto no añade mucha complejidad en el desarrollo y tampoco añade mucha carga de trabajo al lado cliente, ya que se usan algoritmos de cifrado muy eficientes. Además, la generación automática de hooks, encapsulan todo el proceso de obtención de datos, proporcionando estados a los componentes.

5.2.2 Capa Interfaz de Usuario

En esta parte del diseño se ha trabajado con Figma para realizar el diseño de la interfaz, primero maquetando unos wireframes de media fidelidad, los cuales nos darían una perspectiva de los flujos de la aplicación de la usabilidad. A partir de los wireframes, se realizaron unos mockups a partir de los wireframes, se definieron los estilos, los colores de la aplicación, etc. Los wireframes y los mockups forman parte de la base del diseño de la aplicación y sirvieron a nivel conceptual a la hora del desarrollo de la interfaz del usuario. Como la idea de este proyecto no es limitarse solamente a este TFM, algunos diseños se han realizado más en detalle que lo que se mostrará en la aplicación y, de la misma forma, algunas pantallas de los diseños han sido mejoradas en tiempo de desarrollo.

Wireframes

Un wireframe es un diagrama visual que esboza el esqueleto de un proyecto. A veces se conoce como plano de la pantalla y muestra cómo se relacionan los elementos entre sí y cómo están estructurados. El wireframing es un proceso de alto nivel, y es usado por los diseñadores UX (User experience o experiencia de usuario), que es de mucha ayuda a la hora de planificar la estructura de una interfaz y del flujo de la navegación.

A continuación mostraremos unos ejemplos de los wireframes realizados:

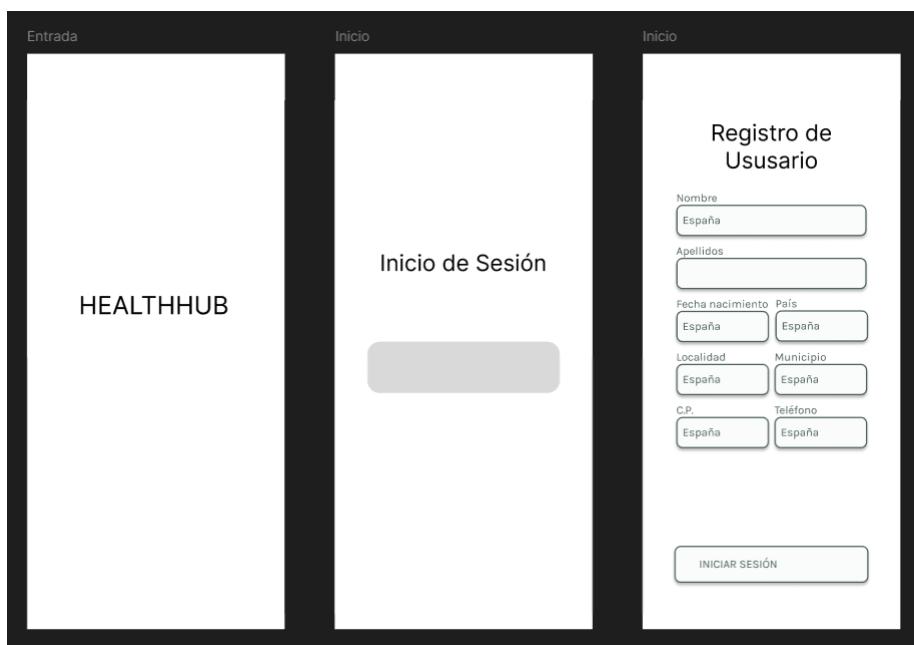


Figura 5.2: Wireframes correspondientes a splash screen, inicio sesión y registro.

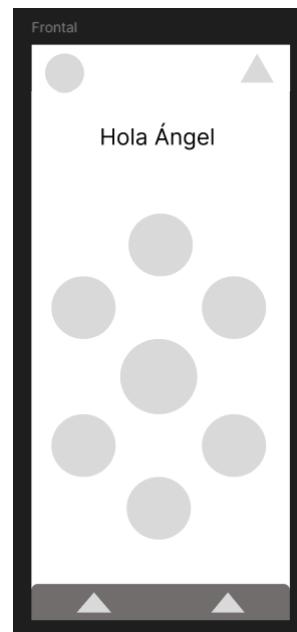


Figura 5.3: Wireframe menu usuario.

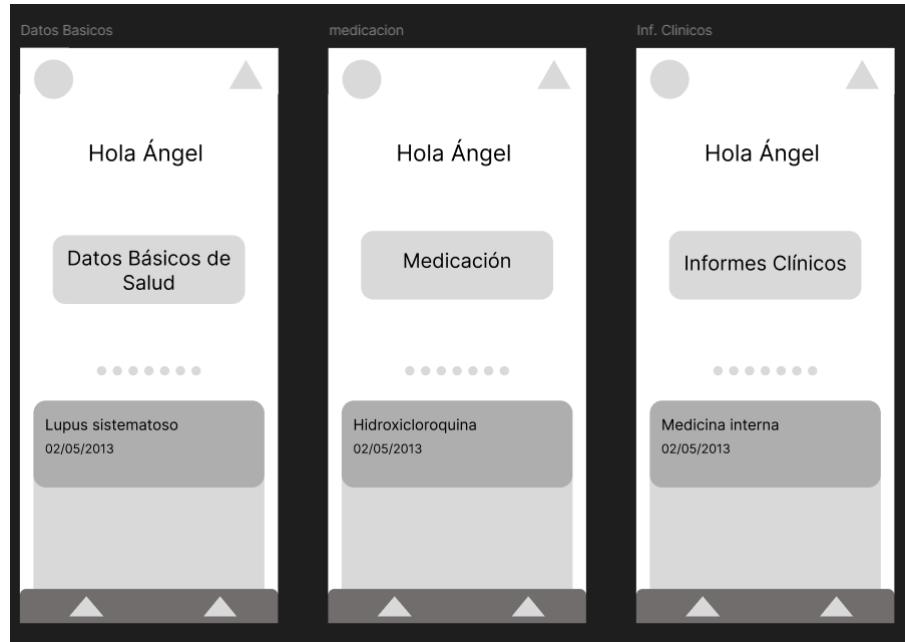


Figura 5.4: Wireframes vistas datos médicos por categorías.

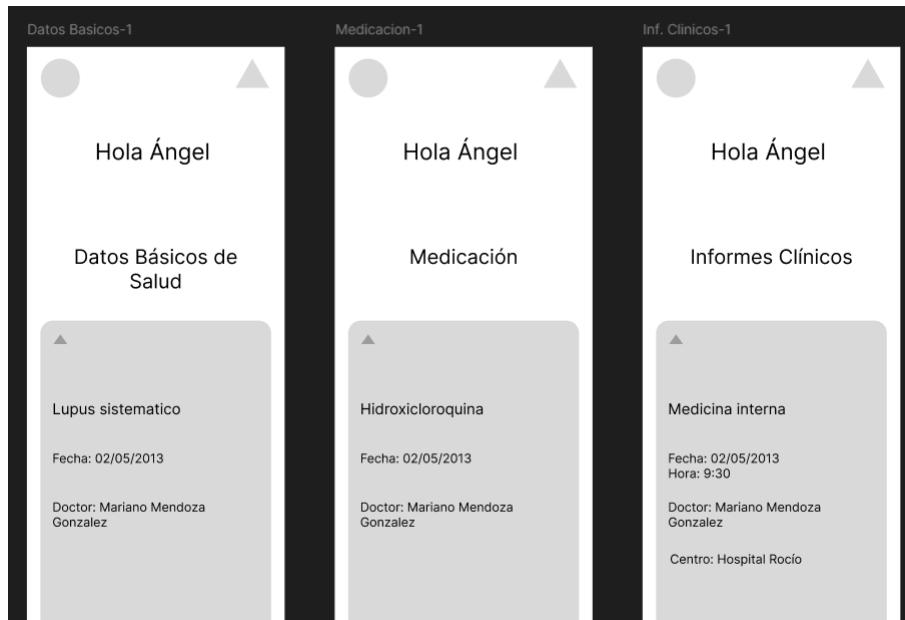


Figura 5.5: Wireframes visualización detalles datos.

Mockups

Los mockups son representaciones visuales similares a los prototipos que simulan el aspecto final de un diseño. Estos van más allá de los wireframes y los diseños, ya que muestra los

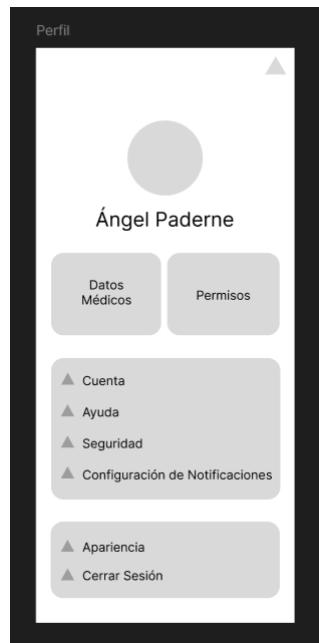


Figura 5.6: Wireframe pantalla perfil usuario.

elementos de diseño reales, como los colores, tipografía y las imágenes. Los mockups proporcionan una vista previa realista de cómo un producto, sitio web o aplicación se mostrará a los usuarios. Sus elementos no son interactivos pero, son lo más parecido a lo que se encontrará el usuario.

A continuación, mostraremos los mockups homónimos a los wireframes mostrados previamente:



Figura 5.7: Mockups correspondientes a splash screen, inicio sesión y registro.

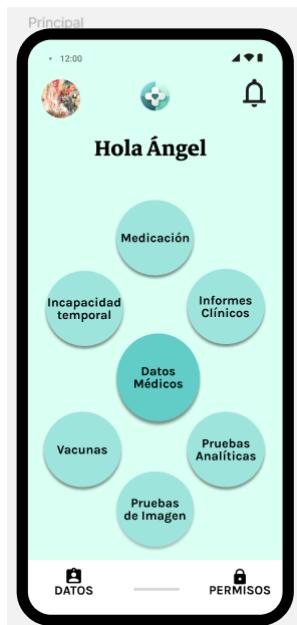


Figura 5.8: Mockup menú usuario.

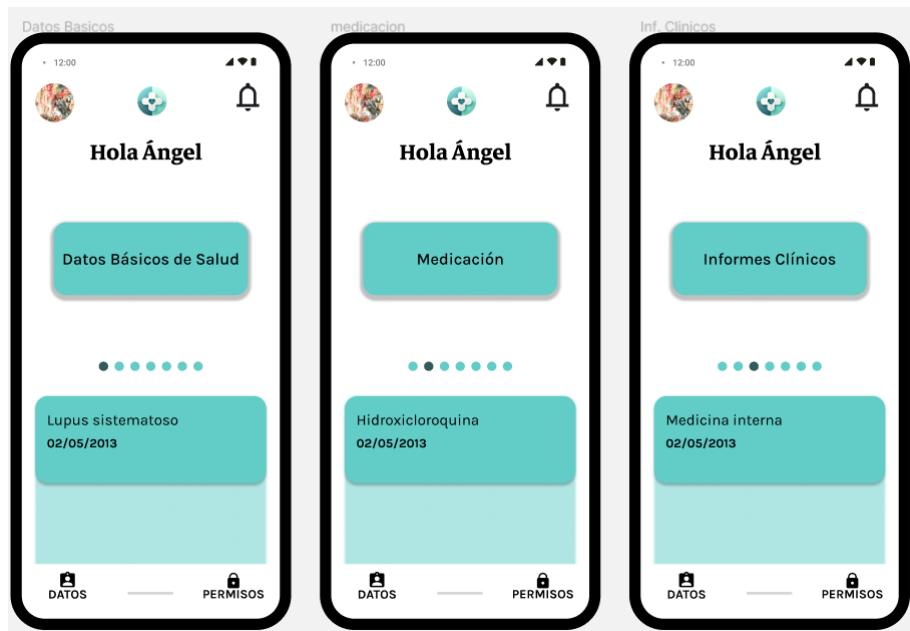


Figura 5.9: Mockups vistas datos médicos por categorías.

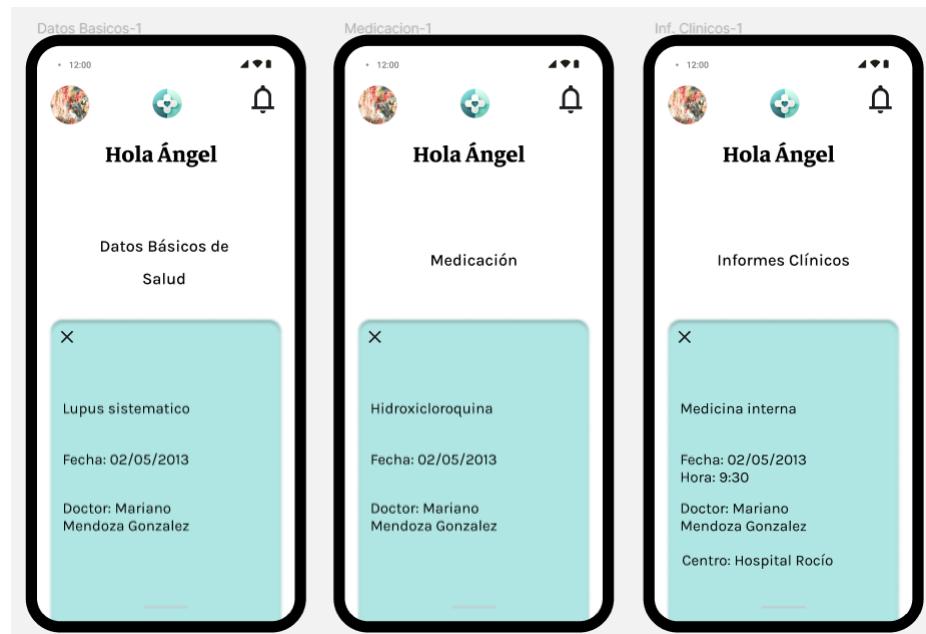


Figura 5.10: Mockups visualización detalles datos.



Figura 5.11: Mockup pantalla perfil usuario.

Capítulo 6

Implementación

En este capítulo hablaremos de los detalles de la implementación del sistema mediante código. Para esto, comentaremos la estructura de ambos proyectos, frontend y backend.

6.1 Estructura global

Para el desarrollo de este TFM hemos creado dos repositorios en github, uno para el backend y otro para el frontend, ambos repositorios bajo el mismo proyecto. Uno llamado ‘healthhub’, que es la parte frontal de la aplicación y otro llamado ‘orbitdb-service’ que corresponde al backend del sistema.

6.2 Implementación del Backend

6.2.1 Estructura del backend

Como podemos ver en la siguiente imagen 6.1, podemos ver cómo hemos organizado nuestro servicio, así cuando hablemos en los siguientes puntos, podamos usarla como referencia,

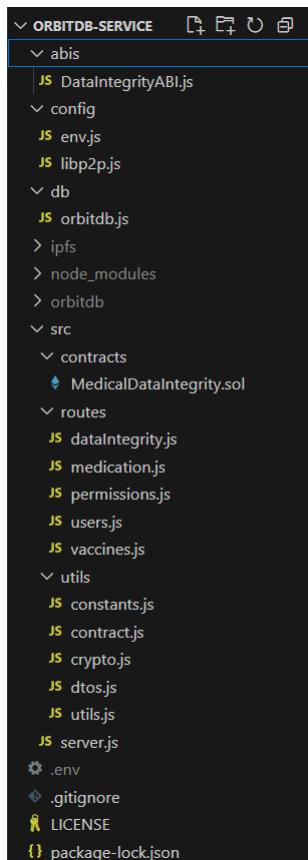


Figura 6.1: Estructura servicio backend.

```
1 import express from 'express';
2 import cors from 'cors';
3 import dotenv from '../config/env.js';
4 import { initOrbitDB } from '../db/orbitdb.js';
5 import permissionsRoutes from './routes/permissions.js';
6 import usersRoutes from './routes/users.js';
7 import dataIntegrityRoutes from './routes/dataIntegrity.js';
8 import vaccinesRoutes from './routes/vaccines.js';
9 import medicationsRoutes from './routes/medication.js';
10
11 const app = express();
12 const port = 3000;
13
14 app.use(express.json());
15 app.use(cors());
16 dotenv.config();
17
18 const { permissionsDB, usersDB, medicationsDB, vaccinesDB } = await
19     initOrbitDB();
20
21 console.log('UserDB', usersDB.address.toString());
22 console.log('PermissionsDB', permissionsDB.address.toString());
23 console.log('VaccinesDB', vaccinesDB.address.toString());
24 console.log('MedicationsDB', medicationsDB.address.toString());
25
26 app.use('/permissions', permissionsRoutes(permissionsDB, usersDB));
27 app.use('/users', usersRoutes(usersDB));
28 app.use('/data-integrity', dataIntegrityRoutes());
29 app.use('/vaccines', vaccinesRoutes(vaccinesDB));
30 app.use('/medications', medicationsRoutes(medicationsDB));
31
32 app.listen(port, () => {
33     console.log(`Server running on port: ${port}`);
34 })
```

Figura 6.2: Server.js.

El fichero principal del sistema es el ‘server.js’, que es donde inicializamos las bases de datos de OrbitDB, usamos las diferentes rutas y ejecutamos el servicio. Como podemos observar, en la figura 6.2 definimos el servidor de Node.js. Primero inicializamos las diferentes bases de datos de OrbitDB, las cuales son utilizadas en las diferentes rutas del servicio, las cuales son añadidas al servicio. Hablemos de la inicialización de OrbitDB.

6.2.2 OrbitDB

Para implementar las bases de datos de OrbitDB, hemos tenido que realizar cierta configuración. La verdad es que OrbitDB es un proyecto muy nuevo que está en constante evolución, lo cual hace que la documentación quede desactualizada muy rápidamente. Tampoco cuenta con un ecosistema de desarrolladores muy grande, por lo que ha resultado un desafío a la hora

de la implementación.

Inicialización OrbitDB

Como hemos visto en la figura anterior(6.2), para inicializar las diferentes bases de datos de OrbitDB, llamamos al método ‘initOrbitDB’. Esta función se sitúa en el archivo ‘orbitdb.js’ de la carpeta ‘db’. Como podemos observar en el código 6.3, para crear una instancia de OrbitDB, necesitamos Helia para almacenamiento en bloque y Libp2p para sincronizar las bases de datos:

- **Blockstore:** Un blockstore es un sistema de almacenamiento eficiente para bloques de datos en IPFS. Es necesario ya que Helia por defecto usa almacenamiento en memoria, lo cual significa que cada vez que cerrasemos la aplicación, se destruirían los bloques.
- **Libp2p:** OrbitDB sincroniza bases de datos entre pares utilizando Libp2p. Helia necesita una instancia para que OrbitDB pueda sincronizar los datos entre bases de datos a través de redes p2p.
- **Helia:** Helia es una implementación moderna de IPFS desarrollada en Javascript. Es la implementación recomendada por la documentación de OrbitDB. Es más ligera, rápida y flexible que la implementación tradicional de IPFS. Helia nos permite interactuar con la red IPFS para almacenar, recuperar y compartir archivos.

Con todo esto, ya podemos tener nuestra instancia de OrbitDB. Una vez tenemos nuestra instancia, podemos crear las diferentes bases de datos. Para esto, definimos un control de acceso libre, para que cualquier usuario o doctor pueda escribir en la base de datos. Para crear cada base de datos, necesitamos la siguiente información:

- **DB ADDRESS:** Aquí le mandamos la dirección de la base de datos para que orbit pueda abrirla, aunque en caso de no estar creada, la crearemos desde 0 indicando el nombre que queramos que tenga esta. Una vez tenemos la dirección de la base de datos, tendríamos que añadirla como variable de entorno para poder abrir esta.
- **Tipo:** Aquí indicamos el tipo de la base de datos.
- **Control de acceso:** Aquí indicamos el control de acceso para la base de datos, se pueden definir numerosas reglas para el control de acceso, pero para este entorno de desarrollo no hemos hecho normas muy estrictas.

Una vez abiertas las bases de datos, podemos devolverlas para ser utilizadas.

```
1 import { LevelBlockstore } from 'blockstore-level'
2 import { createLibp2p } from "libp2p";
3 import { Libp2pOptions } from "../config/libp2p.js";
4 import { createHelia } from "helia";
5 import { createOrbitDB } from "@orbitdb/core";
6
7 export async function initOrbitDB() {
8     const blockstore = new LevelBlockstore('./ipfs/blocks');
9     const libp2p = await createLibp2p(Libp2pOptions);
10    const ipfs = await createHelia({ libp2p, blockstore });
11    const orbitDB = await createOrbitDB({ ipfs });
12
13    console.log('Initializing databases... ');
14
15    const accessController = {
16        type: 'orbitdb',
17        write: ['*'],
18    };
19
20    const USERS_DB_ADDRESS = process.env.USERS_DB_ADDRESS || 'users';
21    const PERMISSIONS_DB_ADDRESS =
22        process.env.PERMISSIONS_DB_ADDRESS || 'permissions';
23    const VACCINES_DB_ADDRESS = process.env.VACCINES_DB_ADDRESS || 'vaccines';
24    const MEDICATIONS_DB_ADDRESS =
25        process.env.MEDICATIONS_DB_ADDRESS || 'medications';
26
27    const usersDB = await orbitDB.open(USERS_DB_ADDRESS, {
28        type: 'keyvalue',
29        accessController,
30    });
31
32    const permissionsDB = await
33        orbitDB.open(PERMISSIONS_DB_ADDRESS, {
34            type: 'documents',
35            accessController,
36        });
37
38    const medicationsDB = await
39        orbitDB.open(MEDICATIONS_DB_ADDRESS, {
40            type: 'documents',
41            accessController,
42        });
43
44    const vaccinesDB = await orbitDB.open(VACCINES_DB_ADDRESS, {
45        type: 'documents',
46        accessController,
47    });
48
49    return {
50        usersDB,
51        medicationsDB,
52        permissionsDB,
53        vaccinesDB,
54    };
55}
```

6.2.3 Rutas

Las rutas definen los puntos de acceso de la API REST. Como pudimos observar previamente, contamos con 5 rutas, donde en cada una se implementa su responsabilidad:

- **Integridad de los datos:** En esta ruta definimos las funcionalidades que necesita el administrador para interactuar directamente con el contrato inteligente. No necesita de ninguna base de datos de OrbitDB, ya que solamente ejecuta operaciones sobre el contrato inteligente, del que hablaremos más adelante.
- **Medicaciones:** En esta ruta definimos las operaciones sobre los medicamentos de los pacientes. Esta ruta necesita de la base de datos de medicaciones de OrbitDB e interactúa con el contrato inteligente para registrar los cambios en los datos.
- **Permisos:** En esta ruta definimos las funcionalidades relacionadas con los permisos. Necesita la base de datos de permisos para almacenar los permisos vigentes y las claves cifradas con la clave pública del doctor para que este pueda acceder a ellas. También se implementan las interacciones con el contrato inteligente para emitir los eventos pertinentes.
- **Usuarios:** En esta ruta definimos las funcionalidades relacionadas con la gestión de usuarios. Además, está la funcionalidad que la otorga a los administradores habilitar a los doctores verificados. Aquí se interactúa con la base de datos de los usuarios para almacenar los datos personales de estos. También se interactúa con el contrato inteligente para emitir los eventos de registro de usuario.
- **Vacunas:** En esta ruta definimos todas las funcionalidades relacionadas con el almacenamiento y acceso de los datos de las vacunas de los usuarios. Aquí se implementan los métodos sobre la base de datos de vacunas.

En estas rutas implementamos las funciones básicas CRUD (Create, Read, Update, Delete) para cada base de datos y los exponemos mediante los métodos post, get, patch, delete de HTTP. A continuación, expondremos ejemplos de algunos métodos implementados. Todas las rutas son inicializadas con el router y el contrato.

```
export default function usersRoutes(userDB) {
  const router = Router();
  const contract = getMedicalDataIntegrityContract();
```

Figura 6.4: Inicialización router y contrato.

Creación de paciente

La creación del paciente se realiza mediante un método POST. En este método, a parte de la escritura de los datos en la base de datos de usuario, también se generan las claves, se emiten los eventos pertinentes, como podemos ver en la siguiente figura (6.5)

```
router.post('/', async (req, res) => {
  try {
    const { key, name, dateOfBirth, phoneNumber, encryptedUserPassword, cipherKey } = req.body;

    const decryptedUserPassword = decryptSym(encryptedUserPassword, cipherKey);
    const { pubkey, privkey } = generateKeys(decryptedUserPassword);
    const encryptedCipherKey = encryptAsym(cipherKey, pubkey);
    const value = { name, dateOfBirth, phoneNumber, pubkey, privkey, encryptedCipherKey };
    const CID = await userDB.put(key, value);
    await contract.updateDataHash(CID, key, key);
    await contract.addPatient(key);
    res.status(201).send({ message: 'Item added' , CID});
    console.log(`Nuevo registro añadido: { key: "${key}" , value: "${value}" , , }`);
  } catch (error) {
    res.status(500).send({ error: error.message });
  }
});
```

Figura 6.5: Método POST creación paciente.

Recuperación de las vacunas del paciente

La figura 6.6 muestra cómo se realiza la recuperación de los datos a través del método GET.

```
router.get('/:patientId', async(req, res) => {
  try {
    const patientId = req.params.patientId;
    const patientVaccines = await vaccinesDB.query((doc) => {
      return doc.patientId === patientId;
    });

    res.status(200).send(patientVaccines);
  } catch (error) {
    res.status(500).send({ message: error.message });
  }
});
```

Figura 6.6: Método GET recuperación de vacunas del paciente.

Actualización de los datos del paciente

Por la naturaleza de nuestro sistema, no soportamos muchos métodos de actualización, pero si permitimos introducir ciertos datos al paciente mediante el método PATCH.

```
router.patch('/:key', async (req, res) => {
  try {
    const key = req.params.key;
    const updates = req.body;
    const existingValue = await userDB.get(key);

    if (!existingValue) {
      return res.status(404).send({ message: 'Item not found' });
    }

    const updatedValue = { ...existingValue, ...updates };

    const CID = await userDB.put(key, updatedValue);
    const tx = await contract.updateDataHash(CID, key, key);

    res.status(200).send({ message: 'Item updated', CID, tx });
    console.log(`Registro actualizado: { key: "${key}", hash: "${CID}" }`);
  } catch (error) {
    res.status(500).send({ error: error.message });
  }
});
```

Figura 6.7: Método PATCH actualización datos del paciente.

Revocar permisos de los datos del paciente

Cuando un paciente revoca los permisos a un doctor, eliminamos de la base de datos de permisos la entrada que contiene la clave simétrica cifrada con la clave pública del doctor, así el doctor no tiene forma de descifrar los datos. Para esto, usamos el método DELETE.

```
router.delete('/', async(req, res) => {
  try {
    const { patientId, doctorId } = req.body;
    if (!patientId || !doctorId) {
      return res.status(400).send({
        message: 'Patient and Doctor are necessary'
      });
    }

    const permissionId = generatePermissionsId(patientId, doctorId);
    await permissionsDB.del(permissionId);
    await contract.requestAccess(patientId, doctorId);

    res.status(200).send({ message: 'Relation removed' });
  } catch (error) {
    res.status(500).send({ message: error.message });
  }
});
```

Figura 6.8: Método DELETE para revocar permisos.

Como pueden haber visto en el método de crear paciente, usamos numerosas funciones de cifrado y descifrado. Estas funciones están en la carpeta de utils, en el fichero crypto.js, aquí es donde se hace uso de todos los algoritmos criptográficos.

Generar claves pública y privada

Se ha implementado la función ‘generateKeys’, la cual genera el par de claves con la contraseña del usuario:

```
export function generateKeys(userPassword){
  const { publicKey, privateKey } = generateKeyPairSync('rsa', {
    modulusLength: 4096,
    publicKeyEncoding: {
      type: 'spki',
      format: 'pem'
    },
    privateKeyEncoding: {
      type: 'pkcs8',
      format: 'pem',
      cipher: 'aes-256-cbc',
      passphrase: userPassword
    }
  });
  return { pubkey: publicKey, privkey: privateKey };
}
```

Figura 6.9: Función para generar claves asimétricas.

Cifrado/Descifrado simétricos

Estos métodos se usan para cifrar y descifrar datos con la clave simétrica aleatoria que se le genera al usuario. También se usa el método de descifrado para mandar los datos de los pacientes descifrados al doctor.

```
export function encryptSym(data, key) {
  const encryptedData = CryptoES.AES.encrypt(data, key).toString();
  return encryptedData;
}

export function decryptSym(encryptedData, key) {
  const decryptSymBytes = CryptoES.AES.decrypt(encryptedData, key);
  const decryptedData = decryptSymBytes.toString(CryptoES.enc.Utf8);
  return decryptedData;
}
```

Figura 6.10: Funciones para cifrar/descifrar simétricamente.

Cifrado/Descifrado asimétricos

Estos métodos se usan para cifrar y descifrar la clave simétrica aleatoria que se le genera al usuario. Usando la clave privada del paciente, este puede descifrar la clave simétrica, y con la clave pública del doctor este pueda cifrarla para compartirla.

```
export function encryptAsym(data, publicKey){
    const encryptedBuffer = publicEncrypt(publicKey, Buffer.from(data));
    return encryptedBuffer.toString('base64');
}

export function decryptAsym(data, privateKey, userPassword){
    const buffer = Buffer.from(data, 'base64');
    const privateKeyObject = createPrivateKey({
        key: privateKey,
        format: "pem",
        passphrase: userPassword,
    });

    const decryptedBuffer = privateDecrypt({
        key: privateKeyObject,
    },
    buffer
);

    return decryptedBuffer.toString('utf-8');
}
```

Figura 6.11: Funciones para cifrar/descifrar asimétricamente.

También la implementación de la ruta de la integridad de los datos, donde se muestra la implementación de la consulta de los eventos (en este caso de modificación de datos):

```
1 import { Router } from 'express';
2 import getMedicalDataIntegrityContract from '../utils/contract.js';
3
4 export default function dataIntegrityRoutes() {
5     const router = Router();
6     const contract = getMedicalDataIntegrityContract();
7
8     router.get('/', async (req, res) => {
9         try {
10             const filter = contract.filters.DataHashUpdated();
11             const logs = await contract.queryFilter(filter);
12
13             logs.forEach((log) => {
14                 console.log(log.args.dataHash);
15                 console.log(log.args.timestamp.toString());
16                 console.log(log.args.owner);
17                 console.log(log.args.updatedBy);
18             });
19
20             res.status(200).send(logs);
21         } catch (error) {
22             res.status(500).send({ error: error.message });
23         }
24     });
25
26     router.post('/', async (req, res) => {
27         try {
28             const { data } = req.body;
29             const tx = await contract.updateDataHash(data);
30
31             res.status(201).send({ message: 'Event added', tx });
32
33         } catch (error) {
34             res.status(500).send({ error: error.message });
35         }
36     });
37
38     return router;
39 }
```

Figura 6.12: dataIntegrity.js.

En el código se han mostrado también varios métodos relativos con los contratos inteligentes, hablaremos a continuación de la implementación de estos.

6.2.4 Contratos inteligentes

Como ya hemos comentado anteriormente, los contratos inteligentes son los encargados de interactuar con Ethereum (Sepolia) en nuestro caso. Estos se implementan usando el lenguaje de programación Solidity. Para implementar, desplegar e interactuar con un contrato inteligente, lo primero es la codificación del mismo. Un contrato inteligente que emite eventos para registrar cambios en la base de datos tiene forma, como se puede observar en esta figura 6.13.

Este contrato inteligente se sitúa en la carpeta de ‘contracts’. Lo hemos añadido a aquí por comodidad, ya que usamos Remix para compilarlo y desplegarlo. En un entorno de producción interesaría que estuviera en su propio repositorio con las herramientas propias para el desarrollo, pruebas y despliegues propios. Como podéis observar, el contrato es ‘Ownable’. Es un patrón muy utilizado en el desarrollo de los contratos inteligentes. Ownable, de OpenZeppelin proporciona una implementación del patrón, el cual permite que ciertas operaciones del contrato inteligente solo puedan ser ejecutadas por la dirección que desplegó el contrato. Al usar Ownable de OpenZeppelin, nos aseguramos de usar una implementación segura y muy probada de este patrón. Una vez codificado el contrato, podemos probarlo y ejecutarlo en Remix, hablaremos más adelante de esto en el capítulo 7 de pruebas, lo que nos interesa ahora es el proceso de compilación y despliegue. El compilado de un contrato inteligente es muy importante, ya que como resultado obtenemos el ABI. El ABI (Application Binary Interface) de un contrato inteligente es un esquema de comunicación que define como interactuar con el contrato, definiendo los métodos que soporta este, en términos simples, es como el manual del contrato inteligente, especificando que funciones pueden llamarse. En nuestro proyecto, lo almacenamos en ‘abis/DataIntegrityABI.js’, y tiene esta forma:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 import "@openzeppelin/contracts/access/Ownable.sol";
5
6 contract MedicalDataIntegrity is Ownable {
7
8     event DataHashUpdated(bytes32 indexed dataHash, uint256 indexed
9                         timestamp, address indexed owner, address updatedBy);
10
11    event PatientAdded(address indexed patient, uint256 indexed
12                        timestamp);
13    event DoctorAdded(address indexed doctor, uint256 indexed
14                        timestamp);
15
16    event AccessGranted(address indexed patient, address indexed
17                        doctor, uint256 indexed timestamp);
18    event AccessRevoked(address indexed patient, address indexed
19                        doctor, uint256 indexed timestamp);
20    event AccessRequested(address indexed patient, address indexed
21                        doctor, uint256 indexed timestamp);
22
23    constructor() Ownable(msg.sender) {}
24
25    function updateDataHash(string calldata _orbitCID, address
26                           _owner, address _updatedBy) public onlyOwner() {
27        bytes32 orbitCIDHash =
28            keccak256(abi.encodePacked(_orbitCID));
29        emit DataHashUpdated(orbitCIDHash, block.timestamp, _owner,
30                            _updatedBy);
31    }
32
33    function addPatient(address _patient) public onlyOwner() {
34        emit PatientAdded(_patient, block.timestamp);
35    }
36
37    function addDoctor(address _doctor) public onlyOwner() {
38        emit DoctorAdded(_doctor, block.timestamp);
39    }
40
41    function grantAccess(address _patient, address _doctor) public
42                         onlyOwner() {
43        emit AccessGranted(_patient, _doctor, block.timestamp);
44    }
45
46    function revokeAccess(address _patient, address _doctor) public
47                         onlyOwner() {
48        emit AccessRevoked(_patient, _doctor, block.timestamp);
49    }
50
51    function requestAccess(address _patient, address _doctor)
52                         public onlyOwner() {
53        emit AccessRequested(_patient, _doctor, block.timestamp);
54    }
55}
```

Figura 6.13: MedicalDataIntegrity.sol.

```
1 export const dataintegrityABI = [
2     {
3         "inputs": [],
4         "stateMutability": "nonpayable",
5         "type": "constructor"
6     },
7     {
8         "inputs": [
9             {
10                 "internalType": "address",
11                 "name": "owner",
12                 "type": "address"
13             }
14         ],
15         "name": "OwnableInvalidOwner",
16         "type": "error"
17     },
18     {
19         "inputs": [
20             {
21                 "internalType": "address",
22                 "name": "account",
23                 "type": "address"
24             }
25         ],
26         "name": "OwnableUnauthorizedAccount",
27         "type": "error"
28     },
29 ]
```

Figura 6.14: Fragmento de DataIntegrityABI.ts.

Una vez compilado el contrato inteligente, podemos desplegarlo. Para desplegarlo usando Remix. Para esto debemos realizar los siguientes pasos.

1.- Configurar red Metamask:

Para esto debemos obtener la URL RPC de nuestro proveedor, Infura en nuestro caso, y añadir la red a Metamask.

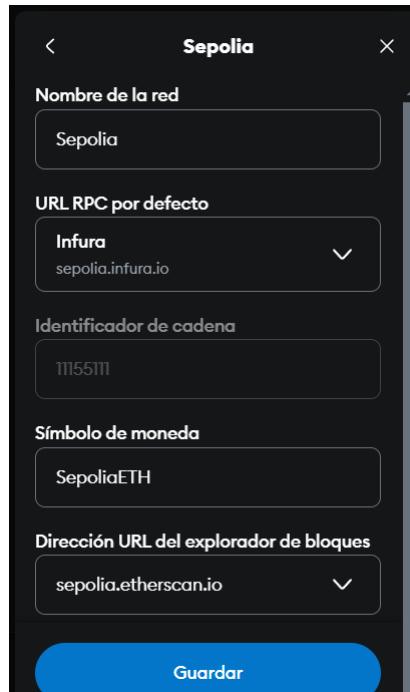


Figura 6.15: Configurar red desde el selector de redes de Metamask.

Una vez hecho esto, podemos dirigirnos al area de ‘Deploy and Run Transacciones de Remix’, indicar el ‘Environment’ donde queremos desplegar nuestro contrato, en nuestro caso será ‘Injected Provider - Metamask’, así desplegaremos el contrato en la red que hemos configurado previamente:

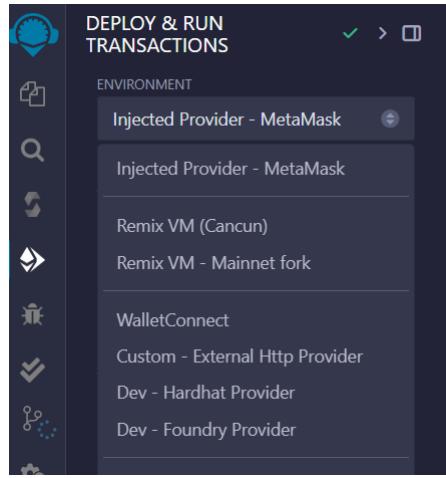


Figura 6.16: Configurar entorno en Remix.

Una vez configurado esto, deberemos escoger nuestra cuenta con la que queremos desplegar el contrato. Esta cuenta debe disponer en este caso de tokens de Sepolia para pagar el coste y las operaciones del contrato. Seleccionando el contrato compilado, podremos desplegarlo:



Figura 6.17: Configuración completada para desplegar.

Al desplegar, se nos abrirá Metamask para confirmar y pagar la transacción y tendremos un mensaje de confirmación:

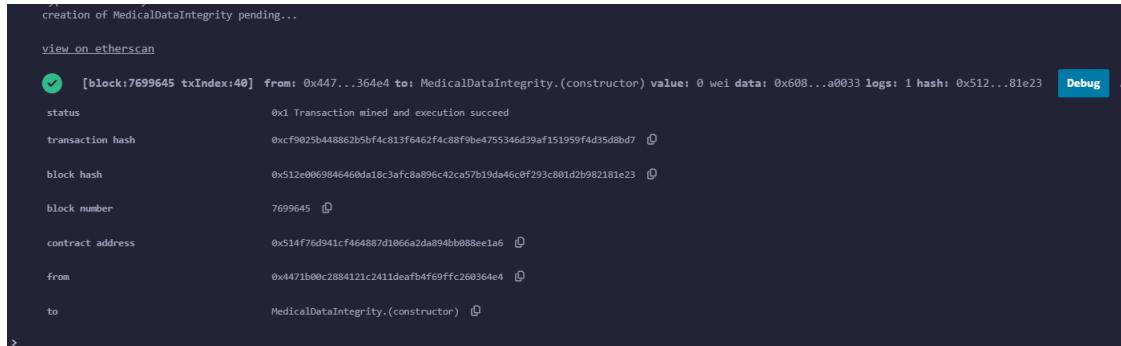


Figura 6.18: Confirmación despliegue contrato exitosa.

Una vez desplegado, necesitaremos la dirección del contrato junto con el ABI poder interactuar con él. Ahora, con esto podemos instanciar nuestro contrato en el backend de nuestra aplicación. Para esto, tenemos el archivo ‘/utils/contract.js’, donde instanciamos al contrato:

```

1 import { Contract, Wallet } from 'ethers';
2 import { JsonRpcProvider } from 'ethers';
3 import {
4     DATA_INTEGRITY_CONTRACT_ADDRESS,
5     DATA_INTEGRITY_CONTACT_ABI,
6     PK,
7 } from './constants.js';
8
9 export default function getMedicalDataIntegrityContract() {
10     const provider = new JsonRpcProvider(URL_INFURA_SEPOLIA)
11     const wallet = new Wallet(PK, provider);
12
13     return new Contract(
14         DATA_INTEGRITY_CONTRACT_ADDRESS,
15         DATA_INTEGRITY_CONTACT_ABI,
16         wallet
17     );
18 }

```

Figura 6.19: contract.js.

Aquí, definimos nuestro proveedor JsonRpcProvider, un proveedor básicamente es un puente entre nuestra aplicación y la blockchain. Este objeto nos permite conectarnos con la blockchain indicándole nuestro proveedor. A continuación, instanciamos la Wallet con la clave privada que podemos obtener de Metamask y nuestro proveedor. Así podemos instanciar el contrato junto con su dirección, el ABI y la billetera. Así, nuestras rutas pueden interactuar con la blockchain.

6.3 Implementación del Frontend

6.3.1 Estructura del frontend

Como podemos ver en la siguiente imagen, podemos ver cómo hemos organizado nuestra aplicación móvil. Así cuando hablamos de los siguientes puntos, podemos usarla como referencia. La estructura del frontend es algo más compleja, por lo que mostraremos todo por partes dependiendo del contexto, pero como referencia general, contamos con esto:

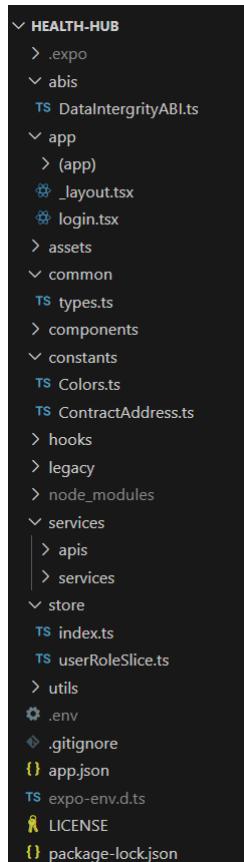


Figura 6.20: Estructura carpetas.

El fichero principal en nuestro caso es el '/app/_layout.tsx'. Esto es así ya que usamos Expo Router para la navegación, más adelante hablaremos expo router. Este fichero tiene la siguiente forma:

```
1 import '@walletconnect/react-native-compat';
2
3 import { QueryClient, QueryClientProvider } from
4   '@tanstack/react-query';
5 import { createAppKit, AppKit } from
6   '@reown/appkit-wagmi-react-native';
7 import { Slot } from "expo-router";
8 import { WagmiProvider } from 'wagmi';
9 import { Provider } from "react-redux";
10 import { store } from '@/store';
11 import { wagmiConfig } from '@/utils/wagmi';
12 import { PaperProvider } from 'react-native-paper';
13
14 const queryClient = new QueryClient();
15
16 const projectId = process.env.EXPO_PUBLIC_PROJECT_ID as string;
17
18 createAppKit({
19   projectId,
20   wagmiConfig,
21   enableAnalytics: true,
22 });
23
24 export default function RootLayout() {
25   return (
26     <Provider store={store}>
27       <WagmiProvider config={wagmiConfig}>
28         <QueryClientProvider client={queryClient}>
29           <PaperProvider>
30             <Slot />
31             <AppKit />
32           </PaperProvider>
33         </QueryClientProvider>
34       </WagmiProvider>
35     </Provider>
36   );
37 }
```

Figura 6.21: contract.js.

Aquí inicializamos:

- **Redux Provider**
- **Wagmi Provider**
- **QueryClient Provider**
- **React Native Paper Provider**
- **Appkit:** De Reown (WalletConnect), es una solución de código abierto para integrar conexiones de wallets y otras funcionalidades Web3. Proporciona a los usuarios autenticarse con un clic, simplificando la experiencia de usuario.

Con estos providers, podemos acceder a diferentes funcionalidades en nuestros componentes. Como veremos a continuación.

6.3.2 Capa de acceso a servicios

En esta capa tenemos los servicios del frontend, encargados de comunicarse con el API REST que hemos implementado anteriormente. Para crearlos, se han implementado APIs, ayudándonos de Redux Toolkit, mediante el uso de querys (operaciones lectura) y mutations (operaciones de creación, actualización e eliminación). Cada API se comunica con su ruta correspondiente del servicio y nos provee automáticamente de un hook para usar en nuestros componentes. En la figura 6.22 también se implementa la estrategia de cacheo, muy útil para evitar peticiones no necesarias y realizarlas cuando se actualizan los datos. Aquí podemos ver un ejemplo de la implementación del API de los usuarios.

```

1 import { decryptData } from '@/utils/crypto';
2 import { createApi, fetchBaseQuery } from
3     '@reduxjs/toolkit/query/react'
4 import { registerDoctor, registerUser, updateUser } from
5     '../services/userService';
6 import { Doctor, User } from '@/common/types';
7
8 const baseUrl = process.env.EXPO_PUBLIC_API_URL as string;
9 // https://redux-toolkit.js.org/rtk-query/usage/queries
10 export const usersApi = createApi({
11     reducerPath: 'usersApi',
12     tagTypes: ['User'],
13     baseQuery: fetchBaseQuery({ baseUrl }),
14     endpoints: (builder) => {
15         // CREATE
16         registerUser: builder.mutation<User, { address: string; user: Partial<User>}>({
17             async queryFn({ address, user }) {
18                 const data = await registerUser(user, address);
19                 return { data };
20             },
21             invalidatesTags: (
22                 _result, _error, { address }
23             ) => [{ type: 'User', address }],
24         }),
25         // READ
26         getUserByAddress: builder.query<User, string>({
27             query: (address) => `users/${address}`,
28             transformResponse: async (response: any) => {
29                 return await decryptData(response);
30             },
31             providesTags: (address) => [{ type: 'User', address }],
32         }),
33         getIsDoctor: builder.query<boolean, string>({
34             query: (address) => `users/is-doctor/${address}`,
35             transformResponse: (response: any) => {
36                 return response.isDoctor;
37             },
38             providesTags: (address) => [{ type: 'User', address }],
39         }),
40         // UPDATE
41         updateUser: builder.mutation<User, { address: string; user: Partial<User>}>({
42             async queryFn({ address, user }) {
43                 const data = await updateUser(user, address);
44                 return { data };
45             },
46             invalidatesTags: (
47                 _result, _error, { address }
48             ) => [{ type: 'User', address }],
49         }),
50     })
51
52 export const {
53     useRegisterUserMutation,
54     useRegisterDoctorMutation,
55     useGetIsDoctorQuery,
56     useGetIsDoctorEnabledQuery,    67
57     useGetIsPatientQuery,
58     use GetUserByAddressQuery,
59     useGetDoctorByAddressQuery,
60     useUpdateUserMutation,
61 } = usersApi;

```

Así, implementaremos para cada ruta de nuestro servicio, una api para que nuestra aplicación móvil pueda comunicarse con ella. Luego de esto, podemos usar estos hooks en nuestras pantallas:

```
const { isConnected, address, isConnecting } = useAccount();
const { data: isPatient, isLoading: isLoadingPatient } = useGetIsPatientQuery(address!);
const { data: isDoctor, isLoading: isLoadingDoctor } = useGetIsDoctorQuery(address!);
const { data: isDoctorEnabled, isLoading: isLoadingDoctorEnabled } = useGetIsDoctorEnabledQuery(address!);
```

Figura 6.23: Uso de hooks para recuperar datos.

Aprovechando este fragmento de la pantalla de login, aparte de ver los hooks del API de usuario, podemos ver el hook useAccount de wagmi. Appkit usa wagmi para gestionar los métodos con la blockchain. Así, podemos usar hooks y métodos en nuestros componentes, como es el caso de useAccount, del cual podemos obtener el address de la cuenta, y comprobar el estado con isConnected e isConnecting.

6.3.3 Capa Interfaz de Usuario

En esta capa contamos con todas las pantallas y los componentes encargados de la vista de la plataforma. Como hemos mencionado con anterioridad, para gestionar la navegación hemos usado Expo Router. Expo Router es un enrutador basado en archivos para React Native y aplicaciones web. Nos permite gestionar la navegación entre pantallas en nuestras aplicaciones, permitiendo a los usuarios moverse sin problemas entre las diferentes pantallas de la interfaz, pudiendo re-usar componentes. Con Expo Router, cuando se añade un archivo a un directorio, se crea automáticamente en una ruta de la aplicación. Por esto, contamos con la siguiente estructura: Así, cada fichero, organizado jerárquicamente en su dirección, es una pantalla de nuestra aplicación. Cada jerarquía debe suelen contar con un archivo ‘_layout.tsx’ donde se suelen definir las páginas de ese nivel, pudiendo compartir estilos entre estas.

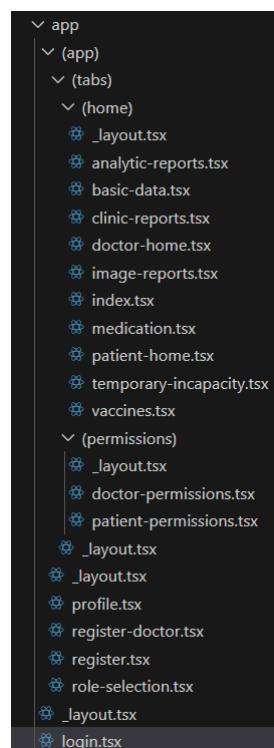


Figura 6.24: Rutas de la aplicación.

Un ejemplo del ‘_layout.tsx’ del layout del menú de pacientes:

```
1 import { Stack } from 'expo-router';
2
3 export default function HomeLayout() {
4   return (
5     <Stack screenOptions={{ headerShown: false }}>
6       <Stack.Screen name="index" options={{ title: 'Home' }} />
7       <Stack.Screen name="basic-data" options={{ title: 'Datos
Básicos' }} />
8       <Stack.Screen name="medication" options={{ title:
'Medicación' }} />
9       <Stack.Screen name="clinic-reports" options={{ title:
'Informes Clínicos' }} />
10      <Stack.Screen name="analytic-reports" options={{ title:
'Pruebas Analíticas' }} />
11      <Stack.Screen name="image-reports" options={{ title: 'Pruebas
de Imagen' }} />
12      <Stack.Screen name="temporary-incapacity" options={{ title:
'Incapacidad Temporal' }} />
13      <Stack.Screen name="vaccines" options={{ title: 'Vacunas' }}>
14    </Stack>
15  );
16}
```

Figura 6.25: _layout.tsx.

En la figura 6.25 podemos observar todas las navegaciones que pueden salir del menú del paciente.

En cada jerarquía, el fichero index.tsx define la ruta base (/). Por ejemplo, nosotros lo usamos para redirigir al home del paciente o del doctor:

```
1 import DoctorHome from "./doctor-home";
2 import PatientHome from "./patient-home";
3 import Login from "@/app/login";
4 import { useAccount } from "wagmi";
5 import { useGetIsDoctorQuery, useGetIsPatientQuery } from
6   "@/services/apis/user";
7
8 export default function Home() {
9   const { address } = useAccount();
10  const { data: isPatient } = useGetIsPatientQuery(address!);
11  const { data: isDoctor } = useGetIsDoctorQuery(address!);
12
13  if (isPatient) {
14    return <PatientHome />;
15  } else if (isDoctor) {
16    return <DoctorHome />;
17  } else {
18    return <Login />;
19 }
```

Figura 6.26: index.tsx.

Siendo así, por ejemplo, ‘PatientHome’:

```

1 const { width, height } = Dimensions.get("window");
2
3 export default function PatientHome() {
4     const { address } = useAccount();
5     const router = useRouter();
6     if (!address) return null;
7     const {
8         data: user
9     } = use GetUserByAddressQuery(address);
10
11    return (
12        <SafeAreaView style={styles.container}>
13            <CommonHeader userName={user?.name} />
14            <View style={styles.menuContainer}>
15                {items.map((item, index) => {
16                    const animatedStyle = useAnimatedStyle(() => ({
17                        transform: [
18                            { translateX: menuPositions[index].x.value },
19                            { translateY: menuPositions[index].y.value }
20                        ],
21                    }));
22
23                    return (
24                        <Animated.View key={index} style={[styles.menuItem,
25 animatedStyle]}>
26                            <TouchableOpacity
27                                style={[styles.itemButton]}
28                                onPress={()=> router.push(item.href)}>
29                                <Text style={styles.itemText}>{item.label}</Text>
30                                </TouchableOpacity>
31                            );
32                    )});
33                <TouchableOpacity
34                    style={styles.centerButton}
35                    onPress={() => router.push("/basic-data")}>
36                    <Text style={styles.centerText}>Datos Médicos</Text>
37                </TouchableOpacity>
38            </View>
39        </SafeAreaView>
40    );
41 }

```

Figura 6.27: patient-home.tsx.

Como se puede observar, al haber definido todas las rutas en el layout, podemos navegar desde aquí fácilmente a todas las pantallas de los datos.

6.3.4 Estilos y Componentes

Como ya se ha podido observar, usamos StyleSheets y React Native Paper para realizar nuestras pantallas y definir los estilos de estos componentes que los conforman. Por ejemplo, del patient-home.tsx, figura 6.27, tendríamos los siguientes estilos definidos:

```
1 const styles = StyleSheet.create({
2   container: {
3     flex: 1,
4     backgroundColor: "#F5F5F5",
5   },
6   header: {
7     width: "100%",
8     height: height * 0.1,
9     flexDirection: "row",
10    alignItems: "center",
11    justifyContent: "space-between",
12    paddingHorizontal: 16,
13  },
14  profileButton: {
15    backgroundColor: "#62CCC7",
16    width: 40,
17    height: 40,
18    borderRadius: 20,
19    alignItems: "center",
20    justifyContent: "center",
21  },
22  profileText: {
23    color: "#fff",
24    fontWeight: "bold",
25    fontSize: 16,
26  },
27  greetingText: {
28    fontSize: 18,
29    fontWeight: "bold",
30    color: "#333",
31  },
32  notificationButton: {
33    padding: 8,
34  },
35  menuContainer: {
36    flex: 1,
37    alignItems: "center",
38    justifyContent: "center",
39  },
40  menuItem: {
41    position: "absolute",
42  },
43});
```

Figura 6.28: Estilos de patient-home.tsx.

También hemos creado componentes propios en algunos casos que lo hemos visto necesario, la verdad es que podríamos componentizar más nuestras pantallas, esto se añadirá al apartado de trabajo futuro.

Capítulo 7

Pruebas

En este capítulo expondremos los diferentes procesos de pruebas seguidos de cara a asegurar el buen funcionamiento del sistema. A continuación explicaremos cómo hemos probado los diferentes elementos del TFM.

7.1 Backend

7.2 Servicio Node.js

Se ha probado el servicio desarrollado en Node.js para evaluar su correcto funcionamiento y rendimiento. Para lograr esto, se hicieron pruebas exhaustivas verificando la ejecución correcta de las rutas, manejo de las peticiones HTTP y respuestas del servidor. Para realizar este ejercicio de pruebas, utilicé la herramienta Postman para mandar solicitudes y analizar las respuestas, asegurando que los endpoints respondieron de acuerdo con lo esperado, así mismo también validando el correcto funcionamiento de las bases de datos de OrbitDB. En la figura 7.1 se muestran algunas colecciones de llamadas de Postman.

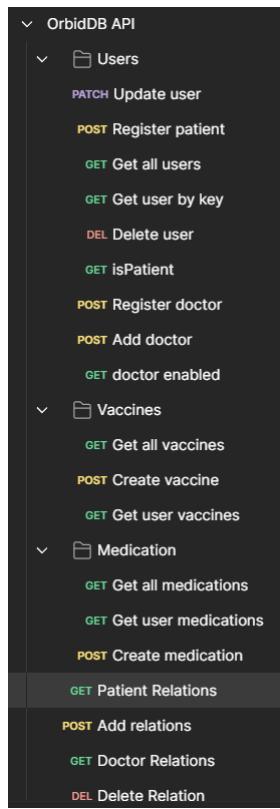


Figura 7.1: Colección llamadas postman.

También todos los endpoints han sido probados a la hora de probar los flujos de la aplicación.

7.3 Smart Contracts

Siguiendo la misma filosofía, para comprobar el funcionamiento de los métodos de los smart contracts y su rendimiento, estos se han probado conjuntamente con las operaciones de las rutas, ya que todos los métodos del contrato son utilizados en uno u otro método de cada ruta. A parte, con la ruta de data integrity, hemos también podido comprobar que los eventos se emiten de forma correcta. Además, durante el desarrollo hemos ejecutado cada uno de los métodos en Remix. Antes de la compilación, hemos podido comprobar que todos los métodos funcionaban correctamente y que solo el dueño podía ejecutarlos. Igualmente que Postman, Remix permite probar todos los métodos y ejecutarlos.

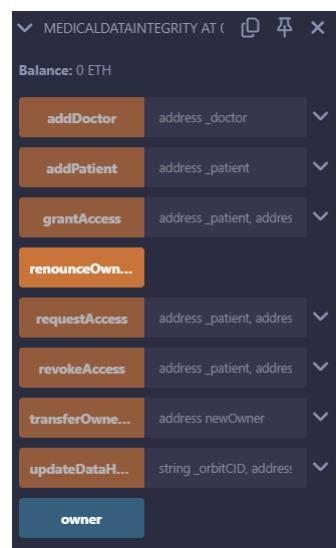


Figura 7.2: Colección llamadas Remix.

Cuando ejecutamos un método, podemos ver en la consola de Remix si se ha ejecutado correctamente:

```
transact to MedicalDataIntegrity.addDoctor pending ...
view on etherscan
[✓] [block:7701983 txIndex:31] from: 0x4471b00C...2D3b6E1Fc to: MedicalDataIntegrity.addDoctor(address) 0x5b0...6e1fc value: 0 wei data: 0x478...364e4 logs: 1
hash: 0x18a...cc60e
```

Figura 7.3: Ejecución correcta Remix.

La consola de Remix también nos proporciona el enlace a Etherscan para comprobar que de verdad la transacción ha sido registrada:

The screenshot shows a transaction details page on Etherscan. At the top, there are tabs for 'Overview' (which is selected), 'Logs (1)', and 'State'. A note says '[This is a Sepolia Testnet transaction only]'. Below are the following fields:

- Transaction Hash: 0x027dea7f1bb65c8167969a50be660cbdf0bc03b87313a9082c03ed5d59c26312
- Status: Success
- Block: 7701983 (9 Block Confirmations)
- Timestamp: 1 min ago (Feb-13-2025 10:44:12 PM UTC)
- Transaction Action: Call Add Doctor Function by 0x4471b00C...C260364e4 on 0x5B0D8EE6...2D3b6E1Fc
- From: 0x4471b00C2884121c2411deafb4f69ffC260364e4
- To: 0x5B0D8EE6B14E3329f7720852372eF882D3b6E1Fc (✓)

Figura 7.4: Verificación Etherscan.

7.4 Frontend

A la hora de testear el Frontend, hemos usado extensivamente la aplicación en varios dispositivos y en varios sistemas operativos. Para lograr esto, nos hemos servido de las facilidades que nos brinda Expo. Con expo, una vez desplegamos nuestra aplicación se nos genera un código QR, como podemos ver en la figura 7.5, el cual podemos escanear con nuestro dispositivo móvil iOS para poder utilizar la aplicación desde nuestro dispositivo. Además, con nuestro emulador de Android Studio, hemos podido probar la aplicación en un entorno Android.

Con esto y con un uso extensivo de la aplicación en ambos sistemas operativos, nos hemos asegurado de que todo funcionaba correctamente y con un buen rendimiento. A continuación mostraremos el resultado de la aplicación.



Figura 7.5: Código QR.

7.5 Pruebas de validación funcional

En esta sección, mostraremos todas las vistas que se han implementado y cómo interactuamos con ellas. La primera pantalla que ve el usuario es la pantalla de inicio de sesión (figura 7.6). Al clicar en ‘Conectar Wallet’, se mostrará el siguiente modal (figura 7.7) donde podemos elegir la billetera a la que seremos redirigidos.



Figura 7.6: Pantalla inicio sesión.

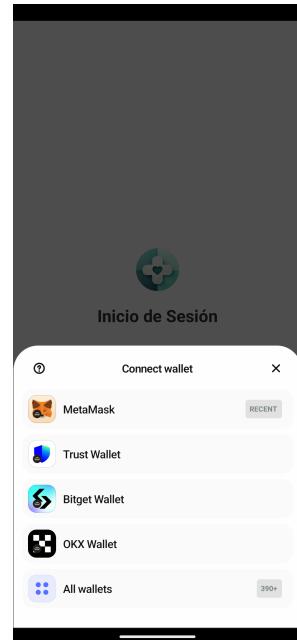


Figura 7.7: Modal para seleccionar billetera.

Una vez iniciamos sesión con Metamask, nos redirigirá a nuestra aplicación de nuevo. Al paciente se le mostrará el siguiente formulario de registro (figura 7.8) donde debe llenar estos datos obligatoriamente.

The screenshot shows a mobile application interface titled "Registro de Usuario". It contains four input fields: "Nombre" (Name) with placeholder "Introduce tu nombre", "Fecha de nacimiento" (Date of birth) with placeholder "Introduce tu fecha de nacimiento", "Teléfono" (Phone) with placeholder "Introduce tu teléfono", and "Contraseña" (Password) with placeholder "Introduce tu contraseña". Below these fields is a teal-colored "Enviar" (Send) button.

Figura 7.8: Pantalla registro paciente.

The screenshot shows the same mobile application interface as Figura 7.8, but with data entered into the fields. The "Nombre" field now contains "Angel", the "Fecha de nacimiento" field contains "23/05/1999", the "Teléfono" field contains "666666666", and the "Contraseña" field contains ".....". The "Enviar" button remains teal.

Figura 7.9: Registro paciente cubierto.

Al hacer clic en ‘Entrar’, se registrará al usuario en nuestra plataforma y lo redirigimos al menú principal del paciente (figura 7.10).

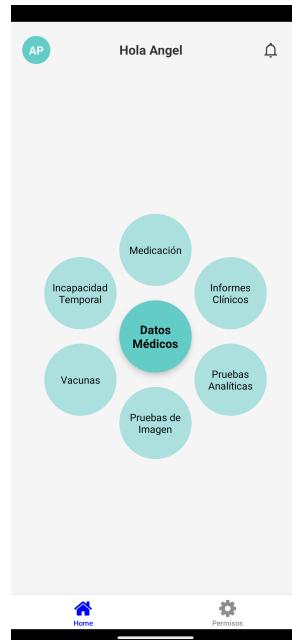


Figura 7.10: Menú principal paciente.

Desde aquí, podemos navegar a cualquier categoría de datos del usuario, también, clicando en la esquina izquierda, podemos acceder a la información del perfil del usuario (figura 7.11). En esta pantalla, aparte de poder ver nuestros datos, podemos añadir algunos datos opcionales como podemos ver en la figura 7.12.

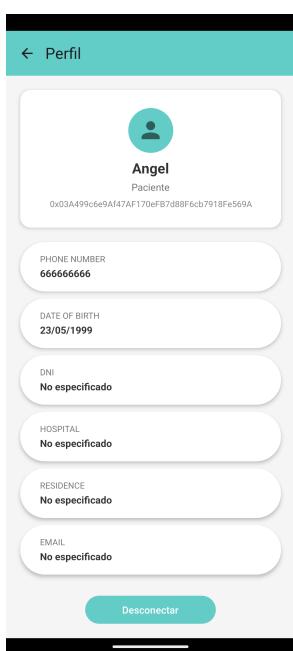


Figura 7.11: Perfil usuario.

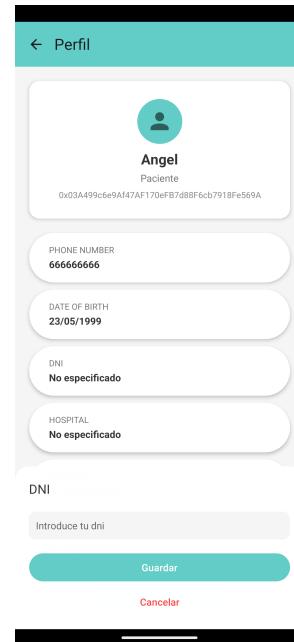


Figura 7.12: Modificando información perfil de usuario.

Volviendo al menú del usuario, si clicamos en cualquier burbuja seremos redirigidos a la pantalla de los datos correspondiente. En nuestro caso, navegaremos a la información de la medicación (figura 7.13).



Figura 7.13: Pantalla medicación del paciente.

Este es un formulario para agregar un medicamento. La parte superior muestra el nombre del paciente "Hola Angel" y un ícono de tres líneas horizontales. El formulario tiene un encabezado "Añadir Medicamento". Los campos incluyen: "Paracetamol" (enfocados), "500 mg", "2 veces al dia", "7 días", "10/01/2025" y "Agua y reposo". Abajo del formulario hay un botón azul "Guardar" y un botón gris "Cancelar".

Figura 7.14: Formulario añadir medicamento.

Una vez añadido, podremos ver el medicamento añadido (ilustrado en la Figura 7.15) podemos ver un medicamento añadido. Si añadiésemos más, aquí se mostrarían hasta 3 medicamentos (figura 7.16). Para ver todos los medicamentos, una vez tenemos más de 3, podemos clicar en ver todo, pudiendo ver la lista entera de medicamentos agrupados por fecha como podemos ver en la figura 7.17.



Figura 7.15: Lista medicamentos.



Figura 7.16: Lista con más de tres medicamentos.

Ahora, en caso de que el paciente quisiera ver los doctores que tienen acceso a añadirle o visualizar sus medicamentos, este debería usar la barra inferior de navegación para ser redirigido a la pantalla de permisos (figura 7.18).



Figura 7.17: Detalle lista medicamentos.

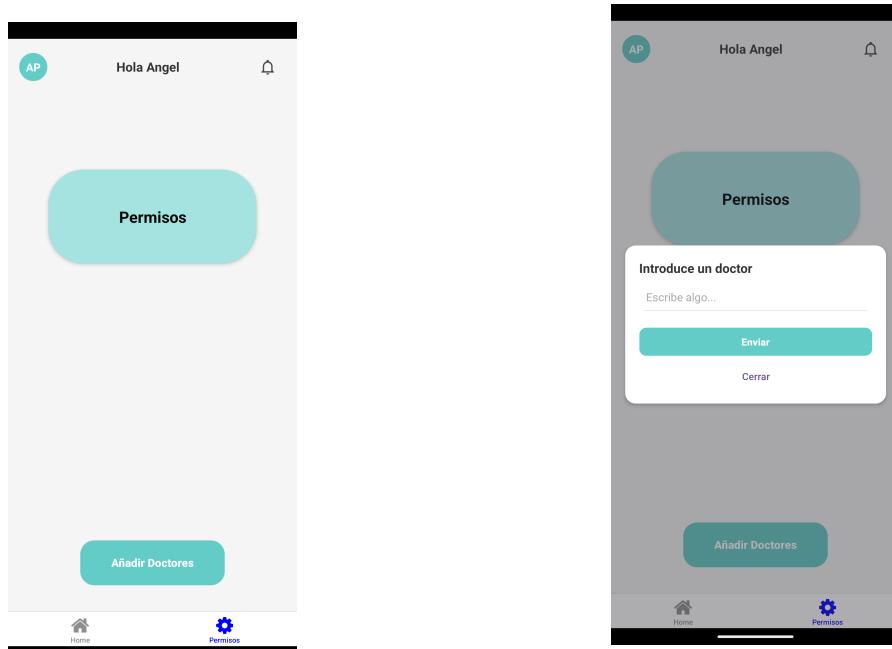


Figura 7.18: Pantalla permisos.

Figura 7.19: Modal conceder permisos a un doctor.

Pero antes que nada debemos ver cómo se registra el doctor. Para añadir un poco de variedad y así demostrar que se ha probado en diferentes dispositivos, las imágenes a continuación han sido sacadas de un dispositivo iPhone. Una vez el doctor ha sido verificado por el administrador, una vez iniciada la sesión con Metamask (figura 7.6) se le redirigirá a un formulario diferente al del paciente (formulario del paciente en figura 7.8). Desde aquí (figura 7.20) el usuario podrá llenar sus datos (figura 7.21) y registrarse.

Registro de Doctor

Nombre
Introduce tu nombre

Hospital
Introduce el hospital donde trabajas

Especialidad
Introduce tu especialidad

Teléfono
Introduce tu teléfono

Contraseña
Introduce tu contraseña

Enviar

Figura 7.20: Pantalla formulario de registro de doctor.

Registro de Doctor

Nombre
Juan Rodríguez González

Hospital
Belen

Especialidad
Cirujano

Teléfono
666666666

Contraseña

Enviar

Figura 7.21: Pantalla formulario de registro de doctor cubierto.

Una vez registrado podemos ver la pantalla del menú inicial del doctor como en la figura 7.22). Desde aquí podemos igualmente navegar a la pantalla del perfil del doctor (figura 7.23) y para ver o añadir nuevos datos.



Figura 7.22: Pantalla menú inicial doctor.

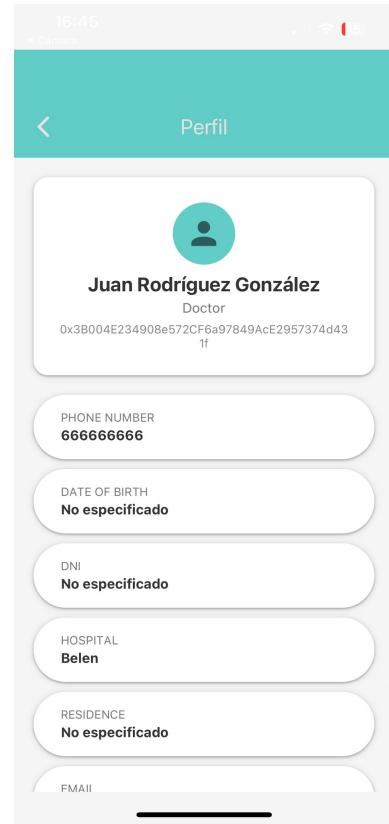


Figura 7.23: Pantalla perfil doctor.

Como hemos podido observar, este doctor no tiene ningún paciente a su cargo. Para solicitar permisos a un paciente, podemos hacerlo desde el menú principal o navegando a la pantalla de permisos (figura 7.24). Donde con un modal igual al del paciente, podrá solicitar permisos (figura 7.25).

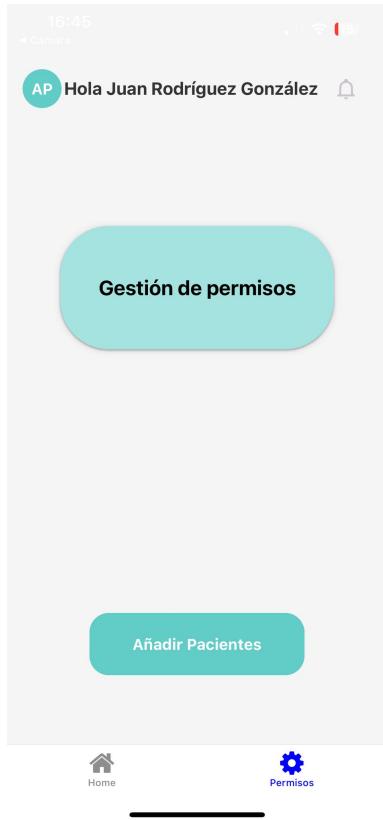


Figura 7.24: Pantalla gestión permisos doctor.

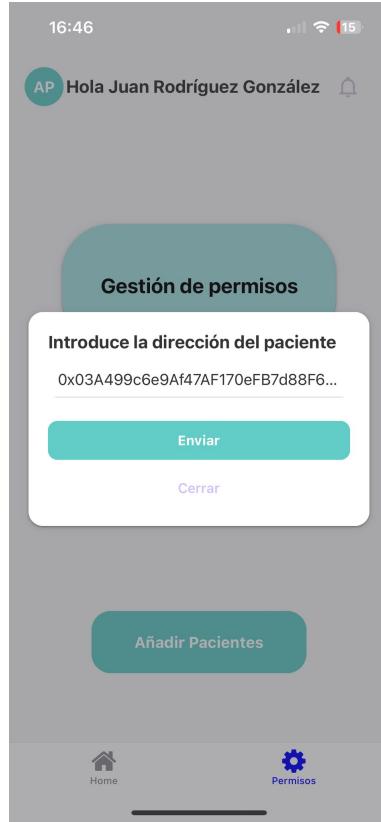


Figura 7.25: Modal solicitud de permisos.

Una vez solicitados estos permisos, si nos vamos de vuelta a la aplicación del paciente, podemos ver que ha recibido una notificación (figura 7.26). Clicando en la campana, podemos acceder a la lista de solicitudes (figura 7.27), clicando en el doctor le concederemos permisos.

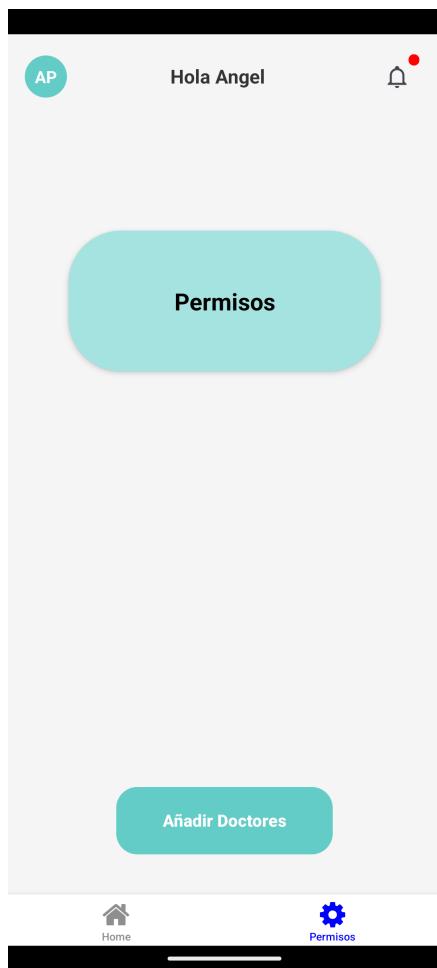


Figura 7.26: Pantalla permisos paciente con notificación.

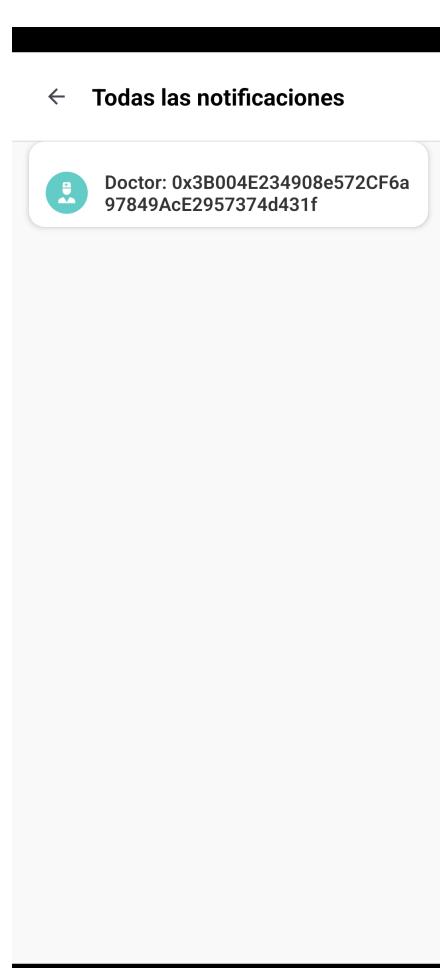


Figura 7.27: Lista de notificaciones.

Y ahora, tanto del lado del paciente (figura 7.29) como del lado del doctor, le aparecerá respectiva a cada uno el paciente o el doctor.



Figura 7.28: Menú inicial con un paciente del doctor.

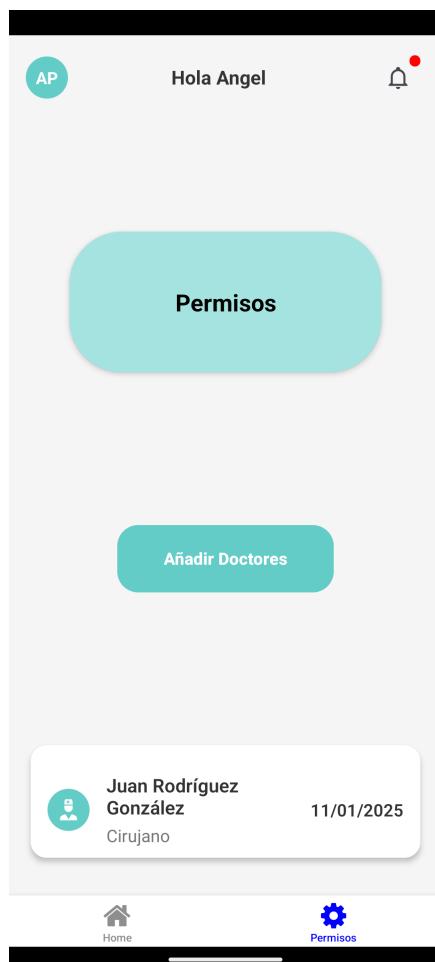


Figura 7.29: Pantalla de permisos del paciente.

Ahora el doctor ya tiene un paciente (figura 7.28), si clica en este podrá acceder al menú del paciente (figura 7.10) de este pudiendo acceder a todos los datos y añadir más. Para acabar con este viaje, el paciente podrá eliminar los permisos clicando en el doctor desde su vista de permisos. Al clicar en él, se mostrará el siguiente modal para confirmar la revocación de permisos como en la figura 7.30.

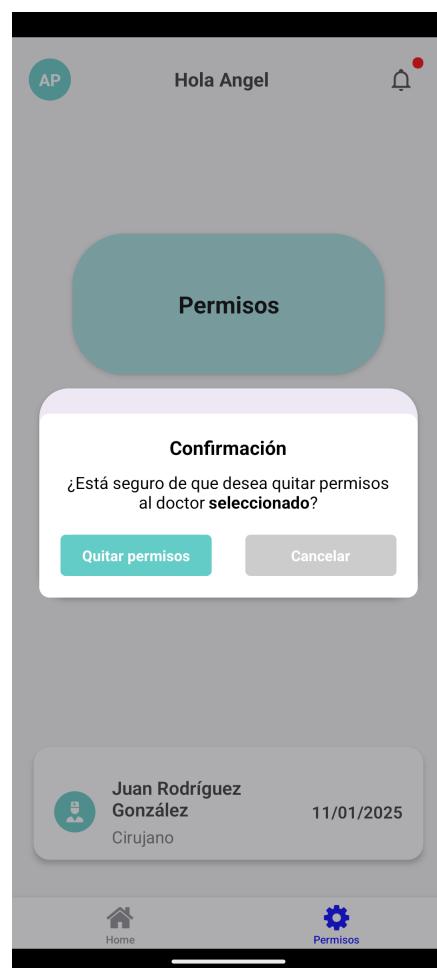


Figura 7.30: Modal confirmación revocar permisos.

Capítulo 8

Incidencias, conclusiones y trabajo futuro

En este capítulo hablaremos de lo que ha supuesto la realización desde el punto de vista del autor y de las posibles mejoras y líneas futuras.

8.1 Conclusiones

Mediante la realización de este TFM hemos podido aplicar la tecnología blockchain y la tecnología de las bases de datos descentralizadas en el ámbito de la salud de manera exitosa. El objetivo de este proyecto no ha sido otro que realizar un MVP (Minimum Viable Product o Producto Mínimo Viable), una prueba de concepto para comprobar que efectivamente el uso de estas tecnologías podría ser efectivo para el ámbito de la salud. Este proyecto tiene mucho margen de mejora, que ahora comentaremos en las líneas futuras para que sea un producto que se pueda usar en el mundo real, pero creo que es un muy buen punto de partida para conseguirlo.

La verdad que ha sido todo un desafío lograr este proyecto. Al usar tecnologías muy nuevas, y que no se han impartido en el MUEI, ha habido mucho trabajo de aprendizaje y hemos tenido que investigar extensivamente como poder aplicarlas de forma correcta y entender bien sus limitaciones. Durante el desarrollo de este TFM, me he dado cuenta de la dificultad que tiene adoptar nuevas tecnologías, me he encontrado con muchos problemas muy poco documentados, he navegado por hilos de github leyendo a otros desarrolladores tener los mismos problemas que yo y me alegra decir que en el desarrollo de este TFM he podido ayudar a otros desarrolladores como por ejemplo, en este bug abierto del repositorio de Metamask:

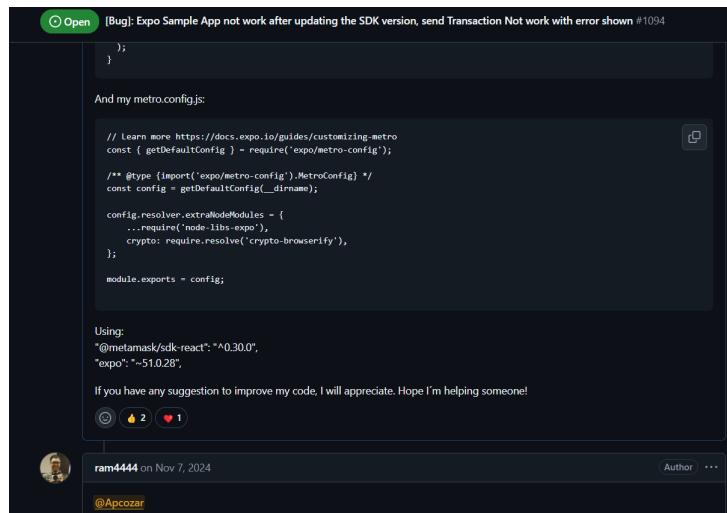


Figura 8.1: Comentario acogida positiva en repositorio de Metamask.

Por lo tanto, podemos concluir que este TFM nos ha servido para poner a prueba un nuevo paradigma de la gestión de los datos médicos, mediante la gestión privada, segura, transparente y descentralizada de los datos. La salud y nuestros datos deberían estar siempre con nosotros, ir con nosotros allá donde vayamos, y no ser dependientes de donde estamos para poder acceder a ellos. Hacer un software donde se pueden realizar auditorías de una forma tan sencilla es de interés para que no exista corrupción en el sistema o al menos intentar reducirla. Los recursos utilizados en el entorno de la salud son muypreciados y deberían ser bien usados, que estos no se vayan en fraudes y estafas. Además, en un mundo en donde la información es poder, el paradigma de tener silos de información va decreciendo con el tiempo. Como hemos comprobado en este proyecto, esto es un gran punto débil que hackers aprovechan para poder lucrarse con nuestros datos.

8.2 Líneas futuras

Este proyecto, aunque este haya cumplido su cometido y con los requisitos definidos, no sale de un entorno de desarrollo. Se podrían realizar muchas mejoras que nos permitan sacar este software a producción, como por ejemplo:

- **Mejoras en la experiencia de usuario:** Aunque este ha sido un objetivo, a la hora de la prueba de concepto de este TFM, procesos como el de insertar manualmente la dirección del paciente o la del doctor para el tema de los permisos puede ser muy mejorables.
- **Contemplar la gestión más datos médicos:** Ahora mismo, gestionamos un conjunto de datos limitados, se podría usar este sistema para almacenar más datos.
- **Realizar una buena colección de tests unitarios, de integración y funcionales.**
- **Desplegar el servicio en una red de nodos distribuidos:** Para agilizar el desarrollo, los datos solo se sitúan en un solo nodo local de OrbitDB, y estos se eliminan cada vez que reiniciamos el servicio.
- **Realizar más pruebas en más dispositivos.**

Apéndices

Apéndice A

Diagrama de Gantt y costes

El TFM se ha realizado en una serie de sprints. Cada ‘Sprint’ ha tenido como objetivo el desarrollo de una o varias funcionalidades. En cada ‘sprint’, menos en el ‘sprint’ inicial, se han llevado a cabo las siguientes fases:

- **Planificación**
- **Análisis de requisitos**
- **Diseño**
- **Implementación**
- **Pruebas**

A.1 Sprints

A.1.1 Sprint 0

Este ‘Sprint’ inicial ha sido pensado para realizar el estudio del arte, el estudio de la tecnología blockchain, IPFS, OrbitDB y de cómo podríamos aprovechar las ventajas de estas tecnologías en el contexto del ámbito de la salud.

A.1.2 Sprint 1

Este ‘Sprint’ nos hemos enfocado en el desarrollo de la Autenticación (sección, [4.5.1](#)).

A.1.3 Sprint 2

Este ‘Sprint’, una vez hemos realizado la autenticación hemos continuado con el desarrollo del Registro de los usuarios (secciones, [4.5.2](#) y [4.5.3](#)).

A.1.4 Sprint 3

Este ‘Sprint’ hemos implementado el desarrollo de Añadir los datos del usuario (secciones, 4.5.11 y 4.5.5).

A.1.5 Sprint 4

Este ‘Sprint’, una vez implementado la funcionalidad de datos, nos hemos enfocado en la Visualización de los datos (secciones, 4.5.8 y 4.5.4).

A.1.6 Sprint 5

Este ‘Sprint’, ya hemos podido dedicarnos al desarrollo de la Creación de permisos (secciones, 4.5.9 y 4.5.7).

A.1.7 Sprint 6

Este ‘Sprint’, para finalizar, se cerró con el desarrollo de la Visualización de permisos (secciones, 4.5.10 y 4.5.6).

A.2 Diagrama de Gantt

Para mostrar la implicación de los Sprints a lo largo del tiempo, hemos desarrollado un Diagrama de Gantt en las figuras A.1 y A.2.

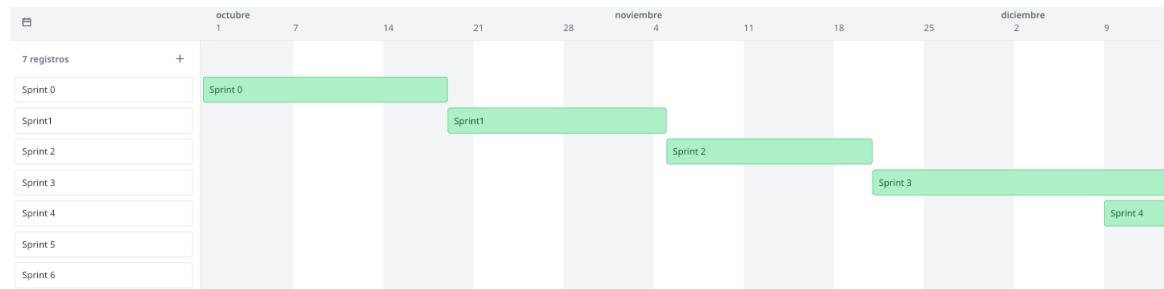


Figura A.1: Primer sección Diagrama de Gantt.

A.3 Costes del proyecto

Para calcular el coste de este proyecto, tomaremos en cuenta el precio/hora del desarrollador solamente. Esto lo haremos ya disponíamos del equipo necesario, no hay gasto de servidores al habernos limitado al entorno local, tampoco existen gastos emitidos por los contratos

Apéndice A. Diagrama de Gantt y costes

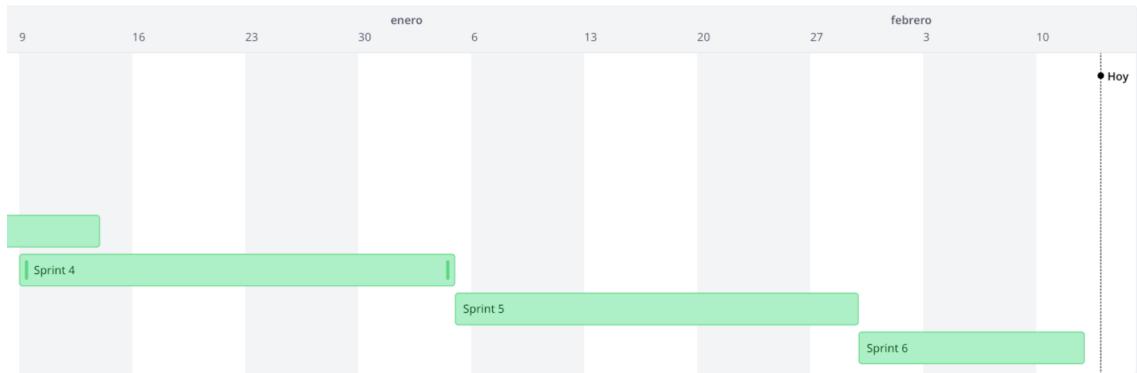


Figura A.2: Segunda sección Diagrama de Gantt.

inteligentes, ya que hemos utilizado una red de prueba. Todos estos gastos que no tenemos ahora deberíamos tenerlos en cuenta en caso de querer producir este TFM. Entonces para calcular el precio, nos ayudamos de la siguiente tabla A.2.

Equipo	Tiempo (h)	Precio (€/h)	Total (€)
Ingeniero de software	550	25	13.750,0 €
Primer director	65	85	5.525,0 €
Coste Total	-	-	19.275,0 €

Tabla A.1: Costes del equipo del proyecto.

Estimando tiempo de vida de un dispositivo móvil de 3 años y de 5 años de un portátil.

Recurso	Coste inicial (€)	Coste por hora (€/h)	Coste total (€)
Ordenador portátil	750	0,02	11 €
iPhone 13	850	0,03	16,5 €
Licencias	0	0	0 €
Figma 48	0	0	48,0 €
Coste Total	-	-	75,5 €

Tabla A.2: Costes del equipo del proyecto.

Bibliografía

- [1] “Historia clínica del sistema nacional de salud (hcdsns).” [En línea]. Disponible en: <https://www.sanidad.gob.es/areas/saludDigital/historiaClinicaSNS/home.htm>
- [2] “Filtración de datos en hospitales.” [En línea]. Disponible en: <https://www.infobae.com/espagna/2024/03/24/los-hospitales-espanoles-tienen-una-cuenta-pendiente-con-la-ciberseguridad-los-datos-medicos-confid>
- [3] “Uhs ransomware attack.” [En línea]. Disponible en: <https://www.techtarget.com/healthtechsecurity/news/366595382/UHS-Ransomware-Attack-Cost-67M-in-Lost-Revenue-Recovery-Efforts>
- [4] “Madrid viola privacidad de los ciudadanos.” [En línea]. Disponible en: https://www.eldiario.es/tecnologia/comunidad-madrid-violo-privacidad-ciudadanos-web-certificado-covid_1_10895587.html
- [5] “Google viola privacidad de los usuarios.” [En línea]. Disponible en: <https://www.aepd.es/prensa-y-comunicacion/notas-de-prensa/la-aepd-sanciona-google-llc-por-ceder-datos-terceros-sin>
- [6] “Patientory.” [En línea]. Disponible en: <https://www.criptonoticias.com/comunidad/eventos/patientory-plataforma-blockchain-cuidado-salud-expone-conferencia-himss18-vegas-5-9-mar>
- [7] “Falsificación medicamentos en España.” [En línea]. Disponible en: <https://www.uria.com/es/publicaciones/8974-nuevos-avances-en-la-lucha-contra-la-falsificacion-de-medicamentos-en-espana>
- [8] “La transformación digital salva vidas: el ejemplo de la sanidad electrónica de Estonia.” [En línea]. Disponible en: <https://www.marcvidal.net/blog/la-transformacion-digital-salva-vidas-el-ejemplo-de-la-sanidad-electronica-de-estonia>

- [9] “¿qué es el blockchain?” [En línea]. Disponible en: <https://www.ibm.com/es-es/topics/blockchain>
- [10] “Ipfs.” [En línea]. Disponible en: <https://ipfs.tech/>
- [11] “Peer-to-peer databases for the decentralized web.” [En línea]. Disponible en: <https://orbitdb.org/>
- [12] “¿qué es el criptografia?” [En línea]. Disponible en: <https://www.ibm.com/es-es/topics/cryptography#:~:text=La%20criptograf%C3%A1%20es%20la%20pr%C3%A1ctica,permiso%20y%20capacidad%20de%20descifrarla.>
- [13] “A statically-typed curly-braces programming language designed for developing smart contracts that run on ethereum.” [En línea]. Disponible en: <https://soliditylang.org/>
- [14] “Javascript.” [En línea]. Disponible en: <https://developer.mozilla.org/es/docs/Web/JavaScript>
- [15] “Typescript is javascript with syntax for types.” [En línea]. Disponible en: <https://www.typescriptlang.org/>
- [16] “Css.” [En línea]. Disponible en: <https://developer.mozilla.org/es/docs/Web/CSS>
- [17] “Ejecuta javascript en cualquier parte.” [En línea]. Disponible en: <https://nodejs.org/es>
- [18] “Comience con metamask portfolio.” [En línea]. Disponible en: <https://metamask.io/es/>
- [19] “Remix project saltar a la web3.” [En línea]. Disponible en: <https://remix-project.org/?lang=es>
- [20] “React native learn once, write anywhere.” [En línea]. Disponible en: <https://reactnative.dev/>
- [21] “The official, opinionated, batteries-included toolset for efficient redux development.” [En línea]. Disponible en: <https://redux-toolkit.js.org/>
- [22] “Wagmi reactivity for ethereum apps.” [En línea]. Disponible en: <https://wagmi.sh/>
- [23] “Toolkits to build onchain ux.” [En línea]. Disponible en: <https://reown.com/>
- [24] “Ai is powered by apis. apis are powered by postman.” [En línea]. Disponible en: <https://www.postman.com/>
- [25] “El ide oficial para el desarrollo de apps para android.” [En línea]. Disponible en: <https://developer.android.com/studio?hl=es-419>

- [26] “Build, scale, disrupt.” [En línea]. Disponible en: <https://www.infura.io/>
- [27] “Figma design.” [En línea]. Disponible en: <https://www.figma.com/es-es/design/>
- [28] “Coste de gas de un evento.” [En línea]. Disponible en: <https://ethereum.stackexchange.com/questions/106772/how-much-gas-does-it-cost-to-emit-an-event>