TRABAJOS FIN DE GRADO

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Desarrollo de una plataforma web para el control de pacientes diabéticos tipo I

Autor: Samuel Valverde García

Grado: Ingeniería informática

Tutor: Prof. D. José Luis López Ruiz
Cotutor: D. Juan Francisco Gaitán Guerrero
Departamento de Informática

Fecha: 17/02/2025

CREA

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Prof. D. José Luis López Ruiz y D. Juan Francisco Gaitán Guerrero, tutores del Proyecto Fin de Carrera titulado: **Desarrollo de una plataforma web para el control de pacientes diabéticos tipo I**, que presenta Samuel Valverde García, autorizan su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Febrero de 2025

El alumno: Los tutores:

Samuel Valverde García

Prof. D. José Luis López Ruiz

D. Juan Francisco Gaitán Guerrero

Tabla de contenidos

Tabla de Figuras

Índice de Tablas

1. Introducción

Para iniciar este proyecto, en primer lugar se presentará el contexto de aplicación en el que se desarrollará la plataforma propuesta, exponiendo los aspectos que han llevado a su creación. A continuación, se detallarán la motivación por el cual se eligió este proyecto, así como los objetivos que se pretenden alcanzar en el desarrollo y finalmente se describe cómo se organiza el contenido de este documento.

El proyecto consistirá en el desarrollo de una aplicación web que facilite la gestión y el seguimiento de pacientes con diabetes tipo I *[1]*. Para lograrlo, la plataforma estará diseñada de forma que los distintos usuarios (pacientes, médicos o familiares) puedan acceder, registrarse, compartir y analizar los datos relacionados con la enfermedad.

1.1. Motivación

En la actualidad, las enfermedades crónicas van en aumento en la población, y la diabetes se sitúa como una de las más comunes. Según estimaciones de la Organización Mundial de la Salud (OMS), más de 800 millones de personas en todo el mundo padecen algún tipo de diabetes, cifra que se ha cuadruplicado en los últimos 30 años ya que por 1990 las cifras rondaban los 200 millones.

Existen varios tipos de diabetes, entre los que destacan principalmente:

- Diabetes tipo I: Se produce cuando el páncreas deja de generar insulina o la produce en cantidades mínimas, forzando al paciente a depender de inyecciones o bombas de insulina para regular sus niveles de glucosa. Es frecuente su aparición en la infancia o adolescencia, y requiere un seguimiento continuo para evitar episodios de hipoglucemia o hiperglucemia.
- Diabetes tipo II: Es la forma más común y está relacionada a menudo con factores como obesidad, mala alimentación y falta de actividad física. Aunque no siempre requiere insulina, sí puede necesitar medicamentos orales y cambios en el estilo de vida. Su prevalencia ha aumentado notablemente en los últimos años debido a hábitos de vida poco saludables.

- Diabetes gestacional: Afecta a algunas mujeres durante el embarazo. Aunque suele remitir después del parto, puede suponer un riesgo para la madre y el bebé si no se controla adecuadamente.

Entre todas ellas, la diabetes tipo I suele considerarse especialmente compleja por la demanda diaria de insulina y la necesidad de un control continuo de la glucosa, lo que requiere a menudo sensores y varios registros al día. Esto implica un esfuerzo constante tanto por parte del paciente como de los profesionales médicos, que deben ajustar el tratamiento con base en la información recabada. Además disponen de sensores de glucosa que en muchos casos están financiados por los sistemas de salud realizando monitorizaciones constantes, generando una gran cantidad de datos. Sin embargo, las aplicaciones comerciales que acompañan a estos sensores, como LibreView u otras soluciones de la industria, suelen centrarse únicamente en el registro básico o en la visualización del histórico, sin ofrecer una integración completa del historial clínico ni una colaboración fluida entre pacientes, médicos y familiares.

Ante esta realidad, mi interés se basa en aprovechar los avances tecnológicos para facilitar un seguimiento personalizado de la diabetes tipo I. Aunque existen iniciativas de eHealth y telemedicina, a menudo se especializan en funciones muy puntuales o no consolidan toda la información (glucemias, hábitos alimenticios, medicamentos, actividad física) en un solo sistema. A la vez, la colaboración con familiares—quienes también se preocupan por el estado del paciente—suele quedar relegada o exenta de funcionalidades integrales.

Por ello, mi principal motivación es, por tanto, crear una aplicación que:

- Facilite la organización de información médica: de modo que los pacientes puedan acceder a sus datos de forma rápida y los médicos dispongan de indicadores completos para tomar decisiones.
- Ahorrar tiempo a los profesionales sanitarios: al centralizar de manera efectiva la información, ya no es necesario que el paciente suministre al médico, información engorrosa de ver como en tablas de excel. Para ello la aplicación cuenta con una interfaz y un entorno más fácil e intuitivo de visualización.

- Participación activa de pacientes y familiares: fomentando el autocuidado y el seguimiento constante, algo esencial en una enfermedad crónica como la diabetes.
- Sistema seguro y accesible: que cumpla con estándares de protección de datos y ofrezca una interfaz clara y atractiva.

El auge de soluciones digitales en el ámbito sanitario y la creciente demanda de herramientas de monitorización me impulsan a enfocar este proyecto precisamente en la diabetes tipo I, donde cada individuo genera decenas o cientos de mediciones al mes. Integrar estos datos en una plataforma robusta contribuye no solo a la eficiencia del sistema sanitario, sino también a la mejora real de la calidad de vida de quienes conviven a diario con esta patología. De ahí que haya decidido volcar mis esfuerzos en diseñar y demostrar un prototipo capaz de combinar diferentes tecnologías web para cubrir las necesidades de pacientes, profesionales médicos y familiares de forma integral.

1.2. Objetivos

Los objetivos principales del proyecto son los siguientes:

- Realizar un análisis de necesidades y soluciones para el contexto propuesto.
    - Identificar los requisitos funcionales y no funcionales del sistema, teniendo en cuenta el perfil de los usuarios (pacientes, médicos, familiares) y las posibles restricciones técnicas y legales (protección de datos, seguridad, etc.).

- Diseñar un sistema informático enfocado en el uso de arquitecturas, principios de diseño y metodologías propias del desarrollo web.
    - Definir la arquitectura a emplear (cliente-servidor, microservicios, etc.).
    - Establecer un modelo de datos claro y extensible que cubra todas las entidades relevantes (usuarios, medidas de glucosa, historial clínico, etc.).

- Elegir un entorno de desarrollo basado en tecnologías web e implementar un prototipo de aplicación que cumpla con los requerimientos identificados.
    - Seleccionar las herramientas y frameworks más apropiados (lenguajes de programación, librerías, ORM *[2]*, frameworks de frontend y backend).
    - Desarrollar un prototipo funcional que integre el back-end, la base de datos y la interfaz de usuario.

1.3. Estructura del documento

El presente documento se organiza en ocho capítulos, desde su planteamiento inicial hasta las conclusiones finales. En la Introducción, se exponen las motivaciones y objetivos generales, contextualizando la relevancia del trabajo y el marco en que se desarrolla. A continuación, el Análisis profundiza en un estudio preliminar del problema, revisando aplicaciones similares y definiendo las necesidades que motivan la propuesta. Durante esta fase, se comparan distintas tecnologías tanto en el ámbito de front-end como de back-end, para determinar las más apropiadas, y se definen los problemas a abordar, la metodología que se empleará, así como los requisitos funcionales y no funcionales, y las historias de usuario que orientan el desarrollo. Además, se establece un modelo de dominio que describe las entidades y relaciones implicadas en la solución.

Una vez sentadas las bases, en la Planificación se expone cómo se distribuyen las tareas en el tiempo, apoyándose en un diagrama de Gantt, y se realiza una estimación de costes que incluye gastos de personal, hardware, software y mantenimiento. Después, en el Diseño se detalla la parte conceptual, presentando el Diagrama Entidad-Relación y el Diagrama de Clases para mostrar cómo se estructuran y relacionan los datos, así como diagramas de casos de uso y de secuencia que ilustran el comportamiento del sistema. Se abordan también los aspectos de diseño de la interfaz de usuario y de la API REST, describiendo los componentes que conforman cada capa y la forma en que se comunican.

Después de estos puntos, se pasa a la Implementación, donde primero se describe la arquitectura general y luego se profundiza en los pormenores de la puesta en marcha de cada subsistema. El desarrollo del servidor se explica detallando la estructura del proyecto, la gestión de la base de datos, las dependencias y la generación de la documentación de la API. Por su parte, la implementación del cliente abarca la creación del proyecto con la tecnología elegida, la organización interna de componentes, los estilos y la responsividad, así como la manera de realizar las peticiones al servidor y las validaciones que se efectúan en el front-end. Además, se incluyen los pasos y configuraciones de Despliegue en entorno local y de producción.

Tras haber completado la construcción funcional del sistema, el capítulo de Pruebas relata cómo se ha verificado la corrección del back-end, fundamentalmente mediante la documentación generada por la aplicación, y cómo se ha validado el front-end a nivel de usabilidad y presentación. A continuación, en Conclusiones, se plasman las reflexiones más destacadas sobre la experiencia de llevar adelante el proyecto, los objetivos alcanzados y las lecciones aprendidas. Finalmente, se enumeran los Posibles desarrollos futuros, proponiendo mejoras o expansiones que darían lugar a una evolución significativa de la aplicación. Por último, se presenta la Bibliografía, con las fuentes que han respaldado al proyecto.

2. Análisis

2.1. Análisis preliminar

En el análisis preliminar se va a asentar las bases de la propuesta que partimos, dividido en cuatro secciones: aplicaciones similares, necesidades que extraemos en conclusión, estudio de tecnologías y descripción de las tecnologías finalmente empleadas.

2.1.1. Aplicaciones similares

En la actualidad, existen diversas aplicaciones y plataformas que abordan la gestión de la diabetes y, en general, la monitorización de la salud. Aunque no todas comparten exactamente el mismo alcance que el proyecto propuesto, resulta de

utilidad estudiar algunas soluciones más destacadas para identificar sus puntos fuertes y sus carencias, y de este modo definir mejor la propuesta de valor de la plataforma que se desarrollará.

A continuación, se describirán algunas aplicaciones consideradas como referentes en el ámbito de la gestión de diabetes tipo I y la mejora de la comunicación médico-paciente:

2.1.1.1. MySugr

Figura 1: Logo de MySugr [3]

- **Funcionalidad principal:**
  Orientada a la monitorización de la glucosa en sangre y al registro de datos diarios como alimentación, actividad física y dosis de insulina.

- **Características relevantes:**
  - Posee un apartado de gamificación que incentiva a los usuarios a registrar sus datos de manera constante.
  - Incluye un sistema de reportes automáticos para que los médicos puedan revisar la evolución del paciente.
  - Interfaz sencilla que facilita la usabilidad desde dispositivos móviles.

- **Limitaciones:**
  - El enfoque se centra en la gestión individual del paciente; no facilita, por ejemplo, la vinculación con familiares en una misma cuenta.

- La comunicación con profesionales sanitarios se realiza principalmente a través de la generación de informes, sin un módulo colaborativo dentro de la aplicación.

- **Apunte:**
  - Puede haber un riesgo en la salud de los usuarios porque según la Agencia Español de Medicamentos y Productos Sanitarios (AEMPS), esta aplicación podría estar proporcionando recomendaciones incorrectas de dosis de insulina, si tienes un dispositivo iOS con una versión anterior a la 3.99.1. *[5]*

2.1.1.2. Glooko

Figura 2: Logo de Glooko [4]

- **Funcionalidad principal:**
  Integración de datos procedentes de diferentes dispositivos de monitorización (medidores de glucemia, bombas de insulina, aplicaciones de ejercicio).

- **Características relevantes:**
  - Compatibilidad con múltiples marcas y modelos de medidores de glucosa, lo que lo hace muy versátil.
  - Representaciones gráficas avanzadas que permiten a los profesionales sanitarios y a los pacientes identificar patrones de control glucémico.

- Opción de compartir la información con el equipo médico en tiempo real o mediante informes detallados.

- **Limitaciones:**
  - No se centra tanto en la comunicación directa entre paciente y médico o en la participación activa de familiares, salvo el envío y revisión de datos.
  - Requiere dispositivos compatibles o integración con sistemas externos de medición.

2.1.1.3. LibreView

Figura 3: Logo de LibreView [6]

- **Funcionalidad principal:**
  Plataforma web asociada a los sensores Flash de la marca FreeStyle Libre para el registro automático de glucosa.

- **Características relevantes:**
  - Recepción de datos de glucosa de forma casi continua, generando gráficos e informes para uso del paciente y del profesional.
  - Posibilidad de analizar patrones a lo largo de semanas o meses y detectar hipoglucemias o hiperglucemias recurrentes.

- **Limitaciones:**
  - Enfoque muy específico en el hardware de FreeStyle Libre; no resulta tan flexible para pacientes que utilizan otros dispositivos.
  - No cubre otras áreas como la gestión de notas diarias sobre alimentación o la interacción familiar.

2.1.1.4. Diasend

Figura 4: Logo de Diasend

- **Funcionalidad principal:**
  Recolección de datos provenientes de bombas de insulina, medidores de glucosa y otros dispositivos médicos para un control centralizado.

- **Características relevantes:**
  - Amplia gama de dispositivos soportados y facilidad de exportar informes detallados.
  - Herramientas de análisis para profesionales, enfocadas en la optimización del tratamiento y la reducción de riesgos.

- **Limitaciones:**
  - Se centra principalmente en el ámbito profesional y en la conversión de datos técnicos en informes claros, sin un énfasis en la comunicación bidireccional con el paciente o con sus familiares.

- No ofrece un módulo de mensajería o la gestión de roles (familiar, médico, paciente).

2.1.1.5. Plataformas de Historia Clínica Electrónica (HCE)

- **Funcionalidad principal:** Son sistemas que integran la información médica de los pacientes en un entorno hospitalario o clínico.

- **Características relevantes:**
  - Permiten al personal sanitario compartir y acceder a la historia médica de un paciente de forma centralizada.
  - Se centran en garantizar la seguridad y el cumplimiento de estándares (por ejemplo, HL7 *[7]*, GDPR, HIPAA, etc.).

- **Limitaciones:**
  - Habitualmente no ofrecen la posibilidad de que el paciente y sus familiares gestionen de manera directa los datos clínicos o interactúen con ellos fuera del entorno hospitalario.
  - No suelen enfocarse en la diabetes tipo I de forma especializada, sino que abarcan múltiples patologías con un grado de personalización limitado.

2.1.2. Necesidades

Al analizar estas soluciones, se observa que muchas están orientadas a la monitorización de datos glucémicos y el registro de información sanitaria. Algunas destacan por su potencia de integración con diferentes dispositivos, mientras que otras simplifican la introducción y el seguimiento de datos, pero tienden a carecer de un entorno colaborativo que involucre a familiares o que facilite la comunicación fluida entre médico y paciente.

En el caso de la aplicación propuesta, se pretende cubrir las siguientes necesidades que no siempre están completamente atendidas en las plataformas mencionadas:

- Enfoque integral: Combinación de registro de datos médicos (glucosa, notas de hábitos, prescripciones, diagnósticos, etc.) con una capa de interacción (mensajes, solicitudes de vinculación, cambios de médico, etc.).
- Gestión de roles: Paciente, familiar, médico y administrador, cada uno con sus propios permisos y nivel de acceso a la información.
- Flexibilidad en la vinculación: Posibilidad de añadir o eliminar familiares y de solicitar cambios de médico directamente por parte del paciente, sin que la plataforma se limite a un único profesional o dispositivo.
- Interfaz accesible y multiplataforma: Diseño pensado para un uso continuo, desde distintos dispositivos y en diferentes contextos.

2.1.3. Estudio de tecnologías

Para el diseño e implementación de una aplicación web sanitaria capaz de gestionar la información de pacientes diabéticos tipo I, se ha valorado un amplio abanico de tecnologías tanto en la parte del front-end como en el back-end. El objetivo principal es encontrar herramientas que ofrezcan:

- Rendimiento y escalabilidad: Que puedan manejar un volumen de datos e interacciones creciente sin degradar significativamente la experiencia de uso.
- Facilidad de desarrollo y mantenimiento: Que permitan iterar con rapidez y hacer cambios conforme evolucione el proyecto o se descubran nuevas necesidades.
- Seguridad y cumplimiento normativo: Al tratar con datos sanitarios, es fundamental contar con mecanismos que protejan la información y cumplan las regulaciones de privacidad.
- Comunidad y soporte: Una comunidad activa y abundante documentación garantizan la resolución más ágil de posibles problemas y la existencia de buenas prácticas.

A continuación, se describen y comparan algunas de las tecnologías evaluadas:

2.1.3.1. Front-end

El front-end [8] es la parte visible de una página web o aplicación con la que los usuarios interactúan directamente. Es la cara visible del sitio web, está ubicada en el lado del cliente. Veamos diferentes alternativas:

2.1.3.1.1. React

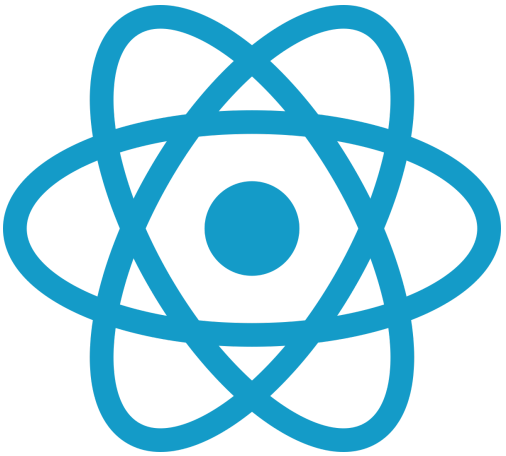


Figura 5: Logo de React

- **Descripción:**

  React *[9]* es una biblioteca de JavaScript enfocada en la construcción de interfaces de usuario a través de componentes reutilizables. Conocida por su Virtual DOM *[13]*, ayuda a optimizar la actualización selectiva de la interfaz sin recargar la página entera.

- **Ventajas:**
  - Amplia comunidad, gran cantidad de librerías y recursos.
  - Favorece la creación de aplicaciones muy interactivas y dinámicas.
  - División en componentes, que facilita la mantenibilidad.
  - Reducción de carga de re-renderizados innecesarios, ofreciendo una buena experiencia de usuario gracias al virtual DOM.

- **Desventajas:**
  - No incluye SSR (Server-Side Rendering) *[12]* de serie, se requiere un framework adicional como Next.js

- Requiere manejar la configuración de múltiples paquetes (enrutado, gestión de estados, etc.) que no vienen de serie.
- Tiene una curva de aprendizaje elevada en patrones avanzados para ser usados correctamente.

2.1.3.1.2. Angular

Figura 6: Logo de Angular

- **Descripción:**

  Angular *[10]* es un framework para crear aplicaciones de una sola página en el lado del cliente usando HTML y TypeScript, desarrollado y mantenido por Google. Incluye un extenso conjunto de herramientas integradas: enrutado, inyección de dependencias, formulación reactiva, servicios, etc.

- **Ventajas:**
  - Arquitectura robusta ya que el patrón MVC e inyección de dependencias facilitan la modularización y la escalabilidad, especialmente para grandes proyectos corporativos.
  - Integración nativa con TypeScript mejorando la robustez y la legibilidad del código, además de la facilidad de testing (Jasmine/Karma).

- CLI muy potente ya que permite generar componentes, servicios y módulos de manera automatizada, lo que acelera el desarrollo y mantiene una estructura consistente.
- Estructura escalable, adecuada para equipos grandes.

- **Desventajas:**
  - Curva de aprendizaje más pronunciada por su arquitectura y los conceptos avanzados como decoradores, guards, resolvers.
  - Sobredimensionado para proyectos pequeños o unipersonales, debido a su complejidad.
  - Menos flexible si se desean patrones de desarrollo muy personalizados.

2.1.3.1.3. Vue.js (y Ext.js)

Figura 7: Logos de Nuxt.js y Vue.js

- **Descripción:**

  Vue.js es un framework progresivo de JavaScript de código abierto, que permite construir interfaces de usuario gradualmente y aplicaciones de una sola página, combinando la reactividad y un enfoque declarativo. Para proyectos que requieran Server-Side Rendering, la herramienta adecuada es Nuxt.js *[11]*, que se sitúa en un nivel de abstracción similar a Next.js para React.

- **Ventajas:**

- Vue destaca por su sintaxis intuitiva y la capacidad de integrar gradualmente sus funcionalidades.
- La comunidad de Vue.js aunque más pequeña que la de React o Angular, sigue creciendo y ofrece multitud de plugins y librerías para la mayoría de necesidades como enrutado, estado global o testing.
- Nuxt.js proporciona SSR y SSG *[12]* de manera nativa, mejorando la optimización para motores de búsqueda y rendimiento.

- **Desventajas:**
  - Ecosistema todavía más pequeño que el de React o Angular, aunque abundan las librerías, su volumen es menor.
  - Al tener menos adopción por las grandes empresas hay un menor soporte en los foros especializados.
  - Hay polémicas en las transiciones de las versiones.

2.1.3.1.4. Svelte (y SvelteKit)

Figura 8: Logo de Svelte

- **Descripción:**

  Svelte en lugar de implementar un virtual DOM *[13]* en tiempo de ejecución, es un compilador genera código de JavaScript ligero y optimizado a partir de nuestro código fuente. SvelteKit agrega SSR *[12]*, enrutado y otras funcionalidades avanzadas.

- **Ventajas:**

- Reduce el tamaño de los bundles y el tiempo de carga, útil para aplicaciones que requieren velocidades de respuesta muy rápidas.
- La reactividad está integrada en el lenguaje de plantillas, y se maneja de forma muy natural.

- **Desventajas:**
  - Comunidad menos extensa, con menos librerías y soluciones a problemas específicos en producción que React o Angular.
  - El ecosistema y la documentación no son tan maduros, ya que Sveltekit todavía introduce cambios de forma frecuente, lo que puede requerir actualizaciones constantes.
  - Las empresas todavía confían en React o Angular, por lo que no siempre hay suficiente documentación sobre casos de uso complejos.

2.1.3.1.5. Gatsby

Figura 9: Logo de Gatsby

- **Descripción:**

  Es un framework basado en React, que se especializa en la generación de sitios web estáticos (Static Site Generation, SSG) *[12]*, además de utilizar GraphQL y Webpack para construir los sitios web. Obteniendo los datos de diferentes fuentes y pre-construye las páginas para un rendimiento más rápido.

- **Ventajas:**
    - La pre-renderización de contenido estático produce tiempos de carga muy bajos.
    - Sistema de plugins para obtener datos de CMS, bases de datos o ficheros Markdown.
    - El modelo de SSG encaja muy bien con sitios informativos o que se actualizan con menor frecuencia, es decir, excelentes para blogs y páginas de contenidos.

- **Desventajas:**
    - Enfoque muy orientado a sitios estáticos o blogs; menos flexible para aplicaciones con mucha interacción en tiempo real, ya que se vuelve poco práctico.
    - Cada vez que se añade o modifica contenido, es necesario reconstruir las páginas, esto puede ser un inconveniente si hay muchos cambios diarios.

2.1.3.2. Back-end

Por otro lado, el back-end *[14]* se encarga de la conexión con la base de datos y el servidor utilizado por la página web. Esta parte está situada en el lado del servidor. Veamos algunas alternativas:

2.1.3.2.1. Django

Figura 10: Logo de Django

- **Descripción:**

  Es un framework de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño conocido como modelo-vista-controlador y cubre todo el ciclo de desarrollo de una aplicación web. Integra un sistema de autenticación, un panel de administración, plantillas, gestión de rutas, entre otros componentes.

- **Ventajas:**
  - Tiene una productividad inicial alta ya que gracias al 'admin papel' y el ORM facilitan la gestión de CRUDos de forma rápida.
  - Comunidad amplia y estable, con abundante documentación y "plugins" (Django REST Framework, etc.).
  - Un enfoque monolítico con una estructura clara y bien definida, facilitando proyectos medianos y grandes.

- **Desventajas:**
  - Al ser bastante "monolítico", puede ser más pesado para proyectos que requieren solamente una API ligera.
  - Escalabilidad compleja si se buscan proyectos que busquen un servicio minimalista y una arquitectura de microservicios puede ser más pesada de lo necesario.

2.1.3.2.2. Flask

Figura 11: Logo de Flask

- **Descripción:**

  Es un micro-framework minimalista de Python, enfocado en la sencillez y en la flexibilidad. Parte de una base mínima como enrutamiento y servidor WSGI, y permite añadir extensiones según las necesidades como autenticación, gestión de base de datos.

- **Ventajas:**
  - Te permite elegir con libertad cada componente (ORM, autenticación, etc.).
  - Fácil de comprender y arrancar, apropiado para APIs sencillas o proyectos de tamaño mediano.
  - Al tener tiempo en el ecosistema Python, existen múltiples extensiones y documentación para casi cualquier caso de uso.

- **Desventajas:**
  - Si el proyecto crece, debes ir añadiendo manualmente librerías, lo que puede derivar en una estructura de directorios compleja. Aunque es posible organizarlo en Blueprints, no hay un modelo definido de microservicios, lo que puede requerir configuración adicional.
  - No tiene por defecto un sistema de autenticación u ORM propios.

2.1.3.2.3. Express.js

Figura 12: Logo de Express

- **Descripción:**

  Framework ligero para Node.js de código abierto y con licencia MIT, muy popular para construir APIs REST y aplicaciones web.

- **Ventajas:**
  - Gran comunidad, cientos de paquetes y middleware para cubrir cualquier función como autenticación, compresión, logging, etc.
  - Permite aplicaciones altamente concurrentes y escalables.
  - Gran flexibilidad, ya que se configura casi todo mediante middleware, permitiendo adaptar la aplicación a diferentes casos de uso.

- **Desventajas:**
  - Necesita librerías adicionales para ORM, validación, seguridad, etc.
  - JavaScript en el back-end puede requerir un enfoque asíncrono que no todos los desarrolladores dominan con fluidez aunque con TypeScript es una ayuda.
  - Tiene una estructura libre similar a Flask, se requiere mayor organización cuando el proyecto crece.

2.1.3.2.4. NestJS

Figura 13: Logo de NestJS

- **Descripción:**

Framework para Node.js basado en TypeScript para el desarrollo progresivo de aplicaciones web, publicado como software gratuito y de código abierto bajo una licencia MIT. Adopta una arquitectura inspirada en Angular (módulos, controladores, servicios), pero orientada al lado del servidor.

- **Ventajas:**
  - Tiene una arquitectura organizada que facilita la escalabilidad y la modularidad.
  - Soporte nativo de TypeScript que introduce robustez y prevención de errores comunes.
  - Integraciones potentes como ORM, websockets, GraphQL, testing, etc.

- **Desventajas:**
  - Menor difusión que Express aunque su adopción crece rápido, la mayoría de ejemplos y tutoriales para Node.js se basan en Express.
  - La curva de aprendizaje exige familiarizarse con ciertos patrones similares a Angular, lo que puede ser más complejo si no se conoce este framework.

2.1.4. Tecnologías utilizadas

A continuación se describen las principales tecnologías utilizadas en el proyecto, tanto en el ámbito del desarrollo front-end como en el del back-end, así como el entorno de desarrollo y el sistema de control de versiones elegido.

2.1.4.1. Tecnologías front-end: Next.js

Figura 14: Logo de Next.js

Next.js *[15]* es un framework JavaScript ligero y de código abierto creado sobre React que ofrece la posibilidad de realizar Server-Side Rendering (SSR) y Static Site Generation (SSG) *[12]* de forma nativa, que permite desarrollar aplicaciones web y sitios web muy rápidos y fáciles de usar. Esto permite:

- Mejorar la optimización para motores de búsqueda al pre-renderizar las páginas en el servidor antes de enviarlas al navegador.
- Optimizar el rendimiento gracias a la generación estática de secciones que no cambian con frecuencia.
- Facilitar el enrutado a través de un sistema file-based routing, en el que las rutas se definen con la estructura de archivos dentro de pages/.
- Simplificar el despliegue mediante servicios como Vercel o en cualquier servidor Node.js, dado que Next.js corre sobre un entorno de Node.

En un contexto sanitario, en el que se maneja información sensible y posiblemente se desea ofrecer carga rápida y acceso ágil a los datos, Next.js resulta muy apropiado. Su integración con React permite el uso de un ecosistema de

librerías y componentes existentes, mientras que las funciones de SSR/SSG *[12]* posibilitan una mejor experiencia de usuario con menor tiempo de carga inicial.

2.1.4.2. Tecnologías back-end: FastAPI

Figura 15: Logo de FastAPI

Para la capa de back-end se ha seleccionado FastAPI *[16]*, un framework de Python diseñado específicamente para la construcción de APIs (REST, WebSockets, etc.). Sus características principales incluyen:

- Modelo asíncrono (ASGI): Proporciona alto rendimiento y escalabilidad, ideal para escenarios con numerosas peticiones concurrentes.
- Tipado de datos y validación automática: Se aprovecha de las anotaciones de tipos de Python para generar validaciones y documentación de forma muy sencilla.
- Documentación automática: Integra de manera nativa Swagger UI *[18]* y ReDoc, permitiendo a los desarrolladores y usuarios finales explorar los endpoints y probarlos de forma interactiva.
- Fácil integración con librerías de terceros: Para aspectos como ORM, autenticación, seguridad y migraciones, se combina sin problemas con SQLAlchemy *[17]*, OAuth, entre otros.

En un sistema sanitario, donde se requiere seguridad, rapidez de respuesta y actualización constante de los datos, FastAPI destaca por su velocidad, su sencillez

en la escritura de código y la claridad que aporta a la hora de mantener la lógica del lado servidor.

2.1.4.3. ORM con SQLAlchemy

Figura 16: Logo de SQLAlchemy

Para la gestión de la base de datos y la persistencia de información, se ha hecho uso de SQLAlchemy *[17]*, un Object-Relational Mapper (ORM) para Python. Su rol dentro del proyecto es esencial, ya que facilita la interacción con la base de datos relacional, permitiendo:

- Abstracción del SQL: Se definen modelos en Python en lugar de escribir manualmente las consultas SQL, haciendo el código más limpio y mantenible.
- Manipulación de datos mediante sesiones y métodos de consulta (filter, join, etc.), lo que simplifica operaciones de lectura y escritura.
- Manejo de relaciones entre tablas (One-to-Many, Many-to-Many, etc.) a través de definiciones de modelos, estableciendo de forma clara cómo se vinculan los datos.
- Amplitud de soporte para múltiples motores de base de datos, como PostgreSQL, MySQL, SQLite o MariaDB, dando versatilidad en la elección del servidor de producción.

Gracias a SQLAlchemy, los desarrolladores pueden concentrarse en la lógica de la aplicación en vez de lidiar con sentencias SQL complejas, a la vez que se obtiene una capa de abstracción que facilita la portabilidad entre distintos SGBD (Sistemas Gestores de Bases de Datos).

2.1.4.4. Entorno de desarrollo

Figura 17: Logo de Visual Studio Code

El entorno de desarrollo principal para este proyecto ha sido Visual Studio Code *[19]* (VSCode). Entre los motivos de su elección destacan:

- Multiplataforma: Compatible con Windows, macOS y Linux, lo que facilita la colaboración.
- Extensa galería de extensiones: Para Python, JavaScript/TypeScript, Docker, Git, entre otros. Estas extensiones aportan funcionalidades como autocompletado, depuración, resaltado de sintaxis y formateo de código.
- Terminal integrada: Permite ejecutar comandos sin salir del editor, optimizando el flujo de trabajo para lanzamientos locales o pruebas unitarias.
- Soporte para Git: Incluye herramientas para hacer commits, push, creación de ramas y revisiones de cambios dentro de la misma interfaz.

Esta flexibilidad y potencia convierten a VSCode en una solución óptima para un proyecto en el que se trabaja tanto con Python (FastAPI) como con JavaScript (Next.js).

2.1.4.5. Control de versiones

Figura 18: Logo de Gitlab

El control de versiones se ha gestionado mediante Git utilizando la plataforma GitLab. Sus características principales, relevantes para el proyecto, incluyen:

- Repositorios privados: Permiten proteger el código y limitar el acceso a colaboradores autorizados, esencial cuando se trabaja con información sensible o confidencial.
- Gestión de incidencias (Issues): Facilita la organización de tareas y el seguimiento de errores o mejoras.
- Integración Continua (CI/CD): Ofrece la posibilidad de configurar pipelines de pruebas y despliegue automatizado, agilizando el desarrollo y fomentando buenas prácticas de calidad.
- Merge Requests: Herramienta para revisar y discutir los cambios antes de unirlos a la rama principal (main o master), fomentando un flujo de trabajo colaborativo y manteniendo la calidad del código.

Gracias a estas funcionalidades, GitLab aporta un entorno robusto para la colaboración en equipo y la trazabilidad de cada versión del proyecto, lo cual es especialmente útil en el contexto de un Trabajo Fin de Grado, donde se valora la organización, la documentación y la transparencia en la evolución del desarrollo.

2.2. Problemas a abordar

El principal reto que busca resolver este proyecto está en la gestión y el seguimiento de la información clínica de pacientes con diabetes tipo I, así como en la comunicación fluida entre los distintos actores involucrados (pacientes, familiares y médicos). Por tanto es necesario implementar un sistema que:

- Centralice los datos como mediciones de glucosas, notas diarias, prescripciones, analíticas ya que normalmente se suelen encontrar en diferentes formatos como papel, aplicaciones sin conexión, hojas de cálculo. Esta dispersión dificulta el acceso rápido y ordenado a los datos, lo que ralentiza los diagnósticos y la capacidad de reacción ante cambios importantes en la salud del paciente.

- Resuelve la limitada comunicación entre actores, porque aunque existen canales tradicionales como las visitas presenciales, llamadas telefónicas, correo electrónico, la falta de un medio centralizado complica la interacción médico-paciente. Esta falta de inmediatez puede impedir un ajuste a tiempo de la medicación o un cambio en las recomendaciones de ejercicio o alimentación.

- Personalizar la experiencia en función del rol del usuario, ya que cada uno requiere funcionalidades específicas y diferentes niveles de acceso a la información. Un sistema que no distinga correctamente estos roles puede provocar brechas de seguridad o frustraciones en la usabilidad por ejemplo un familiar accediendo a datos sensibles que el paciente no ha autorizado.

- Garantizar la privacidad y la protección de datos sanitarios ya que cualquier solución que no implemente controles de acceso adecuados o cifrado de datos podría exponer a los usuarios a vulneraciones de seguridad.

- Registros históricos que permitan comprender la evolución del paciente (picos de glucemia, frecuencia de episodios hipoglucémicos, cambios en medicación,...) ya que sin un sistema centralizado, se vuelve complicado

analizar los datos, obtener estadísticas y ofrecer recomendaciones personalizadas.

- Garantizar la vinculación de los familiares. En muchos casos los menores o personas dependientes requieren un rol familiar que acceda a sus datos y puede resultar complejo gestionar quién puede ver cierta información y durante cuánto tiempo, sin contar con un mecanismo claro para otorgar o revocar permisos.

- Accesibilidad multiplataforma para acceder desde diferentes dispositivos, con un diseño responsive y fácil de usar se vuelve imprescindible para que tanto el médico como el paciente y familiar puedan consultar o introducir datos en cualquier momento y lugar.

2.3. Metodología

En este apartado se describe el enfoque metodológico que se ha considerado para el desarrollo del proyecto. Dado que se trata de un Trabajo Fin de Grado realizado esencialmente por un único desarrollador, conviene revisar brevemente las principales alternativas antes de decantarse por la más adecuada.

2.3.1. Principales metodologías

A lo largo de la evolución del desarrollo de software, han surgido diversas metodologías que ayudan a planificar y organizar el trabajo de forma más eficaz. A continuación se describen algunas de las destacadas, poniendo el énfasis en sus características, ventajas, desventajas y en qué contextos suelen aplicarse.

2.3.1.1. Metodología en cascada (Waterfall)

El modelo en cascada, también conocido como Waterfall *[20]*, es un enfoque secuencial en el que cada fase (análisis, diseño, implementación, pruebas y mantenimiento) se completa antes de iniciar la siguiente. Se basa en la idea de que, una vez definido el alcance y los requisitos, el proyecto avanza de forma lineal hasta su entrega.

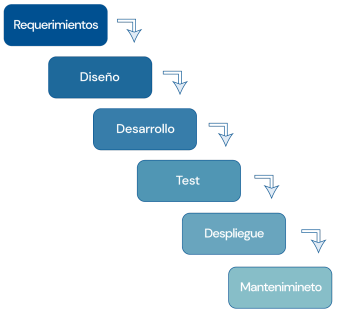


Figura 19: Representación metodología en cascada

- **Ventajas**
  - Claridad en la planificación: Al estar dividido en fases concretas, se facilita la elaboración de cronogramas y la estimación de recursos.
  - Adecuado para proyectos con requisitos estables: Cuando los requisitos no se esperan cambiar de forma significativa, resulta sencillo documentar cada paso.
  - Facilita la documentación: La separación por fases fomenta la elaboración de documentos formales y detallados en cada etapa.

- **Desventajas**
  - Escasa flexibilidad: Si surgen cambios en los requisitos una vez avanzada la fase de diseño o implementación, es costoso retroceder.
  - Riesgo de descubrir errores tardíos: La validación con el cliente suele ocurrir en etapas finales, lo que puede derivar en grandes ajustes de última hora.
  - Limitada retroalimentación continua: No está pensada para revisiones frecuentes con el cliente o usuarios.

- **Contexto**
  - En proyectos con requisitos muy estables y bien definidos desde el principio.
  - Cuando se dispone de un equipo que sigue procesos muy formales y se prioriza una documentación exhaustiva.

2.3.1.2. Scrum

Scrum [21] es una metodología ágil centrada en entregas iterativas llamadas sprints, que suelen durar de dos a cuatro semanas. En cada sprint, se trabaja en un conjunto de funcionalidades priorizadas (Product Backlog), y al concluir, se presenta un incremento de software potencialmente utilizable.

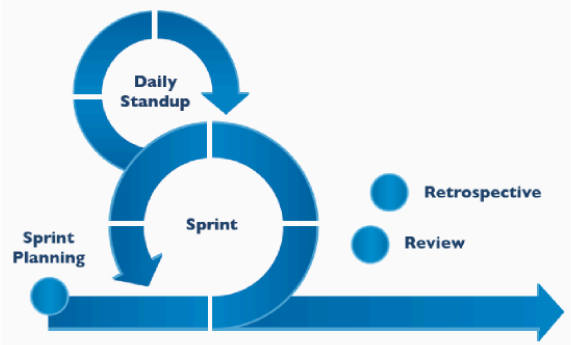


Figura 20: Representación metodología Scrum

- **Ventajas**
  - Retroalimentación rápida: Tras cada sprint, el equipo obtiene comentarios del cliente o stakeholders, lo que permite hacer ajustes en fases tempranas.
  - Transparencia y adaptación: Reuniones diarias (daily stand-up) y revisiones de sprint fomentan la comunicación y la corrección de desvíos.
  - Enfoque en la colaboración: Roles definidos (Scrum Master, Product Owner, Equipo de Desarrollo) que garantizan la eficiencia de la comunicación.

- **Desventajas**
  - Dependencia de un equipo comprometido: Scrum funciona bien si todos los miembros participan activamente y si hay un Product Owner disponible.

- Puede complicarse en proyectos grandes: Aunque existe Scrum a gran escala, en equipos muy numerosos se requiere una coordinación más compleja.
- No está diseñado para desarrolladores en solitario: Su esencia colaborativa y basada en roles hace que sea menos fluido si trabajas individualmente.

- **Contexto**
  - En equipos de tamaño reducido o mediano (5-9 personas) que trabajan en entregas frecuentes.
  - Cuando los requisitos evolucionan con regularidad y se busca adaptarse a los cambios rápidamente.

2.3.1.3. Kanban

Kanban [22] es otra metodología ágil que se basa en la visualización del flujo de trabajo mediante un tablero con columnas (típicamente “Por Hacer”, “En Proceso”, “Realizado”). Se asigna una tarjeta a cada tarea, y esta va moviéndose de columna a columna conforme se progresa en su realización. No se fija la duración de iteraciones (sprints), sino que se promueve un flujo continuo.

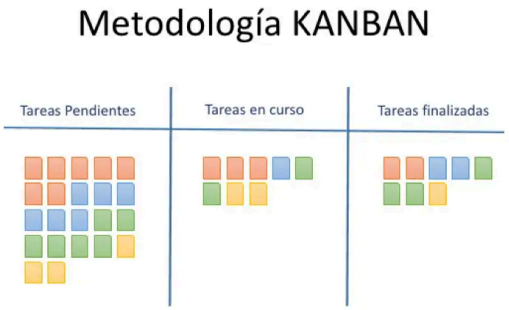


Figura 21: Representación metodología Kanban

- **Ventajas**
  - Flexibilidad ante cambios: No obliga a completar un sprint cerrado antes de incorporar nuevas tareas o requisitos.

- Simplicidad en la gestión: Aporta claridad al ver de un vistazo qué tareas están pendientes, en curso o finalizadas.
- Control del Work In Progress (WIP): Permite limitar las tareas en curso para evitar la sobrecarga de trabajo y los cuellos de botella.

- **Desventajas**
  - Menos estructura formal: Algunas personas o equipos echan en falta la cadencia de las iteraciones o roles más definidos.
  - Riesgo de multitarea ineficiente: Si no se controlan las WIP, el tablero puede llenarse y no se consigue el foco necesario para terminar tareas con prontitud.
  - Seguimiento de plazos más difusos: Al no haber sprints delimitados, se necesita otro método para estimar o planificar fechas de entrega con precisión.

- **Contexto**
  - En proyectos unipersonales o de tamaño reducido, donde la comunicación es sencilla.
  - Cuando la prioridad es la flexibilidad y la adaptación constante a nuevas tareas.
  - Para flujos de trabajo continuos que no encajan bien con la idea de sprints fijos.

2.3.1.4. Extreme Programming (XP)

Extreme Programming (XP) [23] es una metodología ágil que hace hincapié en la calidad del código y la comunicación constante con el cliente. Se caracteriza por prácticas como la programación en pareja (pair programming), la integración continua y la refactorización constante.

Figura 22: Representación metodología XP

- **Ventajas**
  - Alto foco en la calidad del software: La programación en pareja y las pruebas continuas suelen derivar en un código más limpio y testeado.
  - Menor riesgo de errores: La integración y las pruebas frecuentes permiten detectar problemas de forma temprana.
  - Retroalimentación fluida con el cliente: Ideal para proyectos donde los requisitos varían con frecuencia y el producto evoluciona de forma incremental.

- **Desventajas**
  - Requiere alto grado de disciplina: Sin un equipo bien entrenado en sus prácticas, se pierde la esencia de XP.
  - Difícil de implementar para desarrolladores en solitario: Una de las bases de XP es la programación en pareja, que no tendría sentido si solo hay un programador disponible.
  - Curva de aprendizaje: Exige cambios culturales y de hábitos, como la escritura constante de test y el pair programming.

- **Contexto**
  - En entornos con alta variabilidad en los requisitos y necesidad de entregar valor de forma temprana y frecuente.
  - Cuando se dispone de un equipo capaz de adoptar prácticas de desarrollo colaborativo y estricto control de calidad.

2.3.2. Elección de la metodología

Dado que la realización de este proyecto recae principalmente en una sola persona, se ha optado por Kanban. Su flexibilidad y su enfoque en la visualización de tareas lo convierten en una alternativa ágil idónea, que facilita la gestión del flujo de trabajo sin necesidad de estructuras de equipo complejas.

El proceso con Kanban se llevará a cabo de la siguiente forma:

1. Análisis de requisitos
    - Recopilación de información acerca de las necesidades funcionales y no funcionales.
    - Definición del alcance inicial del prototipo y priorización de las funcionalidades más importantes.

2. Creación del tablero Kanban
    - División de las tareas en columnas: “Por Hacer”, “En Proceso” y “Realizado”.
    - Elaboración de tarjetas para cada tarea identificada, describiendo su objetivo y requisitos.
    - Asignación de prioridades según la criticidad o complejidad de cada tarea.

3. Diseño del sistema
    - Creación de diagramas de arquitectura y de datos para definir la estructura de la aplicación.
    - Selección final de patrones de diseño y tecnologías que se ajusten mejor al proyecto.

4. Implementación
    - Desarrollo incremental de funcionalidades, moviendo las tarjetas en el tablero Kanban conforme se avanza.

- Integración y pruebas parciales para detectar y corregir errores de forma temprana.

5. Documentación y resultados
    - Redacción de la memoria del TFG, detallando el proceso, las decisiones adoptadas y los resultados obtenidos.
    - Presentación de conclusiones y posibles líneas de mejora para el futuro.

La sencillez de Kanban permite adaptarse a los cambios y a las nuevas prioridades que surjan durante el proyecto, favoreciendo una buena gestión del tiempo y la realización de entregas parciales de valor.

2.4. Extracción de requisitos

En este apartado se describirán los distintos requisitos que se han considerado para el desarrollo del proyecto.

2.4.1. Requisitos funcionales

- Gestión de usuarios
    - Registro de usuarios: Cada usuario se registrará en la plataforma, proporcionando sus datos, además de elegir el rol correspondiente (paciente o familiar).
    - Registro de médicos: El administrador creará una cuenta para el médico.
    - Los usuarios tendrán una página de configuración
    - En la configuración el usuario podrá cambiar su información personal.
    - En la configuración el paciente podrá desvincular los familiares vinculados al paciente.
    - En la configuración el familiar podrá desvincularse de los pacientes que el familiar es uno de ellos.
    - En la configuración el paciente podrá cambiar el médico asignado, enviando una petición de cambio de médico al administrador.

- El administrador podrá aceptar o denegar la petición de cambio de médico.
- En la configuración el usuario podrá actualizar su rol para tener varios, principalmente iniciaran su cuenta con la elección seleccionada.
  - En caso de ser médico podrá seleccionar los roles del paciente y/o familiar.
  - En caso de ser paciente únicamente se podrá seleccionar el rol familiar.
  - En caso de ser familiar únicamente se podrá seleccionar el rol del paciente.
- En la configuración el usuario podrá cambiar de contraseña.
- Inicio de sesión: El sistema debe proporcionar un inicio de sesión seguro para los usuarios.
- Recuperación de contraseñas: Los usuarios deben poder recuperar sus contraseñas en caso de que se olviden, mediante el correo electrónico
- Los pacientes tendrán un apartado de mensajes en los que los familiares podrán enviarle solicitudes de familiares.
- Los pacientes podrán aceptar o rechazar las solicitudes de familiares.

- Gestión de datos sanitarios
  - Del rol paciente
    - Los pacientes deben poder ingresar una nota por día en la cual tendrá una descripción por campo indicado como por ejemplo campo más como alimentos consumidos, actividad física, etc
    - El sistema debe permitir el almacenamiento y la visualización del historial de la glucosa para el seguimiento del paciente y con intervalos para gráficas.
    - El usuario podrá visualizar las prescripciones médicas, si las tiene
    - El paciente podrá modificar las notas introducidas
    - El paciente podrá eliminar las notas introducidas
    - Los datos de glucosa no se podrán eliminar

        - El paciente tendrá una página en la que aparecerán los diagnósticos que los médicos le hayan recetado.
        - El paciente tendrá una página con los datos de las analíticas realizadas.
    - Del rol médico
        - El médico podrá visualizar los datos del paciente seleccionado en su lista de pacientes
        - El médico podrá agregar una prescripción médica
        - El médico podrá modificar una prescripción médica
        - La prescripción médica incluirá, datos básicos del médico, fecha de realización, tratamiento y descripción de la observación
    - Del rol familiar
        - El familiar podrá ver la información del paciente
    - Del rol administrador
        - El administrador podrá crear usuarios con los roles que desee
        - El administrador podrá aceptar o no las solicitudes de cambio de médico de los pacientes.

- Búsqueda y vinculación entre usuarios
    - Vinculación Médico-Paciente:
        - Cuando el paciente se registra se le asigna un médico disponible
        - Una vez añadido a la lista el médico tendrá acceso completo a la información médica del paciente
        - El paciente podrá ver los datos básicos de su médico asignado
        - El paciente podrá pedir un cambio de médico en el cual podrá seleccionar a través de una lista los médicos disponibles,
        - El administrador acepta o deniega las peticiones de cambio de médico.
    - Vinculación familiar-paciente:
        - Los familiares deberán enviar una solicitud de vinculación al paciente.
        - El paciente recibirá un mensaje con el cual podrá aceptar o rechazar.

- El paciente podrá ver un listado de los familiares a los que les ha dado acceso.
- El paciente podrá administrar los familiares que tienen acceso pudiendo revocarle el acceso a su historial si es mayor de edad

2.4.2. Requisitos no funcionales

- Las funcionalidades de la web deben estar limitadas según el rol del usuario para proteger los datos y la privacidad del paciente
- Disponibilidad en cualquier momento, desde cualquier dispositivo para todos los usuarios.
- Interfaz intuitiva y fácil de navegar para que los usuarios puedan utilizarla sin dificultad, incluso si tienen poca experiencia en tecnología
- Sistema de pruebas unitarias para verificar que cada funcionalidad esté realizada correctamente.

2.5. Historias de usuario

Las historias de usuario son breves descripciones de funcionalidades vistas desde la perspectiva del usuario final. En la práctica, se redactan en un lenguaje sencillo y siguen la fórmula:

***Como** (tipo de usuario), **quiero** (acción) **para** (beneficio/objetivo).*

Estas historias se vinculan con la metodología elegida, ya que ayudan a identificar y organizar las tareas en el tablero según la prioridad y el impacto en el proyecto. A continuación, se listan los requisitos funcionales adaptados:

- Historias de Usuario Genéricas (Registro, Sesión, Configuración)
  - Registro de usuario
    - Como nuevo usuario, quiero registrarme en la plataforma para poder acceder y utilizar las funcionalidades según mi rol (paciente o familiar).
  - Creación de médico (administrador)

        - Como administrador, quiero poder crear la cuenta de un médico para otorgarle acceso a sus funcionalidades específicas sin que él tenga que registrarse de forma convencional.
    - Acceso seguro
        - Como usuario, quiero iniciar sesión en la plataforma de forma segura para proteger mis datos personales y sanitarios.
    - Recuperación de contraseña
        - Como usuario, quiero poder recuperar mi contraseña mediante correo electrónico para restablecer el acceso en caso de olvido.
    - Página de configuración
        - Como usuario (de cualquier rol), quiero acceder a una página de configuración para actualizar mi información personal (nombre, apellidos, email, etc.) y gestionar las opciones de mi cuenta.
    - Cambio de contraseña
        - Como usuario, quiero poder cambiar mi contraseña desde la sección de configuración para mantener la seguridad de mi cuenta.
    - Actualización de rol
        - Como usuario, quiero poder añadir un nuevo rol en mi cuenta para ampliar mis funciones en la plataforma.
        - Como médico, quiero poder seleccionar el rol de paciente y/o familiar si así lo necesito para acceder a más funcionalidades de la plataforma.
        - Como paciente, quiero poder añadir el rol familiar para gestionar a otros pacientes que necesiten mi ayuda.
        - Como familiar, quiero poder añadir el rol paciente para incorporar mi propio control de datos de salud si fuese necesario.

- Historias de Usuario para Paciente
    - Notas diarias

        - Como paciente, quiero registrar una nota al día (comida, ejercicio, etc.) para llevar un seguimiento detallado de mis hábitos y evolución.
    - Historial de glucosa
        - Como paciente, quiero poder ver mi historial de glucosa y graficarlo por intervalos para analizar mi progreso y detectar patrones de control.
    - Visualización de prescripciones
        - Como paciente, quiero consultar las prescripciones médicas que me han asignado para conocer mis tratamientos y recomendaciones.
    - Modificación de notas
        - Como paciente, quiero poder editar las notas diarias que haya registrado para corregir errores o añadir información adicional.
    - Eliminación de notas
        - Como paciente, quiero poder borrar las notas que ya no considere necesarias para mantener mi historial organizado.
    - Visualización de diagnósticos
        - Como paciente, quiero contar con una sección donde ver los diagnósticos que el médico me haya recetado para tener claras mis condiciones o resultados clínicos.
    - Visualización de analíticas
        - Como paciente, quiero disponer de una página con los datos de mis analíticas para revisar parámetros médicos adicionales.
    - Cambio de médico
        - Como paciente, quiero solicitar un cambio de médico desde mi configuración para poder elegir a otro profesional en caso de no sentirme conforme o necesitar una segunda opinión.
    - Desvinculación de familiares
        - Como paciente, quiero poder desvincular a familiares de mi cuenta para mantener el control de quién ve mis datos sanitarios.
    - Recepción y gestión de solicitudes de familiares

        - Como paciente, quiero recibir solicitudes de vinculación de familiares en mi apartado de mensajes para decidir si acepto o rechazo su acceso a mi información.

- Historias de Usuario para Familiar
    - Vinculación con paciente
        - Como familiar, quiero enviar una solicitud de vinculación a un paciente para poder ver su información médica y ayudar en su cuidado.
    - Desvinculación de paciente
        - Como familiar, quiero desvincularme de un paciente desde mi configuración para dejar de acceder a sus datos si mi rol ya no es necesario.
    - Consulta de información del paciente
        - Como familiar, quiero visualizar la información sanitaria básica del paciente al que estoy vinculado para conocer su estado y ayudarlo en caso de necesidad.

- Historias de Usuario para Médico
    - Visualización de pacientes
        - Como médico, quiero ver la lista de pacientes que me han sido asignados para acceder de forma rápida a sus historiales clínicos.
    - Acceso completo a datos del paciente
        - Como médico, quiero consultar la información médica de cada paciente asignado para llevar un seguimiento óptimo de su estado de salud.
    - Gestión de prescripciones
        - Como médico, quiero poder crear nuevas prescripciones para mis pacientes para indicarles tratamientos o pautas concretas.
        - Como médico, quiero editar una prescripción existente para corregir o actualizar los detalles del tratamiento.

- Historias de Usuario para Administrador

    - Creación de usuarios
        - Como administrador, quiero poder crear usuarios que desempeñen los roles correspondientes.
    - Gestión de peticiones de cambio de médico
        - Como administrador, quiero ver las solicitudes de cambio de médico para aprobarlas o denegarlas según la disponibilidad y las normas de la plataforma.

- Historias de Usuario sobre Búsqueda y Vinculación
    - Confirmación o rechazo de cambio de médico
        - Como administrador, quiero aceptar o denegar las solicitudes de los pacientes para cambiar de médico para mantener el correcto reparto de la carga de trabajo entre los profesionales.
    - Envío de solicitud de familiar a paciente
        - Como familiar, quiero buscar a un paciente e iniciar la solicitud de vinculación para acceder a su información sanitaria con su permiso.
    - Aceptación o rechazo de familiar
        - Como paciente, quiero confirmar o rechazar la solicitud de un familiar para controlar quién tiene acceso a mi historial médico.

2.6. Modelo de dominio

A continuación, se presenta una definición conceptual de cada entidad del modelo de dominio [24] que subyace en la aplicación para la gestión y el seguimiento de pacientes con diabetes tipo I. Posteriormente, se anexará un diagrama que representa de manera visual este modelo de dominio.

Comenzaremos con la definición de las entidades:

- Usuario: Representa a cualquier persona que accede a la plataforma, ya sea un paciente, un médico, un familiar o incluso un administrador. Es la pieza central que da identidad a quienes interactúan con la aplicación, permitiendo

gestionar aspectos como la autenticación, la configuración de la cuenta y el control de permisos basados en su rol.

- Rol: Define la función que un usuario cumple en el sistema (por ejemplo, “paciente”, “médico”, “familiar”, “administrador”). Cada rol conlleva ciertos privilegios y acciones específicas que determinan lo que el usuario puede o no hacer dentro de la plataforma.

- Médico: Se refiere a la capacidad profesional de un usuario que ejerce la práctica médica dentro de la aplicación. A diferencia de un simple registro personal, esta entidad extiende el rol de médico, otorgándole acceso a gestiones clínicas, como la elaboración de diagnósticos y la supervisión de pacientes.

- Paciente: Corresponde a la identidad clínica de un usuario que requiere seguimiento y control en el marco de la diabetes tipo I. Como paciente, se centra en el registro de mediciones de salud, la consulta de diagnósticos, la creación de notas sobre su día a día.

- Nota: Refleja un registro periódico donde el paciente apunta información relevante sobre su salud o actividades diarias (p. ej., alimentación, ejercicio, cambios en la rutina). Estas notas ayudan a realizar un seguimiento más completo y personalizado de la evolución del paciente.

- NotaDetalle: Amplía la información de una nota, abordando puntos concretos y clasificándolos según tipos (por ejemplo, “comida”, “deporte”). Sirve para desglosar el contenido de la nota principal, proporcionando detalles granulares de las actividades o eventos que el paciente registra.

- Diagnóstico: Representa la valoración médica que se emite respecto al estado de un paciente en un momento dado (tanto en lo referido a la enfermedad principal como a otras condiciones clínicas). Concentra la opinión profesional del médico, ofreciendo una referencia importante para el tratamiento y el seguimiento.

- Glucosa: Registra las mediciones de glucemia de un paciente a lo largo del tiempo. Esta información es esencial para quienes padecen diabetes tipo I, ya que posibilita el control preciso de los niveles de azúcar en sangre y la toma de decisiones clínicas fundamentadas.

- Bioquímico: Almacena las pruebas de laboratorio o analíticas que el paciente se ha realizado, abarcando resultados más allá de la glucemia (por ejemplo, perfiles lipídicos, hemoglobina glicosilada, etc.). Brinda una visión global del estado de salud del paciente y de su evolución en análisis clínicos.

- Vínculo: Refleja la relación de parentesco o conexión familiar entre dos usuarios, posibilitando que un familiar tenga acceso a cierta información del paciente cuando se requiere supervisión o cuidado. Es clave para gestionar permisos en situaciones en que el paciente es menor de edad o necesita apoyo familiar constante.

- SolicitudesCambio: Agrupa las peticiones formales de un paciente para modificar el médico que lo atiende. Contiene además el estado de esas solicitudes (pendiente, completado, rechazado), permitiendo que la administración del sistema y los médicos gestionen estas solicitudes de manera ordenada y transparente.

- Mensaje: Representa la comunicación interna dentro de la plataforma. Incluye notificaciones, solicitudes (por ejemplo, de vinculación familiar) o simplemente mensajería directa entre usuarios. Permite intercambiar información de forma segura y registrar la fecha y estado de los mensajes, contribuyendo a una comunicación fluida en el entorno clínico.

A continuación, mostraremos el diagrama de modelo de dominio que muestra la representación visual de las entidades principales de nuestro dominio, y las relaciones entre ellas.

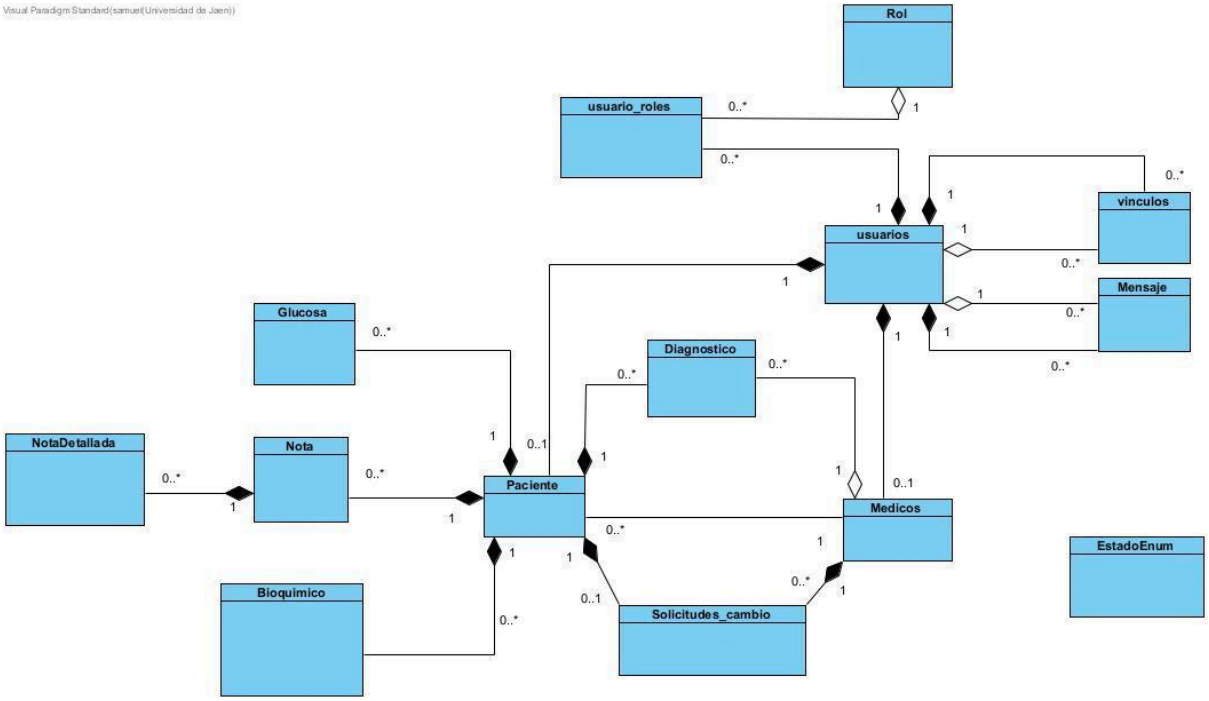


Figura 23: Modelo de dominio

- **Usuario - Médico**
  - Naturaleza: Relación uno a uno (1–1).
  - Descripción: Un objeto de la entidad Médico no puede existir sin su correspondiente Usuario (hereda atributos como nombre, email, etc.). Por ello, podemos asimilar esta relación a una composición: si se elimina el Usuario, el médico carece de sentido. Asimismo, un Médico pertenece a un único Usuario, pues cada médico es una persona concreta en la plataforma.

- **Usuario - Paciente**
  - Naturaleza: Relación uno a uno (1–1).
  - Descripción: De forma análoga a Medico, la entidad Paciente depende de Usuario. Un paciente no existe en el sistema si antes no existe el usuario base. La eliminación del Usuario (por ejemplo, en caso de darse de baja de la plataforma) conlleva la desaparición de su registro como Paciente.

- **Usuario - Rol (usuario_roles)**

- Naturaleza: Relación de muchos a muchos (M–N).
- Descripción: Un Usuario puede poseer uno o varios roles (por ejemplo, “paciente” y “familiar”), mientras que un Rol puede ser asignado a diferentes usuarios. Esta relación se implementa a través de una tabla intermedia (usuario_roles). Es una composición entre usuarios y usuario_roles ya que si desaparece un usuario se borrarán los registros de la tabla usuario_roles, en cambio los roles siguen existiendo.

- **Usuario - Usuario (vínculo)**
  - Naturaleza: Relación de muchos a muchos (N-M).
  - Descripción: Un vínculo representa, por ejemplo, la relación de parentesco o conexión familiar entre dos usuarios. Cada vínculo hace referencia a un par de usuarios. En usuario con vínculo es composición ya que si alguno de los usuarios se elimina, dicho vínculo deja de tener sentido. En la otra parte de la relación se considera una asociación dado que ambos usuarios pueden existir por separado sin que exista el vínculo.

- **Usuario - Usuario (Mensaje)**
  - Naturaleza: Relación de muchos a muchos (N-M) .
  - Descripción: Un Mensaje se asocia a un Usuario que lo envía y a otro que lo recibe. De esta forma, cada Usuario puede emitir o recibir múltiples mensajes. Se considera una composición entre usuario y mensaje ya que si no existe un usuario los registros de mensajes se borrarán, pero a su vez la otra relación sería una asociación ya que los usuarios seguirán existiendo.

- **Paciente - Nota**
  - Naturaleza: Relación de uno a muchos (1–N).
  - Descripción: Un paciente puede registrar múltiples Notas a lo largo del tiempo (una cada día). Se ha visto como una composición ya que, si el paciente es eliminado, sus notas pierden el sentido y se eliminan los registros también.

- **Nota - NotaDetalle**
  - Naturaleza: Relación de uno a muchos (1–N).
  - Descripción: Cada Nota puede desglosarse en varios NotaDetalle según el tipo de actividad (comida, deporte, etc.). Se ha interpretado como composición ya que los detalles forman parte intrínseca de la nota; si se elimina la nota, sus detalles también dejan de existir.

- **Paciente - Glucosa**
  - Naturaleza: Relación uno a muchos (1–N).
  - Descripción: Glucosa representa las mediciones de azúcar en sangre de un paciente. Un paciente puede tener muchas mediciones. Al eliminar un paciente, desaparecerían todos sus registros de glucosa; por lo tanto, se puede considerar composición ya que la medición depende enteramente del paciente.

- **Paciente - Bioquímico**
  - Naturaleza: Relación uno a muchos (1–N).
  - Descripción: Similar a Glucosa, Bioquímico registra valores de analíticas (colesterol, triglicéridos, etc.). Un paciente puede poseer muchos análisis bioquímicos; vistos como composición, ya que sin el paciente no tendría sentido conservar estos datos.

- **Paciente - Diagnóstico**
  - Naturaleza: Relación uno a muchos (1–N)
  - Descripción: Un paciente puede acumular múltiples diagnósticos realizados en diferentes fechas (muchos a lo largo de su historia clínica). Suele interpretarse como composición respecto al paciente, puesto que el diagnóstico es totalmente dependiente de la existencia del paciente.

- **Medico - Diagnostico**
  - Naturaleza: Relación uno a muchos (1–N).

- Descripción: Cada médico puede emitir numerosos diagnósticos, mientras que un diagnóstico en particular corresponde a un único médico. Es una agregación porque el médico puede existir sin que necesariamente existan diagnósticos (y viceversa, un diagnóstico podría conservarse aunque un médico dejara de estar activo).

- **Médico - Paciente**
  - Naturaleza: Relación uno a muchos (1–N).
  - Descripción: Un médico atiende a varios pacientes, y cada paciente está asociado a un médico en concreto (aunque se puedan producir cambios). Cada paciente depende de la asignación de un médico, pero ambos pueden existir de forma independiente antes de vincularse. Suele considerarse asociación con cardinalidades 1–N.

- **Paciente - Médico (Solicitudes_cambio)**
  - Naturaleza: Relación muchos a muchos (N-M)
  - Descripción: En ambas relaciones son composición ya que si al eliminar algunos de los dos tanto usuario como médico o los dos, los registros no tienen sentido porque no debe existir nadie con quien relacionarse.

- **EstadoEnum**
  - Naturaleza: No es una entidad relacional
  - Descripción: Sirve para clasificar y validar las solicitudes dentro de Solicitudes_cambio. No se relaciona de forma cardinal con otras entidades, sino que actúa como atributo en la definición de la solicitud.

3. Planificación

Al emplear Kanban como metodología principal, las tareas no se organizan en sprints cerrados, sino que siguen un flujo continuo en un tablero visual, donde cada tarjeta (tarea) pasa por columnas como Por Hacer, En Progreso y Hecho. Sin

embargo, la mayoría de los trabajos requieren una estimación temporal y un cronograma, para lo cual se mostrará un diagrama Gantt que sirva de visión general y facilite la justificación de las 300 horas totales asignadas al proyecto.

3.1. Planificación de las tareas

La planificación se compone de las siguientes fases o grandes grupos de trabajo, cada una vinculada a las historias de usuario y a la lógica de Kanban. El número de horas estimado se ajusta a los requisitos de carga, sumando 300 horas entre trabajo autónomo y reuniones con tutores.

1. Revisión de Historias de Usuario
    - Objetivo: Validar, detallar y priorizar las historias de usuario con base en las necesidades del sistema.
    - Actividades:
        - Analizar cada historia, definir criterios de aceptación claros.
        - Organizar las tarjetas correspondientes en la columna Por Hacer (tablero Kanban).
        - Reuniones con tutor para solventar dudas y refinar alcance.
    - Estimación: 20 horas.

2. Diseño de Arquitectura y Modelo de Datos
    - Objetivo: Plasmar la arquitectura global (front-end con Next.js, back-end con FastAPI) y el modelo de base de datos (SQLAlchemy).
    - Actividades:
        - Creación de diagramas de arquitectura (componentes, clases, flujo de datos).
        - Definición del modelo de dominio (entidades, relaciones, ORM).
        - Priorización de los primeros endpoints a implementar según las historias de usuario.
    - Estimación: 20 horas.

3. Implementación del Back-end
    - Objetivo: Construir la API principal con FastAPI y configurar la persistencia de datos con SQLAlchemy.
    - Actividades:
        - Modelado de entidades (Usuario, Paciente, Médico, etc.) y migraciones de base de datos.
        - Creación de endpoints y validación de datos.
        - Pruebas iniciales para la API.
        - Colocar cada endpoint o funcionalidad en el tablero Kanban, moverlos de En Progreso a Hecho al completarse.
    - Estimación: 60 horas.

4. Implementación del Front-end
    - Objetivo: Desarrollar la interfaz en Next.js, reflejando las pantallas y flujos de las historias de usuario (paciente, médico, familiar, administrador).
    - Actividades:
        - Configuración de proyecto.
        - Creación de vistas para login, registro, tablero de control (paciente/médico), formularios de notas, etc.
        - Integración de llamadas al back-end.
        - Pruebas básicas de la interfaz (comprobación de usabilidad y datos).
    - Estimación: 80 horas.

5. Integración y Pruebas de Conjunto
    - Objetivo: Combinar back-end y front-end, validando que las historias de usuario funcionen.
    - Actividades:
        - Desarrollar y ejecutar pruebas manuales (registro, vinculación familiar, consulta de glucosa, etc.).
        - Solucionar defectos (bugs) y refactorizar código según sea necesario.
    - Estimación: 40 horas.

6. Pruebas de Usuario
    - Objetivo: Realizar pruebas con casos reales
    - Actividades:
        - Simular escenarios con distintos tipos de usuario (paciente con glucosas, médico emitiendo diagnósticos, etc.).
        - Ajustar funcionalidades según feedback o errores hallados.
    - Estimación: 30 horas.

7. Documentación del TFG
    - Objetivo: Redactar la memoria del proyecto (introducción, análisis, diseño, implementación, conclusiones, anexos).
    - Actividades:
        - Explicar los conceptos
        - Incluir diagramas (modelo de dominio, arquitectura, etc.) y capturas de la interfaz.
        - Revisar estilo, ortografía y adecuación a la normativa del TFG.
    - Estimación: 40 horas.

3.2. Diagrama de Gantt

Columna 1	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5	Semana 6	Semana 7	Semana 8	Semana 9	Semana 10	Semana 11	Semana 12
1. Revisión de historias de usuario												
2. Diseño arquitectura/modelo datos												
3. Implementación back-end (FastAPI)												
4. Implementación front-end (Next.js)												
5. Integración y pruebas de conjunto												
6. Pruebas de Usuario												
7. Documentación TFG (paralelo)												
8. Reuniones con el tutor												

Tabla 1: Diagrama de Gantt

3.3. Estimación de costes

A continuación, se presenta una estimación de costes asociada al desarrollo del proyecto, en la que se toman en cuenta tanto los gastos humanos (horas de trabajo) como algunos gastos de infraestructura y otros elementos necesarios para su ejecución.

3.3.1. Costes de personal

El coste de personal constituye la parte principal de un proyecto de software. Aquí se asume que se va a contratar a usuario con los conocimientos necesarios para la realización de todo el desarrollo.
Pero también se podría haber realizado dividiendo el trabajo mediante las tareas a realizar y contratar a un especialista de cada parte como por ejemplo analistas, desarrollador back-end, desarrollador front-end y un diseñador.

- Perfil: Desarrollador Full Stack
- Tarifa aproximada: El salario full stack promedio en España es de 36.000€ al año o 18,46€ por hora.
- Horas totales: 300 horas.

Para simplificar, se tomará un coste medio de 18 €/hora.

Coste de personal = 300 (horas) × 18 (€/hora) = 5,400 €

Este importe cubre la dedicación del desarrollador a lo largo de las 300 horas destinadas al TFG, considerando tanto el análisis, la implementación (front-end y back-end), las pruebas y la documentación.

3.3.2. Costes de hardware

Se considera la adquisición o disponibilidad de equipo informático para que el desarrollador pueda llevar a cabo el trabajo en buenas condiciones. En este caso, se detallan los dispositivos empleados, sin incluir costes de electricidad o de conexión a internet, que pueden variar según la compañía y la duración efectiva del proyecto.

Hardware	Precio relativo	Precio	Vida útil
Portátil Dell latitude 3440	78€	1249€	4 años
Logitech MX Master 3S	7,91€	95€	3 años
Logitech G Pro X SE	5,83€	70€	3 años
Total	91,74€	1414€	-

Tabla 2: Costes de Hardware

- Precio relativo: es el coste imputado al período del proyecto, en función de la vida útil estimada del dispositivo para lo cual la duración del proyecto es de 3 meses.
- Precio total: coste de adquisición completo de cada dispositivo.
- No se incluyen en esta partida los gastos de luz ni de conexión a internet.

En la práctica, es frecuente que el desarrollador ya disponga de su equipo y no se generen estos gastos directamente. No obstante, aquí se refleja el valor económico relativo que supondría la provisión de hardware.

3.3.3. Costes de Software

En cuanto a software, se listan las herramientas utilizadas. Algunas son gratuitas en sus versiones estándar, mientras que otras requieren suscripciones o licencias.

Software	Precio relativo	Precio	Suscripción
Visual Studio Code	0€	0€	No
Visual Paradigm	267€	89€/mes	Si
GitLab	0€	0€	No
Google Docs	0€	0€	No
Hoja de cálculo	0€	0€	No
Total	267€	-	-

Tabla 3: Costes de Software

- Visual Studio Code es gratuito y no genera costes adicionales.
- Visual Paradigm (herramienta de modelado) puede requerir una suscripción mensual o un pago puntual, dependiendo de la licencia adquirida. En este caso, se ha estimado un desembolso de 267 € (3 meses a 89 €/mes) como coste relativo al desarrollo del proyecto.
- GitLab, Google Docs y el uso de hojas de cálculo no tienen coste en sus planes gratuitos, suficientes para un proyecto de estas características.

3.3.4. Costes de mantenimiento

Para un proyecto, normalmente no se contempla un gran coste de mantenimiento, ya que la mayoría de proyectos académicos finaliza en la entrega y defensa. Sin embargo, si se considerara un escenario profesional donde la aplicación se mantenga activa, se deberían incluir:

- Hosting o servidores: Desplegar el back-end (FastAPI) y el front-end (Next.js) en un servicio en la nube (p. ej., AWS, Azure, Vercel, Heroku, etc.). Los planes gratuitos pueden ser suficientes para pruebas, pero se requeriría un plan de pago si se espera un uso real o tráfico medio-alto, con importes que pueden oscilar entre 10 € y 50 € al mes.
- Dominio web: Alrededor de 10-15 € al año, en función del proveedor.
- Soporte y actualizaciones: Horas adicionales para corregir bugs, añadir nuevas funcionalidades o realizar mejoras en seguridad.

Si se trata estrictamente de un proyecto de TFG y no se va a desplegar a producción, el coste de mantenimiento puede estimarse en 0 €. En un entorno real, el importe es variable según la escala y objetivos del proyecto.

3.3.5. Total

Sumando la parte correspondiente de cada apartado, obtenemos:

Concepto	Coste
Coste de personal	5.400€
Coste de hardware	91,74€
Coste de software	267€
Coste de mantenimiento	0€
Total	5.758,74€

Tabla 4: Costes de Software

Por el contexto del TFG no se ha añadido un valor al mantenimiento, pero en caso de que se necesitara servicios externos (hosting, dominio), o soporte post-entrega, se añadiría una cantidad adicional, dependiente del tiempo y la infraestructura requerida.

4. Diseño

En este capítulo se describe la arquitectura y el diseño general de la aplicación, fundamentados en los requisitos y el modelo de dominio analizados en capítulos anteriores. El diseño aquí expuesto cubre tanto la organización conceptual de los datos (mediante el Diagrama Entidad-Relación) como la representación interna de las entidades y su lógica de negocio (Diagrama de Clases) en las diferentes capas de la aplicación (principalmente en el servidor y, si procede, en el cliente).

De esta forma, el lector podrá:

- Comprender cómo el modelo de dominio (definido en la fase de análisis) se traduce en estructuras de datos y relaciones dentro de la base de datos.
- Observar cómo se articulan las clases que encapsulan la lógica y las reglas de negocio en el back-end.
- Conocer la justificación de las decisiones de diseño relativas a la persistencia, la organización de entidades y la previsión de futuros cambios o escalabilidad.

4.1. Diagrama Entidad-Relación

El diagrama de Entidad-Relación (E-R) [25] describe una representación conceptual de la información con la que trabaja la aplicación, resaltando:

- Entidades: Elementos o “objetos” de la realidad que requieren ser modelados en la base de datos, son los principales del dominio (por ejemplo, Usuario, Paciente, Médico)
- Relaciones: Vínculos entre entidades
- Cardinalidades: Multiplicidad de la relación (1:1, 1:N, M:N)

Este diagrama sirve de base para el posterior diseño de la base de datos, así como para la definición de las clases en la capa de persistencia, tal como se detalla en el Diagrama de Clases del Servidor.

El Diagrama Entidad-Relación (E-R) que se presenta a continuación constituye la versión final del modelo de datos en la aplicación. En el capítulo 2.6 (“Modelo de dominio”), se describieron de forma detallada las entidades y la naturaleza de sus relaciones. En esta sección, por tanto, se mostrará cómo se implementan y conectan estas entidades en el diseño global de la base de datos, sin reiterar al completo la descripción conceptual ya expuesta.

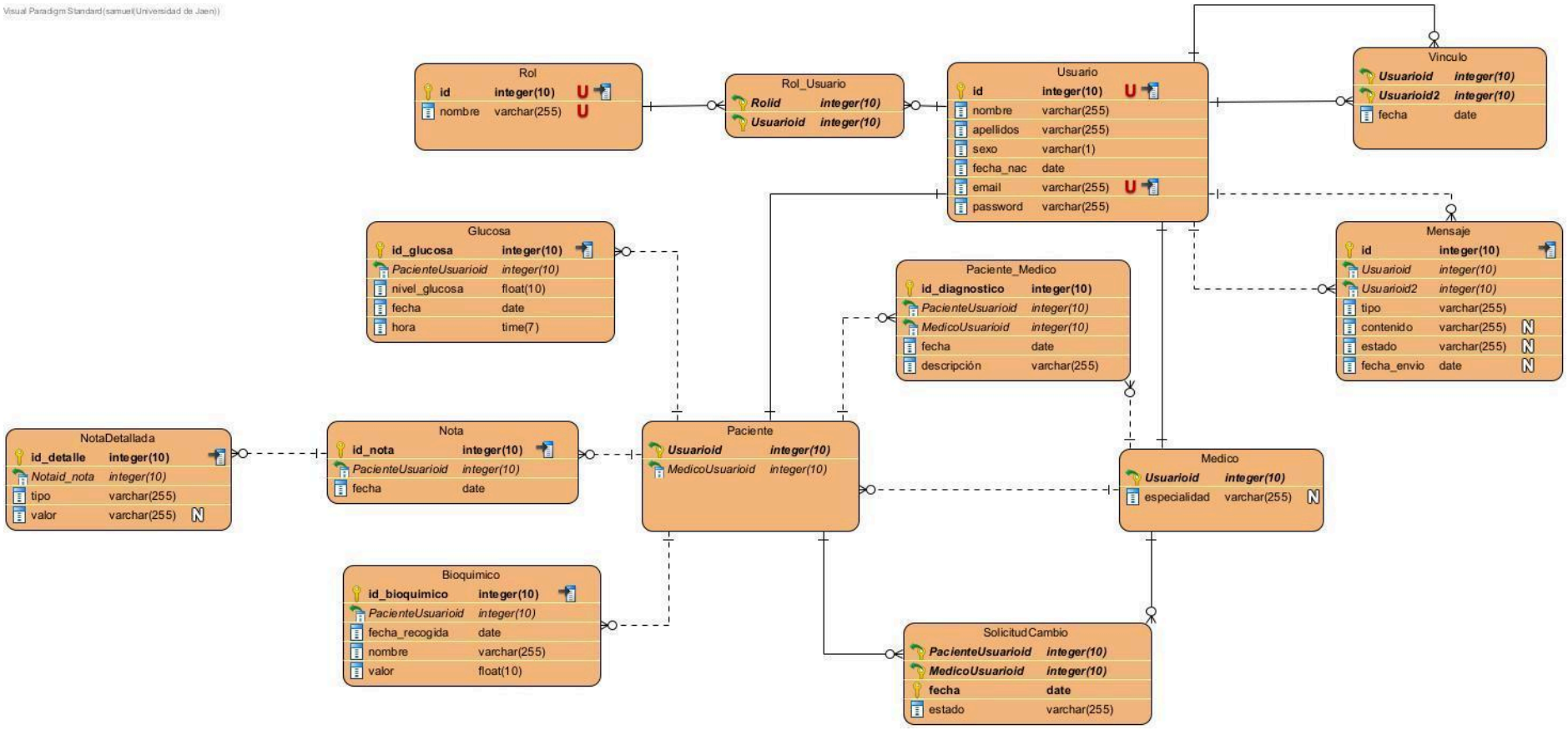


Figura 24: Diagrama Entidad Relación

4.1.1. Visión global y relación con el modelo de dominio

El diagrama E-R se basa en el modelo conceptual definido en 2.6 y mantiene la misma lógica: cada entidad (por ejemplo, Usuario, Paciente, Medico, Rol) tiene un propósito específico y se relaciona con las demás según las necesidades del sistema.

Se han definido claves primarias (PK) y claves foráneas (FK) coherentes con las cardinalidades y reglas de negocio ya explicadas. Por ejemplo:

- usuarios.id como clave primaria en la tabla usuarios;
- pacientes.id y medicos.id que referencian a usuarios.id (relación 1:1);
- Tabla intermedia usuario_roles para la relación M:N entre Usuario y Rol.

4.1.2. Estructura del diagrama E-R

En la ilustración adjunta (Figura 23), se observa el conjunto de entidades principales y sus relaciones:

1. Usuario
    - Sirve de entidad base para toda persona en la plataforma.
    - Se vincula uno a uno con Médico y Paciente.

2. Rol
    - Se asocia al Usuario mediante una tabla intermedia (M:N).

3. Paciente
    - Aglutina información clínica y enlaces a entidades como Nota, Glucosa, Bioquímico, y Diagnóstico.

4. Médico
    - Puede emitir Diagnóstico y tener varios pacientes asignados (relación 1:N).

5. Nota, NotaDetalle
    - Estructuras para almacenar la evolución diaria o periódica que registra el paciente.

6. Glucosa, Bioquímico
    - Entidades donde se guardan mediciones puntuales o analíticas más completas del paciente.

7. Diagnóstico
    - Asociado a un paciente y un Médico, indicando fecha y descripción del diagnóstico.

8. Solicitudes_cambio, Mensaje, Vínculo
    - Muestran la lógica de solicitudes (cambio de médico), mensajería interna y parentesco, respectivamente, completando el sistema.

Cada relación 1:1, 1:N o M:N se ilustra mediante líneas y símbolos de cardinalidad, en línea con la notación estándar de Entidad-Relación o UML. Por ejemplo, se aprecia cómo Paciente puede poseer muchas Glucosa, mientras cada registro de glucosa pertenece a un solo paciente.

4.1.3. Justificación de las claves y Restricciones

- Tablas con claves primarias compuestas: en particular, Solicitudes_cambio define como PK (id_paciente, id_medico, fecha), garantizando que cada petición de cambio sea única por paciente, médico y día.

- Tablas intermedias: para el caso M:N de usuario_roles, se ha creado una tabla que almacena los pares (usuario_id, rol_id) como PK compuesta. Esto asegura que no se dupliquen roles para un mismo usuario y se facilita la extensibilidad si se agregan más roles en el futuro.

- Restricciones de integridad: las FK (claves foráneas) establecen los vínculos. Por ejemplo, notas.id_paciente FK a pacientes.id; solicitudes_cambio.id_paciente FK a pacientes.id y id_medico a medicos.id. De este modo, no es posible generar registros “huérfanos” o inconsistentes.

4.1.4. Coherencia con el modelo de dominio

Al comparar este diagrama con el modelo de dominio descrito en 2.6, se verifica que:

- Cada entidad (Usuario, Paciente, Médico, etc.) mantiene el mismo propósito y atributos principales.

- Las relaciones (1:1 entre Usuario y Paciente/Médico, M:N entre Usuario y Rol, 1:N entre Paciente y Nota, etc.) coinciden con las definidas en el análisis.

- Se añaden detalles de implementación (nombres de tablas, PK, FK) que no se especificaron en 2.6, pues ese capítulo se centraba en la lógica conceptual.

4.2. Diagrama de Clases

Mientras que el Diagrama Entidad-Relación (4.1) ilustra la organización conceptual de las entidades y sus relaciones, el Diagrama de Clases se centra en cómo se implementan dichas entidades en el software. En este apartado:

1. Se expone una visión general de la arquitectura de clases, prestando atención a la lógica de negocio, la persistencia y la gestión de roles y permisos.
2. Se muestra cómo las entidades conceptuales (Usuario, Paciente, Médico, etc.) Se convierten en clases con atributos y métodos específicos, siguiendo patrones de diseño o prácticas recomendadas para el desarrollo.

3. Se subdivide el análisis en Diagrama de Clases del Servidor (4.2.1) y, un apartado para las clases del Cliente (4.2.2), para poder reflejar la organización interna de la capa front-end.

Esta aproximación permite apreciar la evolución del modelo de dominio a su representación técnica, indispensable para garantizar que la lógica descrita en los capítulos anteriores se materialice de forma coherente en el código.

4.2.1. Diagrama de clases del Servidor

El Diagrama de Clases del Servidor muestra cómo se ha estructurado la capa de back-end, donde reside la lógica de negocio y la persistencia de datos. En este proyecto, se implementa con Python, utilizando FastAPI para los endpoints y SQLAlchemy como ORM para mapear las clases a tablas en la base de datos.

4.2.1.1. Objetivos del diagrama de clases del servidor

- Reflejar el modelo de dominio tal como fue concebido a nivel conceptual (ver 2.6 y 4.1), mostrando ahora atributos y métodos relevantes que se utilizan en la práctica.

- Vincular cada clase a los elementos de persistencia (ORM con SQLAlchemy), de modo que se observe qué tablas y qué columnas se generan en la base de datos (sin profundizar en notación E-R, pues aquí priman las clases).

- Mostrar la lógica de negocio: métodos auxiliares (por ejemplo, has_role() en Usuario), validaciones internas, relaciones (OneToOne, OneToMany, etc.) implementadas en el código.

4.2.1.2. Estructura general

A grandes rasgos, el Diagrama de Clases del Servidor comprende las siguientes categorías:

- Clases básicas de dominio:
  - Usuario, que contiene la información necesaria para la identificación en la plataforma (nombre, email, contraseña, etc.).
  - Rol, que define las funciones y permisos.
  - Paciente, Médico como especializaciones (o extensiones) de la información base de Usuario.

- Clases relacionadas con el seguimiento clínico:
  - Nota, NotaDetalle, Glucosa, Bioquímico, Diagnóstico, donde se centralizan las mediciones y registros de la evolución del paciente.
  - Cada una posee atributos específicos (por ejemplo, fecha, descripción, nivel_glucosa) y métodos utilitarios si son precisos para la lógica (cálculo de promedios, validaciones, etc.).

- Clases de soporte:
  - Mensaje (para la comunicación interna),
  - Solicitudes_cambio (para gestionar peticiones de cambio de médico),
  - Vínculo (para reflejar la relación familiar o supervisión),
  - Enumeraciones (por ejemplo, EstadoEnum para los estados de solicitud), si se representan como clases específicas o enumerados en el diagrama UML.

4.2.1.3. Relaciones entre clases

- Relación 1–1 entre Usuario y Médico/Paciente, donde cada instancia de Médico o Paciente está asociada a exactamente un registro de Usuario.

- Relación 1–N en clases como Paciente ↔ Nota, Paciente ↔ Glucosa, Medico ↔ Diagnostico, etc. En UML, a menudo se muestra como un “1” junto a Paciente y un “*” junto a Nota.

- Relación M–N para Usuario ↔ Rol, con una clase (o tabla) intermedia (usuario_roles) que no se ve como entidad de primer nivel, sino como un mapeo en SQLAlchemy.

- Composiciones o agregaciones (según tu diagrama UML), a menudo simbolizadas con rombos, si consideras que la vida de algunas clases depende totalmente de otra (por ejemplo, NotaDetalle depende de Nota).

4.2.1.4. Diagrama

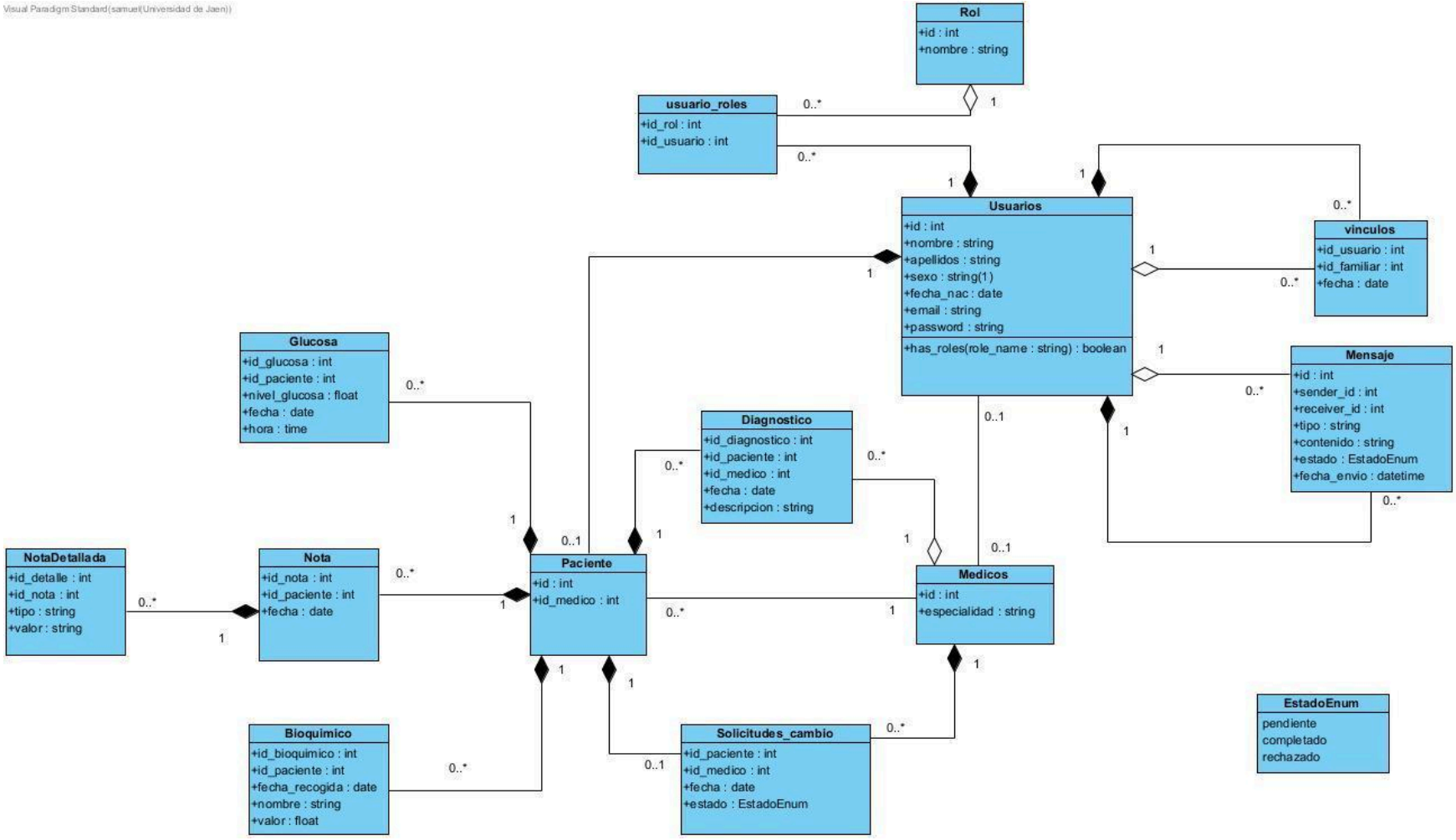


Figura 25: Diagrama de clases del servidor

4.2.1.5. Beneficios de la organización

- Claridad: se separan los datos por responsabilidad (paciente, médico, notas, mediciones, etc.), lo que facilita el mantenimiento.

- Escalabilidad: si se requiere agregar nuevas entidades (por ejemplo, otra métrica de salud), basta con crear la clase y vincularla en UML y en la base de datos.

- Buena integración con FastAPI: cada clase puede traducirse a modelos Pydantic para la gestión de datos en los endpoints, o usar directamente el ORM para las operaciones de base de datos.

4.2.2. Diagrama de componentes del Cliente

En el front-end del proyecto se emplea Next.js, un framework basado en React que aporta funcionalidades como ruteo automático, posibilidad de Server-Side Rendering (SSR) y Static Site Generation (SSG) *[12]*. Aunque React/Next.js no sigue el paradigma de clases clásico en la mayoría de los casos (utiliza principalmente funciones y hooks), resulta útil presentar un Diagrama de Componentes que muestre la relación entre las páginas, componentes y contextos principales del cliente.

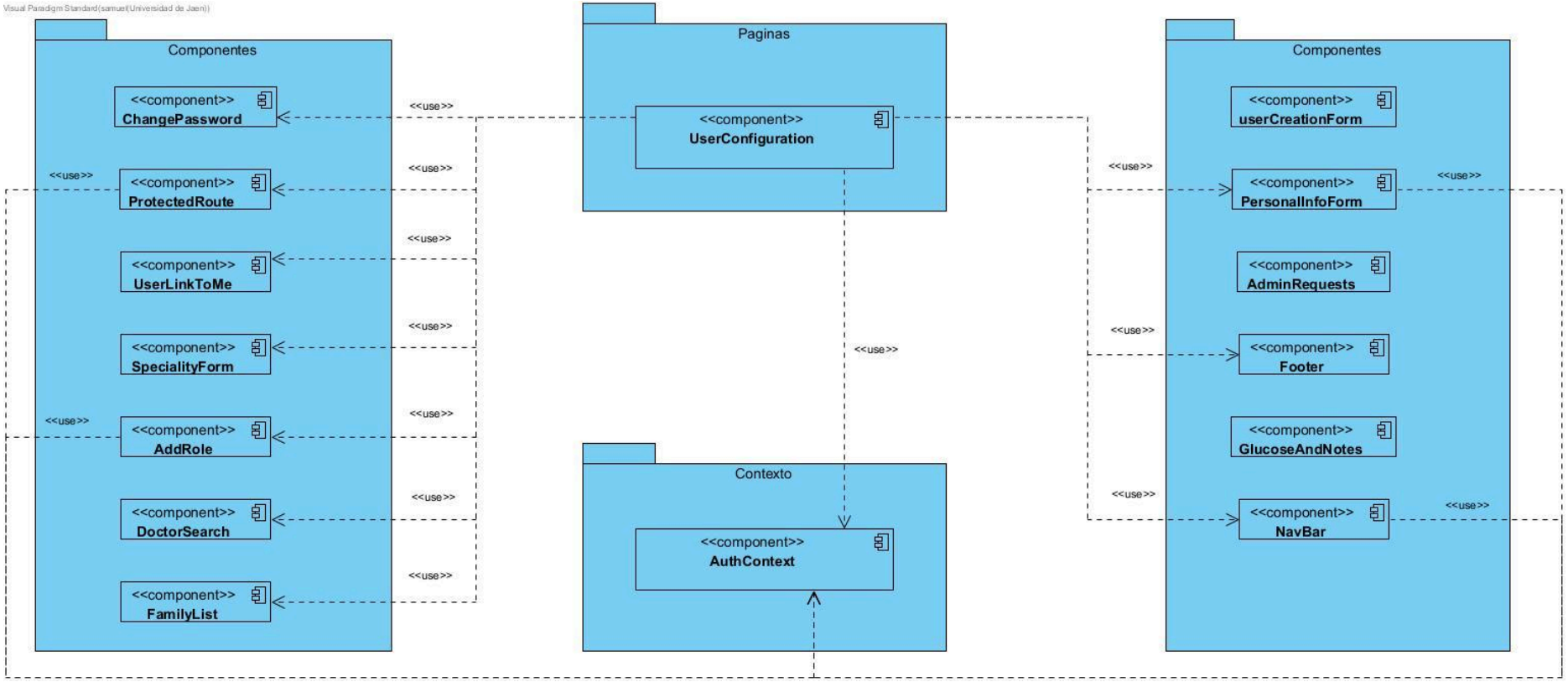


Figura 26: Diagrama de componentes del cliente (1/10)

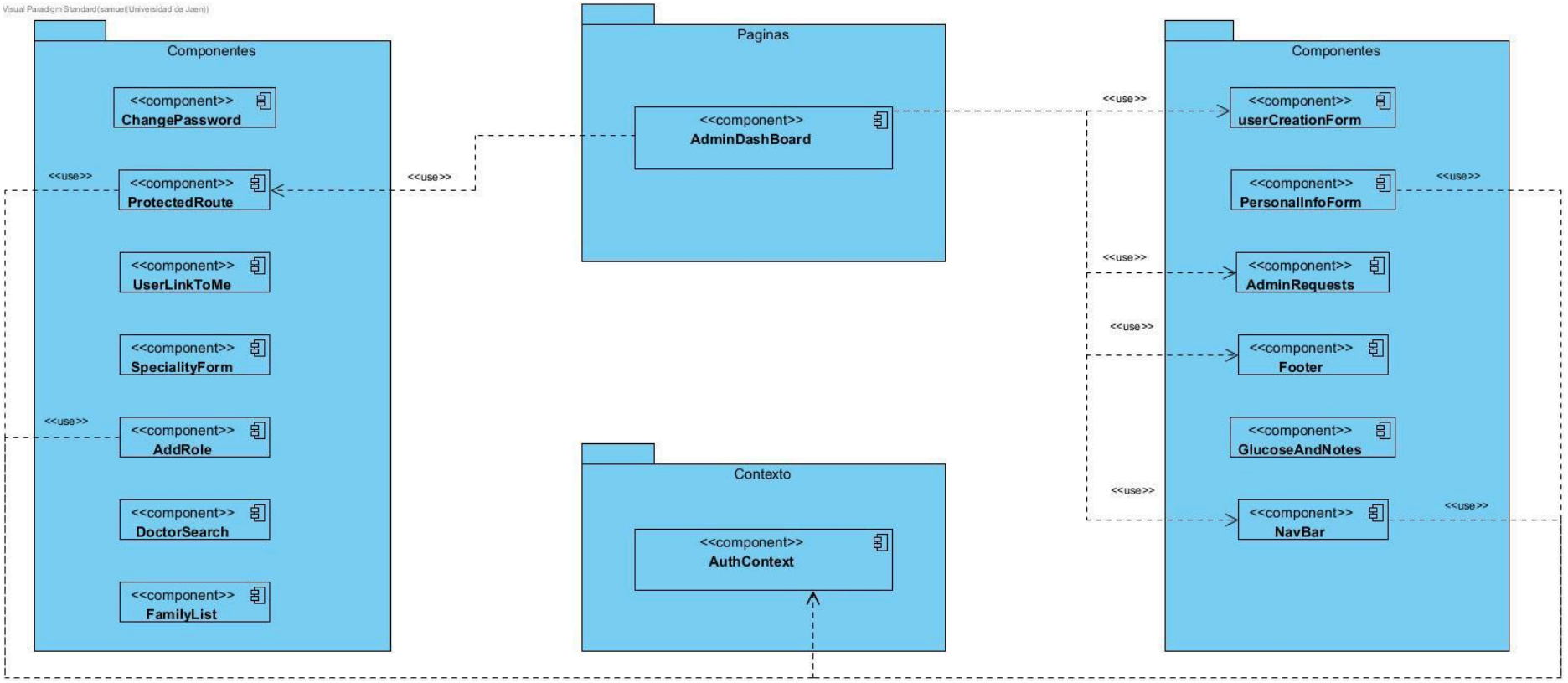


Figura 27: Diagrama de componentes del cliente (2/10)

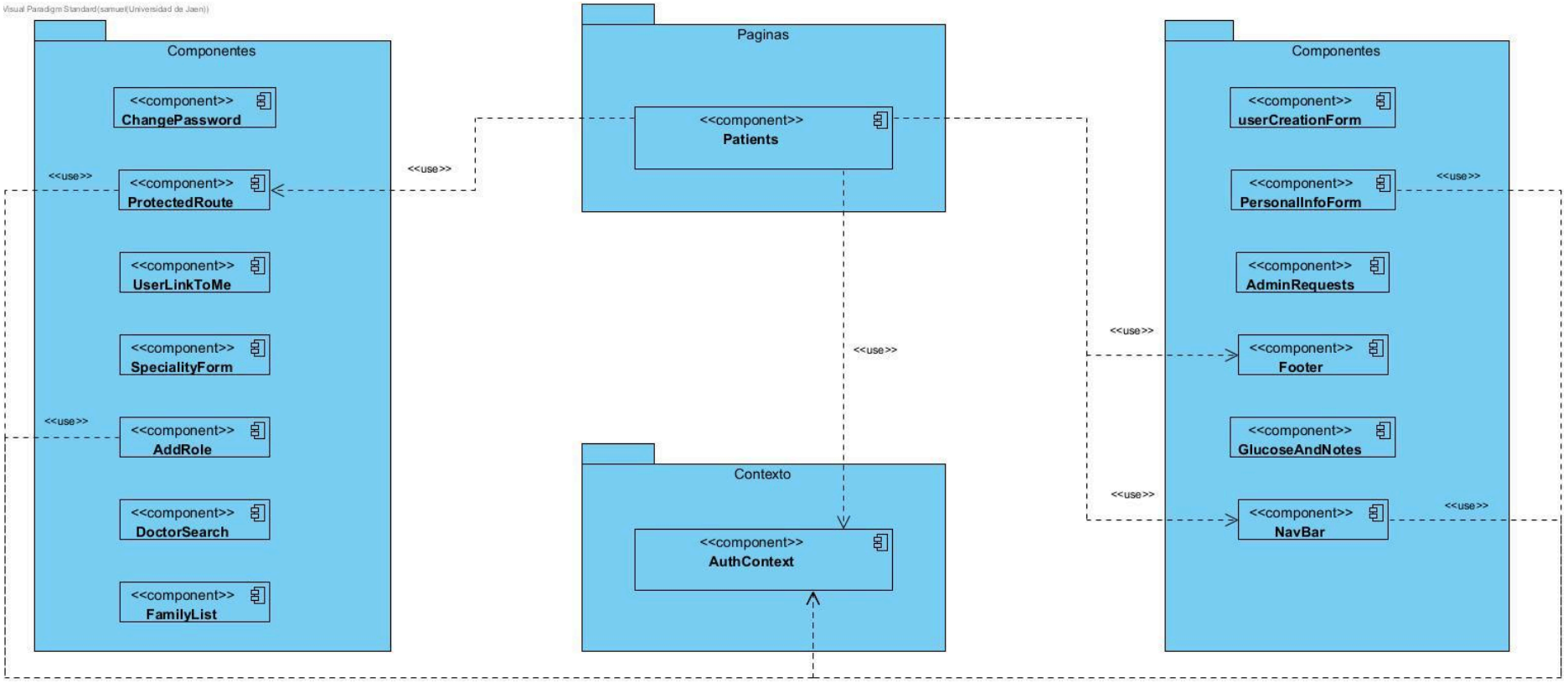


Figura 28: Diagrama de componentes del cliente (3/10)

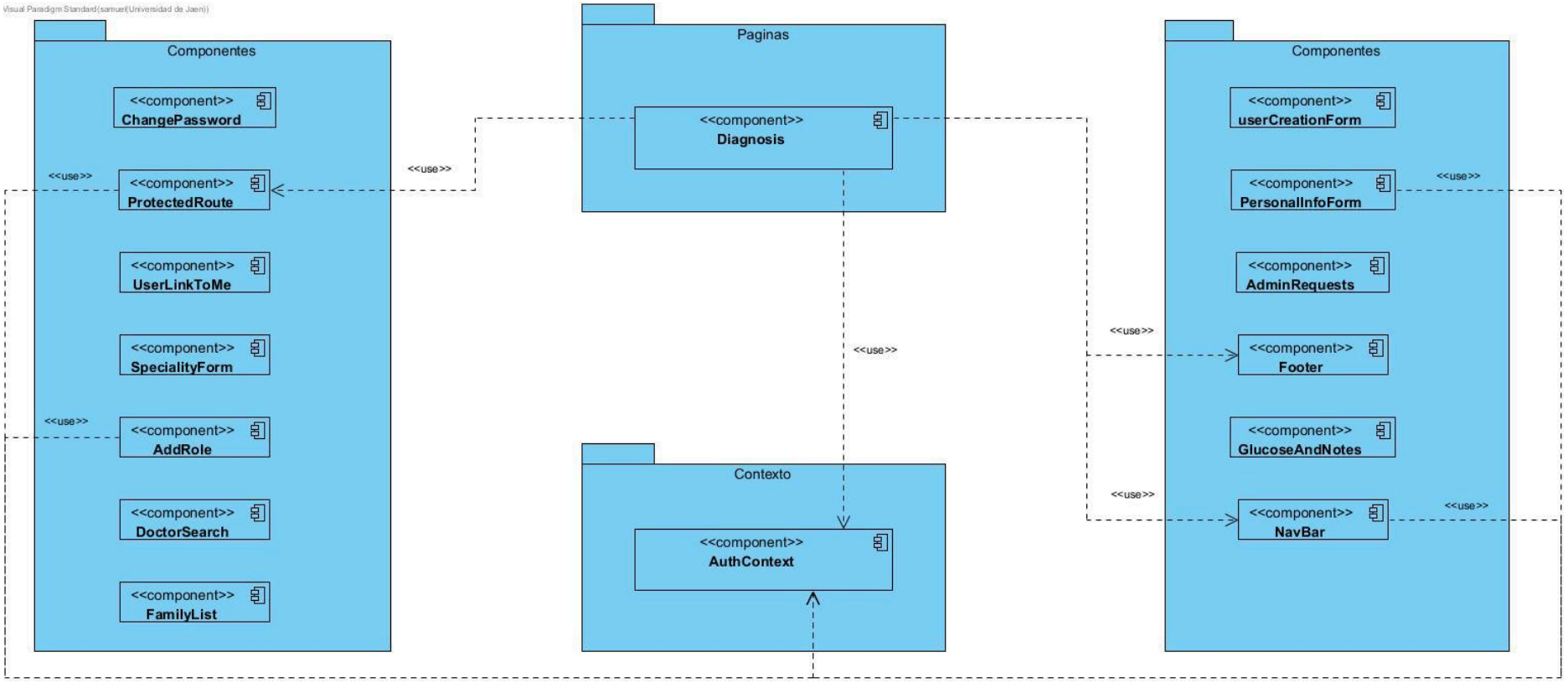


Figura 29: Diagrama de componentes del cliente (4/10)

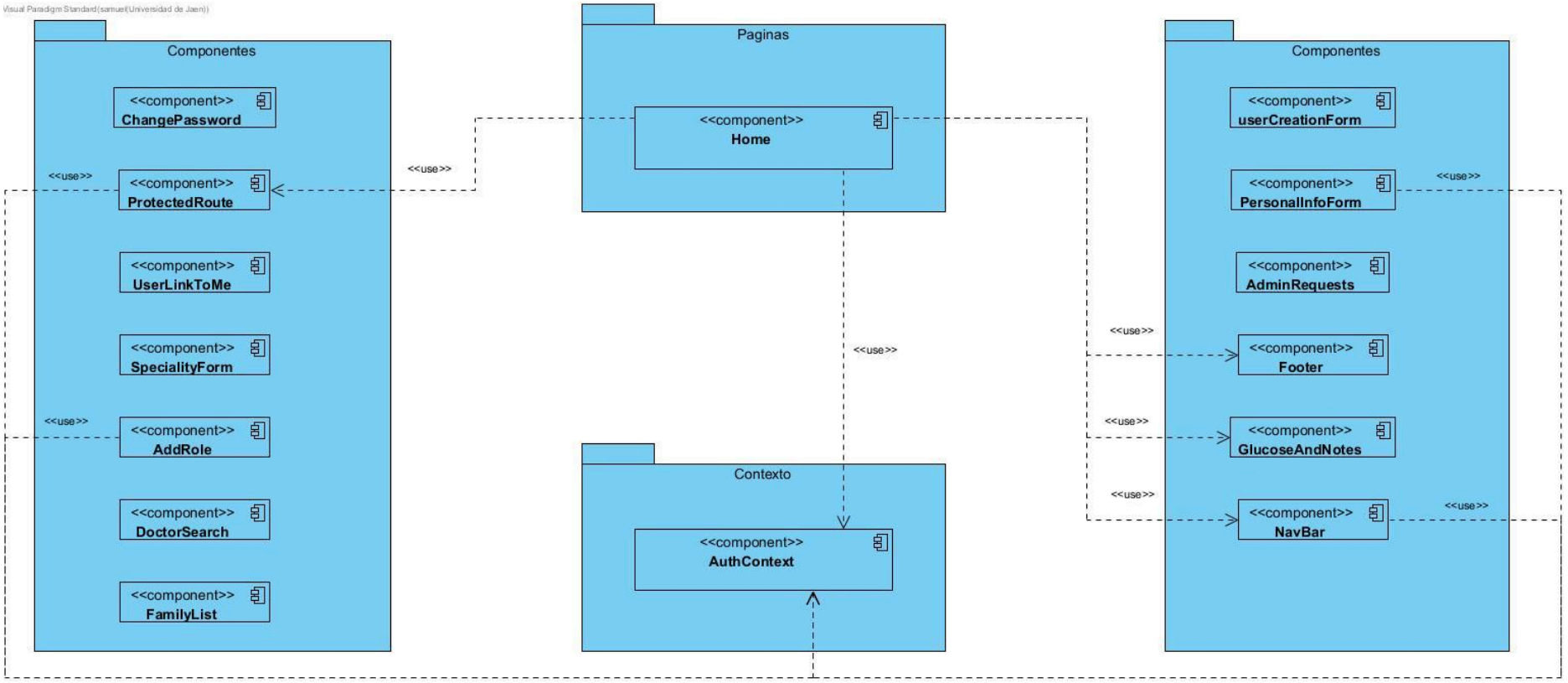


Figura 30: Diagrama de componentes del cliente (5/10)

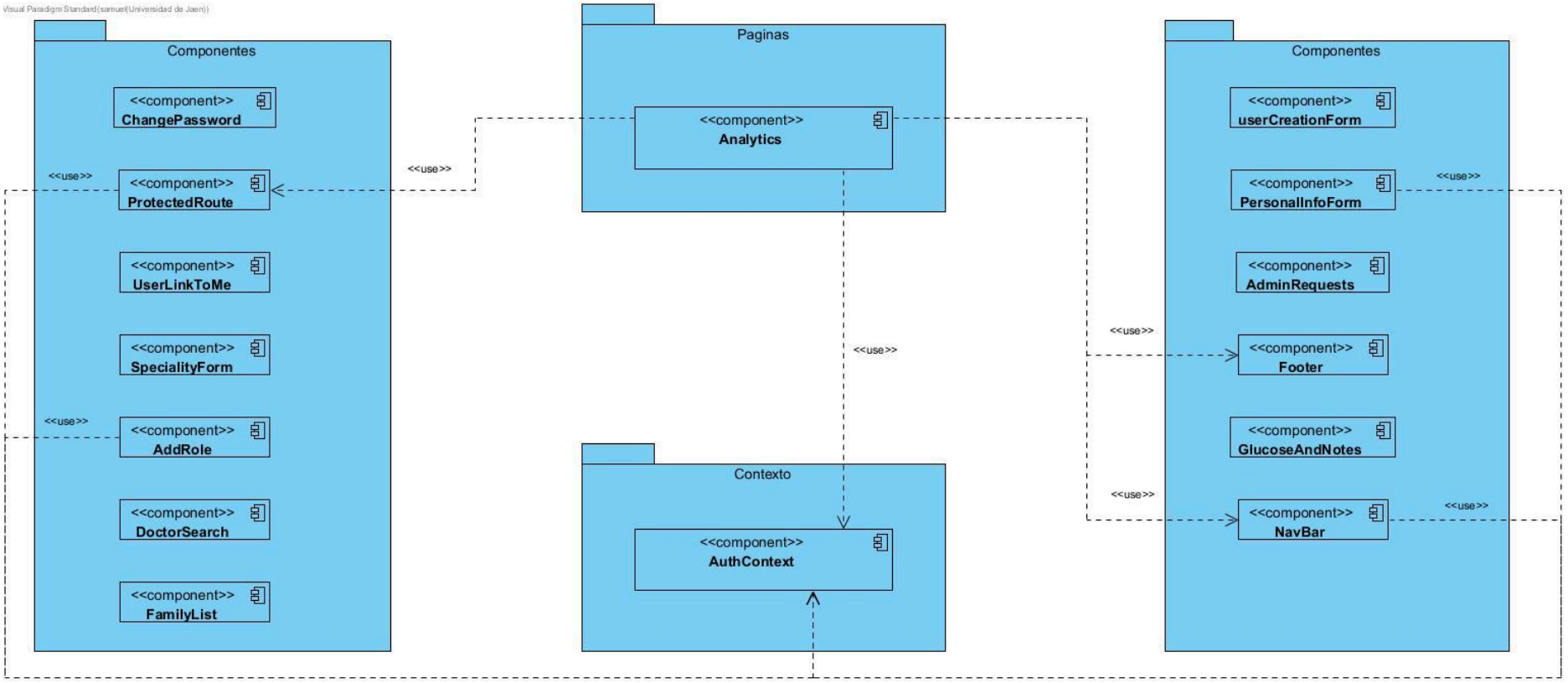


Figura 31: Diagrama de componentes del cliente (6/10)

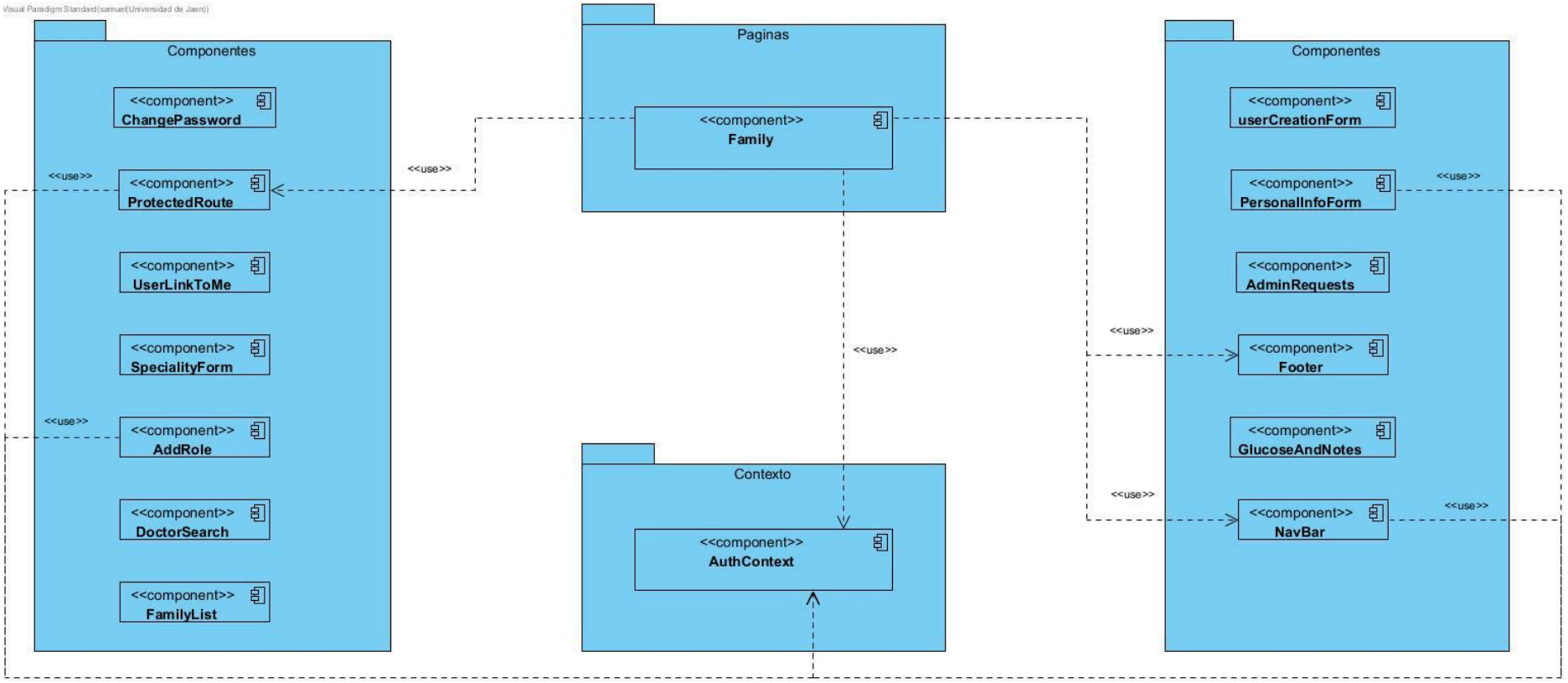


Figura 32: Diagrama de componentes del cliente (7/10)

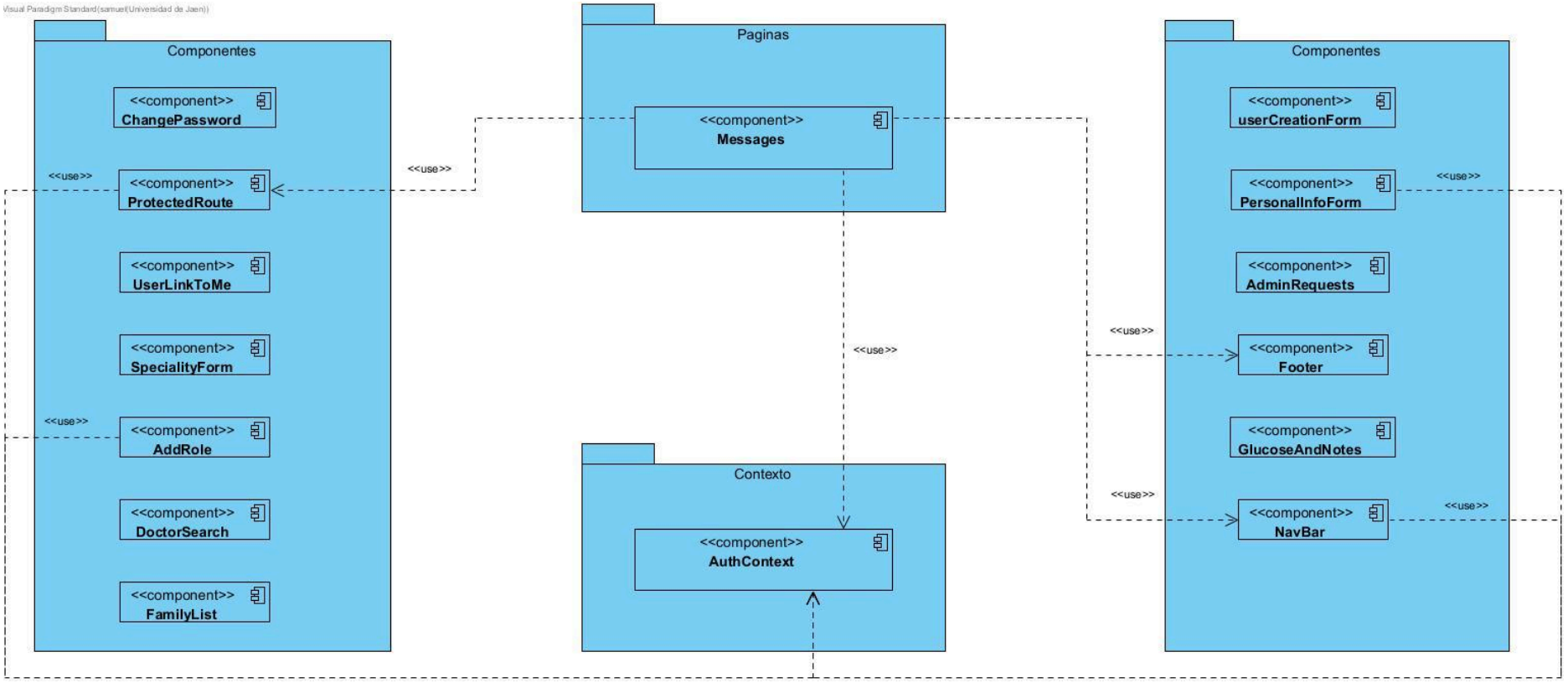


Figura 33: Diagrama de componentes del cliente (8/10)

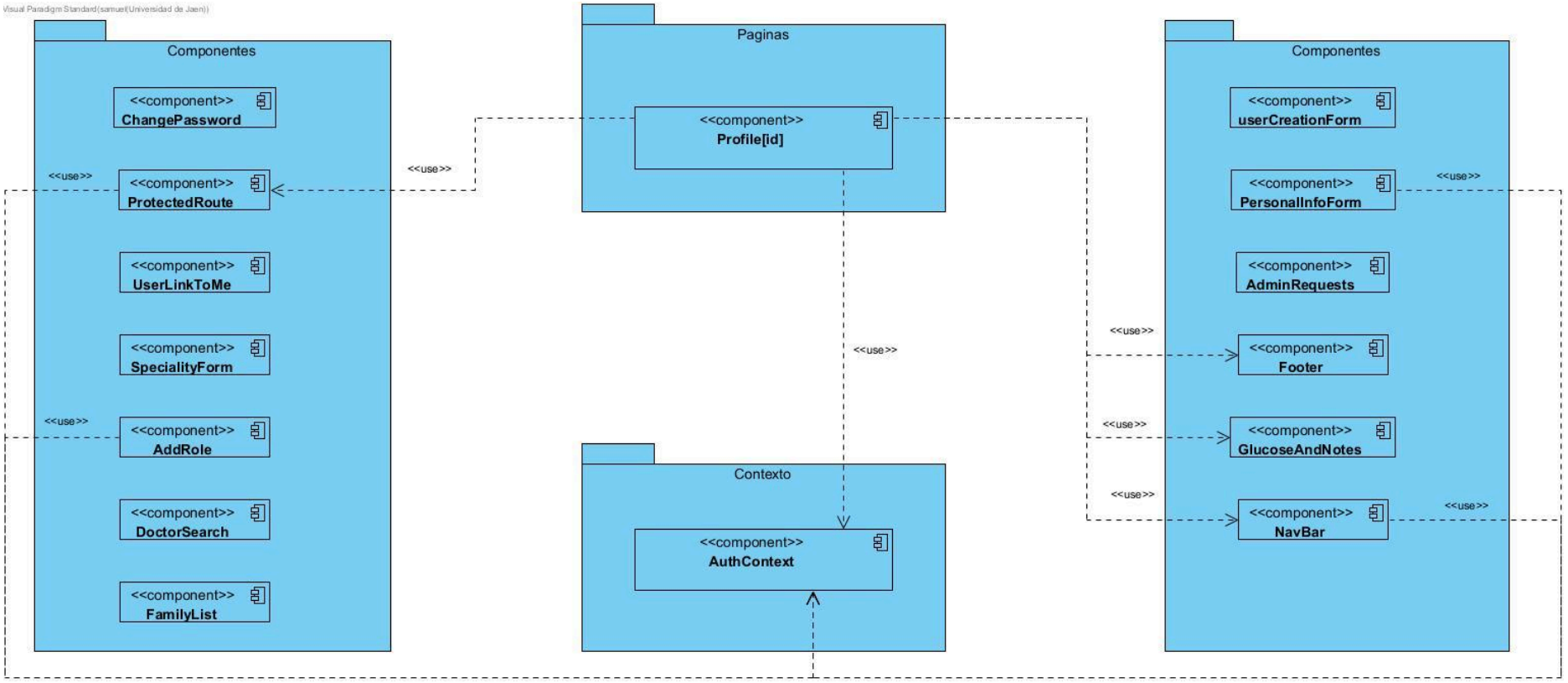


Figura 34: Diagrama de componentes del cliente (9/10)

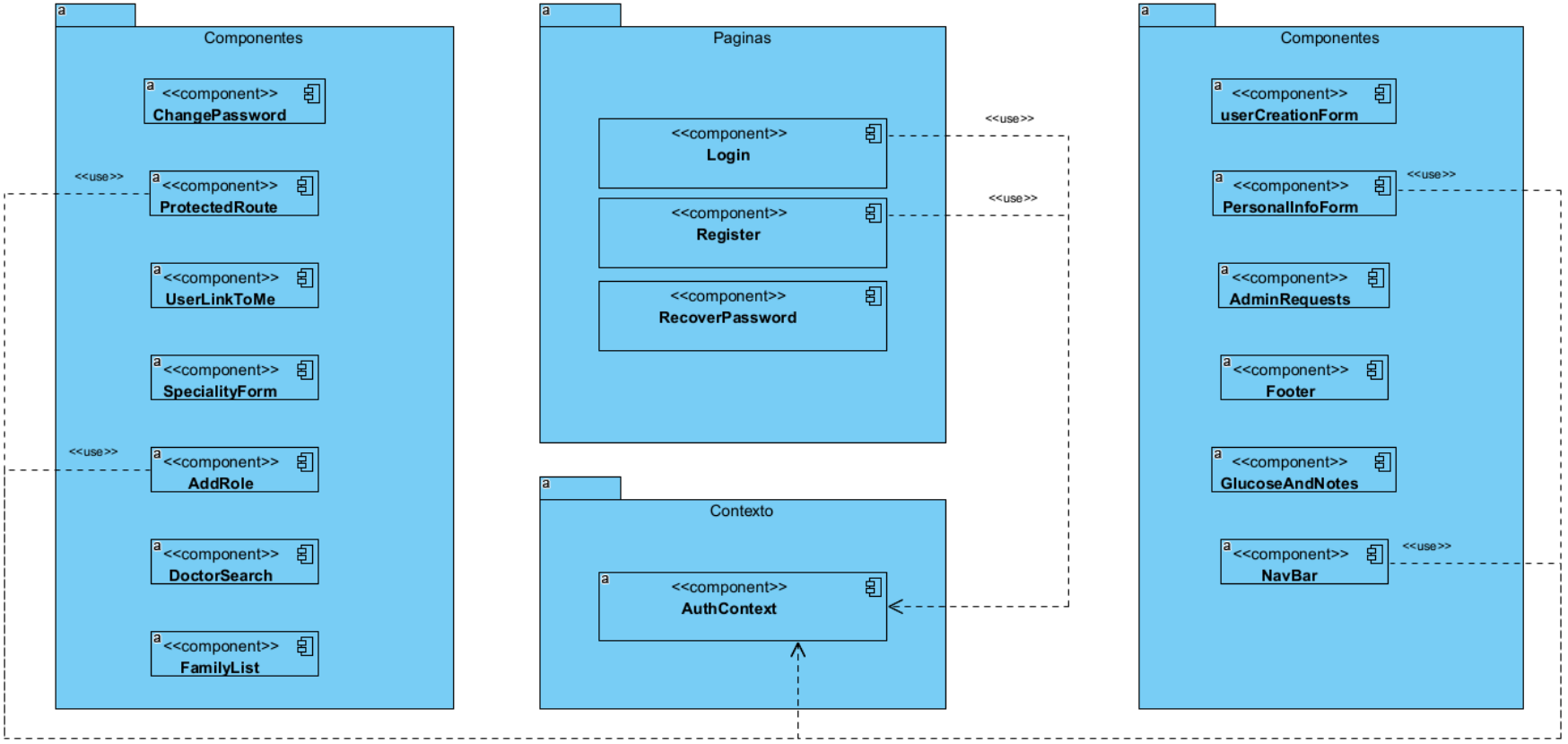


Figura 35: Diagrama de componentes del cliente (10/10)

4.2.2.1. Estructura general del cliente

Se pueden distinguir tres niveles:

- Contextos y lógica global:
  - AuthContext: Proporciona el estado global de autenticación, guarda la información del usuario logueado y maneja funciones (login, logout, refreshUser).
  - Otros contextos o hooks (si los hubiese) que sirven para compartir datos entre componentes.

- Páginas (Next.js):
  - login/page.js: Página de inicio de sesión.
  - register/page.js: Página de registro de usuarios.
  - recoverPassword/page.js: Página para recuperar o actualizar contraseña.
  - home/page.js: Página principal para el rol “paciente”.
  - analytics/page.js: Visualización de analíticas de laboratorio.
  - diagnosis/page.js: Historial de diagnósticos.
  - family/page.js: Gestión de familiares (rol “familiar”).
  - messages/page.js: Bandeja de mensajes o solicitudes.
  - patients/page.js: Listado de pacientes (rol “médico”).
  - profile/[id]/page.js: Perfil de un paciente (accesible por el médico o familiar).
  - userConfiguration/page.js: Configuración de la cuenta y cambio de médico/familiares.
  - adminDashboard/page.js (si corresponde al rol administrador).

- Componentes reutilizables:
  - Layout y navegación:
    - NavBar: barra de navegación que muestra enlaces según el rol del usuario.
    - Footer: pie de página.

        - ProtectedRoute: componente que protege rutas o páginas mediante validación de roles y autenticación.
    - Componentes de lógica o presentación:
        - GlucoseAndNotes: muestra los niveles de glucosa (gráfico con Plotly) y las notas diarias.
        - FamilyList, DoctorSearch, UsersLinkedToMe, AddRole, ChangePassword, SpecialtyForm, etc., todos en la carpeta components/. Se encargan de funcionalidades específicas (manejo de familiares, cambio de médico, edición de contraseñas, etc.).
    - Formularios y helpers:
        - PersonalInfoForm: edición de datos personales (nombre, apellidos, etc.).
        - UserCreationForm (en caso de administrador).
        - AuthContext (en realidad es un contexto, pero conceptualmente aporta lógica común).

4.2.2.2. Diagrama de componentes

Interpretación:

- Cada página (Login, Register, Home, etc.) es un componente en Next.js que se enmarca en su propio directorio, teniendo dentro la página en cuestión.

- AuthContext actúa como proveedor de estados globales y métodos. Las páginas que requieran autenticación o datos del usuario lo consumen mediante el hook useContext(AuthContext).

- ProtectedRoute revisa si el usuario posee los roles requeridos antes de renderizar la página; de lo contrario, redirige a login o a otra ruta apropiada.

- NavBar y Footer se insertan en distintas páginas para mantener la consistencia visual.

- Componentes específicos como GlucoseAndNotes, FamilyList, etc., solo se renderizan en páginas que necesitan esa funcionalidad.

4.2.2.3. Flujo de autenticación y Navegación

- Login: La página login.js muestra un formulario. Al hacer submit, llama a login() en AuthContext, que realiza la petición al servidor (/auth/token). Si es exitoso, se guarda el token y los datos de usuario en el contexto.

- ProtectedRoute: Cada página que requiere autenticación (ej.: Home, Family, Patients) se envuelve con ProtectedRoute o realiza la validación antes de cargar.

- Roles: ProtectedRoute o la propia página comprueba si el user.roles incluye los roles necesarios. Si no, redirige a otra ruta cada rol va explícitamente a su página inicial (p. ej., un paciente va a /home, un médico a /patients).

- Navegación: Se realiza con <Link href="/ruta"> o mediante el router de Next.js. NavBar muestra enlaces distintos según el rol del usuario.

4.2.2.4. Justificación del Diseño

- Separación de responsabilidades: Cada componente y página se encarga de una función específica (gestión de notas, visualización de glucosa, edición de contraseñas, etc.).

- Contexto centralizado: AuthContext evita pasar props de autenticación por múltiples niveles. Cualquier componente puede leer user o llamar a logout().

- Fácil escalabilidad: Añadir nuevas páginas o componentes (por ejemplo, un panel de reportes) implica solo crear una carpeta con el nombre del

componente y crear un archivo page.js dentro e invocar los servicios o contextos necesarios.

- Mantenimiento: Al usar React, se facilita la reutilización de componentes, la modularidad y la evolución del front-end.

4.3. Diagrama de casos de uso

En este apartado veremos los casos de uso explicados en sus correspondientes tablas y el diagrama de casos de uso asociado.

4.3.1. Casos de uso acceso a páginas

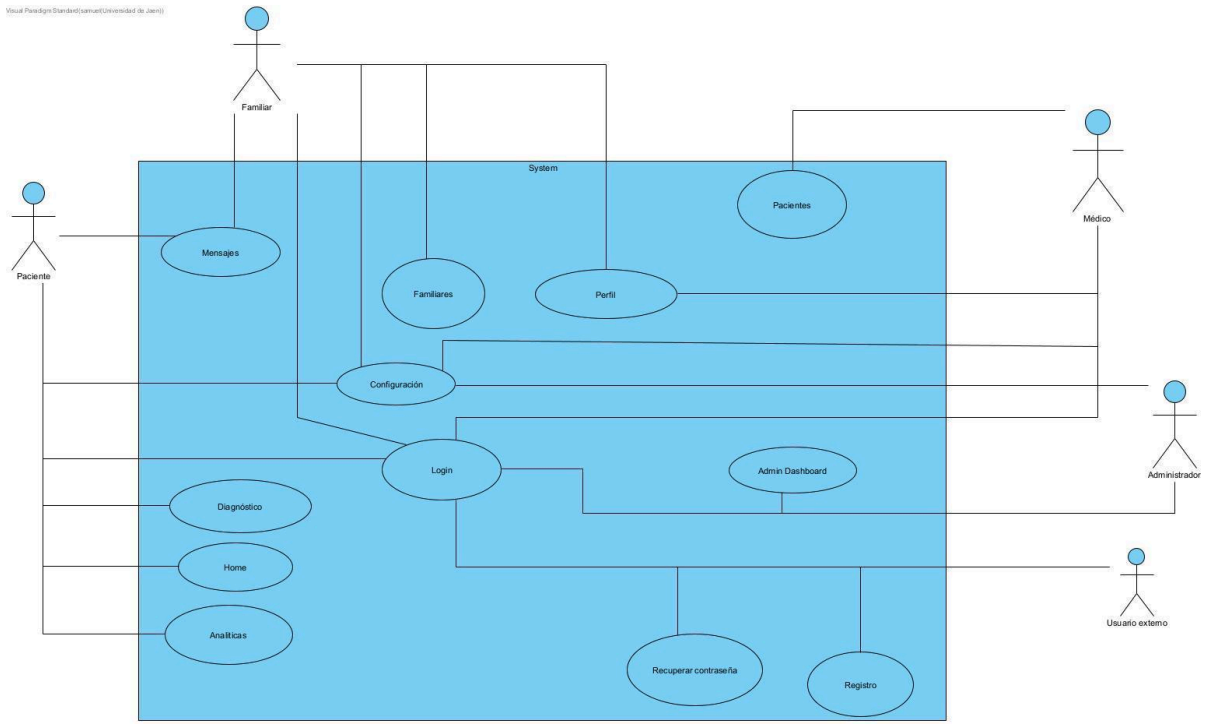


Figura 36: caso de uso acceso a páginas

4.3.2. Casos de uso comunes

4.3.2.1. Caso de uso registro

Nombre	Registro
Actores	Usuario externo (persona que aún no tiene cuenta en la plataforma)
Descripción	El usuario se registra en la aplicación proporcionando sus datos básicos (nombre, apellidos, fecha de nacimiento, etc.) y credenciales (correo electrónico y contraseña). La aplicación valida estos datos y crea una nueva cuenta.
Flujo Principal	1. El usuario accede a la vista o página de Registro. 2. El sistema muestra un formulario donde se solicitan datos personales (nombre, apellidos, sexo, fecha de nacimiento) y credenciales (email, contraseña). 3. El usuario rellena los campos requeridos y hace clic en el botón “Registrar” o “Crear cuenta”. 4. El sistema valida: a. Que todos los campos obligatorios estén completos. b. Que el correo electrónico no esté ya asociado a otro usuario. c. Que la contraseña cumpla con los criterios de seguridad requeridos (si aplica). 5. El sistema almacena los datos en la base de datos, creando un nuevo registro de usuario con su rol correspondiente (por ejemplo, “paciente” o “familiar”, según la elección o la configuración predeterminada). 6. El sistema confirma la creación exitosa de la cuenta y, opcionalmente, envía un correo de verificación si está habilitada la validación de email.

	7. El usuario recibe la notificación de que su cuenta fue creada y puede proceder a iniciar sesión.
Flujos alternativos / Excepciones	• A1: Datos inválidos o incompletos 1. El sistema detecta que falta algún campo obligatorio o que el formato del email no es válido. 2. Muestra un mensaje de error indicando el problema, sin crear la cuenta. 3. El usuario corrige los datos y reintentar el registro. • A2: Correo electrónico ya registrado 1. El sistema detecta que el email proporcionado se halla en uso por otro usuario. 2. Se muestra un mensaje alertando al usuario de que la cuenta ya existe. • A3: Cancelar el registro 1. El usuario decide no continuar con el registro. 2. Vuelve a la página de login.
Precondiciones	• El usuario no debe tener cuenta previa en la aplicación (no existe su correo en la base de datos). • El sistema está funcionando y la base de datos es accesible.
Postcondiciones	• El usuario queda registrado en la base de datos, con un registro que incluye sus datos personales y credenciales.

Tabla 5: Caso de uso registro

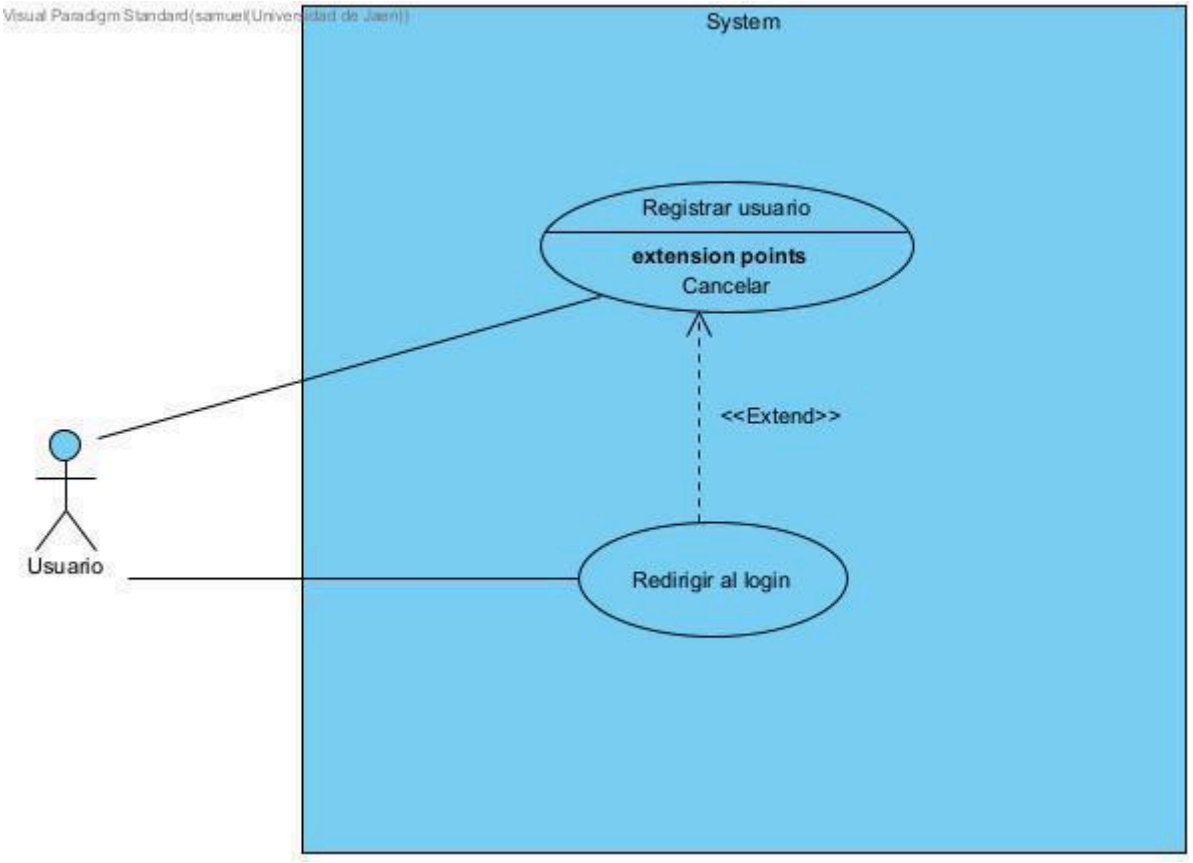


Figura 37:Caso de uso registro

4.3.2.2. Caso de uso login

Nombre	Iniciar sesión (Login)
Actores	Usuario (Paciente, Médico, Familiar, Administrador)
Descripción	El usuario, que ya posee una cuenta, ingresa sus credenciales (correo electrónico y contraseña) para autenticarse en el sistema. El sistema valida las credenciales y, si son correctas, habilita el acceso a las funcionalidades correspondientes a su rol.

Flujo Principal	1. El usuario accede a la página o pantalla de Login.
	2. El sistema muestra un formulario con campos para correo electrónico y contraseña.
	3. El usuario introduce sus credenciales y hace clic en el botón “Login”.
	4. El sistema verifica:
	  a. Que el correo electrónico exista en la base de datos.
	  b. Que la contraseña introducida coincida con la contraseña registrada.
	5. El sistema genera un token de sesión y lo envía al cliente para su uso en posteriores peticiones.
	6. El usuario es redirigido a la página principal correspondiente a su rol.

Flujos alternativos / Excepciones	• A1: Formulario incompleto
	  1. El usuario deja el campo de correo o contraseña vacío.
	  2. El sistema muestra un mensaje indicando que los campos son obligatorios.
	  3. El usuario corrige la información y reintenta.

	• A2: Credenciales inválidas 1. El correo no existe o la contraseña no coincide con la almacenada. 2. El sistema muestra un mensaje de error: “Usuario o contraseña incorrectos”. • A3: Registro 1. El usuario no tiene cuenta. 2. El sistema lo redirige a la página de registro. • A4: Recuperar contraseña 1. El usuario se ha olvidado de la contraseña. 2. El sistema lo redirige a la página de recuperar contraseña.
Precondiciones	• El usuario debe tener una cuenta creada previamente (se registró o fue creado por el administrador). • El sistema está disponible y funcional, con acceso a la base de datos.
Postcondiciones	• El usuario queda autenticado para la sesión actual, guardándose su token. • El sistema habilita a dicho usuario las funcionalidades que corresponden a su rol.

Tabla 6: Caso de uso login

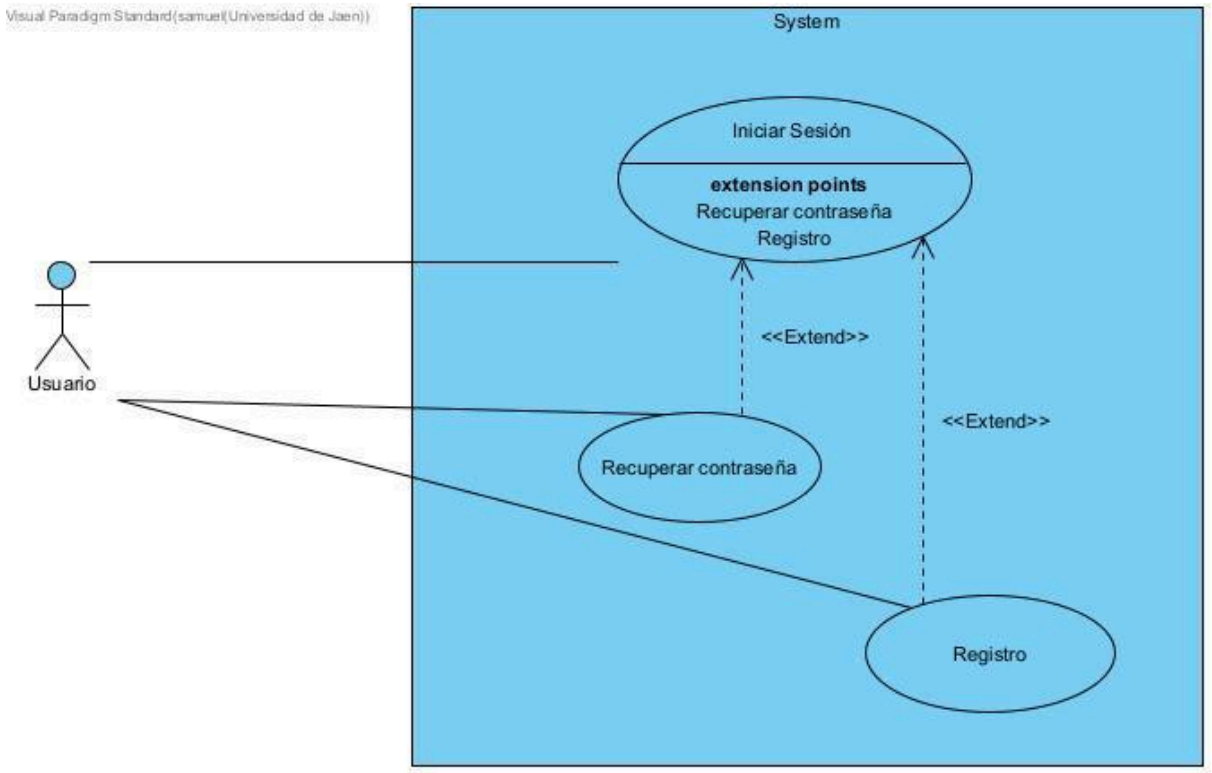


Figura 38: Caso de uso login

4.3.2.3. Caso de uso recuperar contraseña

Nombre	Recuperar contraseña
Actores	Usuario (Paciente, Médico, Familiar, Administrador)
Descripción	El usuario solicita restablecer su contraseña en caso de haberla olvidado. El sistema verifica su identidad mediante el correo y permite al usuario definir una nueva contraseña.
Flujo Principal	1. El usuario ingresa a la pantalla de "¿Has olvidado tu contraseña?". 2. El usuario ingresa su correo y confirma la acción de recuperar contraseña. 3. El sistema verifica si el correo pertenece a un usuario existente. 4. El sistema valida la nueva contraseña y actualiza la base de datos con el nuevo valor. 5. El usuario recibe confirmación de que la contraseña ha sido restablecida exitosamente y puede iniciar sesión con ella. 6. Se le redirige al login
Flujos alternativos / Excepciones	• A1: Correo no registrado 1. El usuario ingresa un correo que no existe en la base de datos. 2. El sistema notifica "Usuario no encontrado".
Precondiciones	• El usuario debe tener una cuenta registrada • El sistema está en línea
Postcondiciones	• La contraseña del usuario se actualiza correctamente en la base de datos. • El usuario ya puede iniciar sesión con la nueva

	contraseña.

Tabla 7: Caso de uso recuperar contraseña

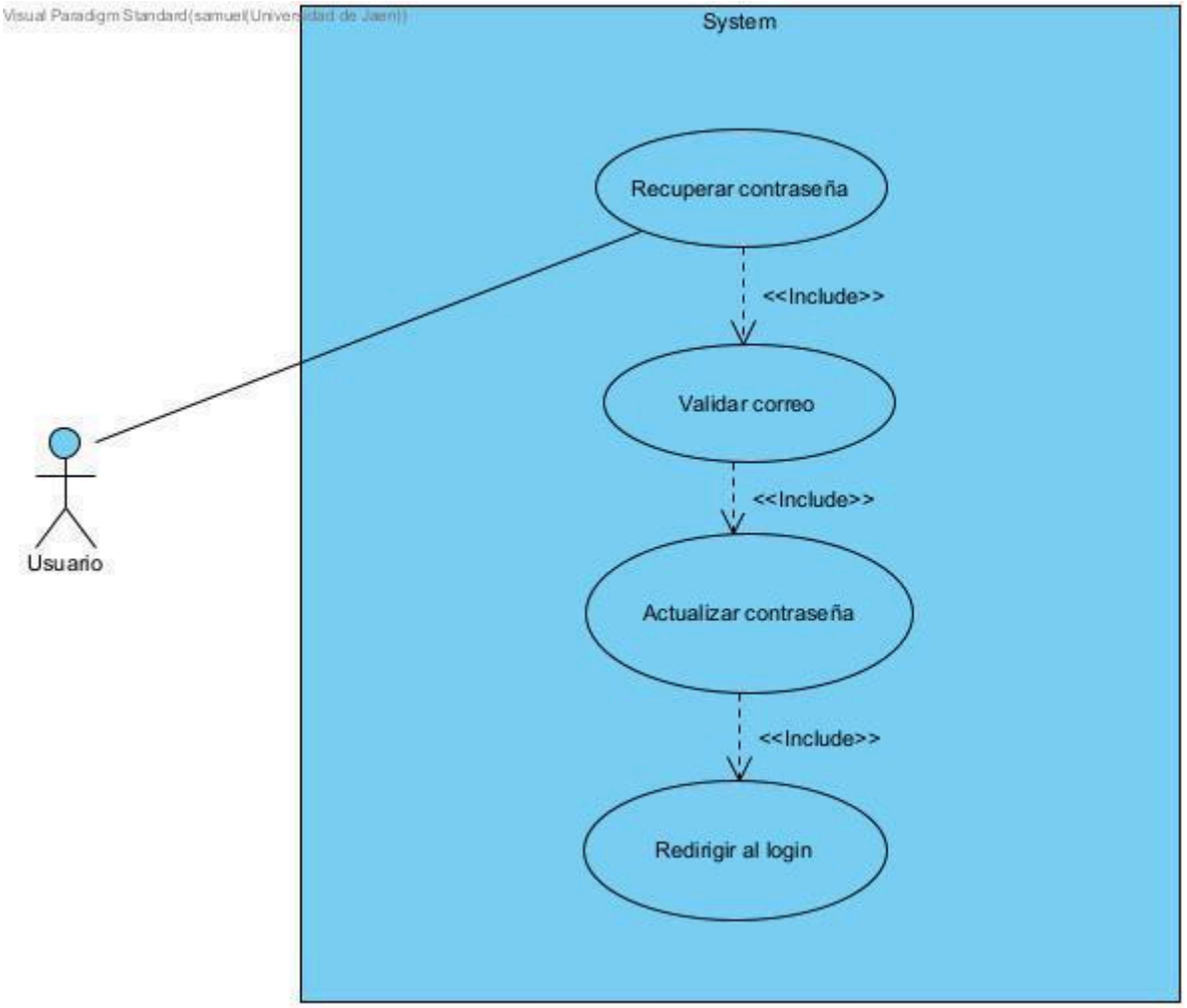


Figura 39: Caso de uso recuperar contraseña

4.3.2.4. Caso de uso cerrar sesión

Nombre	Cerrar sesión (Logout)
Actores	Usuario (Paciente, Médico, Familiar, Administrador)
Descripción	El usuario, tras haber iniciado sesión, decide finalizarla. El sistema invalida la sesión/token y devuelve al usuario a la página del login
Flujo Principal	1. El usuario se encuentra en la aplicación y observa un botón o enlace “Cerrar Sesión” 2. El sistema recibe la solicitud de logout y procede a: a. Eliminar o invalidar el token de acceso local (almacenado en localStorage) 3. El sistema redirige al login 4. El usuario deja de tener acceso a las funcionalidades que requieren autenticación, hasta que inicie sesión nuevamente.
Flujos alternativos / Excepciones	-
Precondiciones	• El usuario ya está autenticado • Se muestra el botón o enlace “Cerrar Sesión” en la interfaz.
Postcondiciones	• El usuario queda no autenticado, imposibilitado de acceder a rutas.

Tabla 8: Caso de uso cerrar sesión

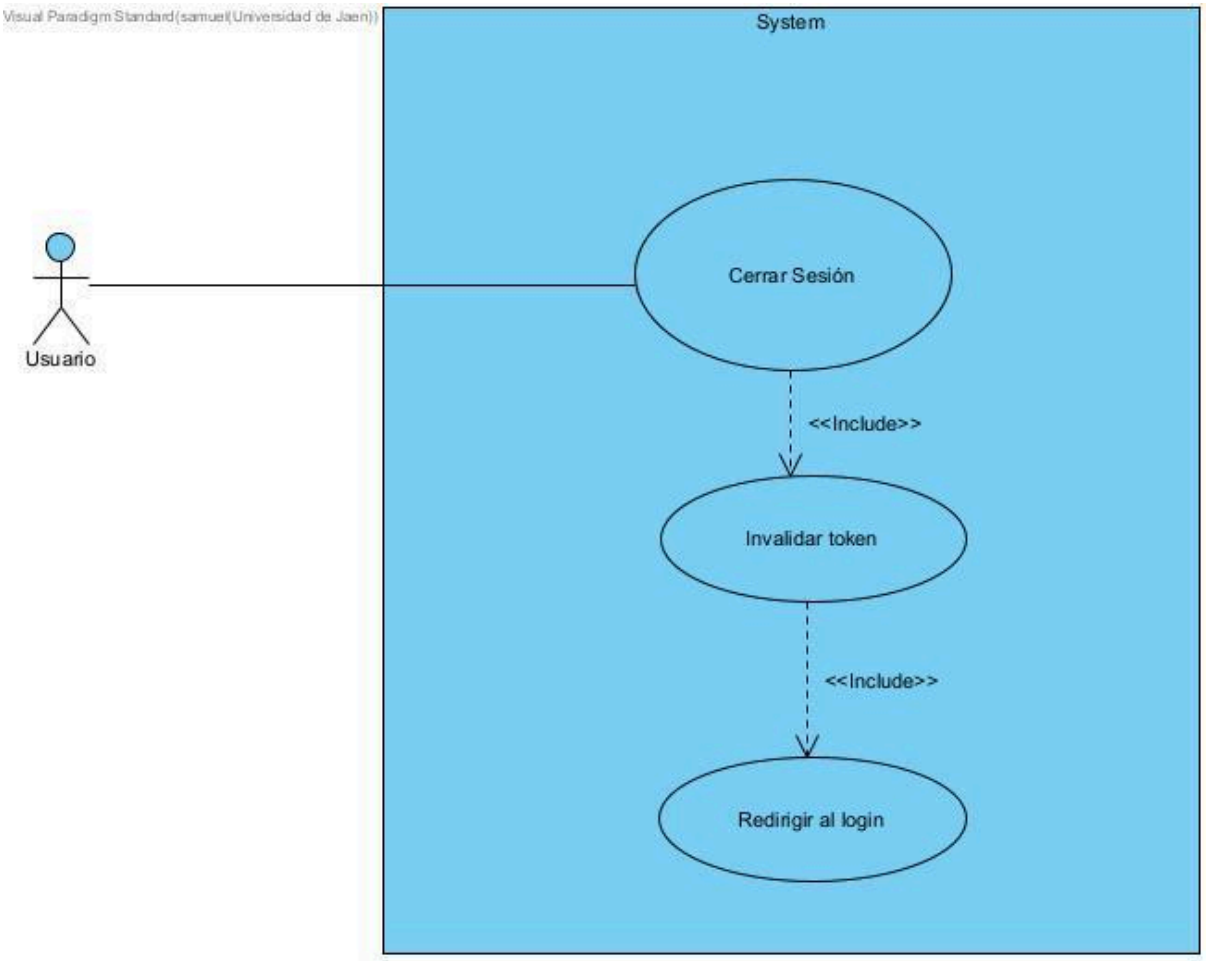


Figura 40: Caso de uso cerrar sesión

4.3.2.5. Caso de uso cambiar contraseña

Nombre	Cambiar contraseña
Actores	Usuario (Paciente, Médico, Familiar, Administrador)
Descripción	El usuario, que ya ha iniciado sesión, decide cambiar su contraseña actual por una nueva. El sistema verifica la contraseña anterior y actualiza la base de datos con la nueva clave tras verificarla.
Flujo Principal	1. El usuario accede a la página de Configuración dentro de la aplicación. 2. El sistema muestra un formulario con campos para la contraseña actual, nueva contraseña y confirmación de la nueva contraseña. 3. El usuario rellena esos campos y confirma la acción de “Cambiar contraseña”. 4. El sistema comprueba: a. Que la contraseña actual coincida con la almacenada. b. Que la confirmación de la nueva contraseña coincida. 5. El sistema actualiza la contraseña del usuario en la base de datos. 6. El usuario recibe la confirmación de que la contraseña ha sido modificada correctamente.
Flujos alternativos / Excepciones	• A1: Formulario incompleto 1. El usuario deja en blanco alguno de los campos obligatorios. 2. El sistema muestra un mensaje de error indicando que todos los campos son requeridos.

	3. El usuario corrige la entrada y reintenta. ● A2: Contraseña actual incorrecta (si se solicita) 1. El usuario introduce una contraseña actual que no coincide con la almacenada. 2. El sistema muestra un mensaje: “La contraseña actual es incorrecta”. 3. El usuario vuelve a intentarlo. ● A3: Contraseña nueva no coincide con la confirmación 1. El sistema detecta que nuevaContraseña y confirmacionContraseña no son iguales. 2. Se notifica al usuario “Las contraseñas no coinciden”. 3. El usuario corrige y envía.
Precondiciones	● El usuario ya está logueado en el sistema y tiene acceso a la sección de configuración de su cuenta. ● El sistema está disponible y se puede acceder a la base de datos para actualizar la contraseña.
Postcondiciones	● El usuario tiene una nueva contraseña almacenada de forma segura.

Tabla 9: Caso de uso cambiar contraseña

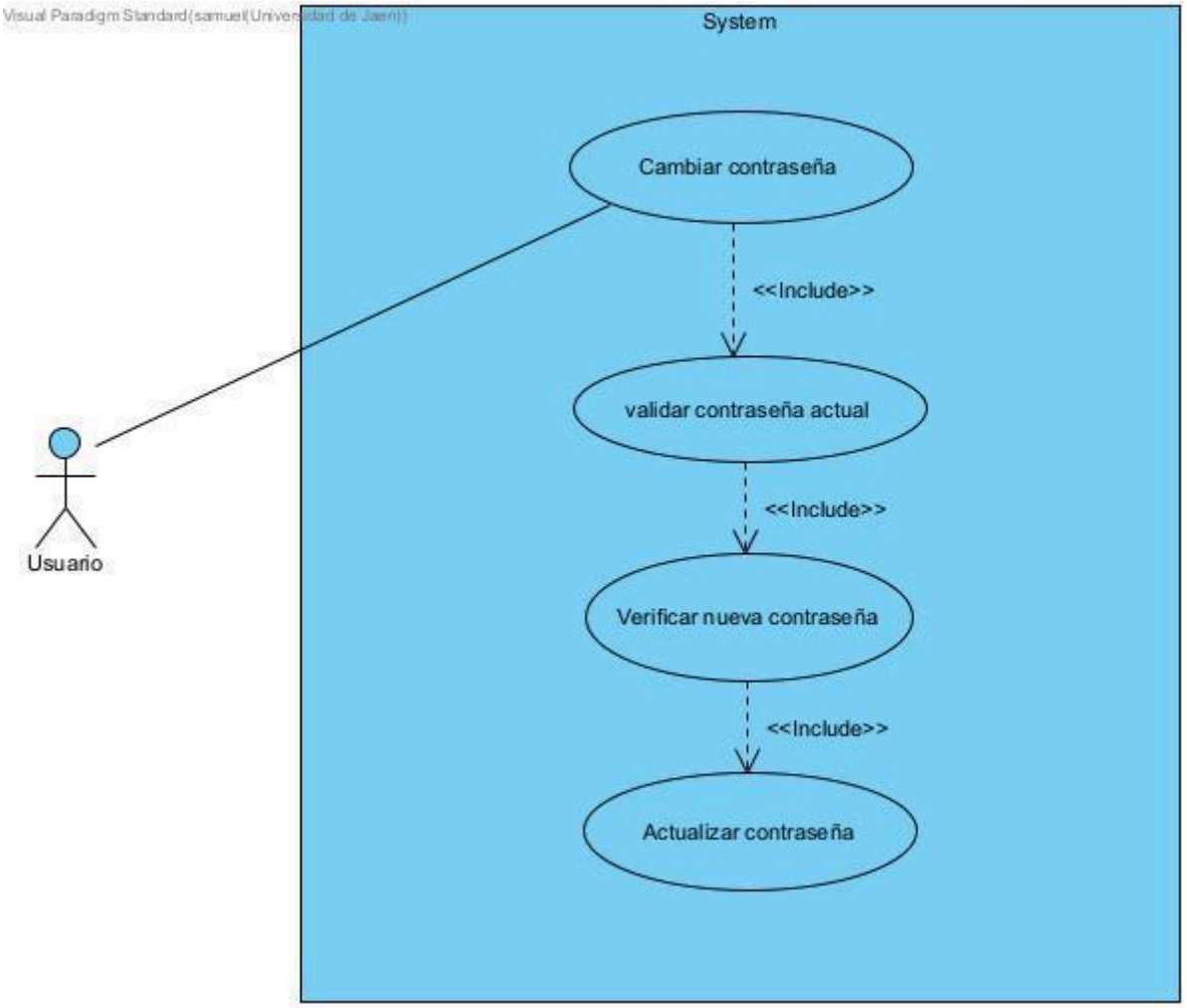


Figura 41:Caso de uso cambiar contraseña

4.3.2.6. Caso de uso editar perfil

Nombre	Editar perfil
Actores	Usuario (Paciente, Médico, Familiar, Administrador)
Descripción	El usuario, ya autenticado, accede a la página configuración y modifica algunos de sus datos (nombre, apellidos, fecha de nacimiento, etc.). El sistema valida y actualiza estos datos en la base de datos.
Flujo Principal	1. El usuario inicia sesión y se dirige a la página de configuración. 2. El sistema muestra un formulario con los datos personales actuales del usuario (nombre, apellidos, fecha de nacimiento, etc.). 3. El usuario modifica uno o varios campos y confirma la acción (por ejemplo, presionando “Guardar cambios”). 4. El sistema actualiza la información del usuario en la base de datos. 5. El usuario recibe una alerta de que su perfil ha sido actualizado correctamente.
Flujos alternativos / Excepciones	-
Precondiciones	• El usuario tiene una sesión iniciada. • El sistema pone a disposición la página de configuración.
Postcondiciones	• Los datos personales del usuario se actualizan en la base de datos. • Las próximas visualizaciones del perfil mostrarán la información actualizada.

Tabla 10: Caso de uso cambiar perfil

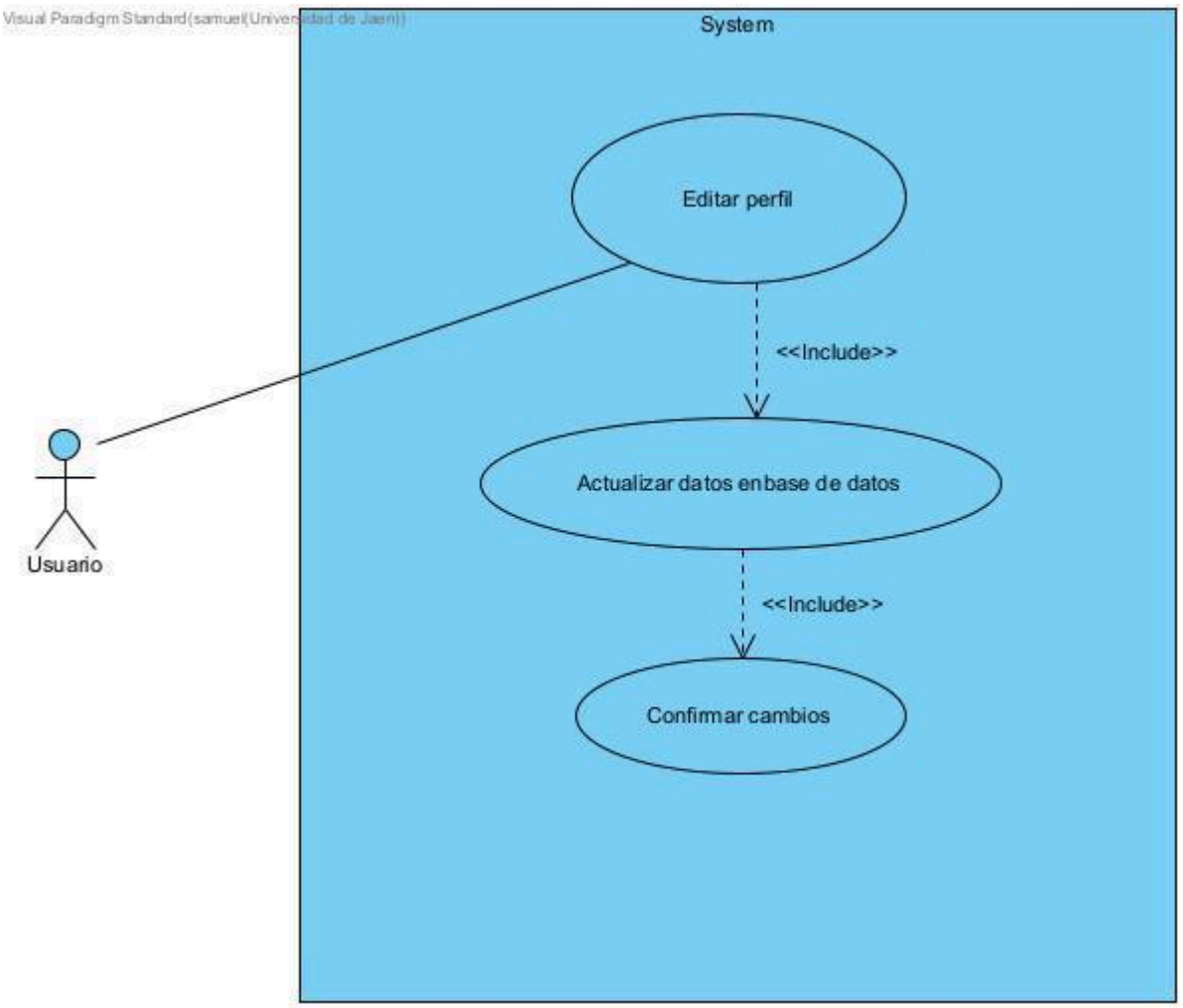


Figura 42: Caso de uso editar perfil

4.3.3. Casos de uso para el rol paciente

4.3.3.1. Caso de uso registrar nota diaria

Nombre	Registrar nota diaria
Actores	Paciente
Descripción	El paciente ingresa, de forma periódica, anotaciones sobre sus hábitos o sucesos relevantes (alimentación, ejercicio, actividades, síntomas, etc.). El sistema almacena estas notas asociándose a la fecha indicada, permitiendo un seguimiento cronológico de su evolución.
Flujo Principal	1. El paciente inicia sesión en el sistema y accede a la sección de Nota del día 2. El sistema muestra un formulario donde el paciente puede añadir uno o varios detalles (por ejemplo, tipo: “comida”, “deporte”, “actividad”) con una descripción. 3. El sistema valida que la fecha sea válida, que los campos requeridos no estén vacíos, etc. 4. El sistema crea un registro de “Nota” asociado al paciente y la fecha, y almacena los detalles en la base de datos. 5. El paciente visualiza la nota reflejada en su historial diario.
Flujos alternativos / Excepciones	● A1: Campos obligatorios vacíos 1. El sistema detecta que algún campo esencial (p. ej., descripción, tipo de detalle) está vacío. 2. El sistema no activa el añadido de la nota.

	3. El paciente corrige y reintentar.
Precondiciones	• El paciente debe tener sesión iniciada.
Postcondiciones	• La nueva nota y la nota detallada quedan almacenadas en la base de datos, asociada al paciente y a la fecha indicada. • El paciente puede consultar la nota en su historial diario y editarla o eliminarla.

Tabla 11: Caso de uso registrar nota diaria

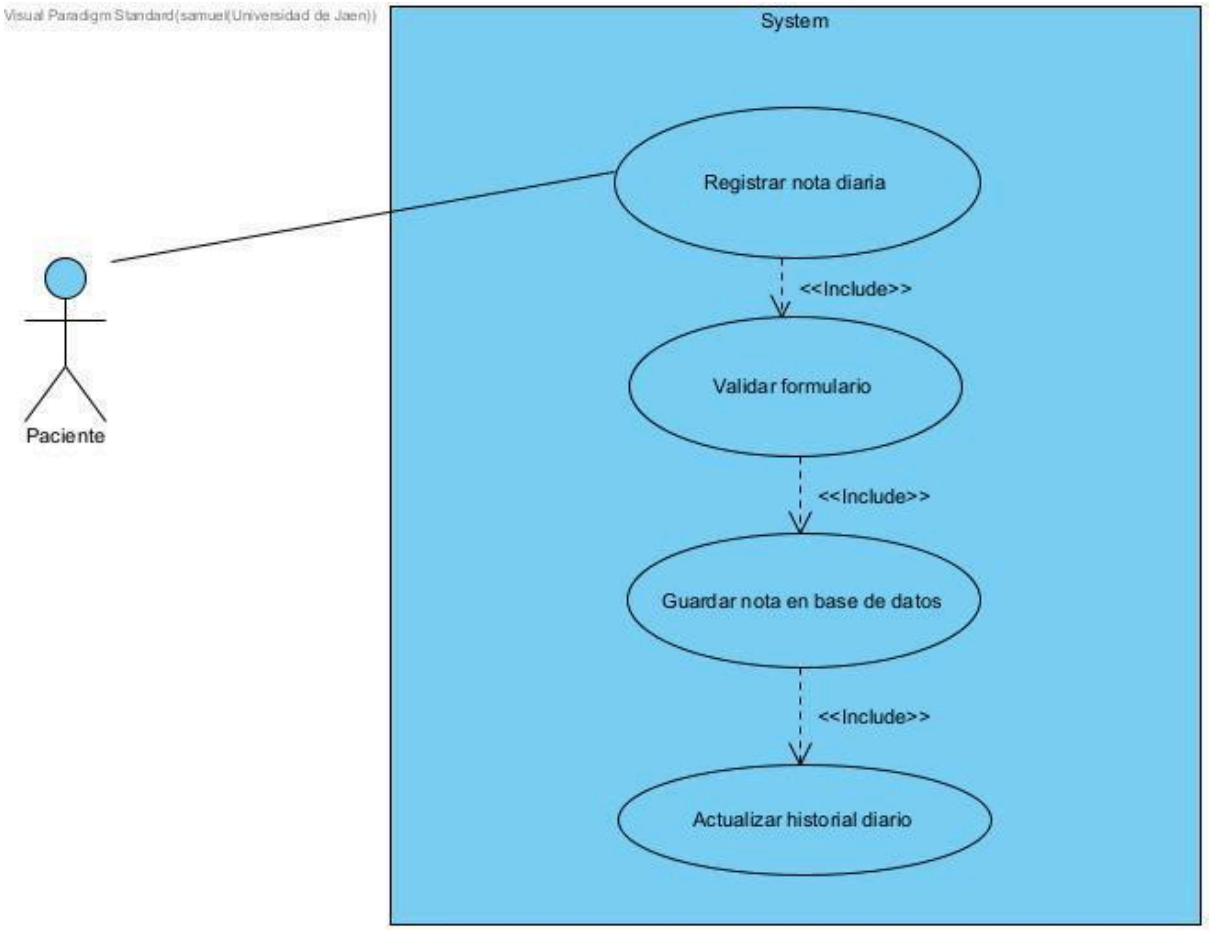


Figura 43: Caso de uso registrar nota diaria

4.3.3.2. Caso de uso visualizar analíticas

Nombre	Visualizar analíticas (Bioquímicos)
Actores	Paciente
Descripción	El paciente accede a la sección de “Analíticas” para revisar los resultados de sus pruebas de laboratorio. El sistema muestra dichas analíticas organizadas por fecha y parámetro, permitiendo así un seguimiento histórico.
Flujo Principal	1. El paciente inicia sesión y se dirige a la sección analíticas dentro de la aplicación. 2. El sistema realiza una consulta a la base de datos para obtener los registros analíticos correspondientes al paciente, ordenados por la fecha. 3. El paciente visualiza la lista de resultados con detalles como fecha de la prueba, nombre del parámetro y su valor.
Flujos alternativos / Excepciones	• A1: No existen registros 1. El paciente no tiene analíticas cargadas todavía. 2. El sistema muestra un mensaje: “No hay datos de analíticas disponibles”. 3. El paciente se queda sin visualización, a la espera de futuras pruebas.
Precondiciones	• El paciente ya tiene una cuenta en el sistema y está logueado.
Postcondiciones	• El paciente puede usar esta información para un seguimiento de su evolución o compartirla con su médico/familiar.

Tabla 12: Caso de uso visualizar analíticas

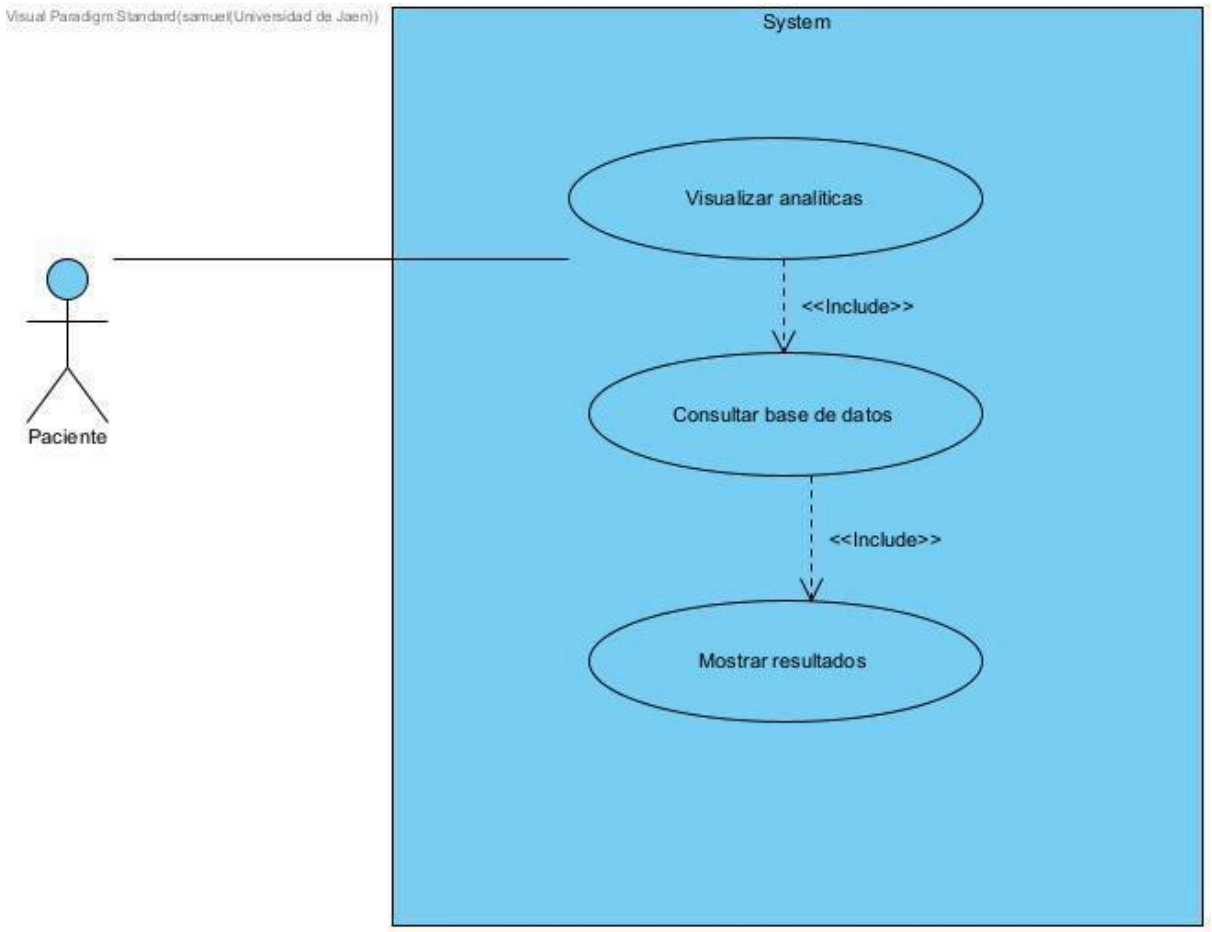


Figura 44: Caso de uso visualizar analíticas

4.3.3.3. Caso de uso visualizar diagnósticos

Nombre	Visualizar diagnósticos
Actores	Paciente
Descripción	El paciente accede a la sección de diagnósticos para consultar los registros médicos emitidos por uno o varios médicos en diferentes fechas. El sistema muestra la información organizada de acuerdo a la fecha y descripción, facilitando la revisión histórica de diagnósticos.
Flujo Principal	1. El paciente inicia sesión y se dirige a la sección “Diagnósticos”. 2. El sistema obtiene de la base de datos los diagnósticos asociados a ese paciente, ordenados por fecha. 3. El paciente ve una vista detallada de diagnósticos, mostrando: ○ Fecha de emisión. ○ Descripción del diagnóstico. ○ Nombre del médico que lo emitió.
Flujos alternativos / Excepciones	● A1: Sin Diagnósticos 1. En caso de que el paciente no tenga diagnósticos registrados. 2. El sistema muestra el mensaje “No hay diagnósticos disponibles”. 3. El paciente vuelve a la pantalla principal o espera a que un médico emita un nuevo diagnóstico.
Precondiciones	● El paciente ya tiene una cuenta en el sistema y está logueado.

Postcondiciones	• El paciente puede usar esta información para un seguimiento de su evolución o compartirla con su médico/familiar.

Tabla 13: Caso de uso visualizar diagnóstico

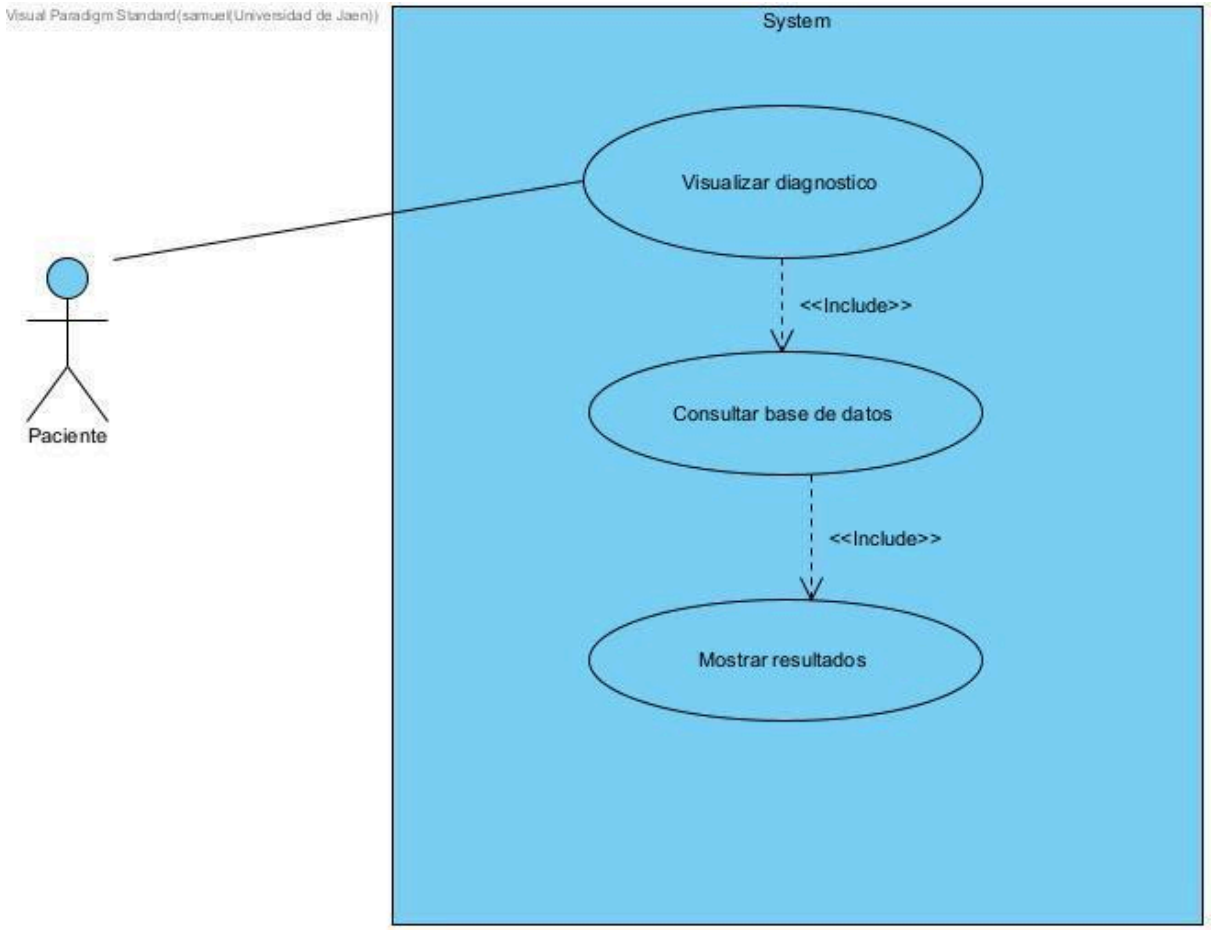


Figura 45: Caso de uso visualizar diagnóstico

4.3.3.4. Caso de uso solicitar cambio de médico

Nombre	Solicitar cambio de médico
Actores	Paciente
Descripción	El paciente, que actualmente tiene asignado un médico en la plataforma, desea cambiar a otro profesional. Mediante este proceso, el paciente solicita su cambio y se envía una solicitud al administrador.
Flujo Principal	1. El paciente inicia sesión en la plataforma y navega hasta la página de configuración. 2. El sistema muestra al paciente el nombre y correo del médico actual y un botón de cambiar de médico. 3. El paciente le da al botón. 4. El sistema registra la "Solicitud de cambio de médico" en la base de datos. 5. El paciente recibe un mensaje de confirmación, indicando que la solicitud ha sido enviada.
Flujos alternativos / Excepciones	• A3: Solicitud de cambio duplicada 1. El paciente ya tiene una solicitud previa "pendiente" con ese mismo médico. 2. El sistema alerta de que no se puede 3. El paciente no puede enviar la solicitud duplicada.
Precondiciones	• El paciente tiene una cuenta y está logueado. • El paciente tiene un médico asignado actualmente y desea cambiarlo.
Postcondiciones	• El paciente queda a la espera de la aprobación o rechazo por parte del administrador

Tabla 14: Caso de uso solicitar cambio de médico

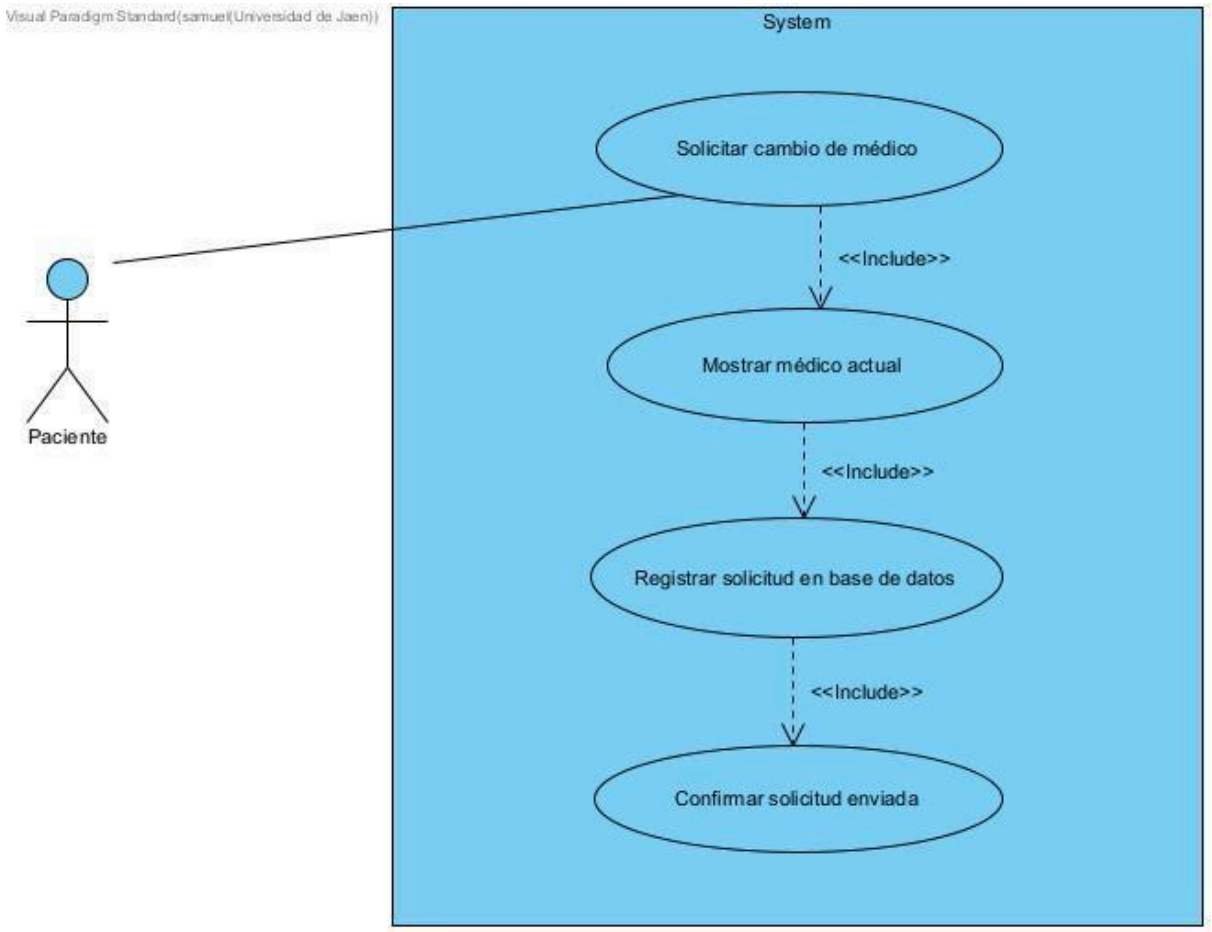


Figura 46: Caso de uso solicitar cambio de médico

4.3.3.5. Caso de uso administrar familiares

Nombre	Administrar familiares
Actores	Paciente
Descripción	El paciente gestiona a las personas que pueden acceder a su información médica o colaborar en su seguimiento. Esto incluye aceptar/rechazar solicitudes provenientes de ellos y desvincular a familiares existentes.
Flujo Principal	1. El paciente inicia sesión y navega a la página de notificaciones.

	2. El sistema muestra una lista de las solicitudes pendientes. 3. El paciente elige. 4. El paciente navega hasta la página de configuración al apartado Usuarios vinculados a ti. 5. Si el usuario lo desea puede desvincularlo para que no tenga acceso a su perfil de datos.
Flujos alternativos / Excepciones	-
Precondiciones	• El paciente tiene una cuenta y está logueado.
Postcondiciones	• La lista de familiares del paciente queda actualizada en el sistema.

Tabla 15: Caso de uso administrar familiares

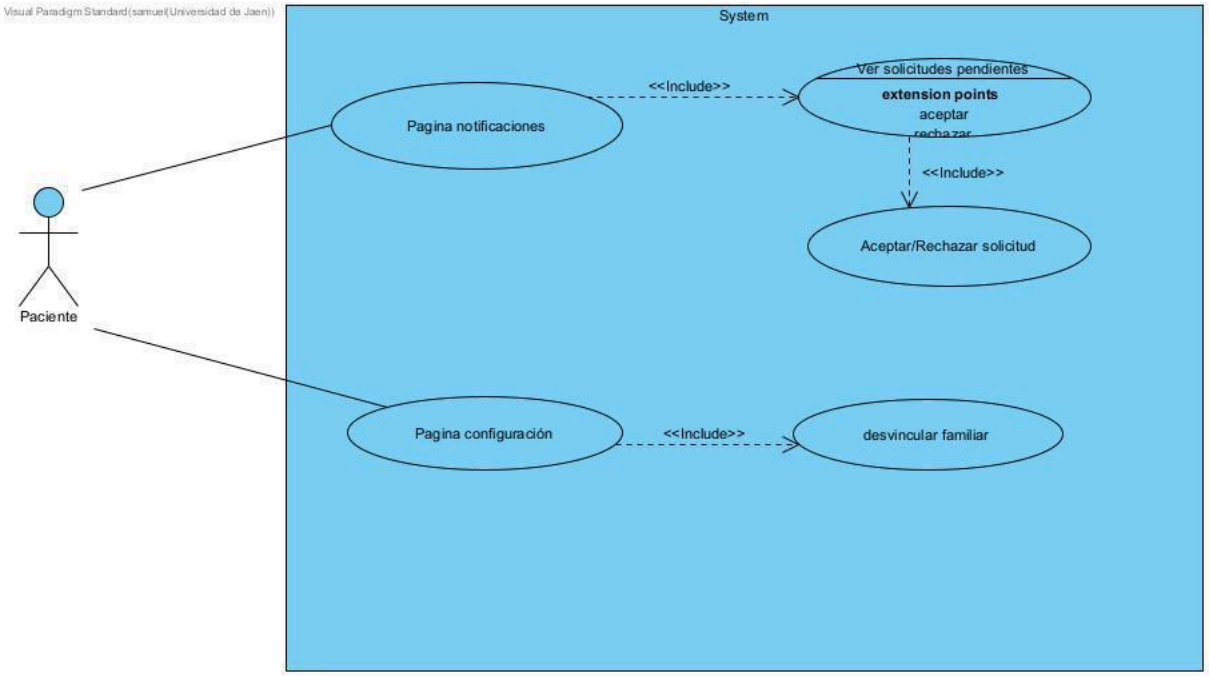


Figura 47: Caso de uso administrar familiares

4.3.4. Casos de uso para el rol médico

4.3.4.1. Caso de uso Consultar pacientes asignados

Nombre	Consultar pacientes asignados
Actores	Médico
Descripción	El médico, previamente autenticado, accede a la sección de la aplicación donde se listan todos los pacientes bajo su supervisión. El sistema muestra los datos fundamentales de cada paciente, facilitando el acceso posterior a información detallada (glucosas, notas, diagnósticos, etc.).
Flujo Principal	1. El médico inicia sesión en la aplicación y navega a la página de pacientes. 2. El sistema realiza una consulta a la base de datos, filtrando aquellos registros de pacientes que tienen asignado a este médico. 3. El sistema muestra una lista de los pacientes, donde pueden figurar: a. Nombre y apellidos del paciente. 4. (Opcional) El médico puede hacer clic en un paciente específico para ver su perfil completo o detalles clínicos adicionales. 5. El médico revisa la lista y, una vez finalizada la consulta, regresa al menú principal o a otra página.
Flujos alternativos / Excepciones	● A1: No hay pacientes asignados 1. Si por alguna razón el médico no tiene pacientes aún, el sistema muestra “No hay pacientes asignados”. 2. El médico se queda sin información que revisar en esta sección hasta que se asignen pacientes.

	3. El médico no puede ver datos hasta que se le asignen pacientes. ● A2: Filtros / Búsqueda 1. El médico decide filtrar por nombre y apellidos. 2. El sistema filtra o busca en la lista de pacientes asignados y presenta los resultados coincidentes. 3. Se retoma el flujo principal en la vista filtrada.
Precondiciones	● El médico tiene cuenta en el sistema y sesión iniciada.
Postcondiciones	● El médico ha visualizado la lista de sus pacientes asignados, pudiendo o no abrir la ficha de un paciente en particular.

Tabla 16: Caso de uso consultar pacientes asignados

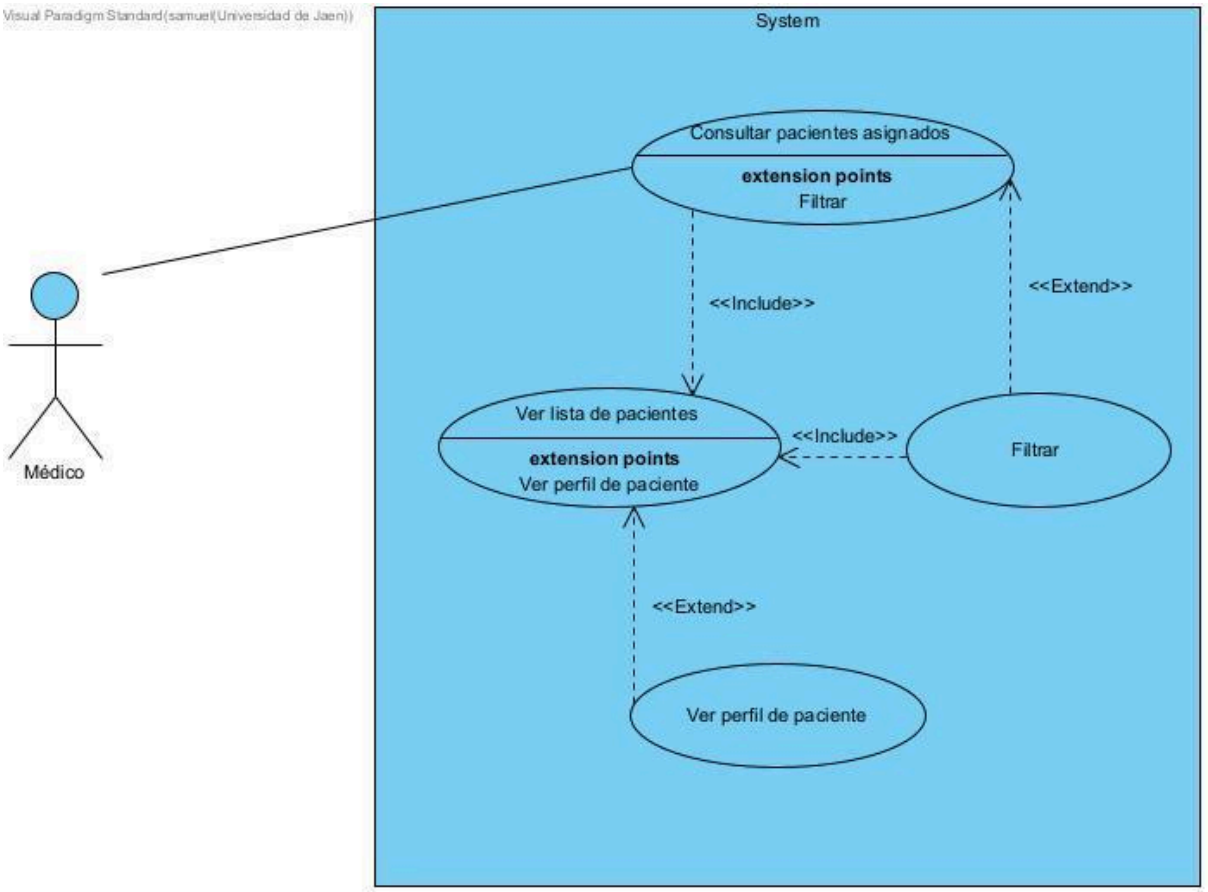


Figura 48: Caso de uso consultar pacientes asignados

4.3.4.2. Caso de uso emitir diagnóstico

Nombre	Emitir diagnóstico
Actores	Médico
Descripción	El médico, tras identificar al paciente dentro de su lista de pacientes asignados, crea un nuevo registro de diagnóstico con fecha y descripción. El sistema almacena dicha información, asociándose tanto al médico que lo emite como al paciente que lo recibe.
Flujo Principal	1. El médico inicia sesión en la aplicación y accede a la página de pacientes. 2. El sistema muestra la lista de pacientes asignados al médico, el médico selecciona uno para visualizar su información. 3. El médico elige la acción “Añadir Diagnóstico”. 4. El sistema presenta un formulario que incluye: a. Campo “Descripción del diagnóstico”, donde el médico detalla la evaluación. 5. El médico completa los campos requeridos y confirma la acción presionando “Guardar”. 6. El sistema guarda el nuevo diagnóstico en la base de datos, relacionándolo con: a. El id_medico (el médico actual). b. El id_paciente (el paciente elegido). c. La fecha y descripción introducidas. 7. El sistema confirma la creación exitosa del diagnóstico y, opcionalmente, redirige a la vista del historial clínico del paciente, mostrando el nuevo diagnóstico agregado.

Flujos alternativos / Excepciones	• A1: Campo descripción incorrecto 1. El médico no rellena bien la descripción. 2. El médico antes de salir del perfil de usuario puede modificar la descripción del diagnóstico realizado. 3. El médico cambia la descripción. 4. El médico le da al botón de guardar. 5. El sistema modifica de la base de datos el diagnóstico.
Precondiciones	• El médico está autenticado en el sistema. • Tiene asignado al paciente correspondiente.
Postcondiciones	• El nuevo diagnóstico queda almacenado con su fecha y descripción, asociado tanto al médico como al paciente. • El paciente o los familiares con acceso podrán ver este diagnóstico en su historial cuando lo consulten.

Tabla 17: Caso de uso emitir diagnóstico

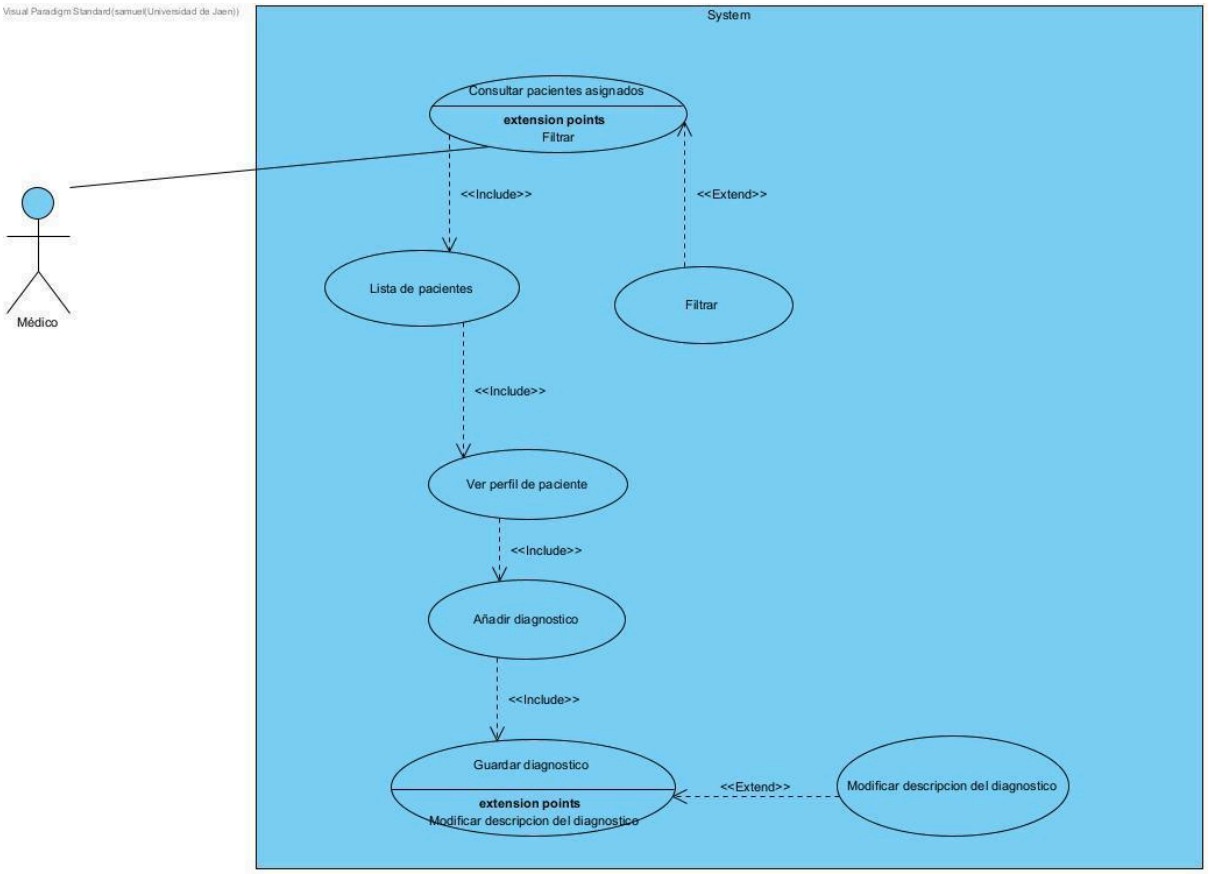


Figura 49: Caso de uso emitir diagnóstico

4.3.5. Casos de uso para el rol familiar

4.3.5.1. Caso de uso solicitar vinculación con paciente

Nombre	Solicitar vinculación con paciente
Actores	Familiar
Descripción	El familiar, registrado en la plataforma, solicita permiso al paciente para poder acceder a su información médica y colaborar en su seguimiento. El sistema registra dicha solicitud como “pendiente”, y el paciente la aceptará o la rechazará posteriormente.
Flujo Principal	1. El familiar inicia sesión en el sistema y navega a la página de familiar. 2. El familiar introduce el correo del paciente y confirma la acción de enviar solicitud de vinculación. 3. El sistema valida que el usuario (paciente) exista en la base de datos y que el familiar no esté ya vinculado. 4. El sistema crea una solicitud de vinculación en estado “pendiente” y envía una notificación al paciente. 5. El familiar recibe la confirmación de que la solicitud fue enviada correctamente y queda a la espera de la respuesta del paciente.
Flujos alternativos / Excepciones	● A1: Paciente no encontrado 1. El familiar ingresa un correo que no corresponde a ningún usuario o a alguien que no es paciente. 2. El sistema notifica “No existe ningún paciente con ese identificador”. 3. El familiar corrige el correo o desiste. ● A2: Solicitud ya existente 1. El familiar intenta enviar una solicitud para un

	paciente con el que ya está vinculado o con el que tiene una petición pendiente. 2. El sistema muestra un aviso: “Ya tienes una vinculación pendiente”. 3. El familiar no puede duplicar la solicitud.
Precondiciones	• El familiar está registrado y logueado en el sistema. • El paciente al que se desea vincular existe como usuario con rol de paciente.
Postcondiciones	• El sistema deja registrada la solicitud con estado “pendiente” en la base de datos, asociando el id_familiar con el id_paciente. • El paciente recibirá una notificación o mensaje y podrá aceptar o rechazar la vinculación.

Tabla 18: Caso de uso solicitar vinculación con paciente

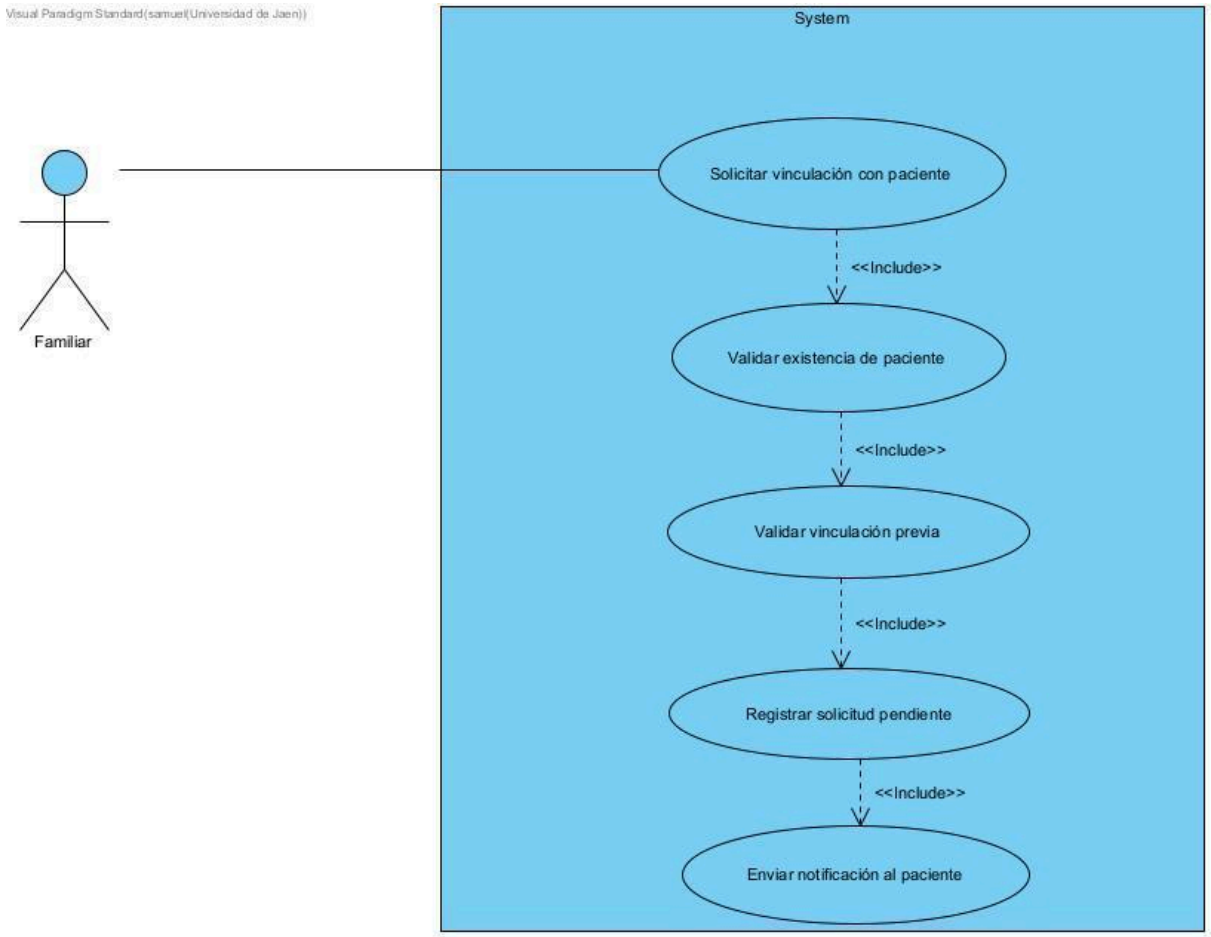


Figura 50: Caso de uso solicitar vinculación con paciente

4.3.5.2. Caso de uso ver información del paciente

Nombre	Ver información del paciente
Actores	Familiar
Descripción	El familiar, tras haber sido vinculado correctamente con el paciente, puede acceder a los datos clínicos y registros de dicho paciente para mantenerse al tanto de su estado de salud y brindarle apoyo.
Flujo Principal	1. El familiar inicia sesión en la aplicación y navega a la página de pacientes.

	2. El sistema muestra la lista de pacientes con los que el familiar está vinculado. 3. El familiar selecciona a un paciente específico para ver su información. 4. El sistema consulta la base de datos y obtiene los datos del paciente. 5. El sistema presenta dichos datos en pantalla, pudiendo incluir: a. Perfil básico del paciente (nombre, apellidos, fecha de nacimiento, etc.). b. Historial de glucosa con fechas y valores. c. Notas diarias o semanales introducidas por el paciente. d. Diagnósticos realizados por el médico, si corresponde. 6. El familiar revisa la información y, una vez finalizada la consulta, puede navegar a otras páginas.
Flujos alternativos / Excepciones	● A1: No existen datos 1. El paciente aún no ha registrado notas ni mediciones de glucosa o no posee diagnósticos. 2. El sistema muestra “No hay datos disponibles” en las secciones correspondientes. 3. El familiar no ve contenido relevante en esa área. ● A2: Pérdida de la vinculación 1. Si, por algún motivo, el paciente revoca el acceso, al intentar ver la información, el sistema podría negar el acceso. 2. El familiar no puede seguir consultando datos hasta que se restablezca la vinculación.
Precondiciones	● El familiar cuenta con una sesión iniciada.

	• Existe una vinculación aceptada entre el familiar y el paciente en la base de datos, lo que confiere los permisos necesarios para ver los datos.
Postcondiciones	• El familiar visualiza la información clínica y/o de seguimiento del paciente, lo que le permite mantenerse al tanto de su evolución.

Tabla 19: Caso de uso ver información del paciente

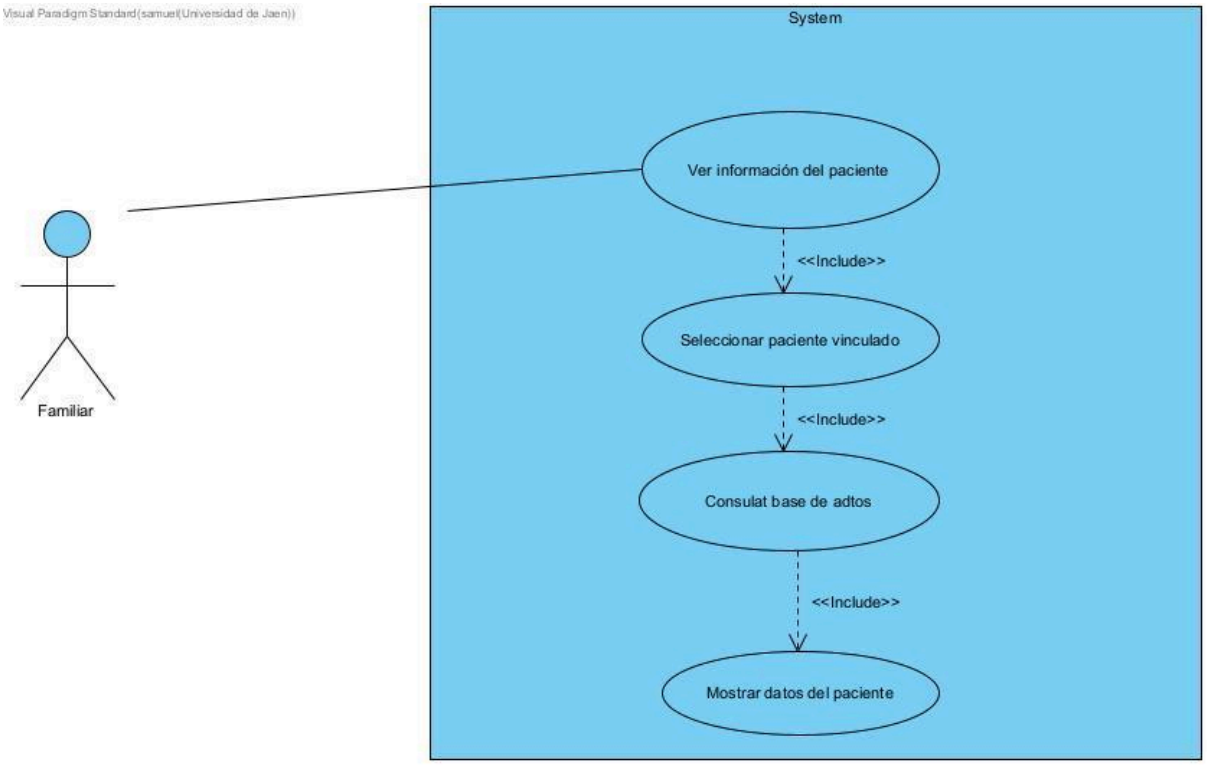


Figura 51: Caso de uso ver información del paciente

4.3.5.3. Caso de uso desvincularse del paciente

Nombre	Desvincularse del paciente
Actores	Familiar
Descripción	El familiar, que previamente se había vinculado a un paciente, decide revocar esa relación. De este modo, deja de tener acceso a la información clínica de dicho paciente.
Flujo Principal	1. El familiar inicia sesión y navega a la página de configuración, a la sección de familiares. 2. El sistema muestra la lista de pacientes a los que el familiar tiene acceso, resultado de una vinculación aceptada. 3. El familiar selecciona al paciente del que desea desvincularse y elige la acción “Desvincular”. 4. El sistema elimina la relación en la base de datos. 5. El familiar ya no aparece en la lista de usuarios vinculados al paciente, y viceversa. 6. El sistema presenta un mensaje de confirmación: “Desvinculación realizada con éxito”.
Flujos alternativos / Excepciones	-
Precondiciones	• El familiar tiene una cuenta y está logueado. • Existe una vinculación activa entre el familiar y el paciente.
Postcondiciones	• El familiar pierde el acceso a la información del paciente: no podrá ver glucemias, notas, diagnósticos, etc. • El sistema actualiza la relación en la base de datos.

Tabla 20: Caso de uso desvincularse del paciente

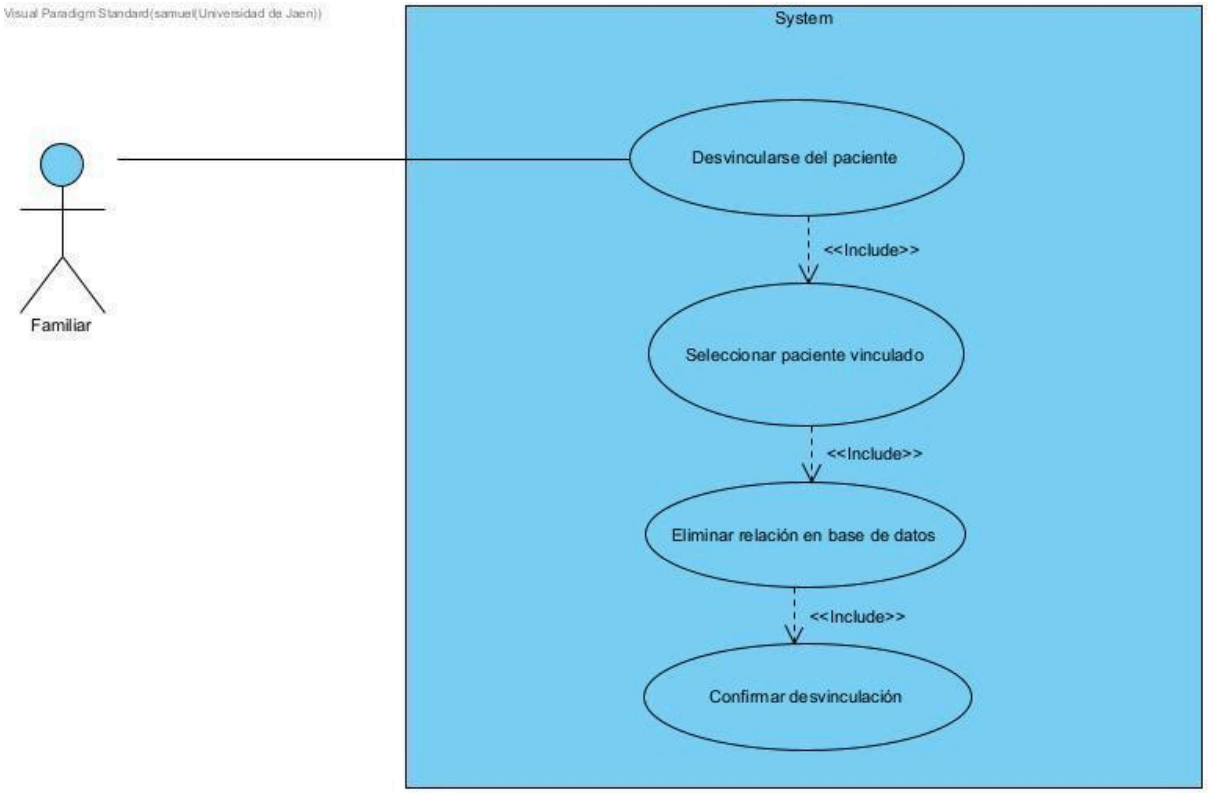


Figura 52: Caso de uso desvincularse del paciente

4.3.6. Casos de uso para el rol administrador

4.3.6.1. Caso de uso crear usuario

Nombre	Crear usuarios
Actores	Administrador
Descripción	El administrador accede a una sección de la aplicación que le permite crear usuarios (pacientes, médicos, familiares).
Flujo Principal	1. El administrador inicia sesión en la aplicación. 2. El administrador ingresa datos básicos (nombre, apellidos, fecha de nacimiento, sexo, email, contraseña, rol). 3. El sistema valida que no exista ya una cuenta con el mismo correo y, si es correcto, crea el registro en la base de datos. 4. Se muestra un mensaje “Usuario creado correctamente”. 5. El administrador puede repetir estos pasos para varios usuarios o regresar al menú principal cuando haya terminado.
Flujos alternativos / Excepciones	• A1: Datos inválidos o faltantes en la creación/edición 1. El sistema detecta que un campo requerido está vacío o que el correo no es válido. 2. El administrador corrige y reintentar.
Precondiciones	• El administrador tiene una cuenta con privilegios para gestionar usuarios. • El sistema está disponible y la base de datos es

	accesible.
Postcondiciones	• Si el administrador crea un usuario, este queda registrado con el rol y datos indicados.

Tabla 21: caso de uso crear usuario

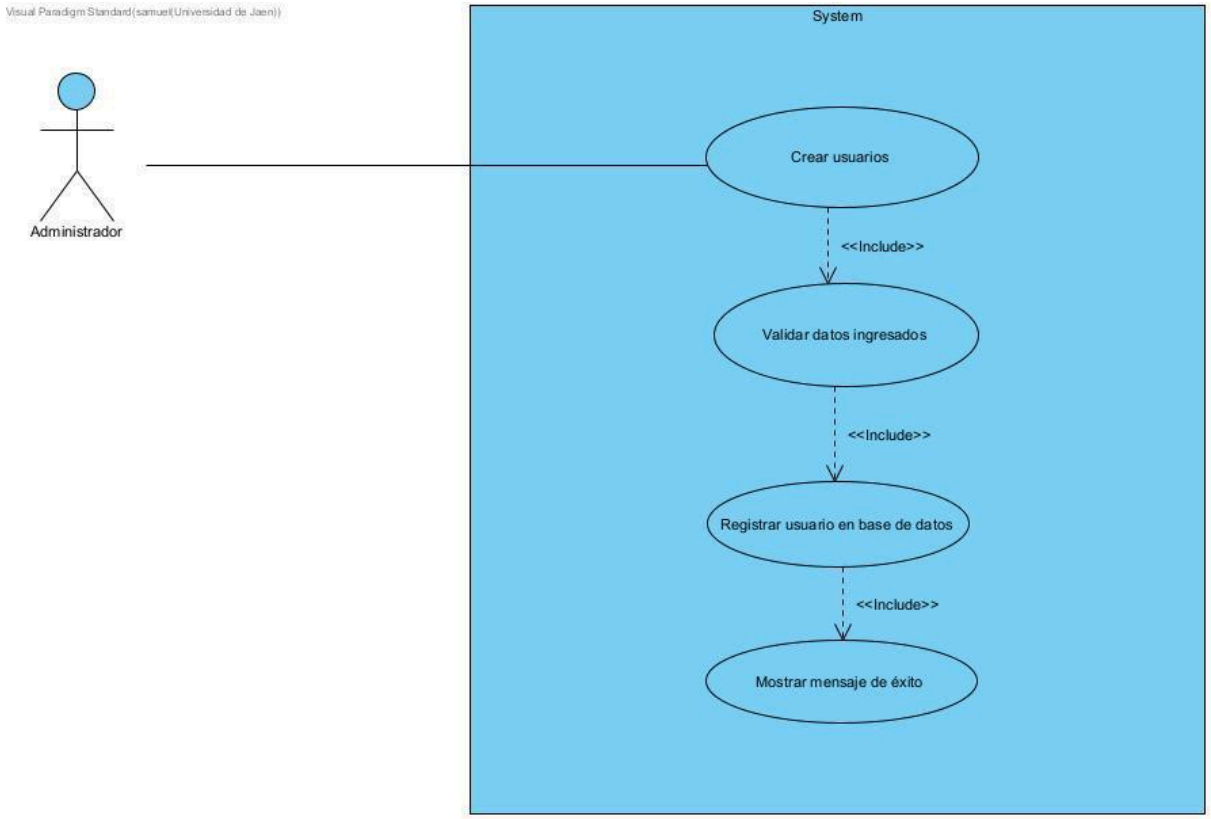


Figura 53: Caso de uso crear usuario

4.3.6.2. Caso de uso administrar solicitudes de cambio de médico

Nombre	Administrar solicitudes de cambio de médico
Actores	Administrador
Descripción	El administrador accede a la sección donde se listan las solicitudes que los pacientes han realizado para cambiar de médico asignado. Aquí puede aprobar o rechazar cada solicitud, definiendo así el nuevo vínculo Médico–Paciente o dejando sin efecto la petición.
Flujo Principal	1. El administrador inicia sesión y navega a la sección solicitudes pendientes. 2. El sistema consulta la base de datos y obtiene todas las peticiones pendientes de cambio, con datos como: 2.1. Nombre y apellidos del paciente. 2.2. Nombre médico solicitado. 3. El administrador ve la lista de solicitudes en estado pendiente. 4. El administrador selecciona una solicitud específica y elige una acción: 4.1. Aceptar: El sistema modifica la relación en la base de datos para asignar el nuevo médico al paciente. 4.2. Rechazar: El sistema deja la solicitud en estado “rechazada” y mantiene el médico actual del paciente. 5. El sistema registra la acción (quién, cuándo) y actualiza el estado de la solicitud (aprobada o rechazada). 6. El administrador recibe una confirmación de que la acción se procesó correctamente.
Flujos alternativos / Excepciones	• A1: No hay solicitudes pendientes 1. El sistema detecta que no existen peticiones nuevas para cambiar de médico.

	2. Muestra un mensaje “No hay solicitudes pendientes en este momento”. 3. El administrador finaliza su revisión sin acciones.
Precondiciones	• El administrador está autenticado y tiene los permisos apropiados para gestionar solicitudes de cambio de médico. • Existen solicitudes pendientes en la base de datos (salvo el caso alternativo A1) que esperan la resolución.
Postcondiciones	• Las solicitudes pasan al estado de aceptadas o rechazadas, según la acción del administrador. • Si se aprueba la solicitud, el paciente en cuestión queda asignado al nuevo médico.

Tabla 22: Caso de uso administrar solicitudes de cambio de médico

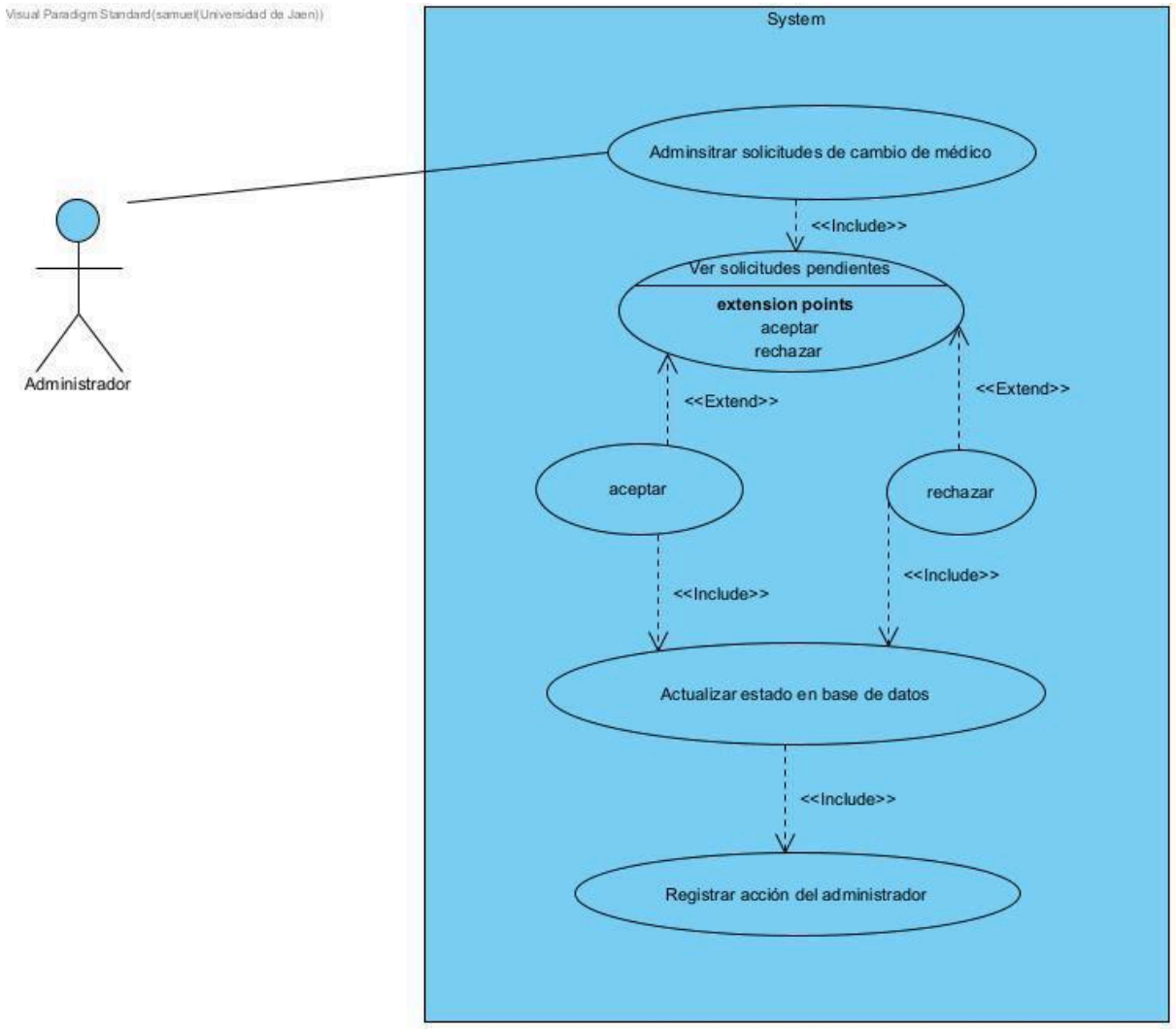


Figura 54: Caso de uso administrar solicitudes de cambio de médico

4.4. Diagrama de secuencias

En este apartado se van a ver distintos diagramas de secuencias, para distintos roles, además de una explicación del flujo de operaciones que se realiza en cada diagrama.

4.4.1. Inicio de sesión

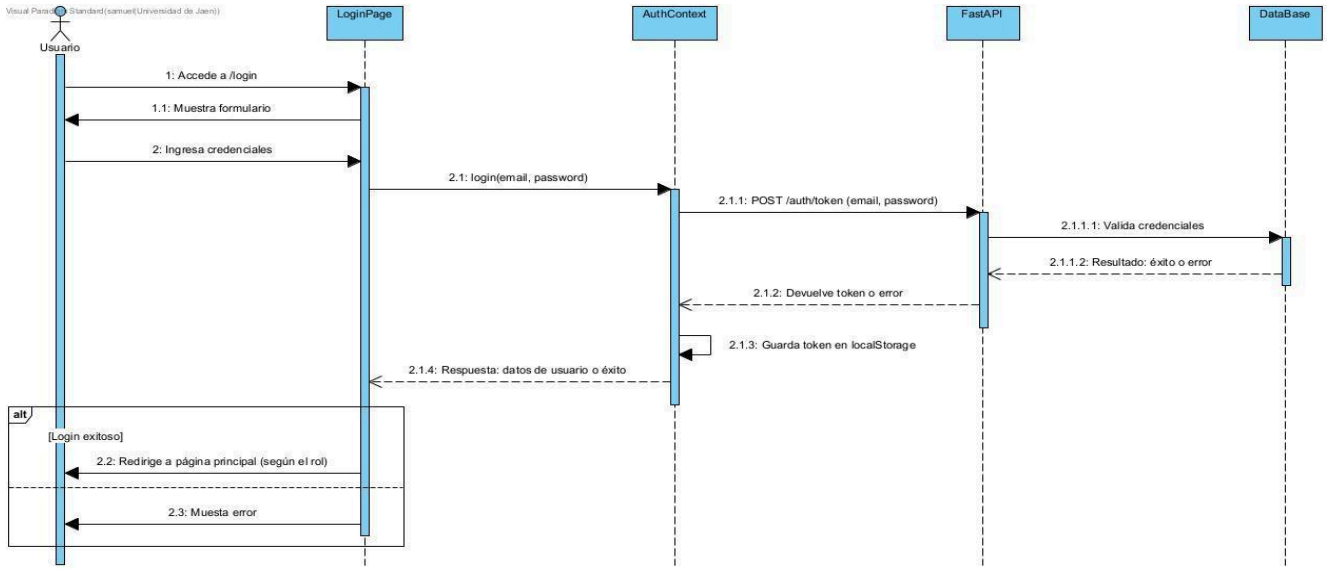


Figura 55: Diagrama de secuencia login

En el diagrama se distinguen los siguientes participantes:

- Actor: Usuario (sin sesión iniciada).
- Front-end (LoginPage): la página de Next.js donde el usuario ingresa credenciales.
- AuthContext: el contexto de autenticación que centraliza las funciones login() y maneja el token en el front-end.
- Back-end (FastAPI): el servidor que recibe las credenciales, valida con la base de datos y emite el token.

El usuario accede a la página de inicio de sesión /login, donde se le muestra un formulario para ingresar su correo y contraseña. Tras completar y enviar los datos, la página llama a la función login() del AuthContext, que realiza una solicitud POST al backend (FastAPI) en el endpoint /auth/token. El backend verifica las credenciales en la base de datos y, si son válidas, genera un token JWT y lo devuelve al AuthContext; en caso contrario, responde con un error.

El AuthContext guarda el token en localStorage y devuelve una respuesta a la LoginPage. Si el inicio de sesión es exitoso, la página redirige al usuario a la página correspondiente según su rol; si las credenciales son inválidas, muestra un mensaje de error como "Usuario o contraseña incorrectos". Este flujo gestiona de forma completa la autenticación y el manejo de errores.

4.4.2. Registrar nota diaria (Paciente)

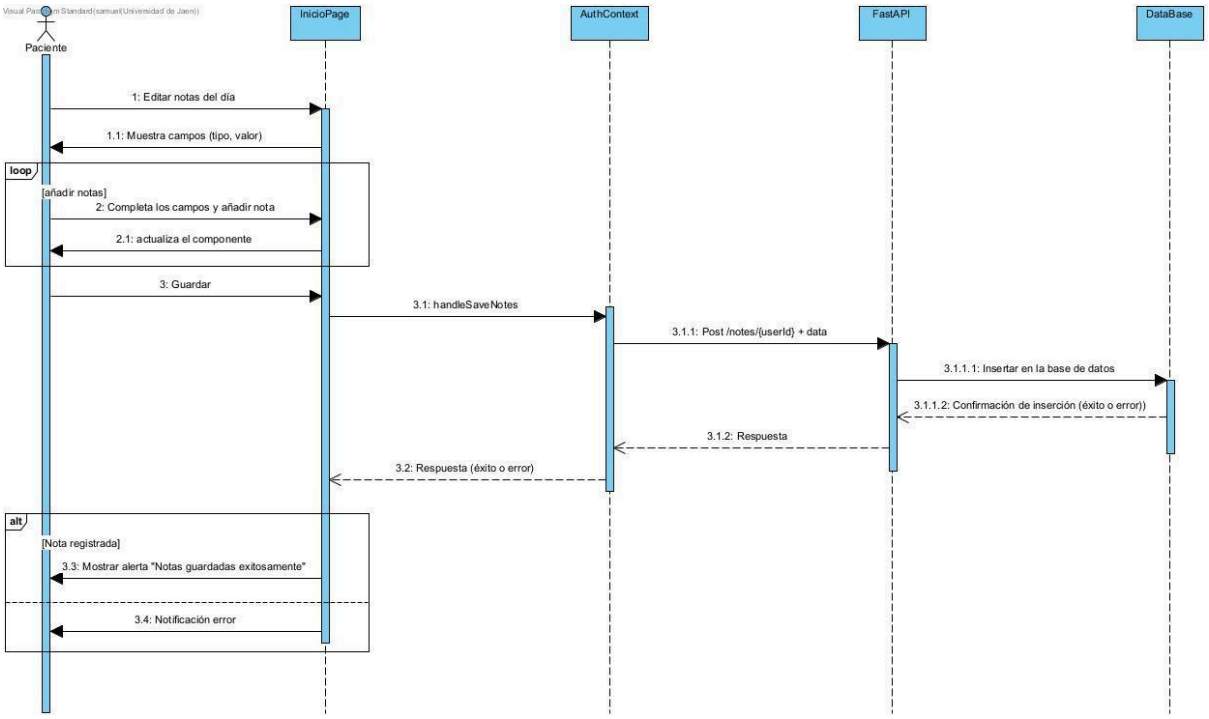


Figura 56: Diagrama de secuencia registrar nota diaria

En el diagrama se distinguen los siguientes participantes:

- Actor: Usuario con rol paciente.
- Front-end (InicioPage): la página inicial del paciente, donde aparece la glucosa y las notas del día.
- AuthContext: el contexto de autenticación que centraliza las funciones de la aplicación.

- Back-end (FastAPI): el servidor que recibe los datos de las notas, para poder agregarlo a la base de datos.

El paciente inicia el flujo abriendo la página de Notas Diarias, donde la interfaz muestra los campos necesarios para registrar una nota, como el tipo (por ejemplo, "comida" o "actividad"), su descripción/valor. Una vez que el paciente completa esta información y añade todas las notas deseadas, pulsa el botón para guardar, el front-end (NotasPage) llama a la función handleSaveNotes() del AuthContext. En este contexto, utilizando un token de autenticación previamente almacenado, realiza una petición POST al endpoint del backend /notes/{userId}, enviando los datos de la nota junto con el identificador del usuario.

En el backend, se procesa la solicitud y realiza un comando INSERT en la base de datos en la tabla correspondiente a "notas", guardando tanto la información básica (fecha, identificador del paciente) y en la tabla “nota_detalles” como los detalles específicos de la nota como el tipo y el valor. Tras completar la inserción, la base de datos responde al backend con una confirmación (indicando éxito o error), que el backend reenvía al AuthContext. Finalmente, el AuthContext devuelve el resultado a NotasPage, que presenta al paciente una alerta indicando si la nota se ha registrado correctamente o si ocurrió algún problema durante el proceso, cerrando así el ciclo de interacción entre los distintos componentes del sistema.

4.4.3. Solicitud cambio de médico

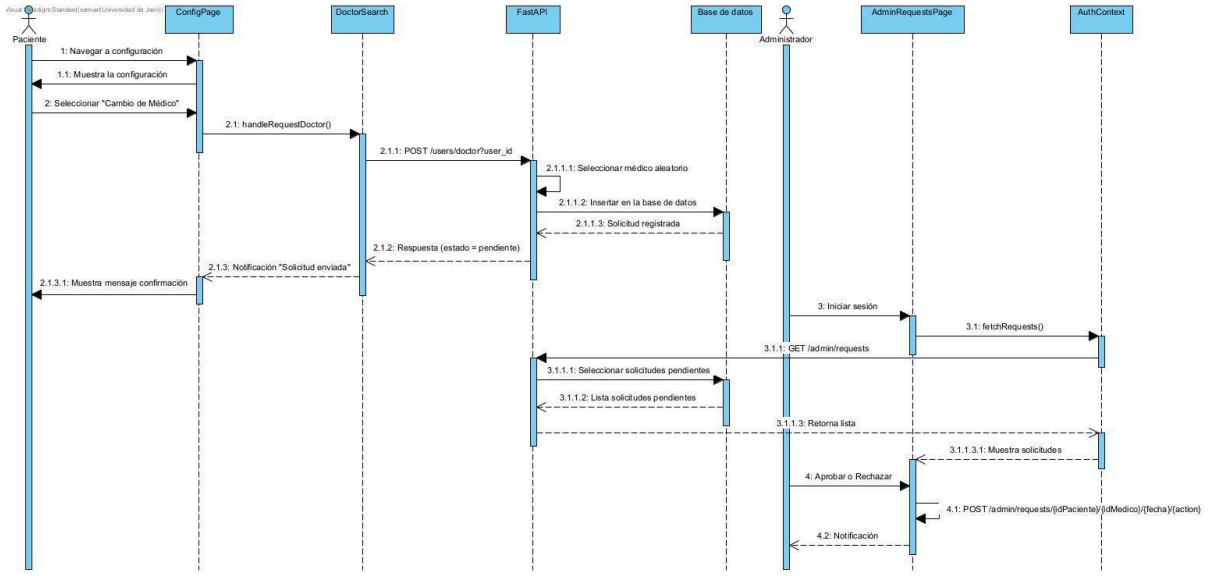


Figura 57: Diagrama de secuencia solicitud cambio de médico

En el diagrama se distinguen los siguientes participantes:

- Actor: Paciente y Administrador.
- ConfigPage: Página en el cual se agrupan los componentes de configuración del usuario.
- DoctorSearch: Componente en el cual muestra la información del doctor actual y tiene un botón para solicitar el cambio de médico.
- Back-end (FastAPI): el servidor que recibe la solicitud de cambio de médico para poder agregarlo a la base de datos.
- AdminRequestPage: Página de los usuarios que tienen el rol de administrador.
- AuthContext: el contexto de autenticación que centraliza las funciones de la aplicación.

El flujo comienza cuando el paciente accede a la página de configuración (ConfigPage) para realizar una solicitud de cambio de médico. Esta acción activa la función handleRequestDoctor, que maneja la autenticación y la comunicación con el back-end. El componente envía una petición POST al endpoint /users/doctor del

servidor FastAPI con el identificador del paciente. El back-end procesa la solicitud e inserta un nuevo registro en la tabla SolicitudCambio de la base de datos, dejándolo en estado “pendiente” y eligiendo aleatoriamente el nuevo médico. Una vez registrado, el back-end devuelve una respuesta que el componente reenvía a la página de configuración, donde se muestra al paciente un mensaje confirmando que la solicitud se ha enviado exitosamente.

Más tarde, el administrador accede a la AdminRequestsPage para gestionar estas solicitudes de cambio. La página solicita a AuthContext la lista de solicitudes pendientes mediante la función fetchRequests(). AuthContext envía una petición GET al endpoint /admin/requests del back-end, que consulta la base de datos y obtiene todas las solicitudes en estado “pendiente”. La lista de solicitudes es devuelta al AuthContext y finalmente mostrada en la AdminRequestsPage, donde el administrador puede revisarla.

Cuando el administrador elige aprobar o rechazar una solicitud específica, la AdminRequestsPage envía una petición POST al endpoint /admin/requests/{id_paciente}/{id_medico}/{fecha}/{action} del back-end, indicando la acción seleccionada. Si la solicitud es aprobada, el back-end actualiza la tabla SolicitudCambio en la base de datos, asignando el nuevo médico al paciente y marcando la solicitud como “completada”. Si es rechazada, la solicitud se actualiza a estado “rechazado”. En ambos casos, el resultado de la acción se devuelve a AuthContext, que informa a la AdminRequestsPage. Esta última notifica al administrador que la solicitud ha sido procesada, mostrando el estado final (aprobada o rechazada).

Este diagrama cubre todo el flujo: desde que el paciente solicita el cambio de médico hasta que el administrador resuelve la solicitud.

4.4.4. Desvincular familiares (Paciente)

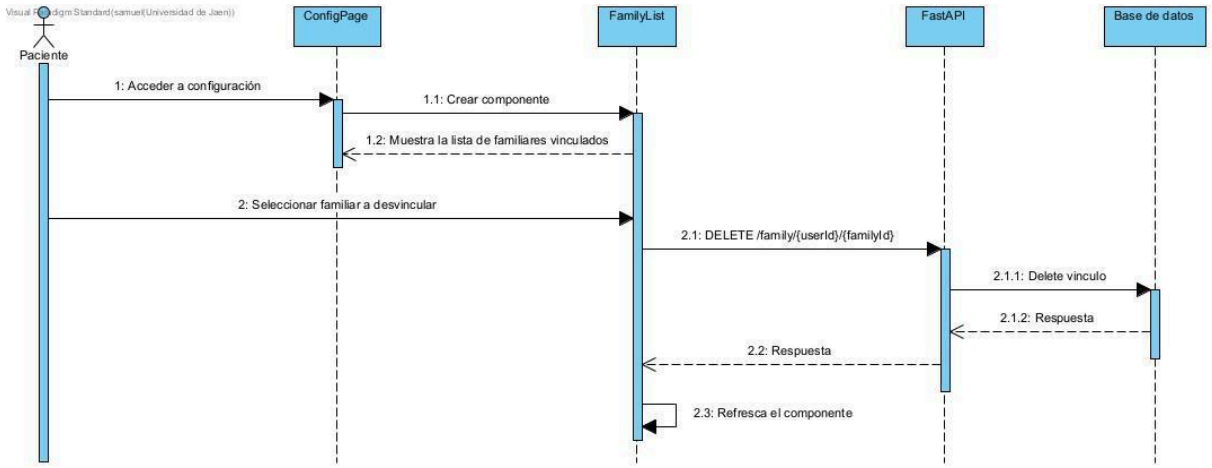


Figura 58: Diagrama de secuencia desvincular familiares

En el diagrama se distinguen los siguientes participantes:

- Actor: Usuario con rol paciente.
- ConfigPage: Página en el cual se agrupan los componentes de configuración del usuario.
- FamilyList: Componente en el que aparecen los familiares que tenemos asociados y podemos desvincularlos.
- Back-end (FastAPI): el servidor que recibe los datos de las notas, para poder agregarlo a la base de datos.

Suponiendo que el paciente ya está autenticado, accede a la sección de configuración (ConfigPage), que muestra la lista de familiares vinculados (FamilyList). El paciente selecciona al familiar que desea desvincular y el componente FamilyList realiza una petición DELETE al backend (FastAPI) en el endpoint /family/{userId}/{familyId}. El backend elimina el vínculo correspondiente en la base de datos, confirma la operación y envía una respuesta al componente. FamilyList refresca la lista para reflejar la desvinculación, mostrando al paciente que el familiar ya no tiene acceso a su información.

4.4.5. Emitir diagnóstico (Médico)

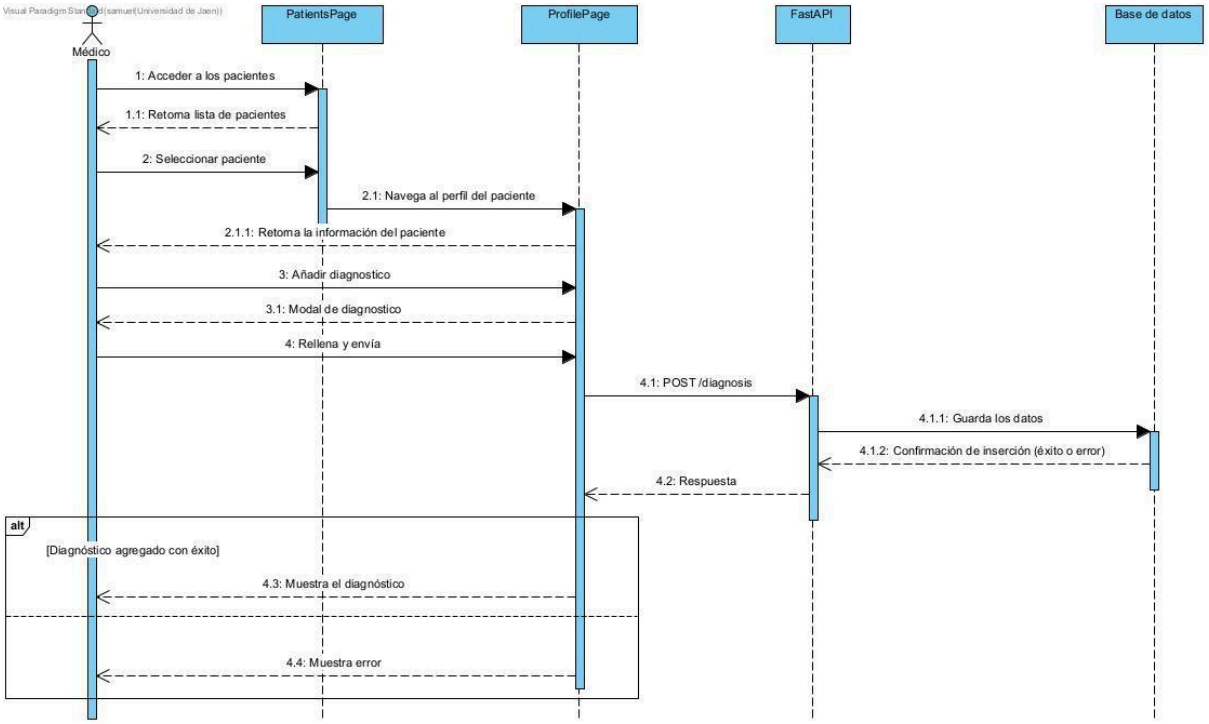


Figura 59: Diagrama de secuencia emitir diagnóstico

El médico inicia el flujo accediendo a la página de pacientes (PatientsPage) y estando autenticado, donde se muestra una lista de los pacientes asignados. Una vez selecciona a un paciente específico, la página navega al perfil del paciente (ProfilePage) y recupera su información para mostrarla en pantalla. Desde el perfil del paciente, el médico elige la opción de “Añadir diagnóstico”, lo que abre un formulario para completar los datos del diagnóstico.

El médico rellena el diagnóstico y envía el formulario. Esto activa una solicitud POST al endpoint /diagnosis del backend el cual procesa la solicitud y guarda la información en la base de datos. Tras completar la inserción, la base de datos responde con una confirmación (éxito o error), que el backend reenvía al ProfilePage. Si el diagnóstico se agrega con éxito, la página actualiza la información

del paciente mostrando el nuevo diagnóstico; en caso de error, se notifica al médico que el guardado no fue exitoso.

4.4.6. Vinculación familiar (Familiar)

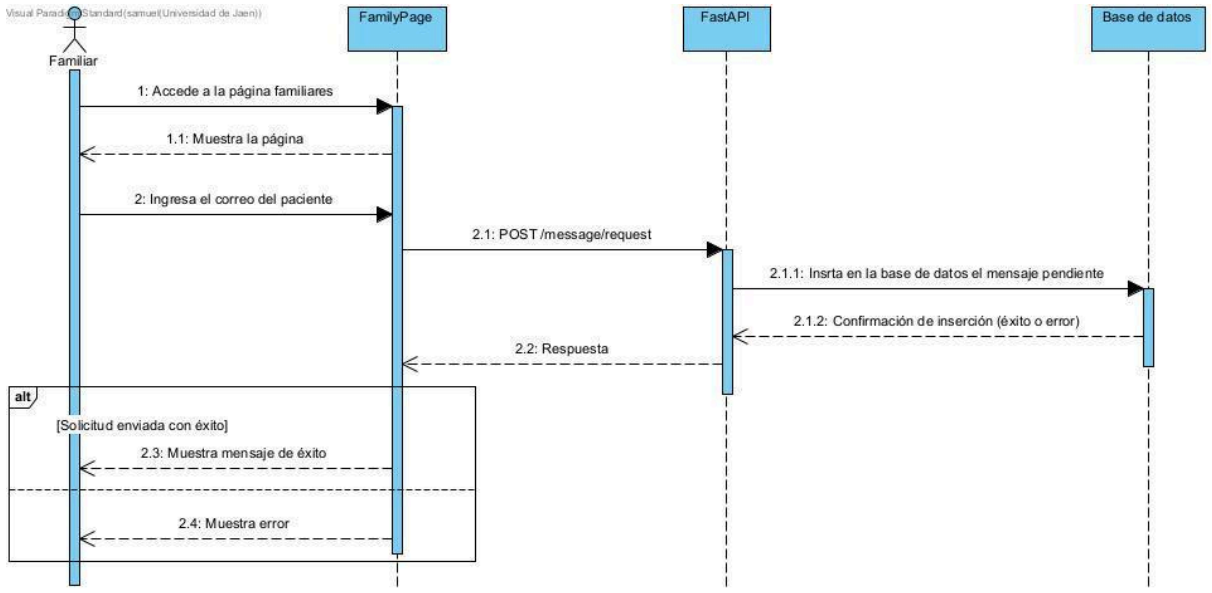


Figura 60: Diagrama de secuencia vinculación familiar

El familiar que ha iniciado la sesión, inicia el flujo accediendo a la FamilyPage, que le permite gestionar las solicitudes de vinculación. La página se carga y muestra la página. A continuación, el familiar ingresa el correo electrónico del paciente al que desea enviar la solicitud de vinculación.

Al confirmar, la FamilyPage realiza una petición POST al endpoint /message/request del backend, enviando los datos de la solicitud. El backend procesa la petición y registra un nuevo mensaje pendiente en la base de datos, asignándole el estado "pendiente". Una vez completado el registro, la base de datos responde al backend con una confirmación, que el backend reenvía a la FamilyPage.

Si la solicitud se envía correctamente, la FamilyPage muestra un mensaje de éxito al familiar. En caso de error, la página notifica al familiar que no se pudo completar el proceso.

4.5. Diseño de la interfaz

En esta sección se describe cómo se organiza y se presenta la interfaz de la aplicación para cada uno de los roles (paciente, médico, familiar, administrador) y cómo se ha llevado a cabo el diseño a nivel front-end. Se busca optimizar la usabilidad y la navegación entre pantallas para que cada usuario tenga un acceso rápido a sus funcionalidades específicas.

La interfaz se ha desarrollado con Next.js y React, estructurando el proyecto en páginas y componentes reutilizables en la carpeta (components/). Cada rol dispone de páginas adaptadas a sus necesidades:

- Paciente: Acceso a la vista principal (`/home`), análisis de glucosa (`/analytics`), diagnósticos (`/diagnosis`), mensajería (`/messages`), configuración (`/userConfiguration`).
- Médico: Página de pacientes (`/patients`), perfil detallado de cada uno (`/profile/[id]`) y configuración de usuario.
- Familiar: Sección para familiares (`/family`), perfil detallado de cada uno y configuración de usuario.
- Administrador: Panel de control (`/adminDashboard`) con herramientas para la creación de usuarios y la gestión de solicitudes de cambio de médico (`/admin/requests`).

Se ha priorizado un diseño responsive, de modo que la interfaz se adapte a diversos dispositivos. Además, se aplica una navegación consistente mediante el uso de un componente de barra de navegación (`NavBar`), que muestra los menús adecuados según el rol del usuario autenticado, y un pie de página (`Footer`). Para la gestión de estados y la autenticación en el front-end, se emplea un AuthContext

que facilita el manejo del token y el control de rutas protegidas con `ProtectedRoute`.

En los siguientes puntos veremos el diseño de la API como se distribuye y los endpoints creados. Además de los diseños de los esquemas de las distintas páginas que tienen los distintos tipos de usuario.

4.5.1. Diseño de la API REST

La aplicación expone una API REST desarrollada con FastAPI, organizada en routers o módulos que agrupan endpoints afines. Cada router define un prefijo y, dentro de él, varios endpoints (métodos HTTP) que responden a las necesidades de cada entidad. A continuación, se describen los endpoint de cada router, en el cual se verá su objetivo, los parámetros usados en la ruta, los parámetros entrantes del cuerpo de la solicitud y un flujo del proceso que hace internamente el endpoint:

4.5.1.1. Auth

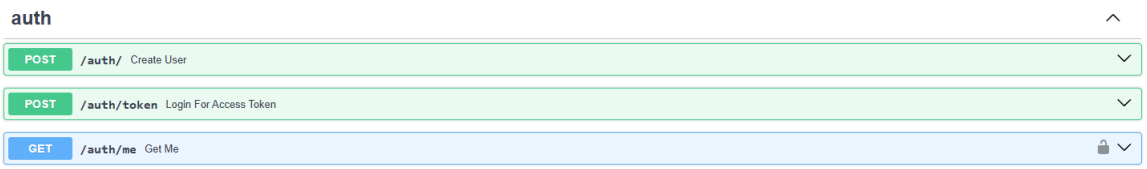


Figura 61: endpoints de la ruta auth

- POST /auth/token
  - Objetivo: Autenticar a un usuario mediante su email y contraseña, generando un token. Se utiliza la clase dada por fastAPI como 'OAth2PasswordRequestForm' para recibir las credenciales, es decir se utiliza para procesar las solicitudes de inicio de sesión y de esta manera se simplifica la gestión de los datos del formulario.
  - Entradas: Email y contraseña.
  - Flujo:
    - El endpoint recibe el email y contraseña.

- Se llama a otra función para que busque al usuario en la base de datos y se verifica el hash de la contraseña.
- Si la validación falla, se lanza un error HTTP 401 con el detalle "No se puede validar el usuario".
- Si es exitosa, se construye la lista de roles del usuario y se crea un token dándole la información del email, el id del usuario, los roles correspondientes y la fecha de expiración 20 de minutos.
- Por último devuelve un JSON que contiene el token de acceso y el tipo de token que será bearer, que indica que el cliente debe enviar el token como un Bearer Token en el encabezado Authorization de las solicitudes HTTP posteriores.

- GET /auth/me
  - Objetivo: Permite a un usuario autenticado conocer los datos de su cuenta.
  - Flujo:
    - Decodifica el token, verifica su validez y extrae el identificador del usuario, consultando la base de datos para obtener todos sus datos.
    - Si hay algún error en el proceso dará un error HTTP 401 y un detalle.
    - En caso de éxito, devuelve el usuario.

4.5.1.2. Admin

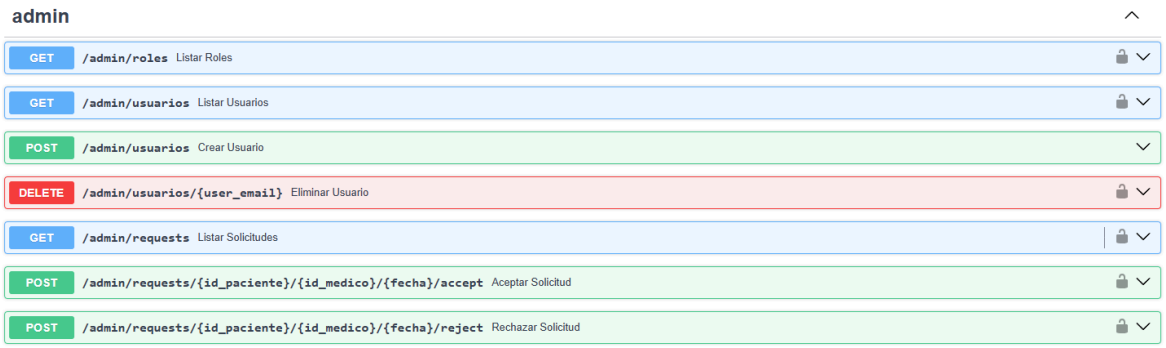


Figura 62: endpoints de la ruta admin

- GET /admin/roles
    - Objetivo: Proporcionar un listado de todos los roles existentes en la base de datos.
    - Flujo:
        - Consulta la tabla de roles, devolviendo todos los registros.
    - Restricción: Solo el administrador puede acceder, si no, se lanza un error HTTP 403 con la descripción “No tiene permisos para acceder a este recurso”.

- POST /admin/usuarios
    - Objetivo: Crear un nuevo usuario con un rol específico, y si el rol es paciente, asignarle un médico aleatoriamente.
    - Entradas: nombre, apellidos, sexo, fecha de nacimiento, email, contraseña y el identificador del rol.
    - Flujo:
        - Primero se verifica si el email ya existe en la tabla Usuario. Si existe, se lanza un error HTTP 400 (el servidor no puede procesar la petición debido a algo que es percibido como un error del cliente).
        - Busca en la tabla Rol si el identificador de rol proporcionado existe, y si no error HTTP 404.
        - Crea un objeto Usuario, hash de la contraseña mediante bcrypt y añade la relación con el rol en la lista de roles del usuario.
        - Si el rol es paciente, se elige un médico aleatoriamente de la tabla Medico. En caso de no existir médicos daría un error HTTP 404.
        - Por último en caso de éxito devuelve un mensaje “Usuario creado exitosamente”con HTTP 201.

- DELETE /admin/usuarios/{user_email}
    - Objetivo: Eliminar un usuario de la base de datos a partir de su email.
    - Parámetros:
        - user_email en la ruta, identificando el usuario por su email.
    - Flujo:

        - Se busca en la tabla Usuario por la columna email.
        - Si no se encuentra, se lanza HTTP 404.
        - De lo contrario, se realiza db.delete(usuario) y db.commit(), removiendo al usuario y sus datos asociados.
    - Restricción: Solo el administrador puede acceder, si no, se lanza un error HTTP 403 con la descripción “No tiene permisos para acceder a este recurso”.
- GET /admin/requests
    - Objetivo: Devolver la lista de solicitudes de cambio de médico que estén pendientes. Se maneja con paginación para que no sea tan abrumador.
    - Flujo:
        - Consulta la tabla SolicitudesCambio filtrando por el estado que estén en pendiente.
        - Se divide en páginas.
        - Para cada solicitud, se traen los datos básicos del paciente y del médico para crear una respuesta completa.
    - Restricción: Solo el administrador puede acceder, si no, se lanza un error HTTP 403 con la descripción “No tiene permisos para acceder a este recurso”.

- POST /admin/requests/{id_paciente}/{id_medico}/{fecha}/accept o reject
    - Objetivo:Aprobar o rechazar la solicitud de cambio de médico para el paciente.
    - Parámetros:
        - id_paciente en la ruta, identificando el paciente.
        - id_medico en la ruta, identificando el médico.
        - fecha en la ruta, identificando la fecha en la que se envió la solicitud
        - accept o reject en la ruta, identificando la respuesta.
    - Flujo:
        - Se busca la solicitud en la tabla SolicitudesCambio.
        - Si no existe o no está pendiente, da un error HTTP 400.

- Se actualiza el estado según la opción escogida y en caso de que sea aceptada se cambia de la tabla Paciente el identificador del médico.
- Por último se confirman los cambios en la base de datos.

- Restricción: Solo el administrador puede acceder, si no, se lanza un error HTTP 403 con la descripción “No tiene permisos para acceder a este recurso”.

4.5.1.3. Glucose

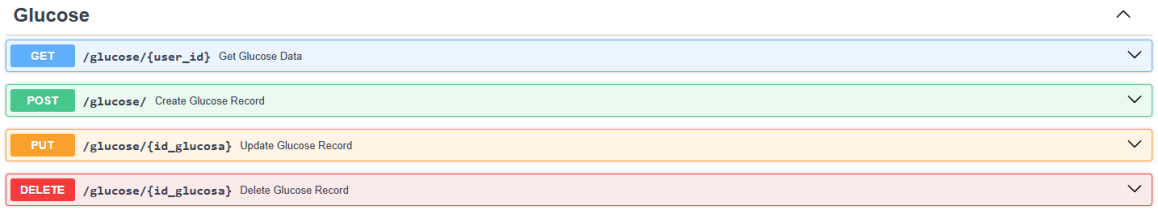


Figura 63: endpoints de la ruta glucose

- GET /glucose/{user_id}
  - Objetivo: Obtener todos los datos de glucosa asociados a un paciente.
  - Parámetros:
    - user_id en la ruta, identificando el paciente.
  - Flujo:
    - El endpoint recibe como parámetros de ruta el identificador del usuario.
    - Se realiza una consulta en la tabla Glucosa filtrando por el identificador del usuario.
    - Si no existen registros, se retorna un array vacío.
    - En caso contrario, se forma un listado con todos los datos.

- POST /glucose
  - Objetivo: Crear un nuevo registro de glucosa para un paciente.
  - Entradas: identificador del paciente, nivel de glucosa, fecha y hora.
  - Flujo:

        - El endpoint recibe una clase de datos definida con pydantic.
        - Se crea un nuevo objeto Glucosa con los campos pasados y se inserta en la base de datos.
        - Al final, se devuelve un mensaje “Registro de glucosa creado exitosamente” junto a los datos guardados.

- PUT /glucose/{id_glucosa}
    - Objetivo: Actualizar por completo un registro de glucosa existente.
    - Parámetros:
        - id_glucosa en la ruta, identificando el registro de la glucosa.
    - Entradas: nivel de glucosa, fecha y hora.
    - Flujo:
        - Se recibe el identificador de la glucosa en la ruta y data Classe en el cuerpo.
        - Se busca el registro en la base de datos. Si no existe, se lanza un error HTTP 404.
        - En caso contrario, se sobreescriben los campos con los nuevos valores.
        - Se confirma la transacción y se retorna un mensaje de confirmación “Registro de glucosa actualizado correctamente” junto a los datos actualizados.

- DELETE /glucose/{id_glucosa}
    - Objetivo: Eliminar un registro de glucosa de la base de datos a partir de su identificador.
    - Parámetros:
        - id_glucosa en la ruta, identificando el registro de la glucosa.
    - Flujo:
        - Recibe el identificador en la ruta y busca el objeto en la tabla Glucosa. Si no se halla lanza un error HTTP 404.
        - Si existe, se borra y se confirma la transacción para confirmar la eliminación.
        - Retorna un HTTP 204 (sin contenido en el cuerpo)

4.5.1.4. Notes

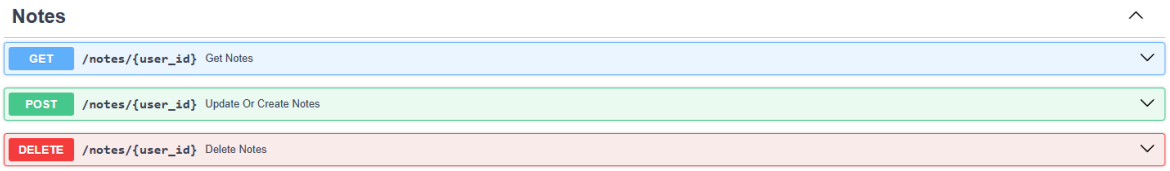


Figura 64: endpoints de la ruta notes

- GET /notes/{user_id}?date=YYYY-MM-DD
  - Objetivo: Recuperar la nota con los detalles para un paciente en una fecha concreta.
  - Parámetros:
    - user_id en la ruta, identificando el paciente.
    - date como parámetro de query, en formato año-mes-dia.
  - Flujo:
    - Se parsea la fecha para asegurar que tenga un formato válido.
    - Se busca en la tabla Nota un registro que cumpla que las identificaciones sean iguales y las fechas también.
    - Si se encuentra la nota, se devuelve sus datos y la lista de detalles.
    - Si no hay coincidencia, retorna un array vacío.

- POST /notes/{user_id}
  - Objetivo: Crear o actualizar la nota y sus detalles para un día dado.
  - Parámetros:
    - user_id en la ruta, identificando el paciente.
  - Entradas: fecha y detalles.
  - Flujo:
    - Se verifica en la base de datos si existe ya una nota para ese identificador de usuario y fecha indicada.
    - Si no existe, se crea una nueva entrada en la tabla Nota.
    - Si existe, se eliminan todos los detalles previos de dicha nota.
    - Se inserta la nueva lista de detalles, construyendo objetos NotaDetalle, asociandolos a la nota.

- Se confirma la operación y retorna un JSON con “Notas actualizadas correctamente”.

- DELETE /notes/{user_id}?fecha=YYYY-MM-DD
  - Objetivo: Eliminar por completo la nota y sus detalles de un paciente en una fecha determinada.
  - Parámetros:
    - user_id en la ruta, identificando el paciente.
    - date como parámetro de query, en formato año-mes-dia.
  - Flujo:
    - Se busca en la tabla Nota la entrada que coincide.
    - Si no existe, se retorna un error HTTP 404.
    - Si existe, se borra y se confirma la operación, lo cual también elimina en cascada los detalles.
    - Devuelve un objeto JSON con un mensaje “Nota eliminada correctamente”.

4.5.1.5. Diagnosis

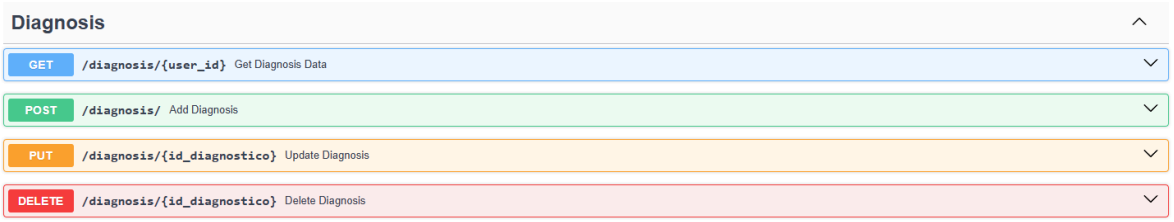


Figura 65: endpoints de la ruta notes

- GET /diagnosis/{user_id}
  - Objetivo: Obtener la lista de diagnósticos correspondientes a un paciente.
  - Parámetros:
    - user_id en la ruta, identificando el paciente.
  - Flujo:
    - El endpoint recibe el identificador del paciente en la ruta.

        - Se ejecuta una consulta que une la tabla Diagnostico con Medico y Usuario, además de filtrar.
        - Si no se encuentran registros, se devuelve un array vacío.
        - En caso contrario, se construye una lista con los datos obtenidos.

- POST /diagnosis
    - Objetivo: Crear un nuevo diagnóstico asociado a un
    - Entradas: identificador de paciente, identificador de médico, fecha y descripción.
    - Flujo:
        - Se comprueba si existe el médico y el paciente. Si alguno falla salta error HTTP 404.
        - Se crea un objeto Diagnóstico con la fecha y descripción. En caso de que no se pase, se usa la fecha actual.
        - Se inserta en la base de datos y se retorna un JSON con los datos insertados.

- PUT /diagnosis/{id_diagnostico}?id_medico=...
    - Objetivo: Actualizar la descripción de un diagnóstico ya existente, si el médico que lo emite coincide.
    - Parámetros:
        - id_diagnostico en la ruta, para identificar el diagnóstico.
        - id_medico en la query o como argumento (en tu caso, como parte de la firma de la función).
    - Entradas: descripción.
    - Flujo:
        - Se busca el diagnóstico. Si no se halla, lanza un error HTTP 404.
        - Se verifica que el médico que intenta actualizarlo sea el autor. Si no coincide, HTTP 403 (“No autorizado para editar este diagnóstico”).
        - Se modifica la descripción con el nuevo texto.
        - Se confirma la transacción.

- Se retorna la información final del diagnóstico, con fecha, paciente y médico.

- DELETE /diagnosis/{id_diagnostico}?id_medico=...
  - Objetivo: Eliminar un diagnóstico, confirmando que el médico que lo borra es el mismo que lo creó.
  - Parámetros:
    - id_diagnostico en la ruta, para identificar el diagnóstico.
    - id_medico en la query o como argumento (en tu caso, como parte de la firma de la función).
  - Flujo:
    - Se busca la entrada en la tabla Diagnóstico.
    - Si no se halla da un error HTTP 404.
    - Se verifica que el médico es el mismo. En caso de no coincidencia, HTTP 403.
    - Se borra y se confirma la operación.
    - Devuelve un HTTP 204.

4.5.1.6. Analytics

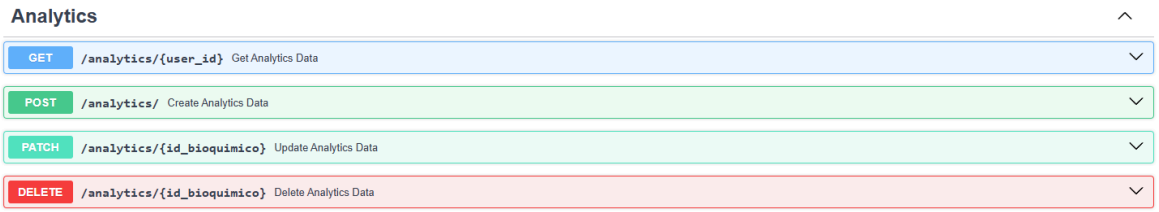


Figura 66: endpoints de la ruta analytics

- GET /analytics/{user_id}
  - Objetivo: Consulta todos los registros de analíticas de un paciente.
  - Parámetros:
    - user_id en la ruta, identificando el paciente.
  - Flujo:

    - El endpoint recibe el identificador del usuario como parámetro de ruta.
    - Se realiza una consulta para que el identificador del paciente coincida.
    - En caso de que no existan registros se devuelve un array vacío.
    - Si hay resultados, se construye un objeto por cada registro.

- POST /analytics
    - Objetivo: Crear un nuevo registro de analítica para un paciente
    - Entradas: identificador del paciente, fecha , nombre y valor.
    - Flujo:
        - Se recibe un data Class con pydantic para la estructura y los campos necesarios.
        - Se crea un objeto Bioquímico, asignando los campos y guardando la operación.
        - Retorna un JSON con “Analítica creada exitosamente” junto al objeto resultante.

- PATCH /analytics/{id_bioquimico}
    - Objetivo: Actualizar parcialmente un registro ya existente de analítica. Se usa PATCH en lugar de PUT para indicar que los campos pueden ser opcional.
    - Parámetros:
        - id_bioquimico en la ruta, para identificar la analítica.
    - Entradas: fecha, nombre, valor, con posibilidad de dejarlos como None si no se actualizan.
    - Flujo:
        - Se localiza el registro en la tabla Bioquímico filtrado por el identificador. Si no existe lanza un error HTTP 404.
        - Por cada campo presente en la petición, se actualiza el objeto en la base de datos.
        - Se confirma la operación.
        - Devuelve un objeto con “Analítica actualizada correctamente” y los datos resultantes.

- DELETE /analytics/{id_bioquimico}
    - Objetivo: Eliminar un registro de analítica por su identificador.
    - Parámetros:
        - id_bioquimico en la ruta, para identificar la analítica.
    - Flujo:
        - Se busca el registro en la base de datos.
        - Si no se encuentra lanza un error HTTP 404.
        - En caso contrario, se borrar la analitica y se confirma la operación.
        - Se retorna un HTTP 204 sin contenido.

4.5.1.7. Family

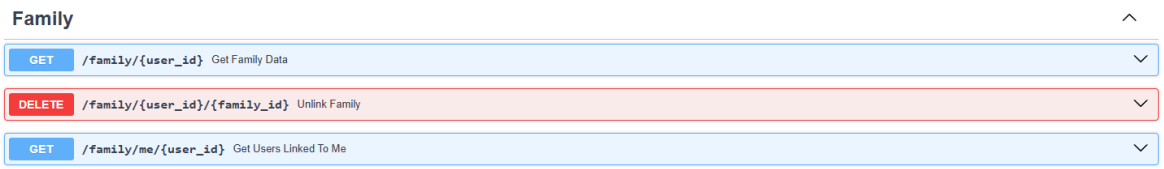


Figura 67: endpoints de la ruta family

- GET /family/{user_id}
    - Objetivo: Obtener la lista de familiares que se han vinculado con un paciente.
    - Parámetros:
        - user_id en la ruta, identificando el paciente.
    - Flujo:
        - Se recibe el identificador del usuario
        - Se hace una consulta a la tabla Vínculo donde coinciden los identificadores y se une con la tabla Usuario para obtener los datos del familiar.
        - Si no se encuentra registros, devuelve un array vacío.
        - En caso contrario, se construye un array con la información del familiar y la fecha en la que se estableció la vinculación.

- DELETE /family/{user_id}/{family_id}
    - Objetivo: Anula el acceso del familiar a la información del paciente.
    - Parámetros:
        - user_id en la ruta, identificando el paciente.
        - family_id en la ruta, identificando el familiar.
    - Flujo:
        - Se consulta la tabla Vínculo filtrando por los parámetros obtenidos.
        - Si la relación no existe, se lanza un error HTTP 404.
        - En caso contrario, se elimina el registro en la base de datos y se confirma la operación.
        - Retorna un JSON con “Familiar desvinculado correctamente”.

- GET /family/me/{user_id}
    - Objetivo: Devolver la lista de pacientes con los que un familiar está vinculado.
    - Parámetros:
        - user_id en la ruta, identificando el familiar.
    - Flujo:
        - Se consulta la tabla Vínculo pero invertido se filtra por el identificador del familiar y se une con la tabla usuario para obtener los datos del paciente.
        - Si no existen registros, se retorna un array vacío.
        - En caso contrario, se construye un array con los datos básicos del paciente con el que está vinculado.

4.5.1.8. Users

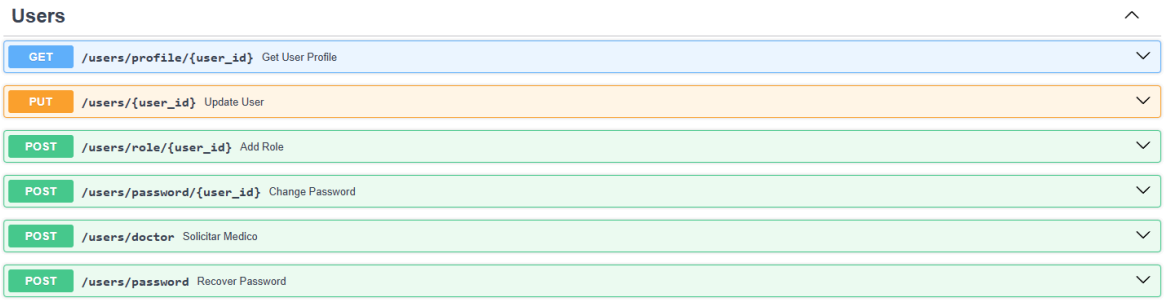


Figura 68: endpoints de la ruta users

- GET /users/profile/{user_id}
  - Objetivo: Obtener la información de perfil de un usuario
  - Parámetros:
    - user_id en la ruta, identificando el usuario.
  - Flujo:
    - Se consulta la tabla Usuario con la relación de roles para ver si el usuario existe.
    - Se examina la lista de roles, lo cual determina si se necesita cargar la tabla Paciente o Médico para datos adicionales y se retorna los datos específicos.
    - Si el usuario no es ninguno de los roles anteriores se devuelve simplemente sus datos básicos junto a los roles.

- PUT /users/{user_id}
  - Objetivo: Actualizar los datos de un usuario específico.
  - Parámetros:
    - user_id en la ruta, identificando el usuario.
  - Entradas: JSON con los campos a cambiar nombre, apellidos, fecha de nacimiento y email.
  - Flujo:
    - Se busca el usuario mediante el identificador. Si no existe, se lanza el error HTTP 404.
    - Se revisan los campos proporcionados. Aquellos presentes se reasignan al objeto user.

        - Se valida el formato de la fecha de nacimiento si se incluye
        - Se confirma la operación.
        - Se retorna un JSON con los datos finales.

- POST /users/role/{user_id}
    - Objetivo: Añadir un rol al usuario.
    - Parámetros:
        - user_id en la ruta, identificando el usuario.
    - Entradas: roles
    - Flujo:
        - Se busca al usuario por su identificador. Si no existe, se lanza el error HTTP 404.
        - Se verifica si el rol proporcionado existe en la tabla Rol. Si no existe se lanza el error HTTP 404.
        - Se comprueba si el usuario ya posee dicho rol, si es así, se lanza error HTTP 400.
        - Se inserta en la tabla intermedia “usuario_roles”
        - Se confirma la operación y retorna el mensaje de confirmación.

- POST /users/password/{user_id}
    - Objetivo: Cambiar la contraseña de un usuario si proporciona la contraseña antigua.
    - Parámetros:
        - user_id en la ruta, identificando el usuario.
    - Entradas: antigua y nueva contraseña.
    - Flujo:
        - Se busca al usuario por su identificador. Si no existe, se lanza el error HTTP 404.
        - Se verifica la contraseña antigua. Si no coincide, se lanza el error HTTP 400
        - Se cambia la contraseña en la base de datos encriptados y se confirma la operación.
        - Se retorna el mensaje “Contraseña cambiada correctamente”.

- POST /users/doctor?user_id=...
    - Objetivo: Asignar un médico aleatorio a un paciente, o crear una solicitud de cambio de médico si el paciente ya tiene uno.
    - Parámetros:
        - user_id en la ruta, identificando el usuario.
    - Flujo:
        - Se busca al usuario por su identificador. Si no existe, se lanza el error HTTP 404.
        - Si el paciente no tiene un médico asignado, se le asigna uno aleatoriamente. Si no hay médicos, HTTP 404.
        - Si el paciente ya tiene un médico, se crea una solicitud de cambio de médico y marcándose en pendiente. Se verifica que no exista una solicitud pendiente.
        - Se confirma la operación y se retorna un mensaje “Médico asignado automáticamente” o “Solicitud de cambio enviada correctamente”.

4.5.1.9. Message

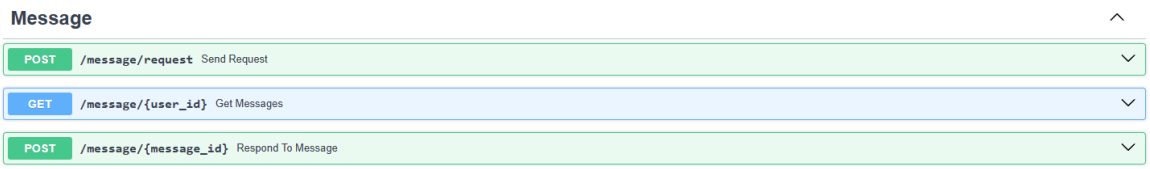


Figura 69: endpoints de la ruta message

- POST /message/request
    - Objetivo: Enviar una solicitud de vinculación del familiar hacia el paciente.
    - Entradas: identificador del usuario que lo envía y el correo del usuario que lo recibe.
    - Flujo:

        - Se busca el usuario con el mismo email. Si no se encuentra, se lanza un error HTTP 404. Si se encuentra, se verifica que tenga el rol paciente. Si no es paciente se lanza el error HTTP 400.
        - Se crea un objeto Mensaje, se almacena la solicitud y se responde con "Solicitud enviada correctamente".

- GET /message/{user_id}
    - Objetivo: Listar los mensajes dirigidos a un usuario, mostrando los que tienen el estado “pendientes” y su contenido.
    - Parámetros:
        - user_id en la ruta, identificando el usuario.
    - Flujo:
        - Se filtra la tabla Mensaje por el identificador del usuario, ordenando por fecha de envío más antiguo primero.
        - Retorna un array de objetos de la clase de Mensaje.
        - Si no hay mensajes, se devuelve un array vacío.

- POST /message/{message_id}
    - Objetivo: Responder un mensaje, marcándolo como “aceptado” o “rechazado”. En el caso de aceptar significa crear un vínculo.
    - Parámetros:
        - message_id en la ruta, identificando el mensaje.
    - Entradas: respuesta aceptar o rechazar.
    - Flujo:
        - Se busca el mensaje por el identificador. Si no existe se lanza el error HTTP 404.
        - Se valida que la respuesta sea “aceptado” o “rechazado”, si el valor es distinto, daría un error HTTP 400.
        - Se actualiza la respuesta.
        - En caso de “aceptado”, se inserta un registro en la tabla Vínculo con la fecha actual.
        - Se confirma la transacción y se retorna el mensaje con la elección.

4.5.1.10. Doctor

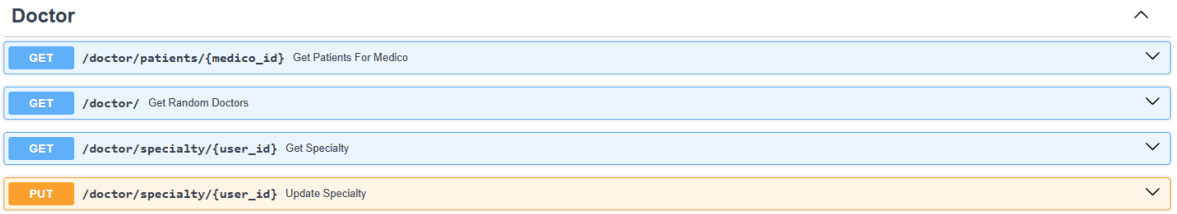


Figura 70: endpoints de la ruta doctor

- GET /doctor/patients/{medico_id}
  - Objetivo: Devolver la lista de pacientes asignados a un médico en concreto.
  - Parámetros:
    - medico_id en la ruta, identificando el médico.
  - Flujo:
    - Se ejecuta una consulta a la tabla Paciente unida con la tabla Usuario y filtrando por el identificador del médico.
    - Si no se encuentran registros, retorna un array vacío. De lo contrario, se construye un array con los datos básicos del paciente.
    - Cada paciente se asocia a un Usuario porque los datos se encuentran en la tabla Usuario.

- GET /doctor/specialty/{user_id}
  - Objetivo: Obtener la especialidad de un médico.
  - Parámetros:
    - user_id en la ruta, identificando el médico.
  - Flujo:
    - Se busca en la tabla Medico por el identificar que se nos ha proporcionado. Si no existe, se lanza un error HTTP 404.
    - Se retorna un JSON con especialidad.

- PUT /doctor/specialty/{user_id}
  - Objetivo: Actualizar la especialidad del médico.

- Parámetros:
  - user_id en la ruta, identificando el médico.
- Entradas: nueva especialidad
- Flujo:
  - Se busca en la tabla Médico por el identificador. Si no se encuentra se lanza el error HTTP 404.
  - Se asigna la nueva especialidad en la base de datos y se confirma la transacción.
  - Se retorna el mensaje “Especialidad actualizada correctamente” y la nueva especialidad.

4.5.2. Diseño Interfaz del cliente

En este punto se va hablar de forma general de la estructura de la interfaz y se va a mostrar algunos wireframes *[26]* de las vistas y elementos principales.

4.5.2.1. Estructura General de la interfaz

En el diseño del front-end, se ha partido de la funcionalidad que nos proporciona next.js que a su vez viene de react que son las páginas y componentes:

- Páginas: Cada carpeta que tiene dentro un archivo ‘page’ se convierte en una ruta que los usuario pueden visitar. Estas páginas están orientadas a roles y funcionalidades muy concretas, aquí se ven algunas de ellas:
  - Login: Inicio de sesión.
  - Home: Vista principal del paciente.
  - Patients: Panel del médico, donde tiene una lista de los pacientes bajo su supervisión.
  - Family: Gestión de familiares.
  - AdminDashBoard: Control de usuarios y solicitudes, exclusivo para administradores.
- Componentes: en la carpeta ‘/components’ se alojan bloques de código reutilizables como NavBar, Footer, ProtectedRoute y muchos más. Cada uno

cumple un propósito específico y se inserta en páginas que requieren su funcionalidad y visibilidad.

- Contexto de autenticación (AuthContext): Se encarga de mantener el estado de autenticación, es decir token, roles, datos del usuario, además de facilitar las funciones de login, logout, actualización del usuario. Con ello, las páginas y componentes pueden consultar el contexto para saber si un usuario está logueado y que rol posee.

Al apostar por una aplicación responsive, cada vista se diseña para reorganizar los elementos al pasar de escritorio a móvil. El layout de escritorio puede tener dos o tres columnas, mientras que la versión móvil suele reducirse a una sola columna en scroll vertical. Además, algunos elementos secundarios se ocultan y agrupan en menús compactos, para que la interfaz se quede más limpia y clara, a la hora del usuario utilizarla.

Existen ciertas desventajas que esto conlleva una mayor complejidad a la hora de reestructurar la página cuando se redimensiona y no saturar la pantalla. Sin embargo, la ventaja principal es una experiencia del usuario unificada es decir un usuario puede ingresar desde su teléfono móvil, ordenador o tablet, encontrando la misma información y lógica de menús, aunque adaptada en distribución.

4.5.2.2. Wireframes de la interfaz

Un wireframe es una representación esquemática de la estructura de una vista en una aplicación web o móvil. Se dibuja con formas sencillas y colores neutros como la escala de grises, para enfocarse en la asignación del espacio, priorización del contenido, las funcionalidades disponibles y los componentes deseados, dando una jerarquía visual, sin distraer con detalles de estilo tipográfico, color o aplicaciones gráficas. Así los wireframes ayudan a los equipos de diseño y desarrollo a discutir y validar la disposición y navegación de la interfaz antes de pasar a la implementación de los aspectos estéticos y funcionales.

El wireframe conecta la estructura conceptual, o arquitectura de la información, con el diseño visual de la web o aplicación.

A continuación se va a describir las distintas vistas que tenemos en la aplicación:

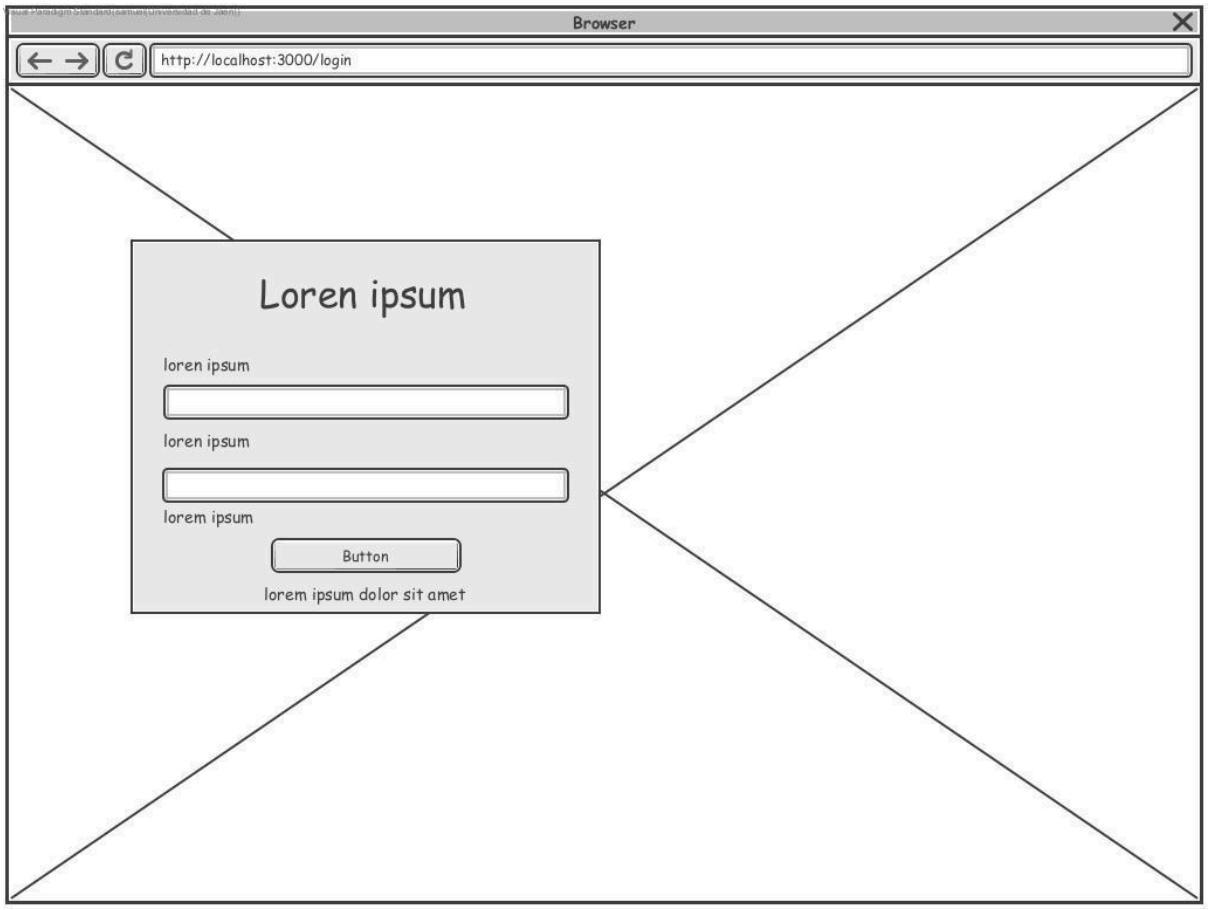


Figura 71: wireframe vista login web

Este wireframe representa la vista de la página de login donde el usuario accede para iniciar sesión. Al cargar la página, se presenta un diseño centrado en el formulario de autenticación. En la parte superior, se muestra la barra de navegación del navegador con la URL del login (http://localhost:3000/login).

El contenido principal de la pantalla , se puede ver que la estructura se compone de una imagen de fondo y una ventana flotante donde aparecerán unos campos de texto y un botón para acceder. En los textos habrá enlaces para acceder a la página de registro y recuperación de contraseña.

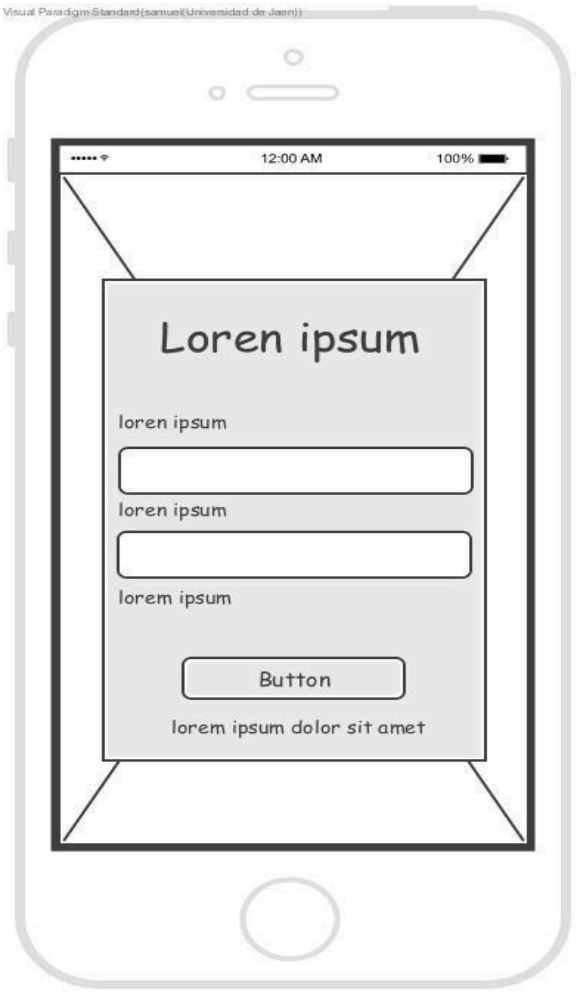


Figura 72: wireframe vista login móvil

En el wireframe se puede ver la adaptación de a móvil en el cual la imagen queda más en segundo plano porque al tener una ventana más pequeña debemos que hacer que el usuario pueda visualizarlo bien, manteniendo la estructura del contenedor que contiene los campos de texto y el botón.

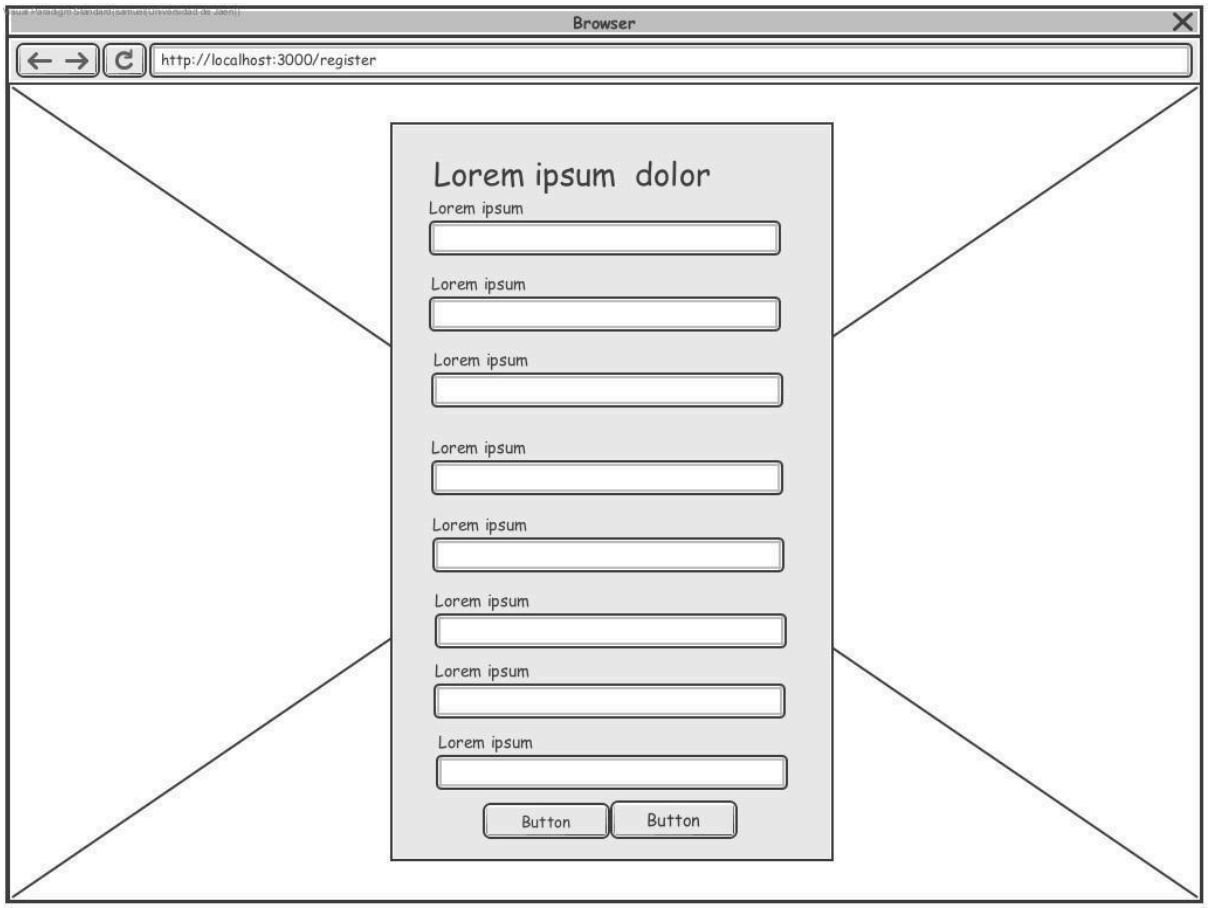


Figura 73: wireframe vista login móvil

La página está diseñada para centrar toda la atención del usuario en el formulario de registro, que ocupa el área central de la interfaz. Alrededor del formulario, el diseño es una imagen en el fondo de la ventana.

Los componentes que podemos visualizar, en la parte superior del formulario, se encuentra el título de registro. Este elemento proporciona contexto inmediato sobre la funcionalidad de la página.

El formulario contiene una lista vertical de campos de entrada que permiten al usuario ingresar la información necesaria para registrarse.

Además en la parte de abajo tenemos botones de acción con lo cual podemos enviar y cancelar.

Esta página para el diseño responsive puede adaptarse fácilmente a diferentes tamaños de pantalla, manteniendo la funcionalidad principal gracias a la disposición vertical.

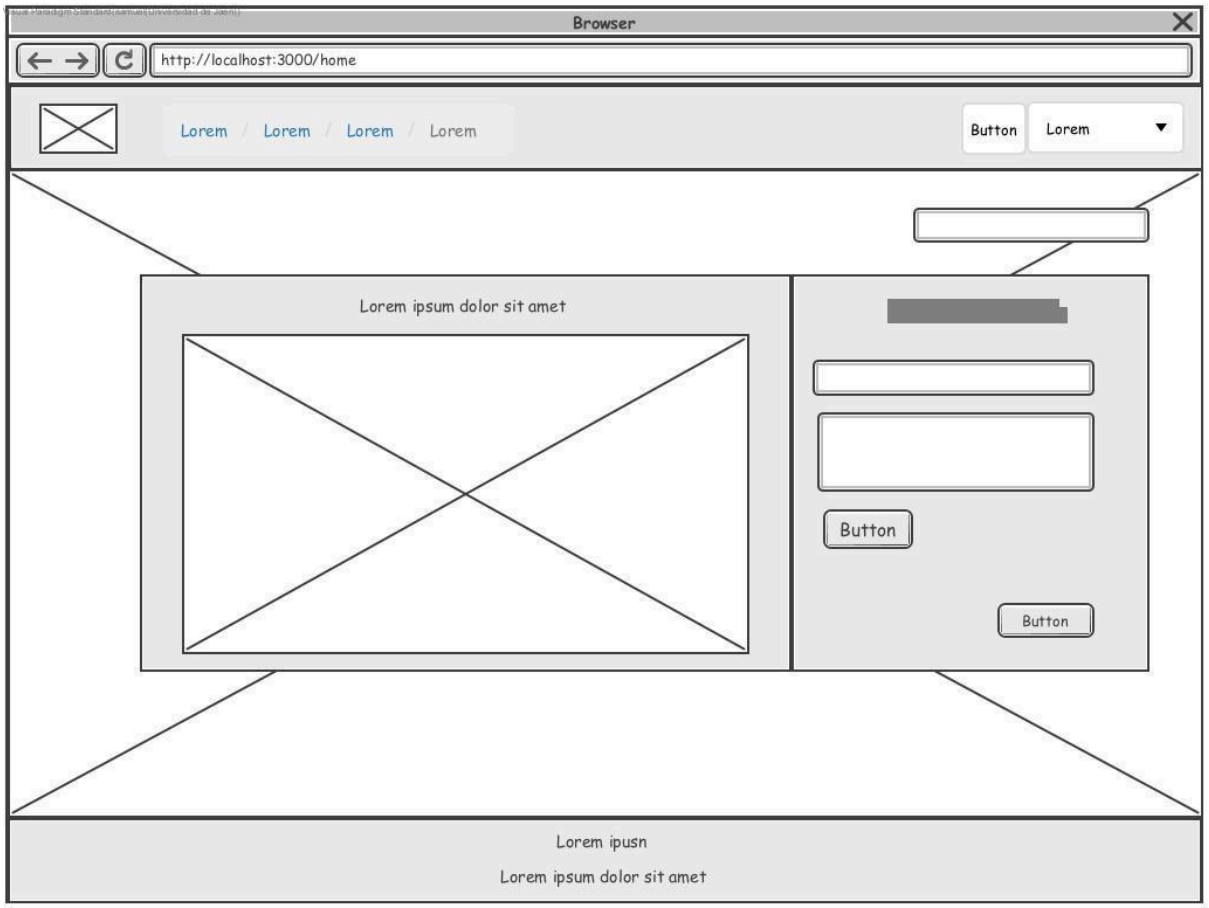


Figura 74: wireframe vista home web

El diseño para web presenta una estructura más horizontal, aprovechando el espacio adicional de las pantallas grandes. La navegación se encuentra en la parte superior, con enlaces visibles que permiten moverse entre diferentes secciones de la aplicación. A la derecha de la barra de navegación, se encuentran un botón de acción y un desplegable, lo que representa opciones relacionadas con la sesión del usuario.

En la parte principal, el contenido se divide en dos columnas:

- Izquierda: Una gran área visual para mostrar gráficos.
- Derecha: Un formulario con campos de entrada y botones. Los usuarios pueden interactuar desde esta página para enviar la información de las notas.

En la parte inferior, hay una barra con texto adicional, que sirve como pie de página web.

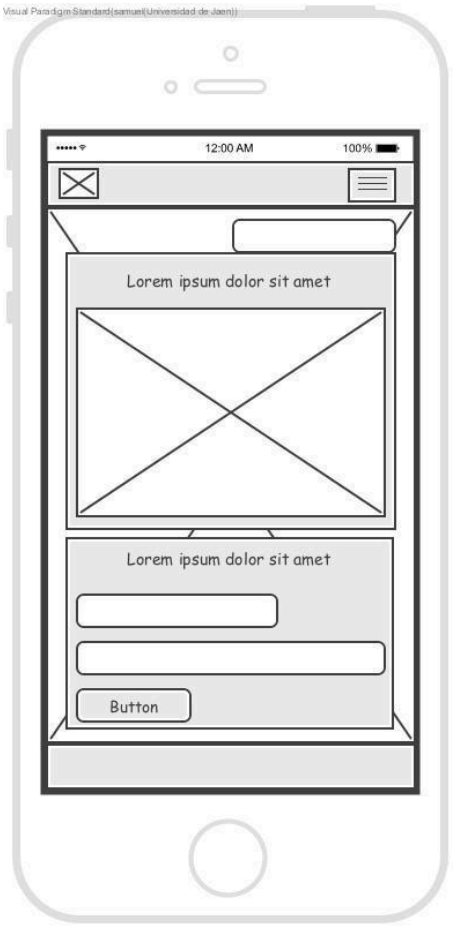


Figura 75: wireframe vista home móvil

El diseño para dispositivos móviles está optimizado para pantallas verticales, con elementos apilados en una sola columna para facilitar el uso en pantallas pequeñas. En la parte superior, se encuentra una barra de navegación que incluye un ícono de menú a la izquierda y un ícono adicional a la derecha, que sirve como desplegable de las vistas de la aplicación.

El contenido principal se organiza verticalmente de arriba hacia abajo tenemos el campo de texto que funcionará como seleccionador de fecha, debajo el gráfico de la glucosa y por último el formulario de las notas diarias.

El diseño está simplificado para garantizar una experiencia fluida en dispositivos móviles, con un enfoque en la usabilidad y el acceso rápido a funciones esenciales.

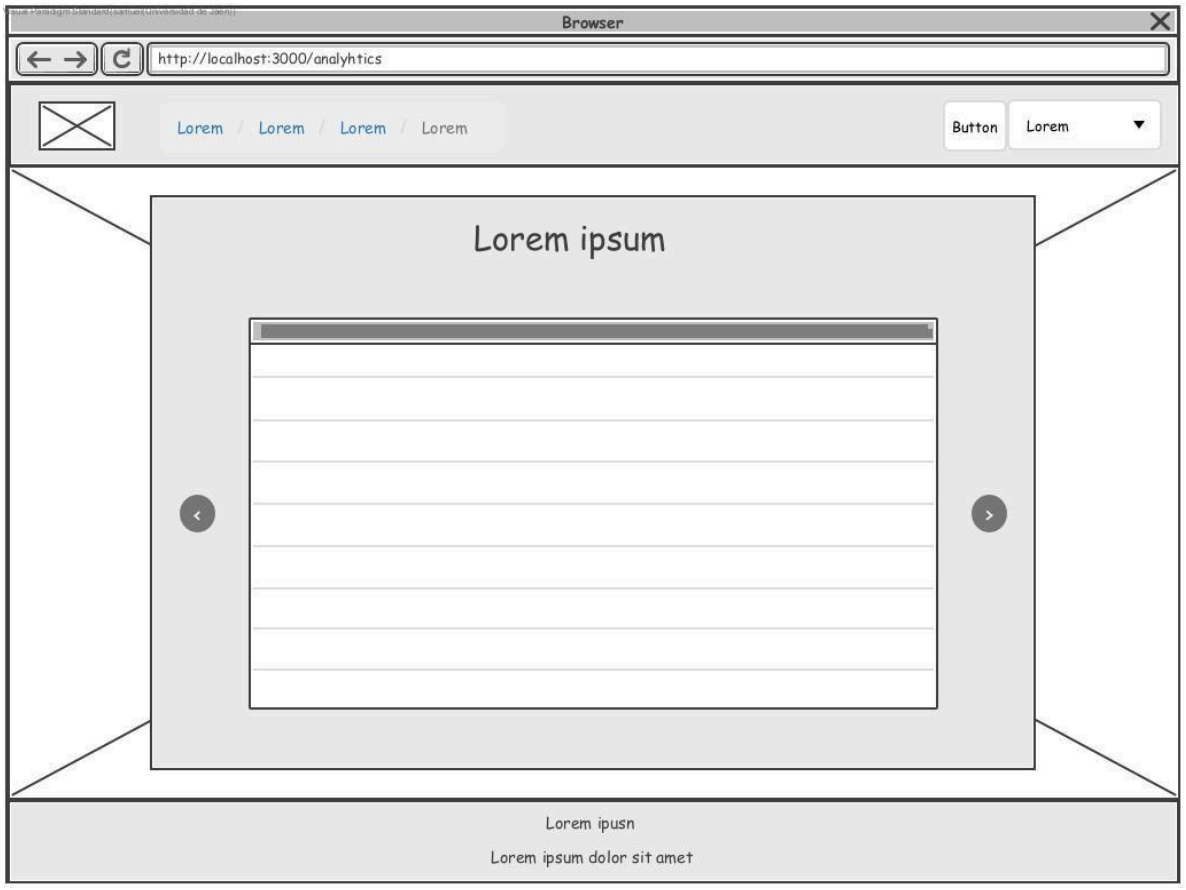


Figura 76: wireframe vista analíticas web

La pantalla de análisis presenta un enfoque más visual. En el centro hay una tabla grande que permite al usuario explorar datos relacionados con analíticas en el cual aparecen el tipo y valor. Los controles laterales permiten navegar entre diferentes períodos de tiempo, proporcionando una experiencia interactiva para explorar datos históricos.

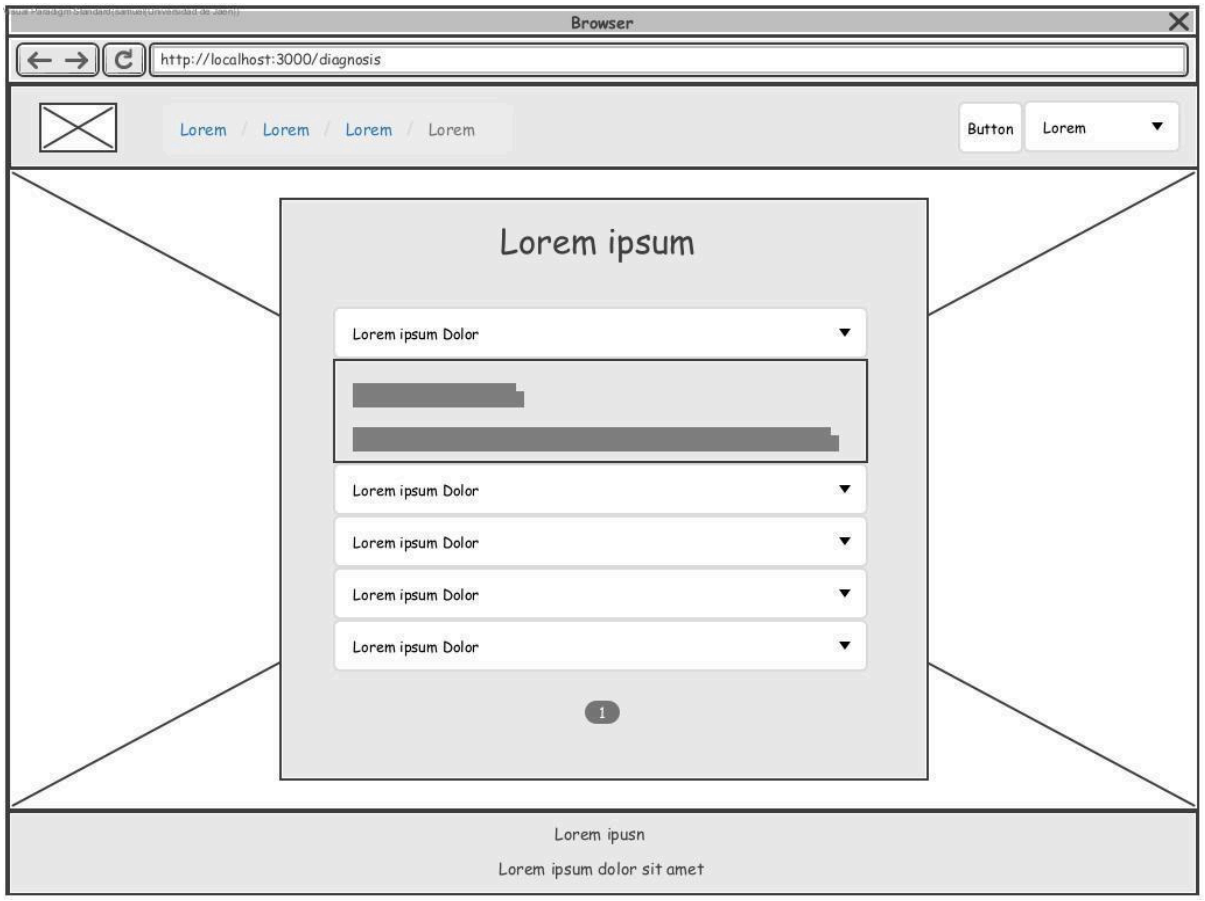


Figura 77: wireframe vista diagnóstico web

Esta pantalla está diseñada para permitir la visualización de diagnósticos. En el centro se encuentra una sección con el encabezado, seguido de una lista desplegable o acordeón donde los usuarios pueden seleccionar opciones relacionadas con los diagnósticos. dentro de cada diagnóstico se verá quien lo emitió y la descripción del diagnóstico. También podemos observar debajo una paginación en el caso de que el número de diagnósticos superen los límites impuestos.

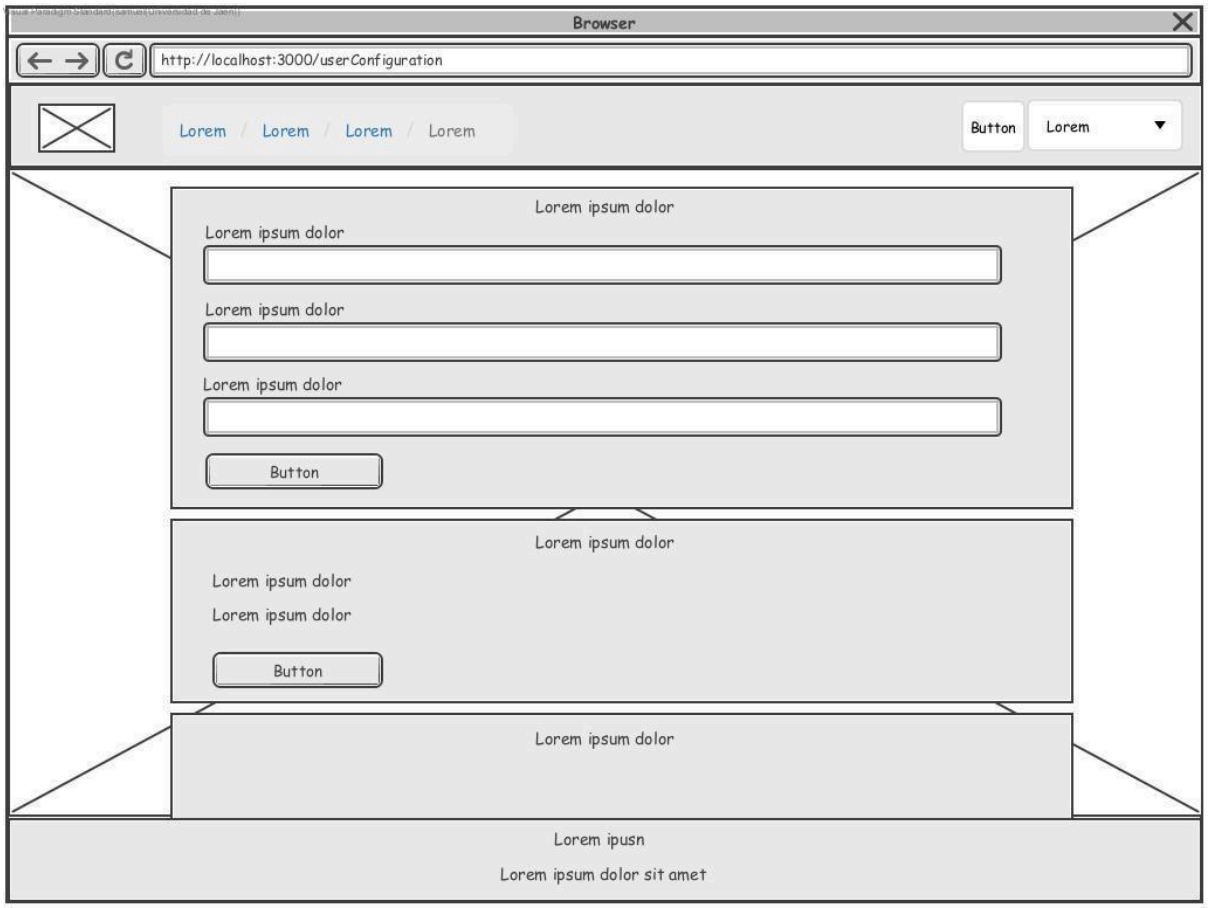


Figura 78: wireframe vista configuración web

Esta ventana está orientada a la configuración de usuario. Vemos un fragmento ya que al tener más de un componente la vista baja más hacia abajo, pero aún así en la parte superior se encuentra un formulario con múltiples campos de entrada y un botón para guardar los cambios, lo que permite editar información personal. Debajo hay secciones adicionales como la solicitud de médico en el cual podemos ver la información del médico actual y con el botón que aparece debajo se podría lanzar la solicitud de cambio de médico.

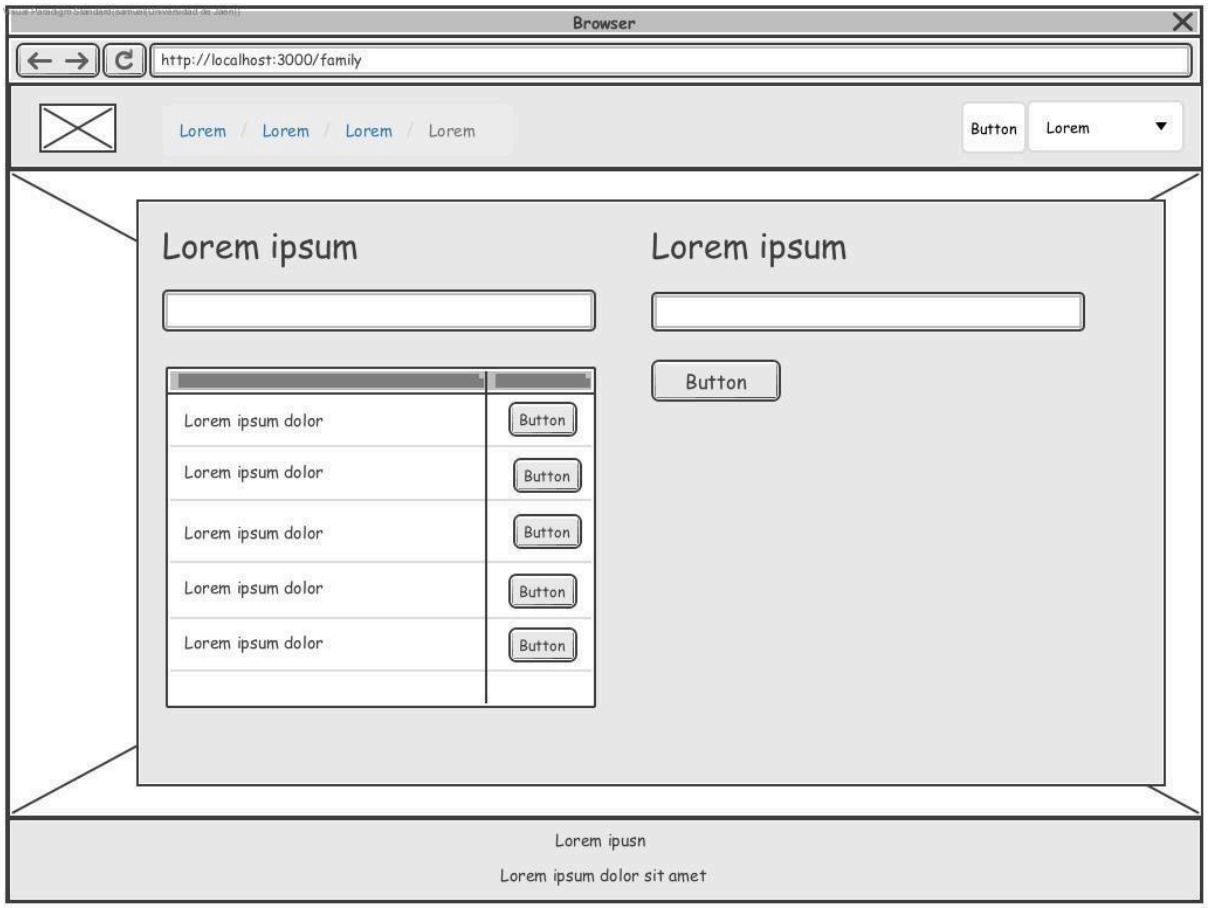


Figura 79: wireframe vista familiares web

Este wireframe está dividido en dos secciones principales, distribuidas horizontalmente. Cada sección tiene un encabezado ya que las áreas están destinadas a diferentes funcionalidades relacionadas con la gestión de familiares.

En la parte izquierda contiene un campo de texto que es un buscador de los familiares, debajo hay una lista de familiares vinculados. Cada fila de la lista contiene el nombre del familiar y un botón asociado a su perfil.

En la parte derecha hay otro campo de texto, en este caso se puede escribir el correo electrónico para buscar al usuario y debajo tenemos un botón con el cual le enviamos la solicitud de familiar.

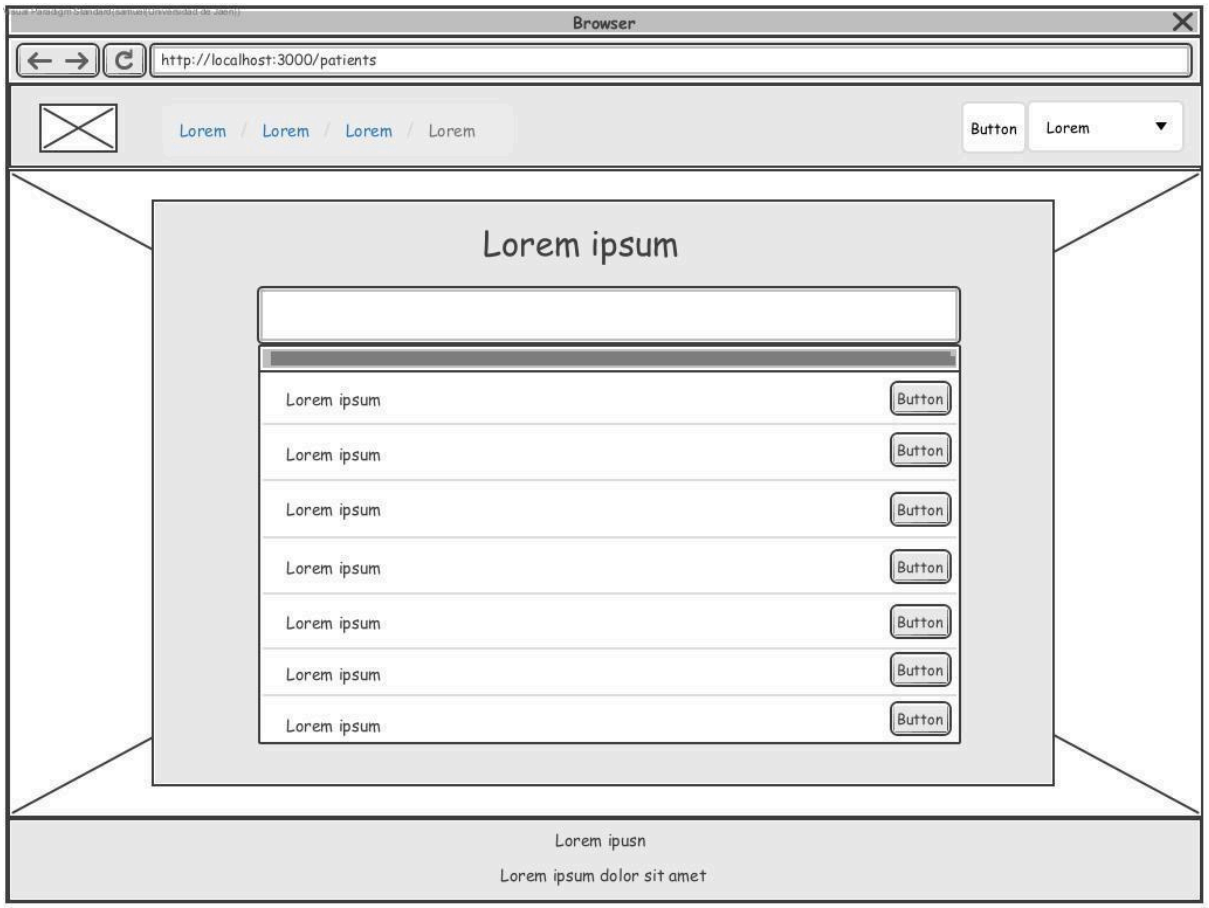


Figura 80: wireframe vista pacientes web

En esta página se centra en una lista de pacientes, presentada de manera ordenada y fácil de navegar. Todo el contenido principal está enmarcado en un cuadro centrado, con un diseño limpio y funcional.

En la parte superior podemos encontrar un campo de texto que se asocia a un buscador de los pacientes que el médico tiene asociado.

En la parte de abajo contiene la lista vertical de elementos, donde cada uno contiene texto en el cual corresponde al nombre del paciente y un botón que sirve para navegar hacia el perfil del paciente.

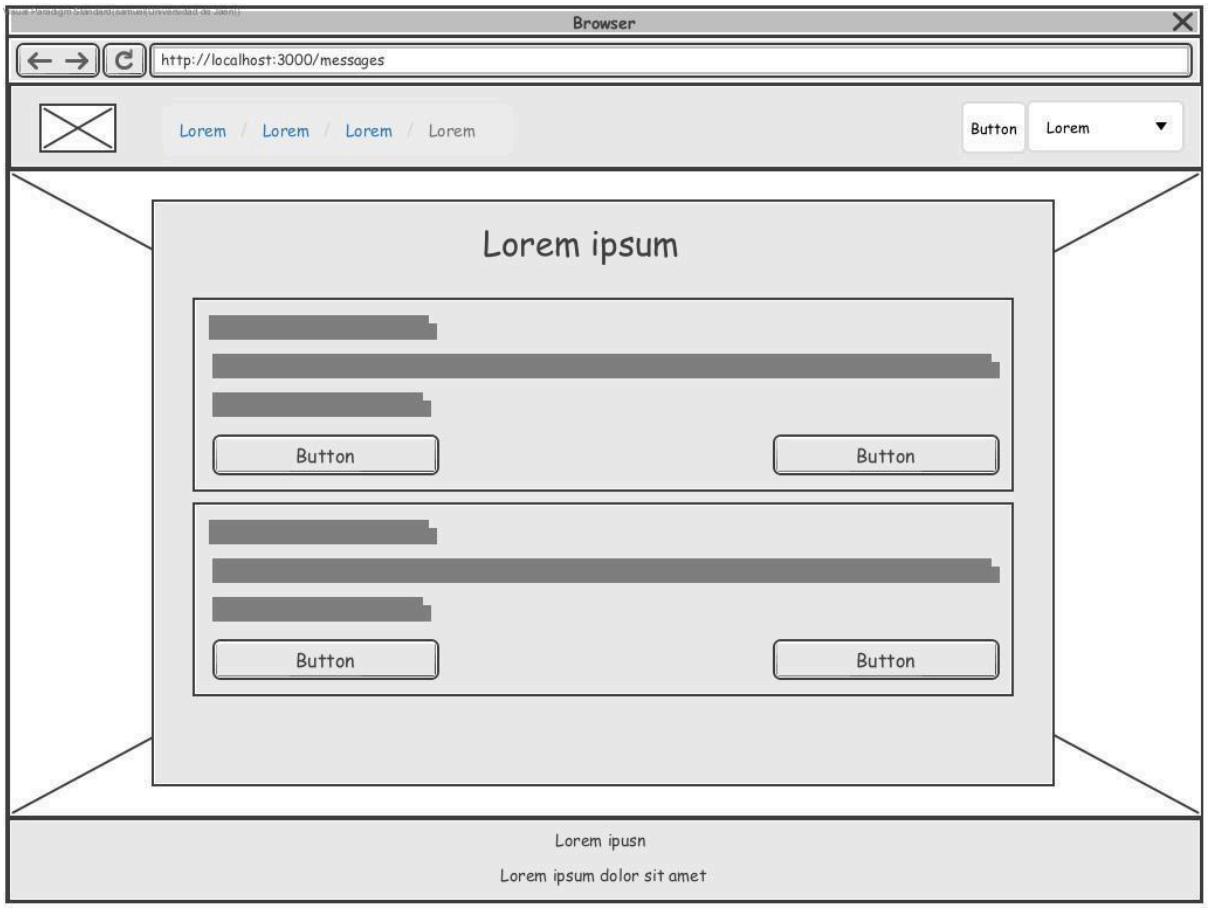


Figura 81: wireframe vista mensajes web

La página está organizada de forma vertical, con una barra superior de navegación, un encabezado que introduce la sección de mensajes, y un área central donde se listan los mensajes de manera estructurada. Todo está enmarcado en un diseño minimalista y funcional.

La parte central muestra una lista de mensajes, cada uno representa como un bloque horizontal, el cual contienen el texto descriptivo representado como líneas horizontales grises, donde mostrarán el tipo de mensaje y la descripción. Además de dos botones a cada lado asociados a aceptar o rechazar los mensajes entrantes.

5. Implementación

Este capítulo explica cómo se ha llevado a cabo el desarrollo práctico de la aplicación, partiendo de la fase de diseño descrita anteriormente. Se profundiza en la arquitectura global, las decisiones tecnológicas, la configuración y la estructura del proyecto, tanto en backend como en front-end. Además, se describe cómo se han implementado las funcionalidades, la lógica interna y la interacción entre servidor y cliente para cubrir los requisitos definidos en capítulos anteriores.

5.1. Arquitectura

La aplicación se compone de dos grandes subsistemas:

- Servidor
  - Lenguaje y framework: Se ha elegido Python como lenguaje, junto con FastAPI para crear la API REST.
  - ORM y base de datos: Se utiliza SQLAlchemy como ORM, conectando con una base de datos relacional.
  - Autenticación y seguridad: Implementada con JWT (JSON Web Tokens), librería python-jose para firmar y verificar tokens. Contraseñas hasheadas con bcrypt.
  - Organización: El servidor se ha dividido en routers que agrupan endpoints temáticos (por ejemplo, /auth, /admin, /users, etc.), cada uno con su propia lógica.
- Cliente
  - Framework: Next.js , aprovechando su enrutado automático y la posibilidad de server-side rendering.
  - Estado global / Contexto: Se hace uso de un AuthContext para gestionar el token y la información del usuario, permitiendo a cada componente y página saber si el usuario está autenticado y con qué roles.
  - Páginas: Vistas concretas para cada rol y funcionalidad: pacientes (/home, /analytics...), médicos (/patients), familiares (/family), administradores (/adminDashboard) y un conjunto de páginas comunes como /login, /register, /recoverPassword.

- Estilos y Responsividad: Uso de librerías CSS para un diseño responsive que se adapte a distintos dispositivos.

La comunicación entre ambos subsistemas se realiza por medio de peticiones HTTP a la API, donde se envía el token de autenticación en la cabecera para las operaciones que lo requieran. El servidor responde con datos en formato JSON, que el front-end consume para mostrar en las vistas y componentes.

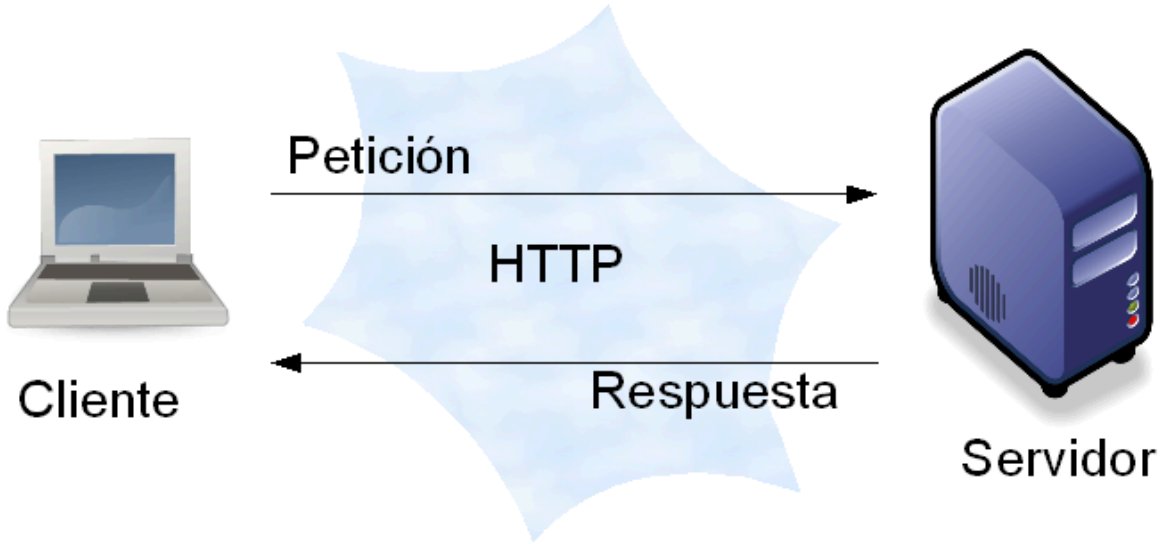


Figura 82: Modelo Cliente-Servidor

Esta arquitectura cliente-servidor *[27]* facilita la separación de responsabilidades: el back-end maneja la persistencia y la lógica de negocio, mientras que el front-end ofrece una experiencia interactiva y responsive, mostrando formularios y datos según el rol del usuario. La autenticación con JWR unifica el control de acceso en el back-end, permitiendo al front-end simplemente enviar el token y obtener la respuesta adecuada según las credenciales y permisos del usuario.

5.2. Detalles sobre implementación

A continuación se desarrolla sobre la Implementación, donde se profundiza en cómo se han materializado tanto la lógica de servidor back-end, como la interfaz front-end. En la primera sección, se presentan los aspectos más relevantes de la configuración y arquitectura del back-end, abarcando desde la creación y estructura del proyecto hasta la gestión de la base de datos y la documentación de la API. Posteriormente, en la segunda sección, se describe cómo se implementa el cliente con Next.js, detallando la organización de sus componentes, la estrategia de comunicación con el servidor y la manera de manejar estilos y validaciones. De esta forma, se ofrece una visión unificada del proceso de desarrollo, evidenciando las decisiones técnicas y las herramientas empleadas en cada capa.

5.2.1. Implementación del servidor

A continuación, se describe la Implementación del Servidor, donde se desarrollan los aspectos fundamentales que dan soporte a la lógica de negocio y a la comunicación con la base de datos. Veremos como en FastAPI se han organizado los ficheros y las dependencias de modo que cada parte del sistema se defina con claridad y resulte fácil de mantener y ampliar.

En los siguientes apartados se detalla, primero, cómo se inició el proyecto y cómo se gestiona el entorno de desarrollo, para luego exponer la estructura adoptada y el tratamiento de la base de datos y sus migraciones. Se profundiza también en la configuración de dependencias, mostrando cómo se inyectan la sesión de la DB o los métodos de autenticación, y se concluye explicando la generación y consulta de la documentación de la API REST. De esta manera, se ilustra en conjunto la arquitectura y la implementación que sostienen el núcleo del sistema.

5.2.1.1. Creación del proyecto

Vamos a ver como se ha ido creando el proyecto paso a paso en los siguientes puntos:

- Preparación del entorno de desarrollo

Se elige Python como lenguaje base. Para aislar las dependencias del proyecto y evitar conflictos con librerías del sistema, se crea un entorno virtual.

Al activar el entorno mediante el comando source venv/bin/activate en Unix o venv\Scripts\activate en Windows, se procede a instalar las librerías imprescindibles para el back-end como fatsapi, sqlAlchemy, bcrypt, python-jose, entre algunos ellos.

Se va a crear el fichero de dependencias requirements.txt para durante el proceso podemos ejecutar el comando pip freeze > requirements. txt . Este comando registra la lista de paquetes actual de su entorno en el archivo requirements, de modo que cualquier desarrollador pueda replicar el mismo entorno con el comando pip install -r requirements.txt

- Configuración inicial de fastAPI

A la hora de configurar FastAPI de forma inicial, se crea un archivo principal, denominado main.py, que será el punto de entrada de la aplicación. A continuación se describen con más detalle los pasos y opciones disponibles:

  - Creación y configuración del archivo main

Se inicia declarando una instancia de la aplicación con:

```
from fastapi import FastAPI

app = FastAPI()
```

De este modo, la app se convierte en el objeto principal sobre el cual se definirán rutas, middlewares, excepciones.

- Middleware y configuración: Si se requieren manejar solicitudes desde orígenes distintos, o interceptar peticiones para análisis, se configuran los middlewares.

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,  # CORSMiddleware permite que la API gestione solicitudes desde otros puertos
    allow_origins=['http://localhost:3000'],
    allow_credentials=True,  # Permite el envío de cookies y encabezados de autenticación en las solicitudes CORS
    allow_methods=['*'],  # Permite todos los métodos (GET, POST, PUT, DELETE, ...)
    allow_headers=['*']  # Permite que la solicitud CORS incluya cualquier encabezado
)
```

- Endpoint básico: Se pueden definir un endpoint en la raíz para verificar el correcto levantamiento del servidor, en nuestro caso tenemos:

```
@app.get("/")
def health_check():
    return 'Health check complete'
```

Con esto, al visitar `http://localhost:8000/`, se recibe un JSON confirmando que la API está en funcionamiento, lo que resulta útil en entornos de desarrollo y monitorización.

- Organización de rutas

En proyectos medianos o grandes, las rutas suelen definirse en archivos separados para cada módulo, en nuestro caso aunque es un proyecto pequeño pero

para tener un estilo de modularización, evitando que el main se sobrecargue con la lógica de las rutas, se ha creado el siguiente creado:

```
from .routers import auth, admin, glucose, notes, diagnosis, analytics, family, users, message, doctor

app.include_router(auth.router)
app.include_router(admin.router)
app.include_router(glucose.router)
app.include_router(notes.router)
app.include_router(diagnosis.router)
app.include_router(analytics.router)
app.include_router(family.router)
app.include_router(users.router)
app.include_router(message.router)
app.include_router(doctor.router)
```

- Ejecución con Uvicorn

Para arrancar la aplicación en modo de desarrollo, se usa el comando:

```
uvicorn main:app --reload
```

Aquí, ‘main:app’ hace referencia al archivo main.py y a la variable app. El flag ‘--reload’ habilita el hot-reloading, de manera que cualquier cambio en el código provoca el reinicio automático del servidor. En producción, se omite ‘–reload’ y se suelen aumentar los ‘workers’ (número de procesos de trabajo), o integrar un ‘process manager Docker’ (automatiza el despliegue de aplicaciones dentro de contenedores software) para asegurar la disponibilidad constante de la API.

Si se desea, se puede colocar un bloque condicional al final del main para ejecutar directamente uvicorn desde ahí:

```
import uvicorn

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)
```

Esto permite levantar la aplicación llamando a python con ‘python main.py’ en lugar de llamar manualmente a uvicorn.

- Definición del archivo conexión a la base de datos
    - Creación de database.py

En este proyecto con FastApi y SQLAlchemy, se genera un fichero denominado database.py que alberga la URL de conexión y la configuración principal de la base de datos. Al utilizar SQLite, el código adopta la siguiente forma:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
SQL_ALCHEMY_DATABASE_URL = f"sqlite:///{os.path.join(BASE_DIR, 'diabetes.db')}"

engine = create_engine(
    SQL_ALCHEMY_DATABASE_URL,
    connect_args={"check_same_thread": False}
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

El motor (`engine`) se construye con create_engine(), recibiendo la URL (sqlite:///...) y, en caso de SQLite, el parámetro check_same_thread (permite a FastAPI usar la misma base de datos SQLite en diferentes hilos, esto es necesario ya que una sola request podría usar más de un hilo). Para motores SQL (PostgreSQL, MySQL, etc.), se aplican otras URLs y configuraciones específicas.

- Creación de la sesión

SessionLocal = sessionmaker(...) define una fábrica de sesiones con las opciones `autocommit=False` (desactiva confirmación automática) y `autoflush=False` (desactiva el vaciado automático). Cada petición a la aplicación creará una instancia de sesión y luego se cerrará, a fin de manipular los datos de la DB.

- Gestión de variables de entorno
  - Fichero .env

Con el fin de proteger información sensible (claves secretas, conexiones a la base de datos en producción, etc.), se crea un archivo .env en el que se definen variables como:

```
AUTH_SECRET_KEY=**LaClaveSecreta**
ALGORITHM=HS256
API_URL=http://localhost:3000
```

En este fichero se excluye del repositorio de git para no comprometer la seguridad.

- Integración con python-dotenv

Se instala la librería python-dotenv y en el fichero deps lo llamamos con load_dotenv().

Esto carga automáticamente las variables definidas en .env y las hace accesibles a través de os.getenv(“Nombre de la variable”)..

Así se evita hardcodear datos confidenciales en el código, manteniendo la flexibilidad para cambiar configuraciones entre entornos sin alterar los archivos fuente.

- Inicialización básica

En el fichero main tenemos la función init_admin con el cual comprobamos la existencia de los roles predefinidos y si no existen, los crea en la base de datos.

Tras asegurarse de la presencia de roles, se revisa si hay un usuario administrador existente. Si no, se crea uno con el correo admin@example.com, con contraseña admin123 hasheada, y se le asigna el rol de administrador.

Esta rutina de inicialización garantiza que siempre haya un primer usuario de tipo administrador en el sistema con el cual gestionar la aplicación. Además, evita la inconsistencia en roles, sobre todo si en producción se requieren roles fijos que no pueden faltar.

La llamada a está función está al final del main, para que se ejecute automáticamente cuando se corre la aplicación. Si los roles y el usuario ya están creados, simplemente imprime el mensaje “El usuario administrador ya existe” y no hace cambios.

- Creación de Scripts auxiliares

Para cargar los datos proporcionados por el profesor, se requiere leer ficheros CSV que tiene información de pacientes, glucosa, diagnósticos y analíticas.

Por tanto se ha creado el script data_migration que hace uso de pandas para leer los ficheros y crear instancias de los modelos con la sesión de la base de datos.

Se ha usado para poblar la base de datos durante el proceso de prueba.

Se ha mantenido el código de migración separado, para evitar complicar el main con rutinas de carga de datos y únicamente ejecutarlo en el momento necesario.

5.2.1.2. Estructura del proyecto

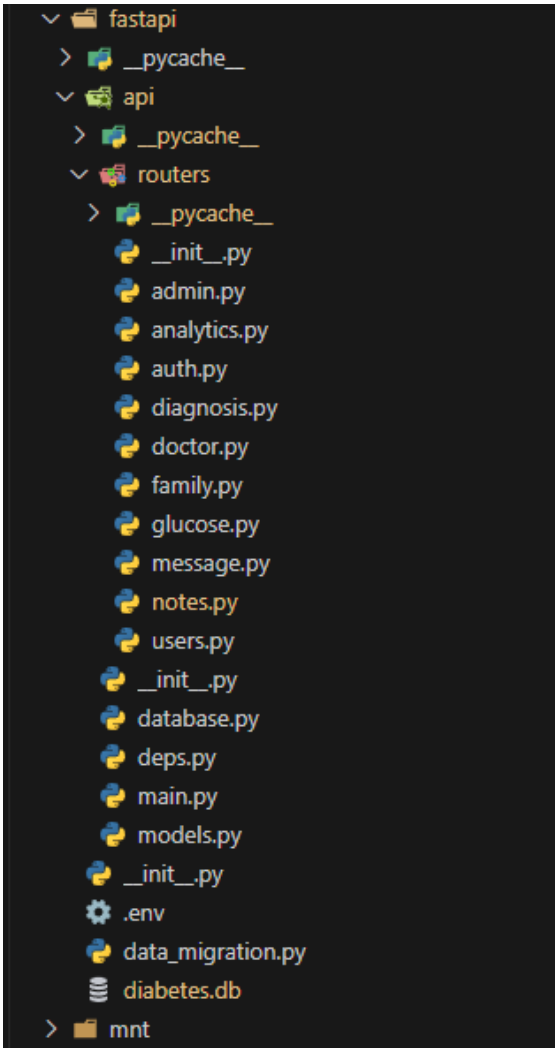


Figura 83: Jerarquía de carpetas en el servidor

Esta organización modular facilita la mantenibilidad: cada carpeta se hace responsable de una parte concreta modelos, rutas, dependencias, bases de datos, y main.py permanece limpio, limitándose a incluir los routers y realizar configuración general. Así cuando se añade una nueva entidad, se crea el modelo dentro de models.py, sus endpoint en la carpeta routers/. Se podría modularizar más dividiendo el fichero models.py por una carpeta models/ y dentro se encontrará separado cada modelo.

Así es una buena forma de escalar el proyecto sin caer en un único archivo sobrecargado.

5.2.1.3. Manejo de la base de datos o migraciones

- Conexión y creación de tablas

  Tal como se describió en la sección anterior, se ha implementado un fichero database.py que define la URL de conexión, un engine y además se establece la clase base de SQLAlchemy sobre la cual se declaran los modelos. En el fichero principal main se ejecuta:

```
Base.metadata.create_all(bind=engine)
```

  Para generar automáticamente las tablas en la base de datos si no existen. Esta operación se hace en el arranque de la aplicación, garantizando la consistencia del esquema sin necesidad de ejecutar instrucciones manuales.

- Modelos y Relaciones

  Las entidades como Usuario, Paciente, Médico, ... Se definen como clases que heredan de Base, estableciendo el nombre de la tabla con tablename, columnas con Column, llaves primarias, foráneas y relaciones con relationship.

  Aquí se muestra el modelo de Rol:

```
class Rol(Base):
    __tablename__ = 'roles'

    id = Column(Integer, primary_key=True)
    nombre = Column(String, unique=True, nullable=False)

    # Relación bidireccional con usuarios
                usuarios = relationship("Usuario", secondary=usuario_roles,
back_populates="roles")
```

Esta organización permite que SQLAlchemy mapee cada objeto en Python a la fila correspondiente en la tabla de la base de datos.

5.2.1.4. Dependencias

Dentro de la arquitectura de FastAPI, se usan las dependencias para inyectar automáticamente recursos en los endpoints.

Cualquier módulo que necesite interactuar con la base de datos se importa la sesión local con:

```
from .database import SessionLocal
```

- Inyección de de la sesión de base de datos

  Se define la función get_db() ubicada en deps.py que crea una instancia de sesión a partir de SessionLocal y la cierra al terminar la petición. De esta forma, cualquier endpoint que necesite acceso a la base de datos declara un parámetro que sea la llamada a la función.

  Esto evita que cada router tenga que abrir y cerrar conexiones manualmente, centralizando la gestión de la base de datos.

- Verificación de usuarios y roles

  Para el sistema de autenticación, se crea una función get_current_user que decodifica el token, busca el usuario y lo retorna si es válido.

```
async def get_current_user(token: oauth2_bearrer_dependency, db: Session = Depends(get_db)):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        user_id: int = payload.get('id')
        if user_id is None:
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail='No puede validar el usuario')
```

```
        #realizar la consulta a la base de datos para obtener el usuario completo
        user = db.query(Usuario).filter(Usuario.id == user_id).first()
        if user is None:
                raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail='Usuario no encontrado')

        return user
    except JWTError:
                raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail='No puede validar el usuario')
```

Asimismo, se define verificación de rol, como la función verificar_rol_administrador, que lanza una excepción si el usuario no posee el rol administrador.

Al centralizar la lógica de cada dependencia en funciones específicas, cada endpoint es más conciso, y se reduce la duplicación de código.

FastAPI resuelve la cadena de dependencias de forma automática, de modo que si “verificar_rol_administrador” a su vez depende de “get_current_user”, el framework se encarga de proveer, como vemos en el código de abajo.

```
def verificar_rol_administrador(user: Usuario = Depends(get_current_user)):
    #comprobamos si el usuario tiene el rol de administrador
    for rol in user.roles:
        if rol.nombre == "administrador":
            return user

    # si el usuario no tiene el rol de administrador, lanzamos una excepción
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail='No
tiene permisos para acceder a este recurso')
```

5.2.1.5. Generación y consulta de documentación API REST

Una de las ventajas más destacadas de FastAPI es su capacidad para generar automáticamente la documentación de la API, gracias al uso de pydantic y las anotaciones de tipos.

- Autodocumentación con OpenAPI

Al levantar el servidor de FastAPI, se crean dos rutas especiales:
http://localhost:8000/docs
http://localhost:8000/redoc

En docs, aparece una interfaz interactiva que permite ver todos los endpoints, sus métodos, parámetros y probar peticiones con distintos valores, un estilo a lo que haríamos con Postman.

En redoc, se muestra la documentación en un formato más estético, limpio y organizado, sin la posibilidad de poder ejecutar nuestros endpoints.

Más abajo se ve una comparación.

- Especificación de los modelos

Al definir modelos de entrada/salida con pydantic, FastAPI aprovecha esas definiciones para describir la forma de los datos que se envían o reciben. Eso se refleja en la documentación, donde cada endpoint tendrá un apartado “Request body” y “Response body” detallado.

Aqui vemos un ejemplo para crear las analiticas:

```
class BioquimicoCreate(BaseModel):
    id_paciente: int
    fecha_recogida: date
    nombre: str
```

```
    valor: float

@router.post("/", status_code=201)
def create_analytics_data(
    bioquimico_data: BioquimicoCreate,
    db: Session = Depends(get_db)
):
...
```

Así es como se vería en redoc

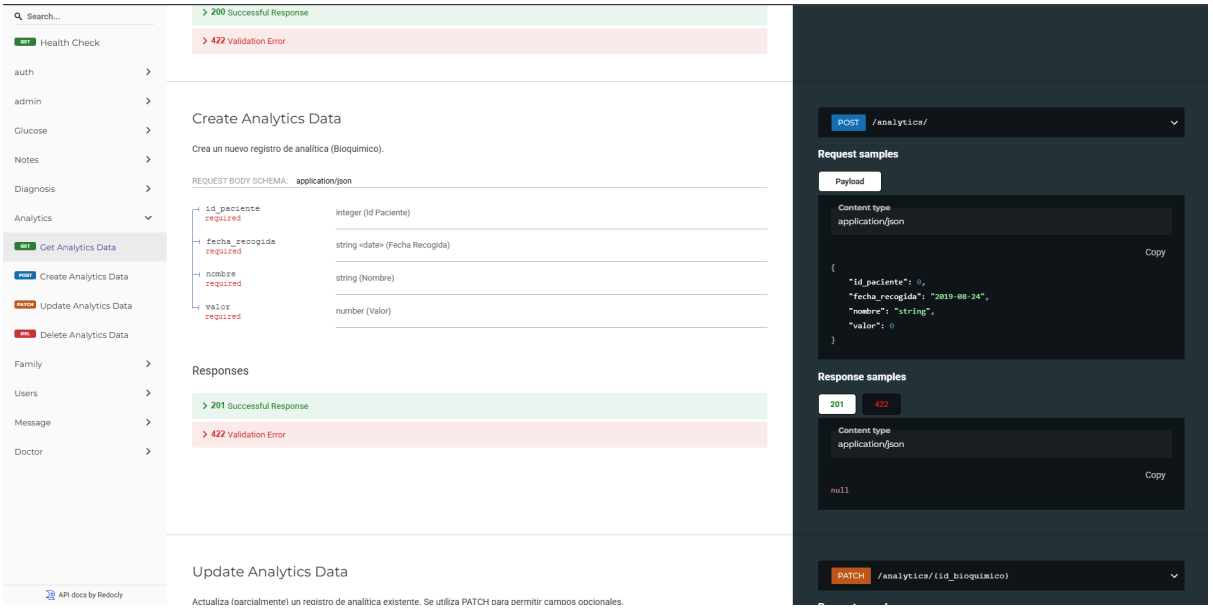


Figura 84: Documentación de la API con redoc

- Descripción de endpoints y parámetros

Cada ruta puede proporcionar descripciones, sumario, ejemplos:

```
@router.post("/", status_code=201)
def create_analytics_data(
```

```
    bioquimico_data: BioquimicoCreate,
    db: Session = Depends(get_db)
):
    """
    Crea un nuevo registro de analítica (Bioquímico).
    """

...
```

Así la documentación generada mostrará la descripción en la vista de docs, facilitando la compresión y el uso por parte de terceros.

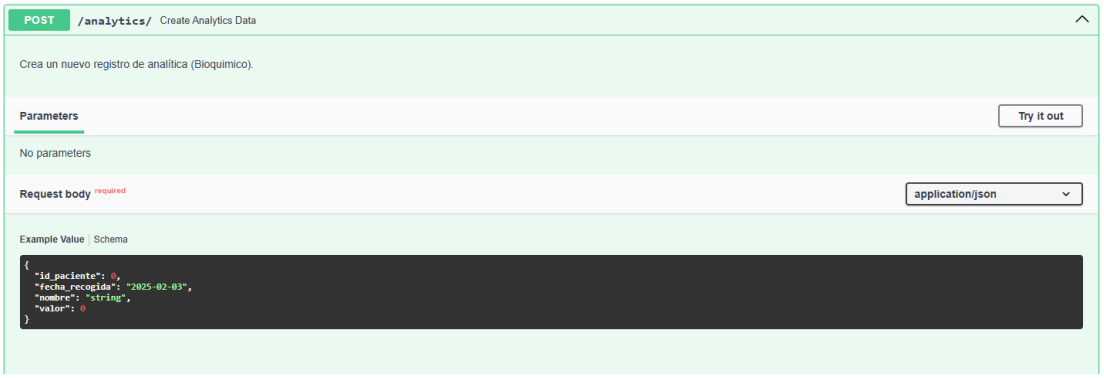


Figura 85: Documentación de la API con docs parte 1 de 2

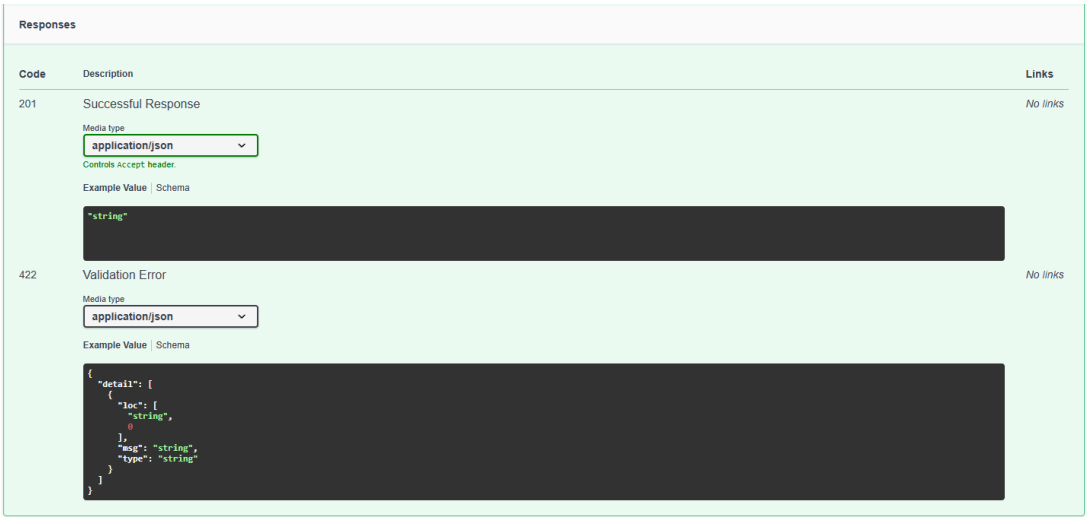


Figura 86: Documentación de la API con docs parte 2 de 2

5.2.2. Implementación del cliente

A continuación, se expone la Implementación del Cliente, donde se describe en detalle cómo se ha desarrollado la interfaz de la aplicación con Next.js. En los distintos apartados, se aborda desde la creación del proyecto y la organización de carpetas, hasta la definición de componentes, la manera de manejar estilos y responsividad, la comunicación con el servidor y, finalmente, las validaciones en el cliente. Así, se proporciona una visión completa de la arquitectura front-end, reflejando cómo se han coordinado la lógica y el diseño para ofrecer una experiencia de usuario fluida y coherente.

5.2.2.1. Creación del proyecto

- Inicialización con create-next-app

  Para agilizar el proceso de configuración de Next.js, se ejecuta el siguiente comando en una terminal

```
npx create-next-app@latest nombre-del-proyecto
```

  Este generador crea una estructura básica con carpetas, un archivo package.json, y varios scripts útiles para su despliegue.

  Tras completar, se obtiene el directorio listo para ser usado.

- Instalación de dependencias

  Una vez se accede a la carpeta del proyecto, se instalan las librerías adicionales que se requieran:

  - axios para realizar peticiones HTTP al back-end.
  - bootstrap para facilitar el diseño responsivo.
  - react-plotly.js para realizar las gráficas de datos de la glucosa.
  - react-datepicker para el manejo de fechas.
  - react-dom permite el enrutamiento de manera eficiente y sencilla.
  - react-icon permite implementar fácilmente iconos.

El package.json refleja las nuevas dependencias agregadas, garantizando la reproducibilidad del entorno para todos los integrantes del equipo.

- Scripts y Arranque

  Por defecto, Next.js genera varios scripts dentro de package.json:

  - dev: Corre la aplicación en modo desarrollo (next dev), con hot-reloading.
  - build: Genera un build optimizado para producción (next build).
  - start: Arranca el build ya generado (next start).

  Para iniciar el proyecto localmente en modo desarrollo, se usa:

```
npm run dev
```

La aplicación, por defecto, se ejecuta en http://localhost:3000, lo cual encaja con la configuración CORS del back-end que se habilitó allow_origins=["http://localhost:3000"].

- Integración con Git

  Es muy importante la iniciación de un repositorio Git dentro de la carpeta del proyecto y crear un fichero .gitignore para excluir la carpeta node_modules, ya que al ser muy pesada dependiendo del proyecto y en caso de recrear el proyecto se puede usar el siguiente comando para crear la carpeta node_modules automáticamente a partir de nuestro package.json

```
npm install
```

5.2.2.2. Estructura del proyecto

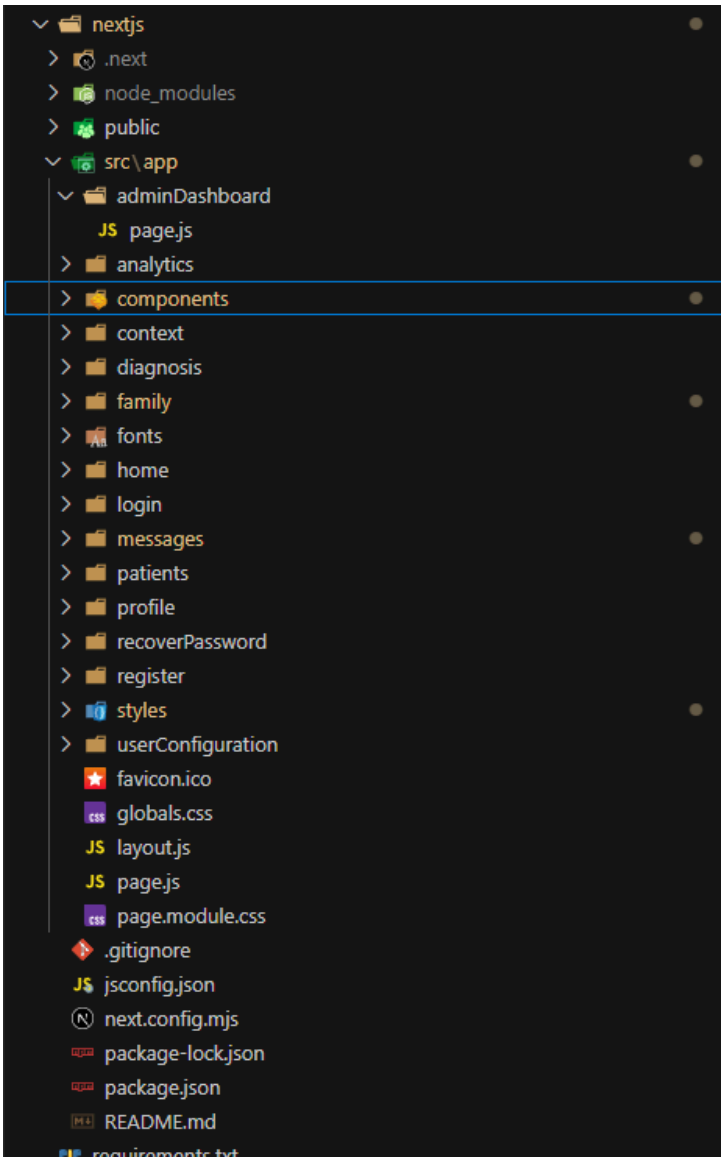


Figura 87: Jerarquía de carpetas en el cliente

5.2.2.3. Definición de componentes

La filosofía de React y por ende de Next.js es facilitar la descomposición de la interfaz en bloques funcionales independientes. Esto facilita la mantenibilidad, ya que un cambio en un solo archivo afecta a toda la aplicación sin tener que modificar múltiples archivos. Así, en la carpet components/ se agrupan archivos .js que representan cada componente, eso hace a la aplicación más escalable ya que si se

introduce nuevos componentes, se pueden ubicar en la carpeta correspondiente, sin mezclarse con el resto. A continuación se enseña una imagen de todos los componentes creados y se destacan algunos de ellos más usados:

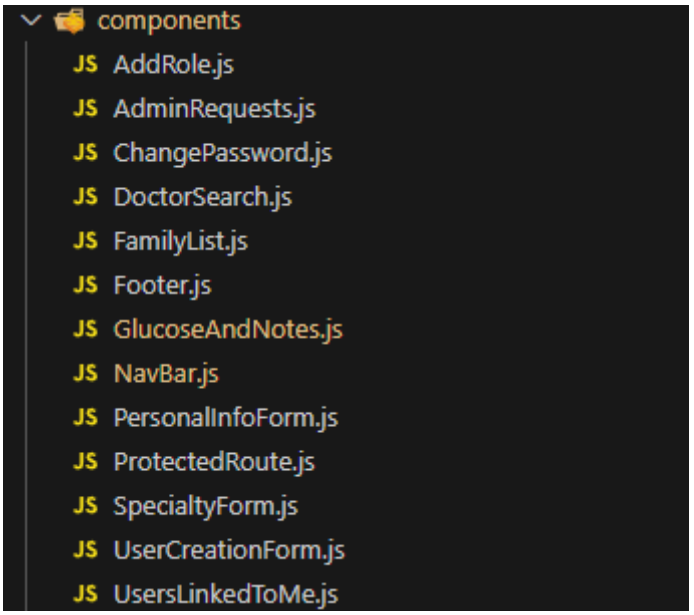


Figura 88: Componentes

- NavBar

  Barra de navegación que se muestra en la parte superior de las vistas.

  Dependiendo de los roles del usuario (paciente, médico, familiar, administrador), mostrará algunos enlaces u otros.

- Footer

  Pie de página común para todas las secciones, conteniendo enlaces, créditos o información de contacto. Se ubica al final de la interfaz y no presenta gran complejidad, pero aporta consistencia visual.

- ProtectedRoute

  Un componente que comprueba si el usuario posee un token o rol antes de renderizar la vista solicitada. Su tarea fundamental es redirigir al login o mostrar un mensaje de no autorizado si el usuario no cumple las condiciones.

Cabe destacar que cada componente tiene un estilo particular que está en la carpeta styles y se asocia un .module.css con el mismo nombre y se importa solo en ese archivo, garantizando el alcance de clases en Next.js

5.2.2.4. Estilos y responsividad

- Estrategia de Estilos con Next.js

En Next.js, normalmente se cuenta con un CSS [28] global (`globals.css`), que puede ser importado en las páginas principales para aplicar reglas generales para resetear los márgenes, tipografías base, colores globales, etc.

Para cada componente o página, se puede crear un archivo `.module.css` que define clases de forma local. Esto evita colisiones de nombres y promueve la modularidad.

Durante el build, Next.js genera nombres de clase únicos, garantizando que los estilos no se sobreescriben accidentalmente entre componentes.

- Uso de Frameworks CSS y Librerías

Para agilizar la creación de una interfaz atractiva, se ha recurrido a Bootstrap *[29]*. Se importa la librería en la página principal layout, quedando disponible en toda la aplicación. Esto permite usar su sistema de grid y clases de componentes interno de bootstrap y facilitando el diseño responsive.

Algunas vistas requieren componentes de interfaz más avanzados, como modales, menús desplegables, etc. Se integran con JavaScript de Bootstrap.

- Responsividad

Para adaptarse a pantallas pequeñas (móviles) y grandes (escritorio), se definen reglas media queries en `.module.css`algo como:

```
@media (max-width: 768px) {
  .container {
    flex-direction: column;
```

```
  }
}
```

O también se puede codificar con bootstrap que trata el contenedor como un grid de 12 columnas el cual especificamos cuantos va a ocupar en cada momento por ejemplo:

“col-md-6 col-sm-12”, cambiando de dos columnas en pantallas medianas a una sola en móviles.

5.2.2.5. Servicios y peticiones al servidor

- Enfoque general de comunicación

El cliente necesita realizar peticiones al back-end para autenticar usuarios, obtener datos y enviar nueva información, entre algunos casos. Para ello, usamos axios a fin de enviar y recibir datos en formato JSON [30]. Al cual vamos hacer la llamadas a la dirección de nuestro servidor back-end en nuestro caso http://localhost:8000.

- Peticiones dentro de cada componente

En lugar de centralizar la lógica de peticiones en un archivo de servicios, la aplicación efectúa llamadas al back-end directamente desde los componentes que las necesitan. Este enfoque conlleva que cada componente hace llamadas y define sus funciones de obtención o envío de datos, además de controlar sus propias peticiones y estados.

Ventajas que podemos encontrar es la lógica de cada funcionalidad dentro de su componente, lo que puede resultar intuitivo si no se comparten peticiones entre varias partes de la interfaz.

Desventajas pueden ser que se duplican en parte las llamadas, o los encabezados y URL base, en varios componentes. Sin embargo, en proyectos de tamaño mediano o pequeños, no son tan engorrosos.

5.2.2.6. Validaciones en el cliente

- Por qué validar en el cliente

La validación en el front-end agiliza la retroalimentación al usuario, por ejemplo:

Impedir que se envíe un formulario de registro si faltan campos obligatorios o el email no tiene un formato válido. Esto evita peticiones que claramente resultan en error, optimizando la comunicación con el back-end.

Aun así, el servidor seguirá validando todos los datos, por razones de seguridad y consistencia, pero se mejora la usabilidad al evitarle al usuario idas y venidas por motivos básicos.

- Metodología de Validación
  - Validación manual: Cada componente o formulario chequea los campos de manera imperativa.
  - Hooks o librerías: Hay bibliotecas como React Hook Form, Formik, o Yup que facilitan la gestión de formularios y la declaración de reglas.
  - HTML5: Para validaciones simples, se puede usar atributos nativos (required, type="email") y capturar los errores en el evento onSubmit.

- Ejemplo práctico del formulario de login

...

```
const handleSubmit = async (e) => {
  e.preventDefault();
  setError("");
  try {
    await login(email, password);
  } catch (err) {
    setError(err.message);
  }
```

```
};

...
        <form onSubmit={handleSubmit}>
          <div className="mb-3">
            <label htmlFor="email" className="form-label">
              Email
            </label>
            <input
              type="email"
              className="form-control"
              id="email"
              value={email}
              placeholder="Introduce el correo electrónico"
              onChange={(e) => setEmail(e.target.value)}
              required
            />
          </div>
          <div className="mb-3">
            <label htmlFor="password" className="form-label">
              Contraseña
            </label>
            <input
              type="password"
              className="form-control"
              id="password"
              value={password}
              placeholder="Introduce la contraseña"
              onChange={(e) => setPassword(e.target.value)}
              required
            />
...
```

Arriba he añadido un fragmento del código de login en el cual podemos ver que se hace una validación mínima con los tipos de los inputs y los campos obligatorios (required).

5.3. Despliegue

El despliegue de la aplicación implica definir cómo se lanzan los servicios de back-end (FastAPI + Base de datos) y front-end (Next.js) de forma coordinada. Dependiendo de la envergadura del proyecto, se puede optar por un entorno local (para desarrollo y pruebas) y por un entorno de producción en un servidor o plataforma en la nube.

5.3.1. Entorno local

- Instalación de Dependencias
    - Servidor (Python + FastAPI)
        1. Clonar o descargar el repositorio del proyecto.
        2. Entrar en la carpeta correspondiente (por ejemplo, /app o /backend).
        3. Crear y activar un entorno virtual:

```
python -m venv venv
venv\Scripts\activate # Windows
# o source venv/bin/activate en Unix
```

        4. Instalar dependencias:

```
pip install -r requirements.txt
```

        5. Revisar el archivo .env o crear uno si no existe, para definir variables de entorno.

    - Cliente (Node.js + Next.js)
        1. Ir a la carpeta del front-end.

2. Ejecutar npm install para descargar las librerías definidas en package.json.

- Arranque de la Aplicación
  - Back-end
    1. Desde la carpeta del servidor, se lanza el servidor con:

```
uvicorn main:app --reload
```

El flag --reload permite recargar el back-end al guardar, se suele usar en desarrollo.

2. Verificar que se haya creado la base de datos.
3. Si se necesita poblar datos, se corre el script data_migration.py

  - Front-end
    1. Desde la carpeta client, ejecutar:

```
npm run dev
```

2. Por defecto, Next.js abre en http://localhost:3000.
3. Si el CORS está configurado en el back-end para permitir http://localhost:3000, se podrán realizar peticiones sin problema.

- Verificación
  - Abrir en el navegador:
    - Front-end: http://localhost:3000
    - Back-end: http://localhost:8000 (opcionalmente http://localhost:8000/docs o redoc).
  - Probar un login o visitar alguna ruta para constatar que la comunicación entre front-end y back-end funciona.

5.3.2. Entorno de producción

El despliegue en producción puede variar según la infraestructura disponible (servidores propios, servicios en la nube, contenedores, etc.). A continuación se describe un escenario genérico:

- Construcción de la aplicación
  - Back-end:
    1. Se opta por un entorno (por ejemplo, Ubuntu) con Python 3.9+ instalado.
    2. Se crea un entorno virtual y se ejecuta pip install -r requirements.txt.
    3. Se define un .env con las variables adecuadas .
    4. Se levanta con uvicorn main:app --host 0.0.0.0 --port 8000. En producción, se puede usar gunicorn junto a uvicorn workers o un process manager.
  - Front-end:
    1. Se corre npm install y luego npm run build en la carpeta del cliente.
    2. Se lanza con npm start o se implementa el proyecto en un hosting específico de React/Next.js (por ejemplo, Vercel).
    3. Se asegura que las peticiones al back-end apunten a la URL real.

- Configuración de Proxy o SSL
  - A menudo se coloca un reverse proxy (Nginx, Caddy, Apache) que maneja los certificados y enruta /api/ a la máquina o contenedor donde corre FastAPI, y el resto a la aplicación Next.js.
  - Por ejemplo, se puede tener un dominio principal midominio.com que sirva la app Next.js, y midominio.com/api que redirija internamente al servidor uvicorn en el puerto 8000.

- Manejo de Base de Datos

- Se puede optar por una instancia de PostgreSQL o MySQL en un servicio gestionado (AWS, Azure, etc.) o en el mismo servidor, manteniendo la variable DATABASE_URL.
- Si se requiere escalabilidad, se añade un cluster de bases de datos o un contenedor Docker.

- Estrategia de Actualización
  - Cada vez que se quiere actualizar la aplicación:
    1. Se hace git pull o se despliega la nueva versión.
    2. En el back-end, se ejecuta pip install -r requirements.txt si las dependencias cambiaron.
    3. En el front-end, se realiza npm install && npm run build.
    4. Se reinicia el servidor uvicorn.

- Uso de Docker
  - Una alternativa popular es empaquetar tanto el back-end como el front-end en contenedores Docker con sus respectivos Dockerfile. Se define un docker-compose.yml que inicia ambos servicios más la base de datos. Esto facilita la portabilidad y la repetición del entorno local/producción con la misma funcionalidad.

6. Pruebas

En este capítulo el objetivo es asegurar que la aplicación cumple los requerimientos planteados y que cada flujo funcione de forma adecuada. A continuación se detallan las dos vertientes principales de pruebas tanto al back-end como al front-end, además de las herramientas o metodologías usadas para la validación.

6.1. Pruebas al Back-end

Dado que el servidor se basa en FastAPI, nos vamos aprovechar de su documentación automática para probar de manera sencilla las rutas (también se

podría hacer con Postman). No se va a implementar una suite de testing automático en esta parte del proyecto, sino que se realiza una validación manual a través de la interfaz generada por Swagger en http://localhost:8000/docs.

- Acceso a Swagger

Se arranca el servidor con uvicorn main:app --reload y se navega hasta http://localhost:8000/docs, donde se lista cada endpoint junto con sus métodos y los cuerpos de petición y respuesta.

- Ejecución de los test

Se pincha en cada operación, se rellenan los parámetros requeridos, en caso de los endpoints protegidos se hace clic en Authorize y se inicia sesión como administrador.

Al pulsar “Try it out” → “Execute”, la herramienta envía la petición al servidor y muestra la respuesta en JSON, junto con el código de estado.

- Tipos de pruebas
    - Casos correctos: Llamadas con datos válidos, que deberían devolver un código de estado 200 y la respuesta deseada.
    - Casos con errores: Llamadas con credenciales erróneas, endpoints que esperan un identificador inexistente, o datos insuficientes en la petición.
    - Verificación de Roles: Probar que un endpoint restringido al rol “administrador” retorne 403 si se accede con otro rol.

- Pruebas realizadas

A continuación se muestran algunos ejemplos de pruebas a diversos endpoints, demostrando su comportamiento y el resultado reflejado en la base de datos.

Para crear un usuario, no se requieren parámetros en la ruta; simplemente se envía en el cuerpo (body) de la petición un objeto con los atributos necesarios. En la figura siguiente (Figura 89), se ha rellenado un ejemplo de usuario de prueba, asignándole el rol de “paciente”. Al pulsar “Execute”, se envía la solicitud al servidor.

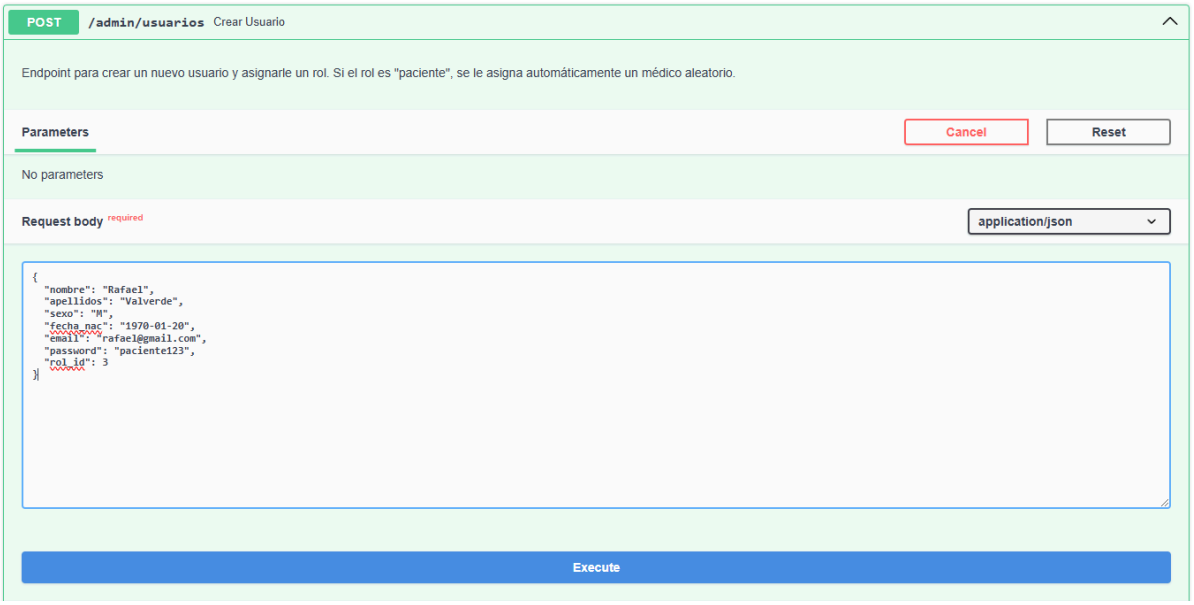


Figura 89: Método crear usuario parte 1

Tras ejecutar la llamada, el servidor responde con un código 201 y el mensaje “Usuario creado exitosamente” (ver Figura 90), lo que confirma que la operación se ha completado correctamente

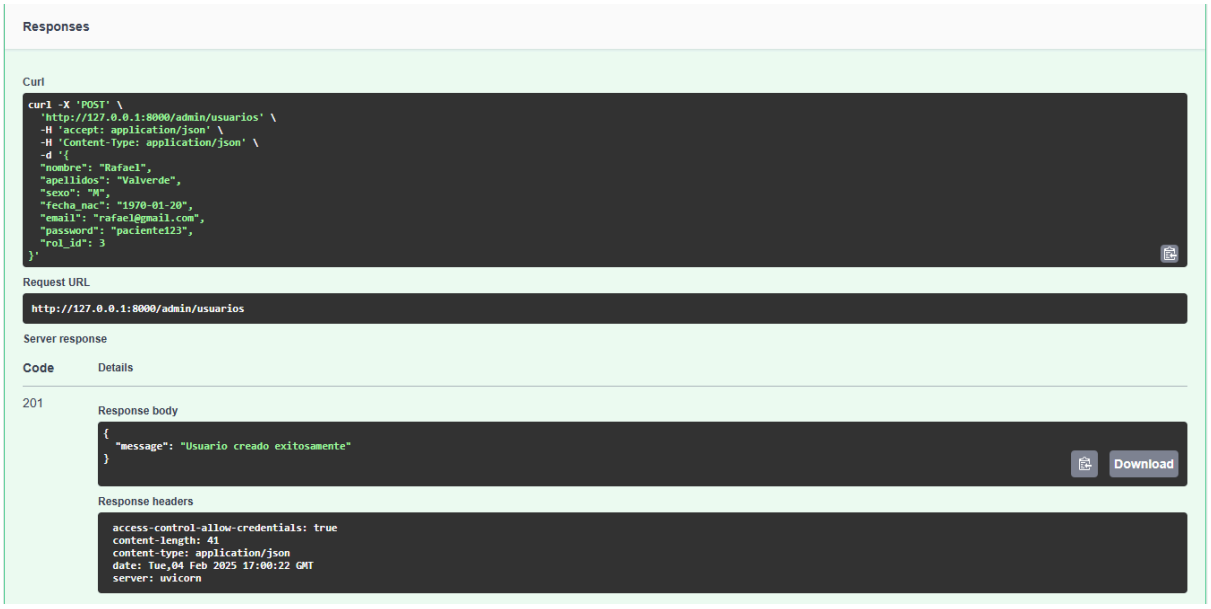


Figura 90: Método crear usuario parte 2

Para verificar que el nuevo usuario realmente se registra en la base de datos, la Figura 91 muestra una captura de la tabla usuarios, donde aparece el identificador del usuario y sus datos, incluida la contraseña hasheada.

839	Rafael	Valverde	M	1970-01-20	rafael@gmail.com	$2b$12$A0rDBlKQGUtx8zQ4NGfIJeUTdHneSaIZ7vr4ymX6cfoF2WXrW4oae

Figura 91: Método crear usuario comprobación BD

En la segunda prueba, se intenta crear un usuario omitiendo el campo correo electrónico en el JSON que se envía al endpoint (Figura 92).

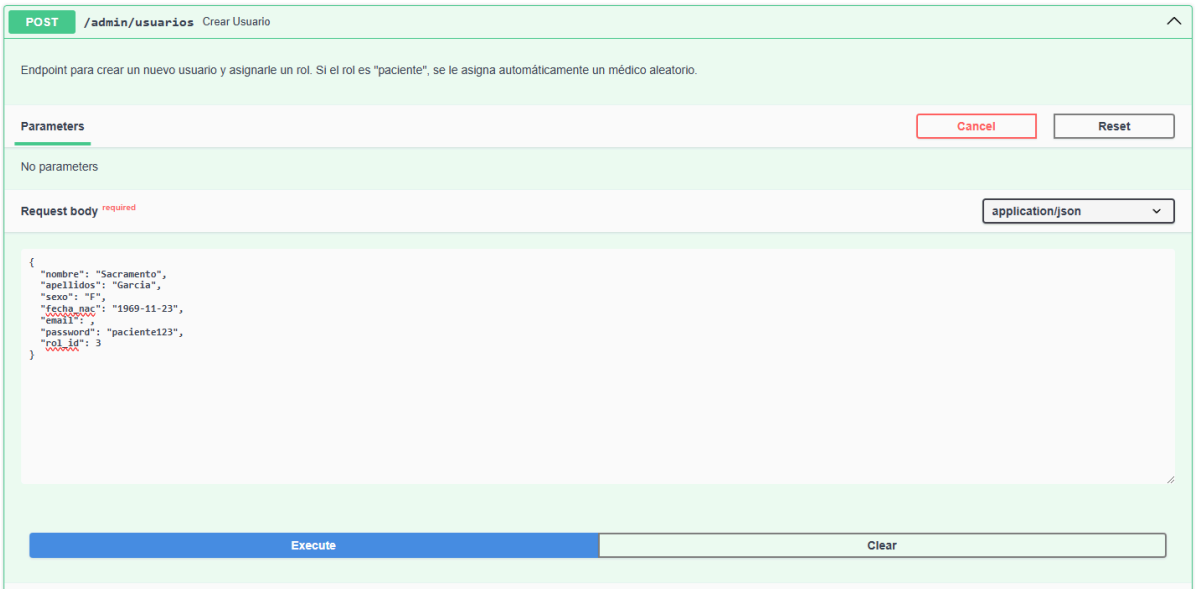


Figura 92: Método crear usuario erróneo parte 1

Al ejecutar la petición, se recibe un error 422 (Figura 93), indicando que el atributo requerido no está presente y, por tanto, la solicitud no se procesa.

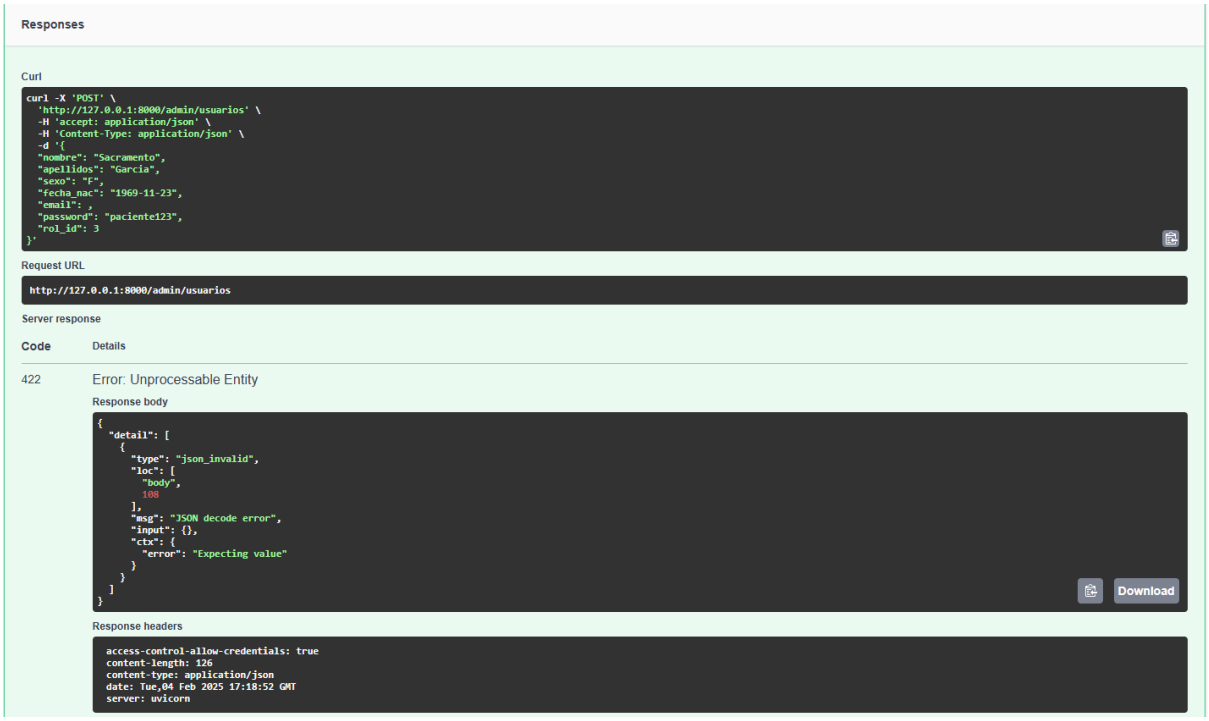


Figura 93: Método crear usuario erróneo parte 2

En la tercera prueba, se intenta crear un usuario añadiendo un rol que no existe en el JSON que se envía al endpoint (Figura 94), ya que los roles en la base de datos están definidos como 1 administrador, 2 médico, 3 paciente y 4 familiar.

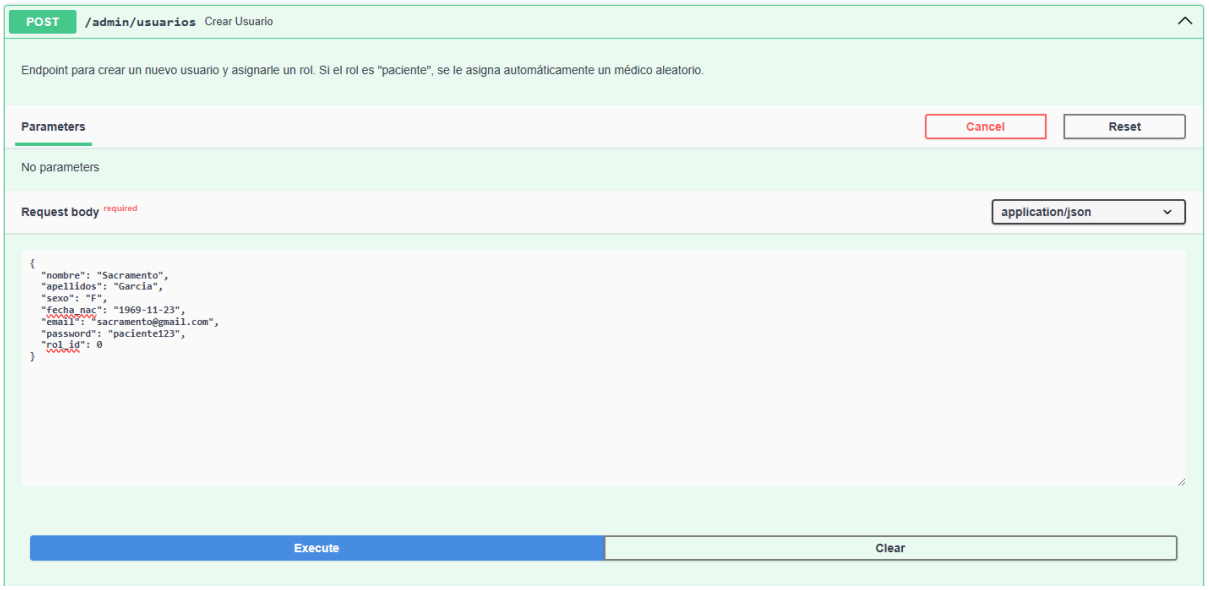


Figura 94: Método crear usuario erróneo rol parte 1

Al ejecutar la petición, se recibe un error 404 (Figura 95), indicando el mensaje 'El rol especificado no existe'.

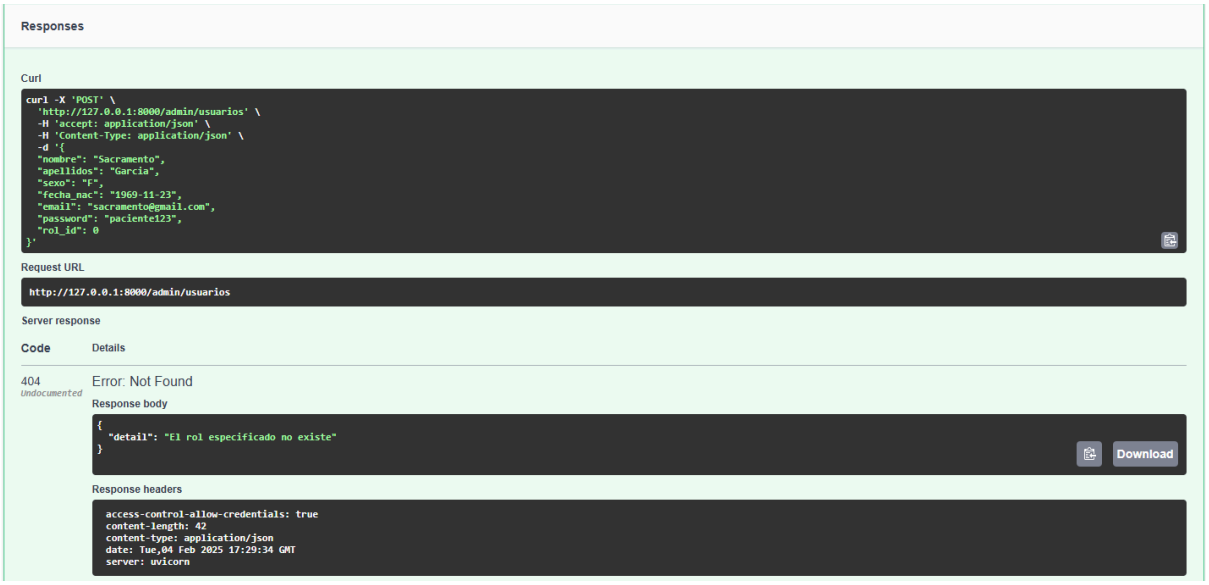


Figura 95: Método crear usuario erróneo rol parte 2

Para crear un registro de glucosa, no se necesitan parámetros en la ruta; en el cuerpo de la solicitud se proporciona el identificador del paciente, el nivel de glucosa, la fecha y la hora. En la Figura 96 se observa cómo se rellena este objeto antes de ejecutar la petición.

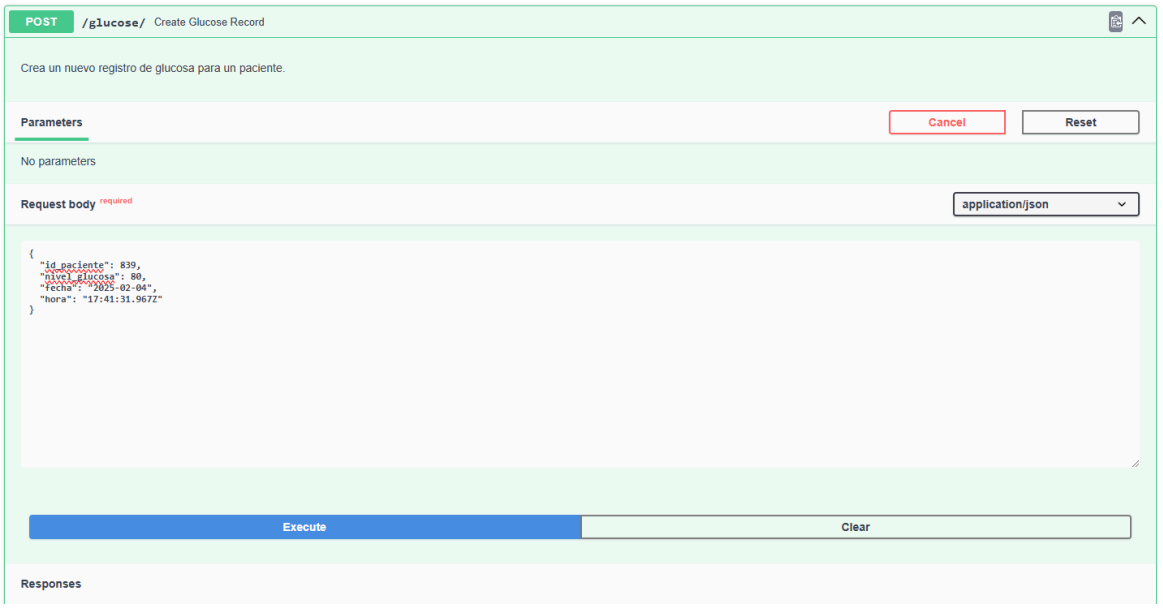


Figura 96: Método crear glucosa parte 1

Al pulsar “Execute”, la respuesta muestra un código 201 y el mensaje “Registro de glucosa creado exitosamente”. Además, devuelve el objeto creado con los datos registrados (Figura 97), confirmando la inserción correcta.

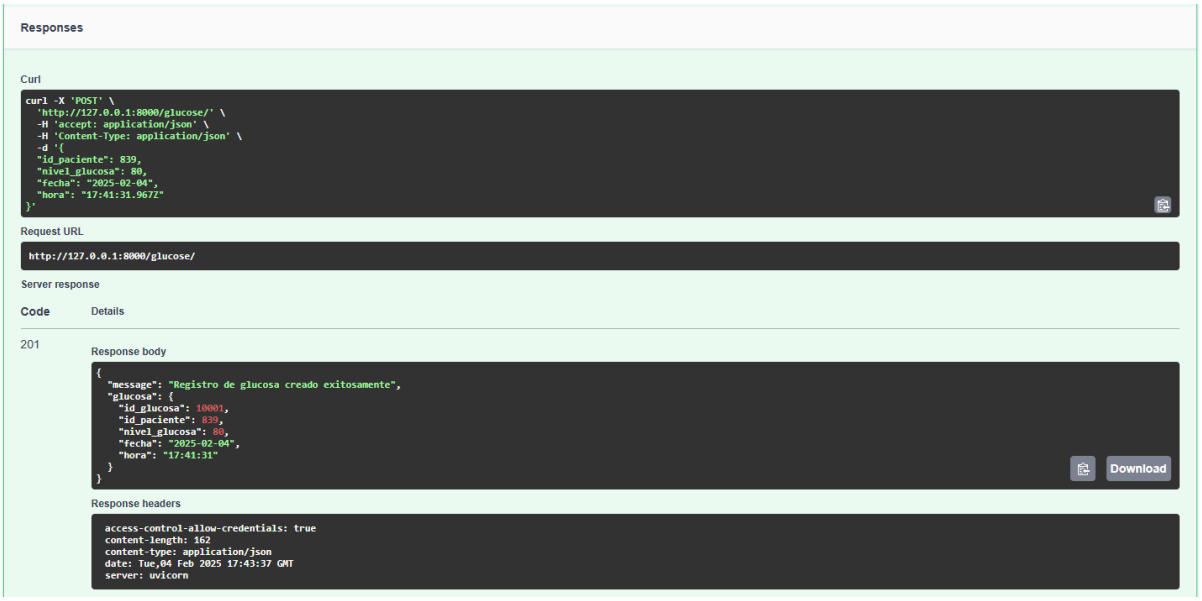


Figura 97: Método crear glucosa parte 2

En la siguiente prueba, se omite el identificador del paciente en el JSON. Como se ve en la Figura 98, esto busca comprobar la respuesta del servidor frente a datos incompletos.

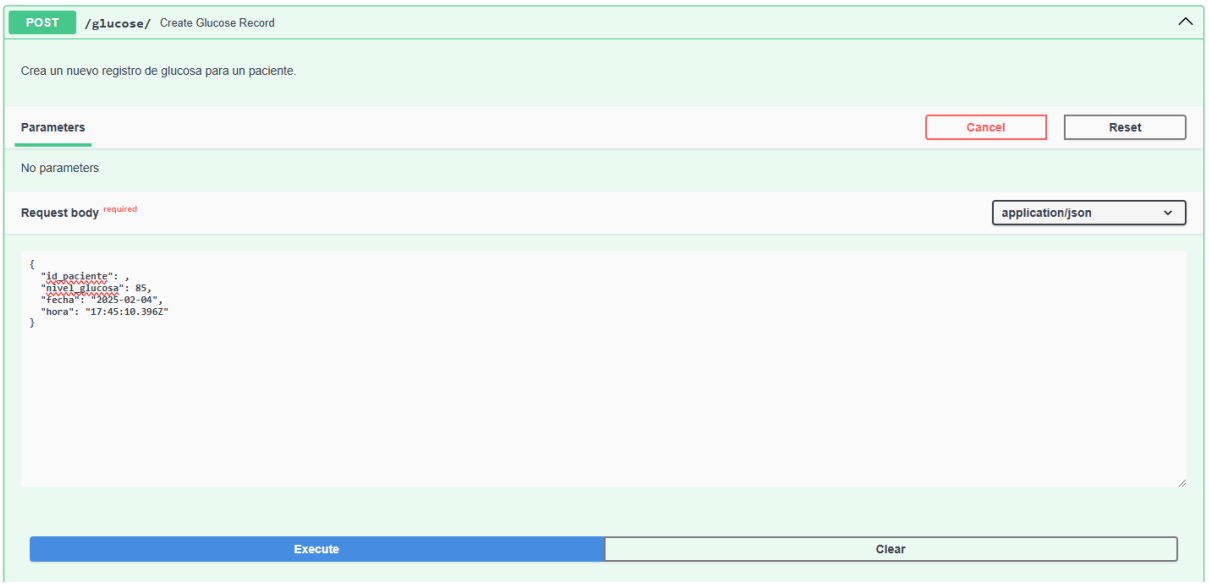


Figura 98: Método crear glucosa erróneo parte 1

Al lanzar la solicitud, el sistema emite un error 422 (Figura 99), indicando “Expecting value”. Esto ocurre porque el endpoint requiere el campo “id_paciente” para procesar la creación del registro.

Figura 99: Método crear glucosa erróneo parte 2

En este endpoint, se solicita la lista de mediciones de glucosa de un usuario concreto. Es preciso indicar el identificador del usuario en los parámetros de ruta. Al ejecutar la llamada, el servidor devuelve un código 200 y un array con todas las mediciones registradas, incluido el nivel de glucosa, fecha y hora (Figura 100).

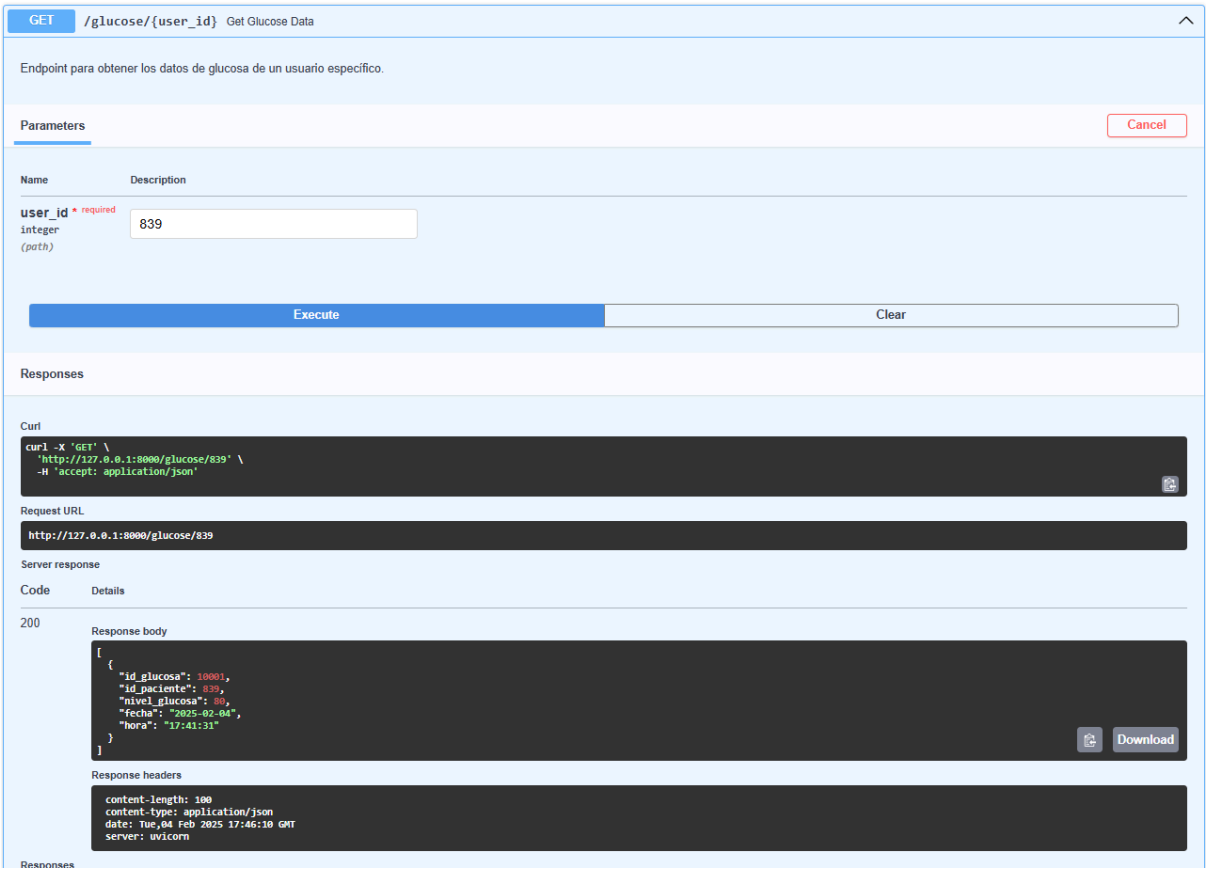


Figura 100: Método obtener glucosas

Para eliminar un registro de glucosa, se necesita el identificador de la misma en los parámetros. En la Figura 101 se muestra que, tras enviar la petición con un ID existente, se retorna un código 204, indicando que la operación se completó sin contenido adicional en la respuesta.

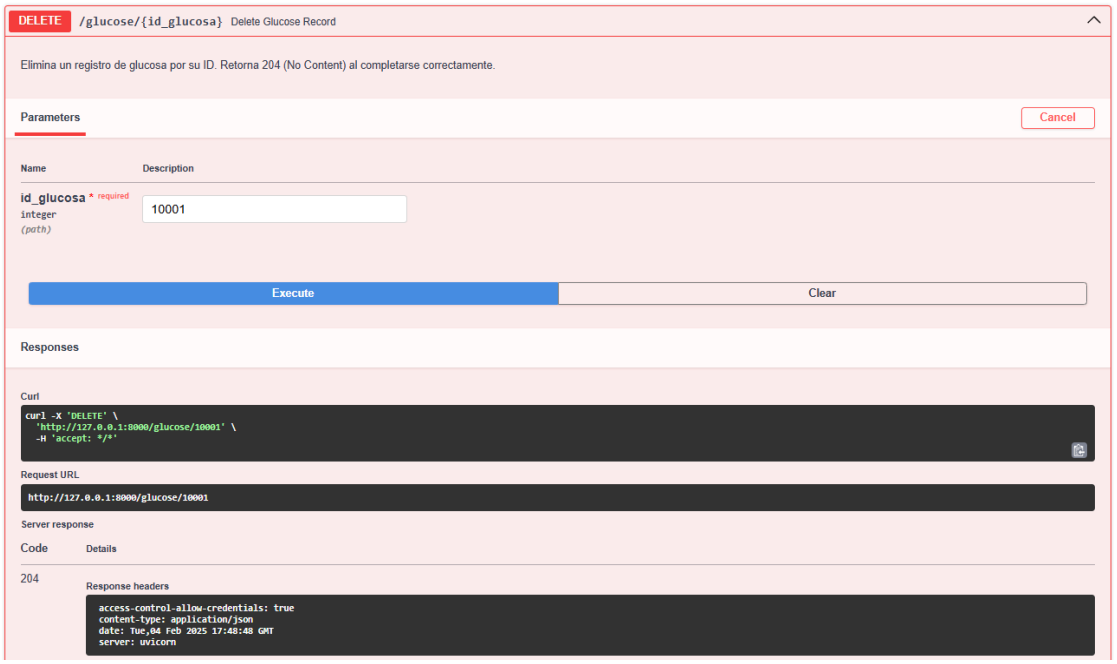


Figura 101: Método eliminar glucosa

Si se intenta eliminar una glucosa inexistente (pasando un ID que no está en la base de datos), el servidor responde con un error 404 y el mensaje “Registro de glucosa no encontrado” (Figura 102). Con esto se confirma la validación de IDs que no corresponden a ningún registro.

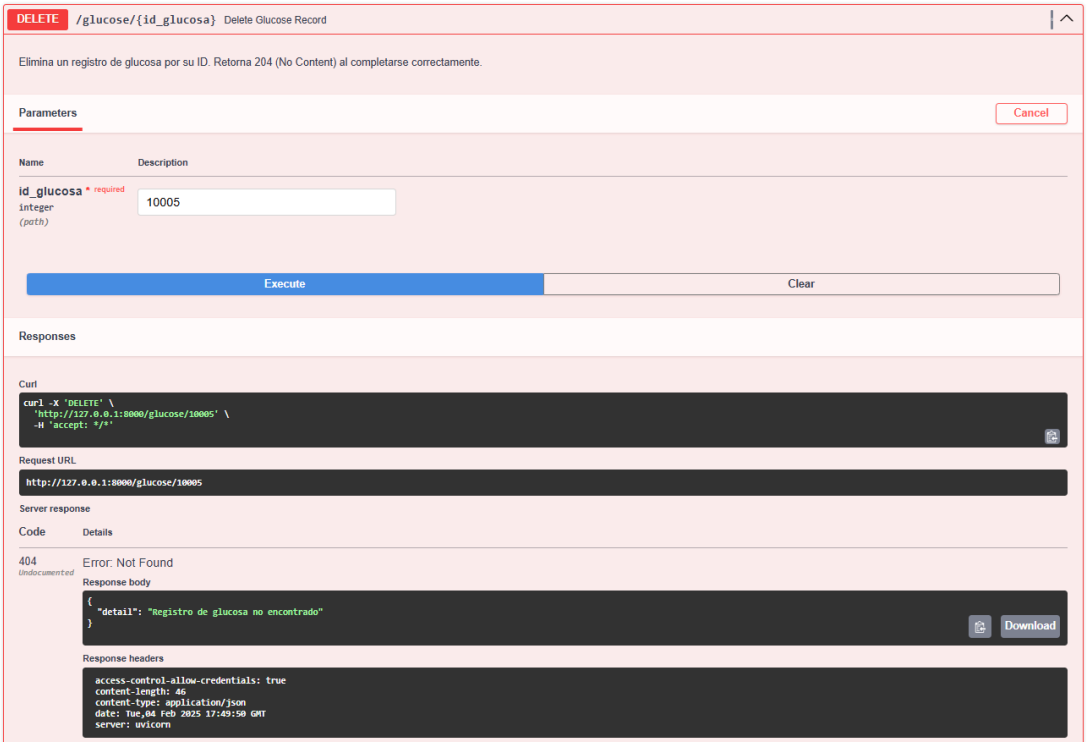


Figura 102: Método eliminar glucosa erróneo

6.2. Pruebas de front-end

El Front-end, desarrollado con Next.js, se valida de forma manual y visual, recorriendo cada vista e interacción para asegurar la usabilidad y correcta respuesta de la aplicación, todo ello sin instalar frameworks de test automatizados.

Se va a poner algunos ejemplos:

- Login
    - Probar credenciales válidas e inválidas

Como se observa en la siguiente captura (Figura 103), al introducir datos erróneos, la aplicación muestra el mensaje “Usuario o contraseña incorrectos”.

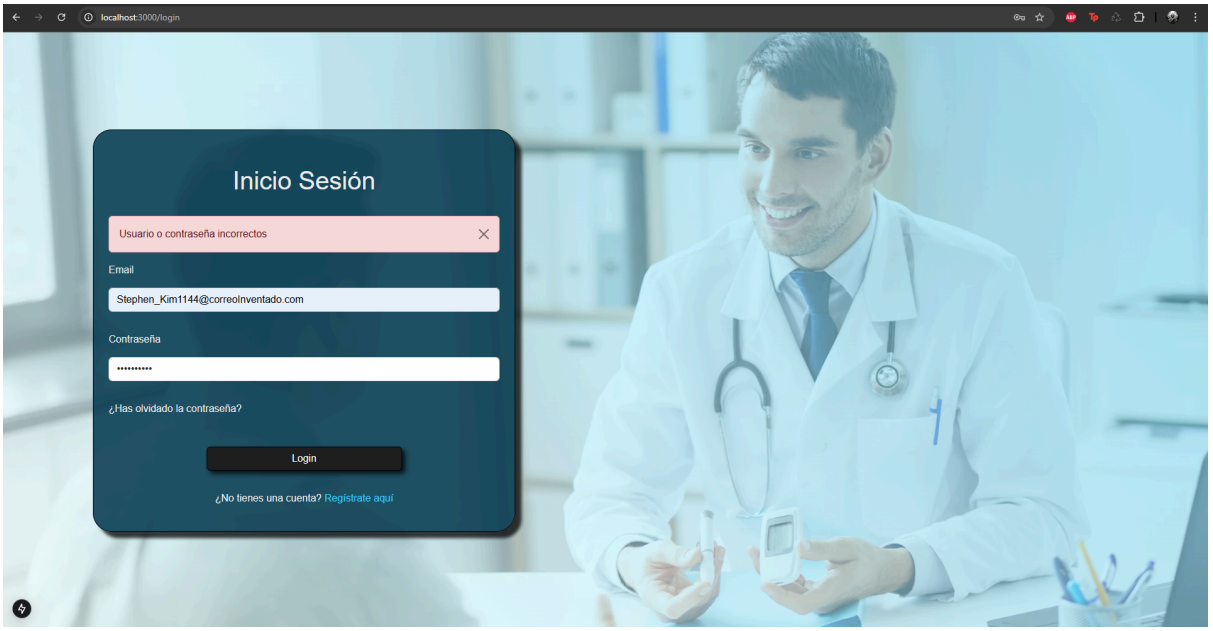


Figura 103: Login incorrecto

Una vez se ingresan los datos correctos, el sistema redirige a la pantalla inicial. En el ejemplo (Figura 104), el usuario tiene rol de paciente y familiar, motivo por el cual va a la vista principal del paciente.

Figura 104: Redirección a inicio

- verificar la redirección correcto a la vista principal de cada rol

En la imagen de abajo podemos observar la vista principal del rol del paciente.

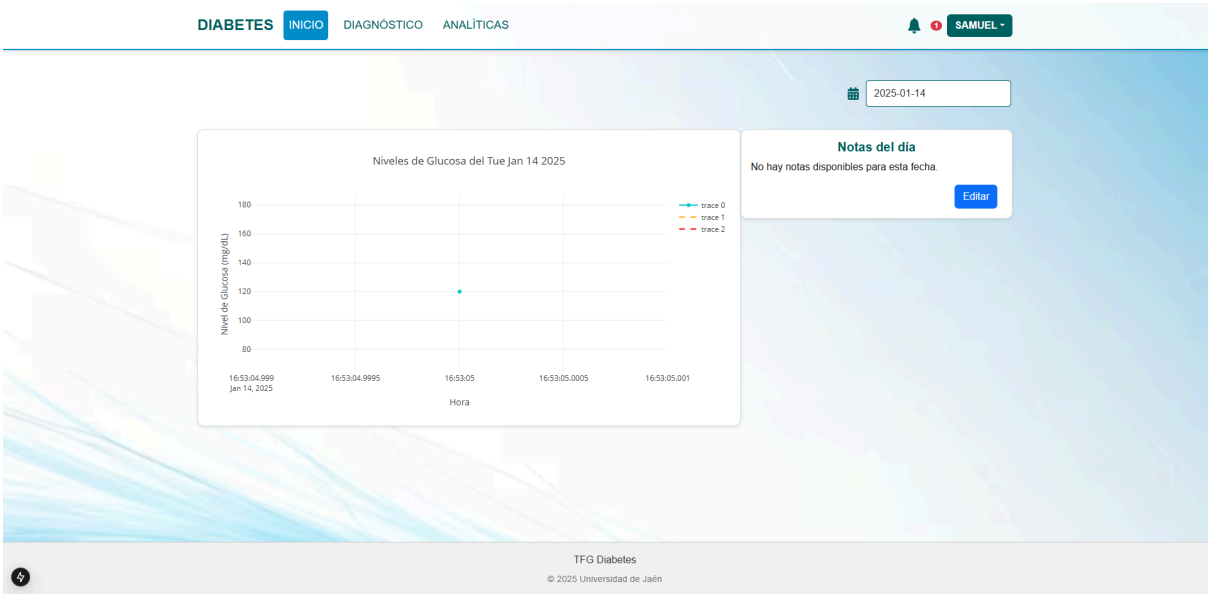


Figura 105: Vista inicio

En la imagen de abajo podemos observar la vista principal del rol del familiar.

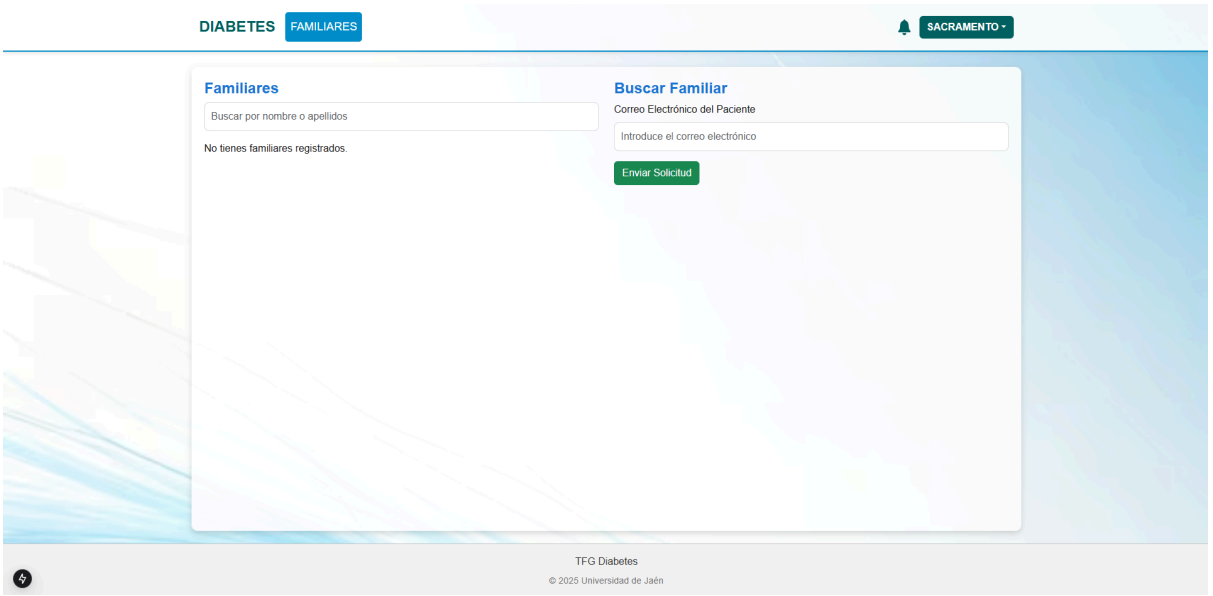


Figura 106: Vista familiares

En la imagen de abajo podemos observar la vista principal del rol del médico.

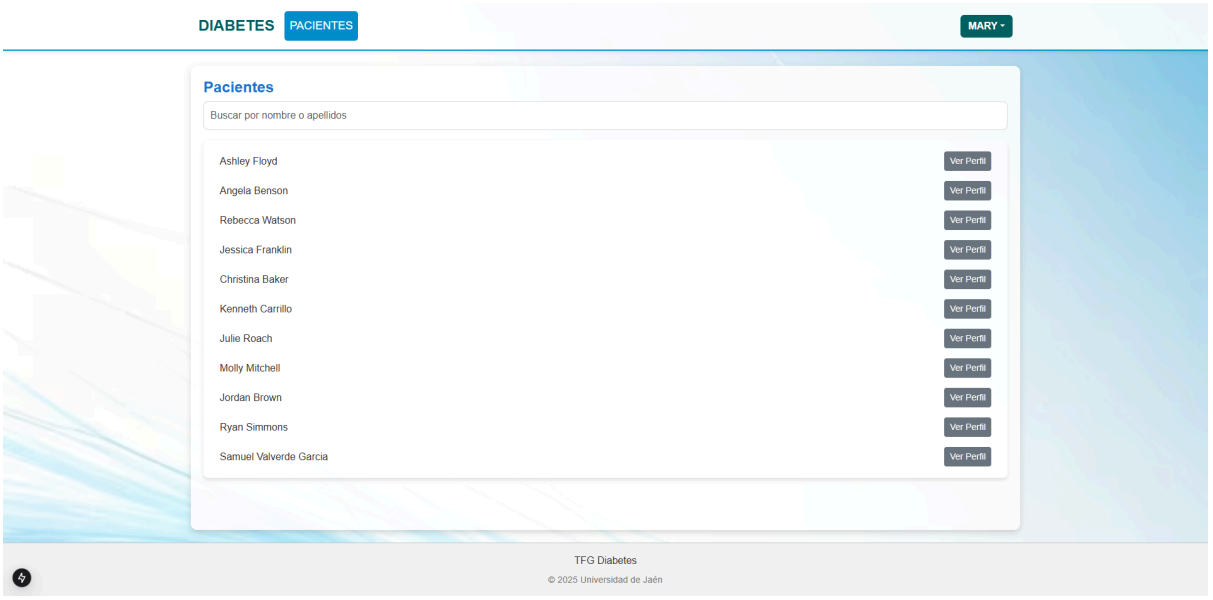


Figura 107: Vista pacientes

En la imagen de abajo podemos observar la vista principal del rol del administrador.

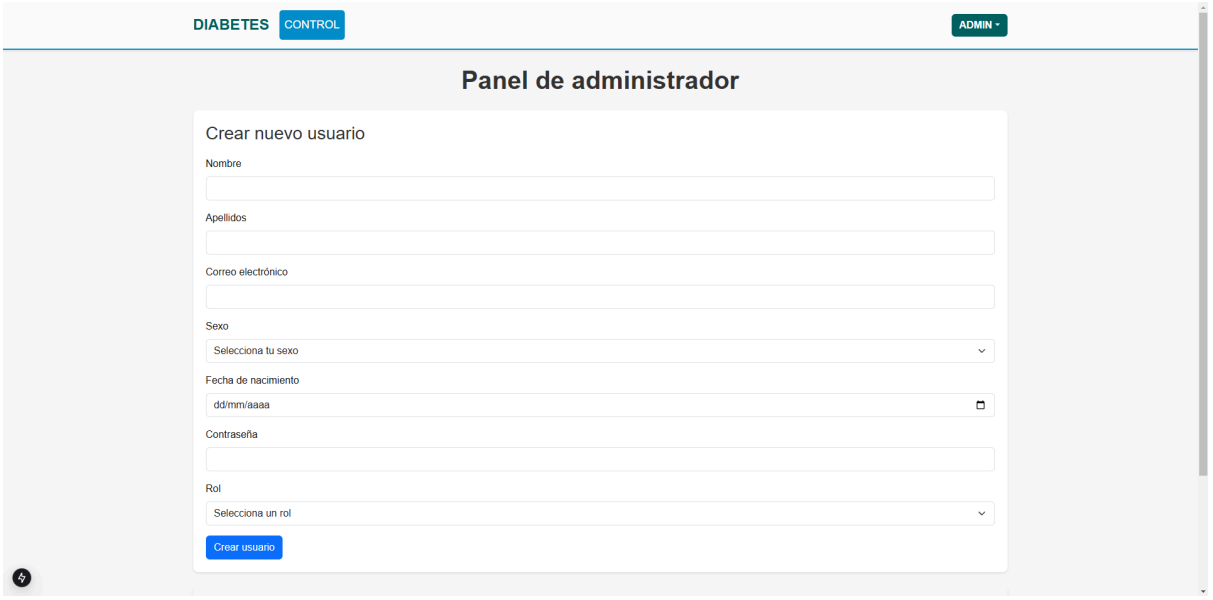


Figura 108 Vista control

- Rol paciente
    - Agregar nota

En las siguiente imágenes vamos a ver una secuencia de agregar una nota.

Para empezar en la vista inicial (Figura 109), se pulsa el botón “Editar” para desplegar un formulario que permite añadir notas con un “tipo” y un “valor”.

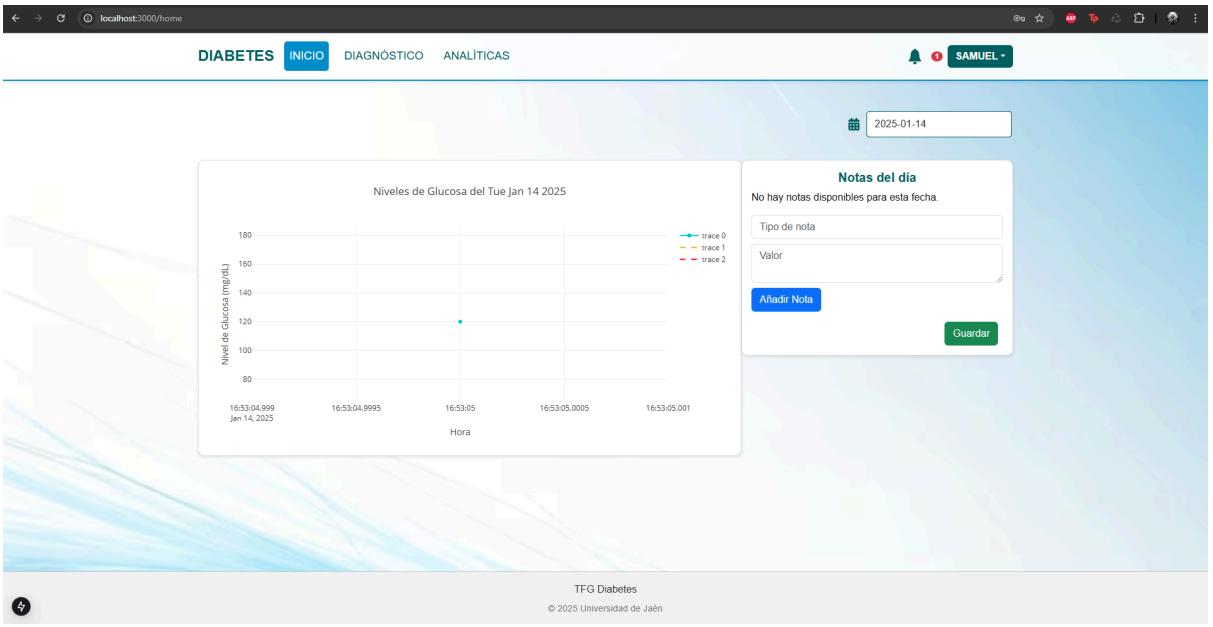


Figura 109: Vista inicio notas

Tras rellenar los campos, se presiona “Añadir Nota” y se visualizan las notas creadas, con la opción de eliminarlas (Figura 110). Puede añadirse cualquier número de notas.

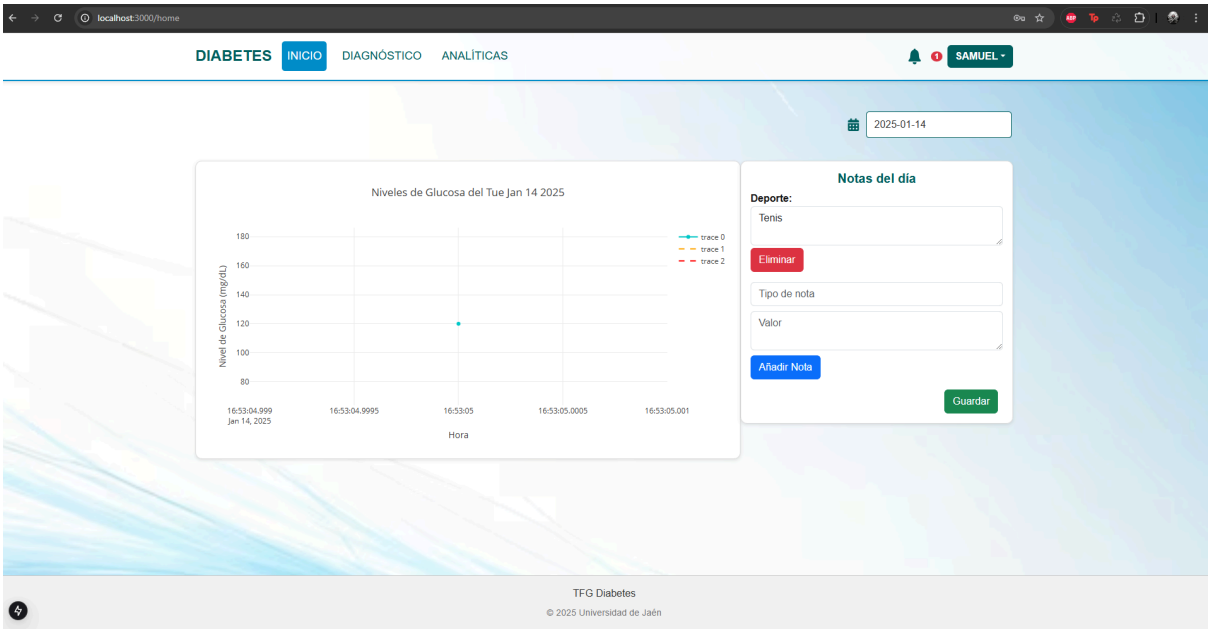


Figura 110: Vista inicio crear notas

Al pulsar “Guardar”, la aplicación muestra una alerta indicando “Notas guardadas exitosamente” (Figura 111).

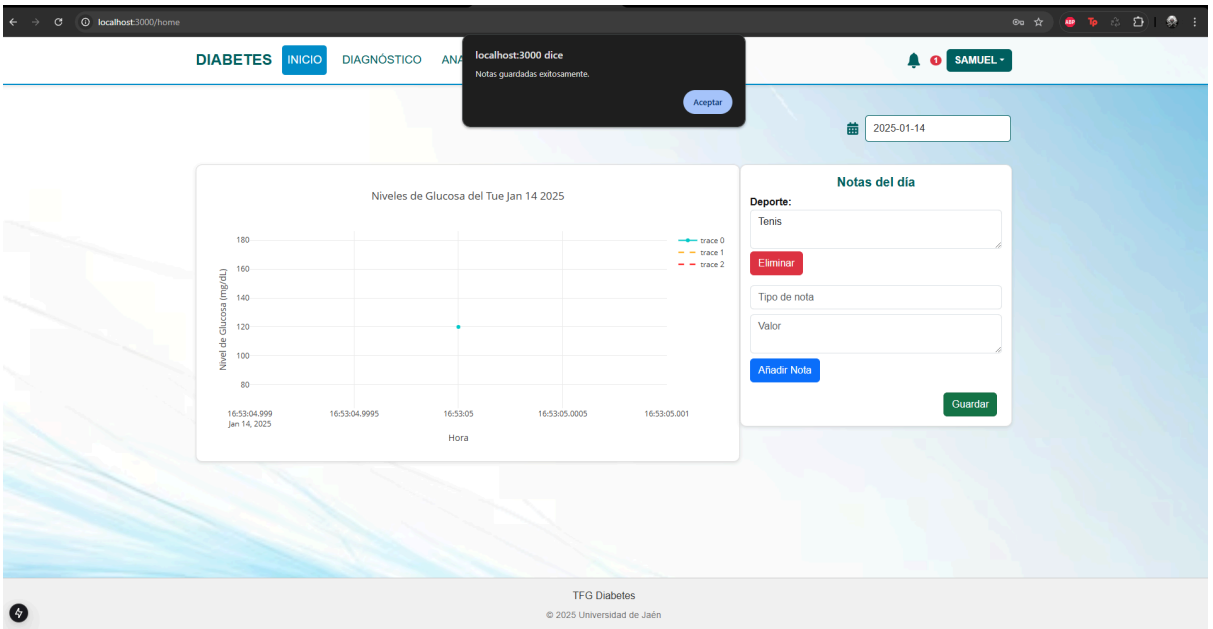


Figura 111: Vista inicio guardar notas

En la Figura 112 se aprecia el resultado final con las notas añadidas reflejadas.

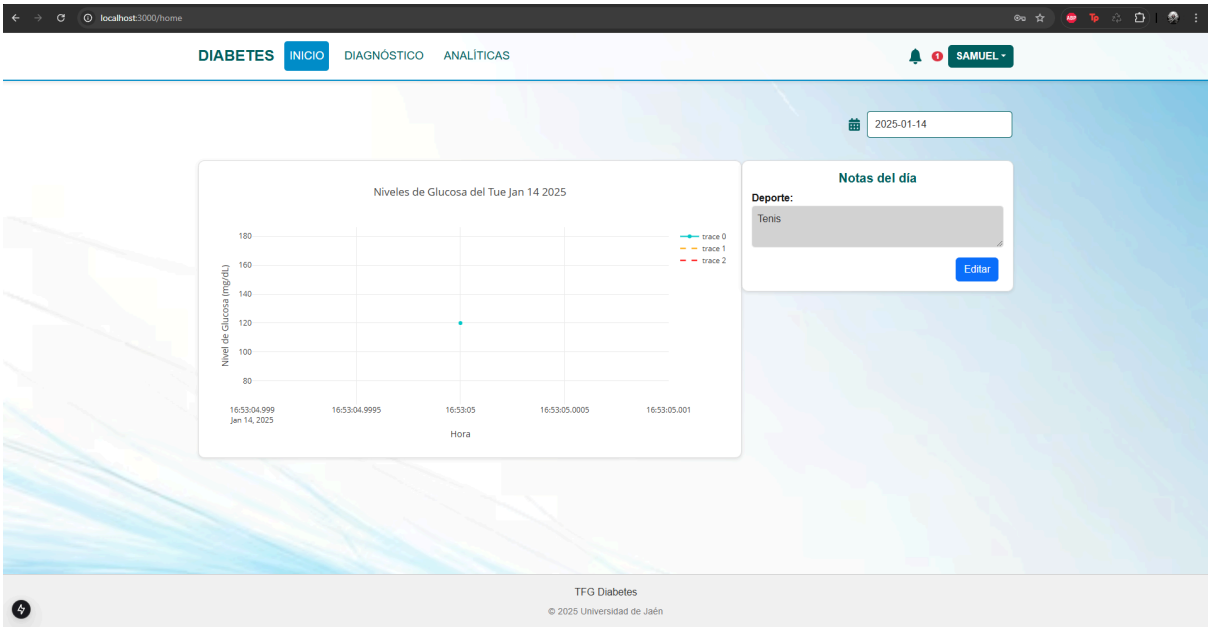


Figura 112: Vista inicio nota guardada

- Página diagnósticos

En la Figura 113 se observa la sección de diagnósticos del paciente, con un acordeón que muestra cada registro: la fecha de emisión, el nombre del médico que lo redactó y la descripción asociada.

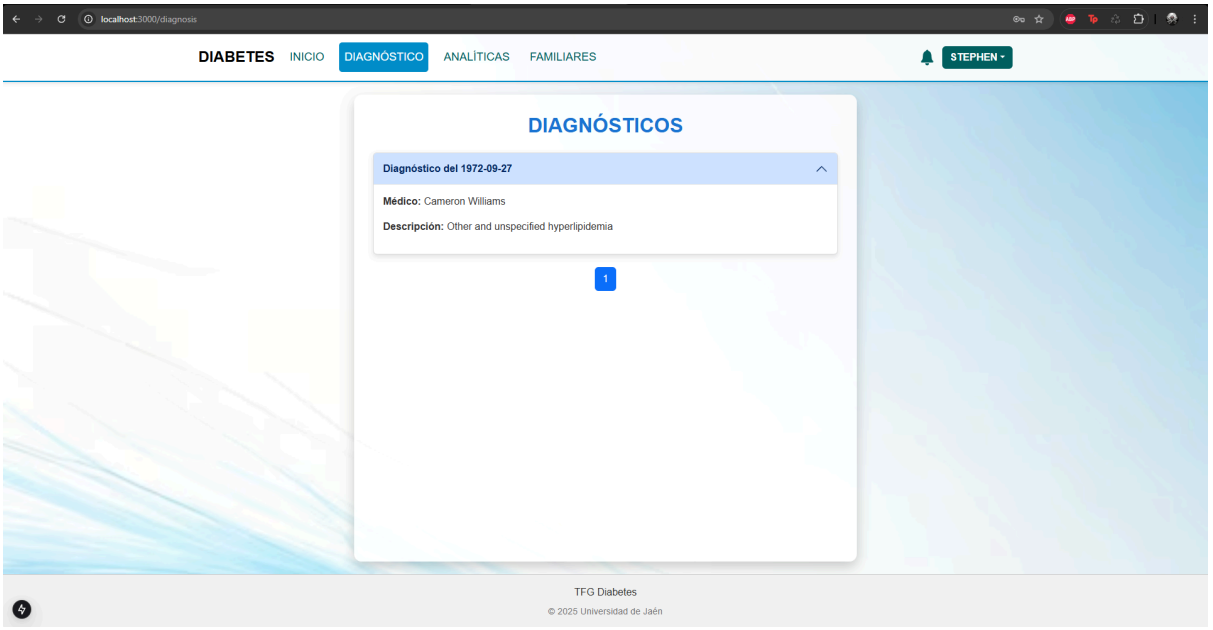


Figura 113: Vista diagnósticos

- Página analíticas

La vista de analíticas (Figura 114) consiste en un carrusel que recorre los distintos registros según su fecha. Cada pantalla del carrusel contiene una tabla con el nombre y valor de cada parámetro bioquímico.

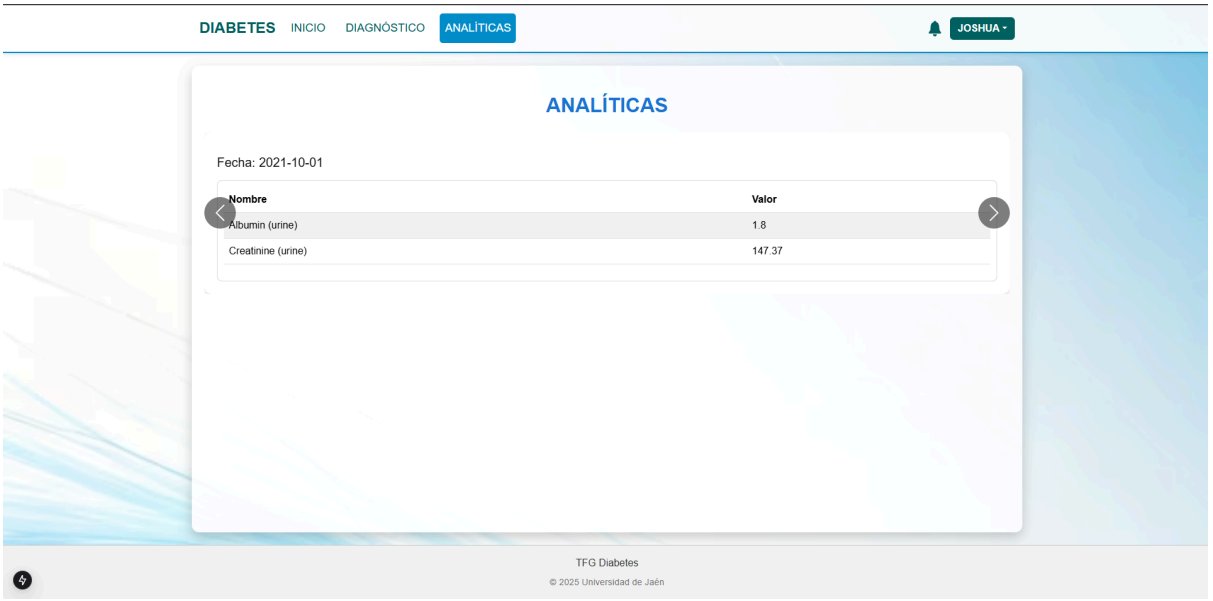


Figura 114: Vista analíticas

- Rol familiar
    - Buscar familiar

En las siguientes imágenes vamos a ver una secuencia de un usuario con rol de familiar busca un paciente por el correo electrónico y al paciente le llega un mensaje para que acepte la solicitud del familiar.

El familiar introduce el correo del paciente que desea vincular (Figura 115) y envía la solicitud, recibiendo un mensaje en verde de confirmación (Figura 116).

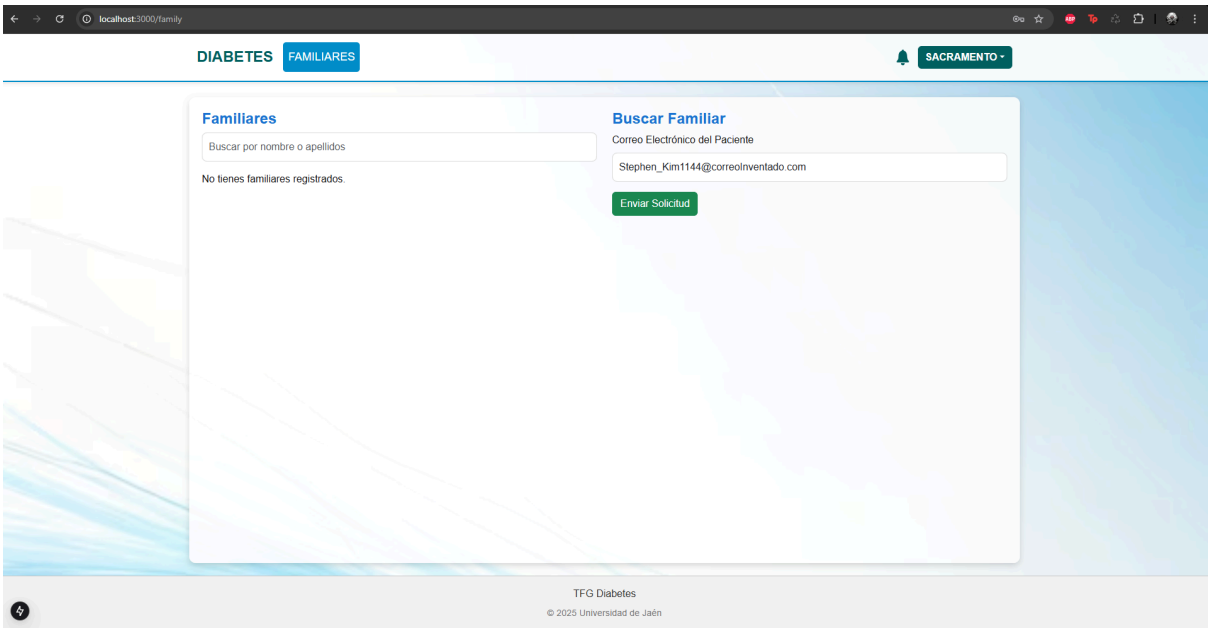


Figura 115: Vista familiar

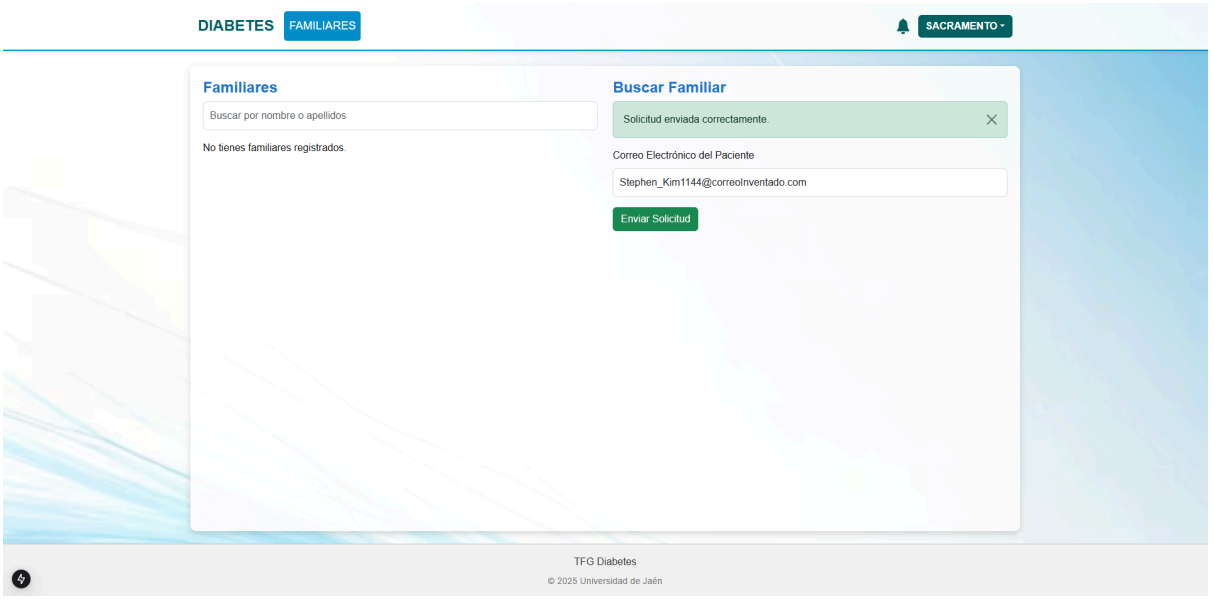


Figura 116: Vista familiar solicitud

Si se ingresa un correo inexistente (Figura 117), la aplicación notifica el error correspondiente.

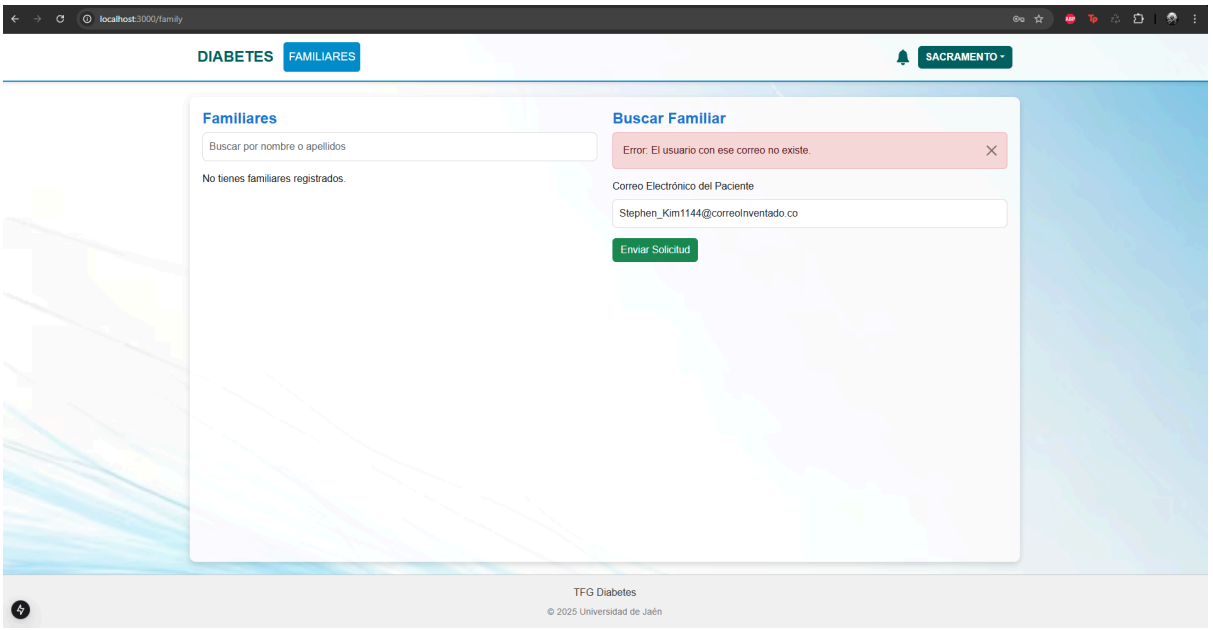


Figura 117: Vista familiar solicitud errónea

- Mensaje a paciente

El paciente, en su barra de notificaciones (campana), ve un indicador numérico (Figura 118). Al pulsar, se abre la lista de mensajes (Figura 119).

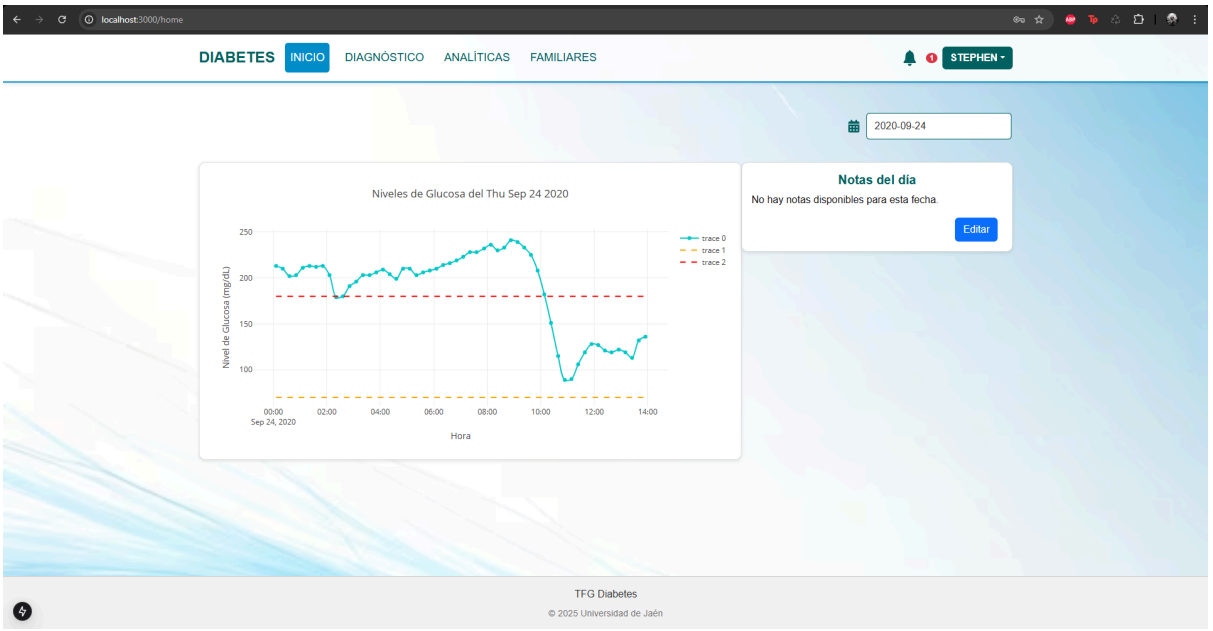


Figura 118: Vista inicio paciente

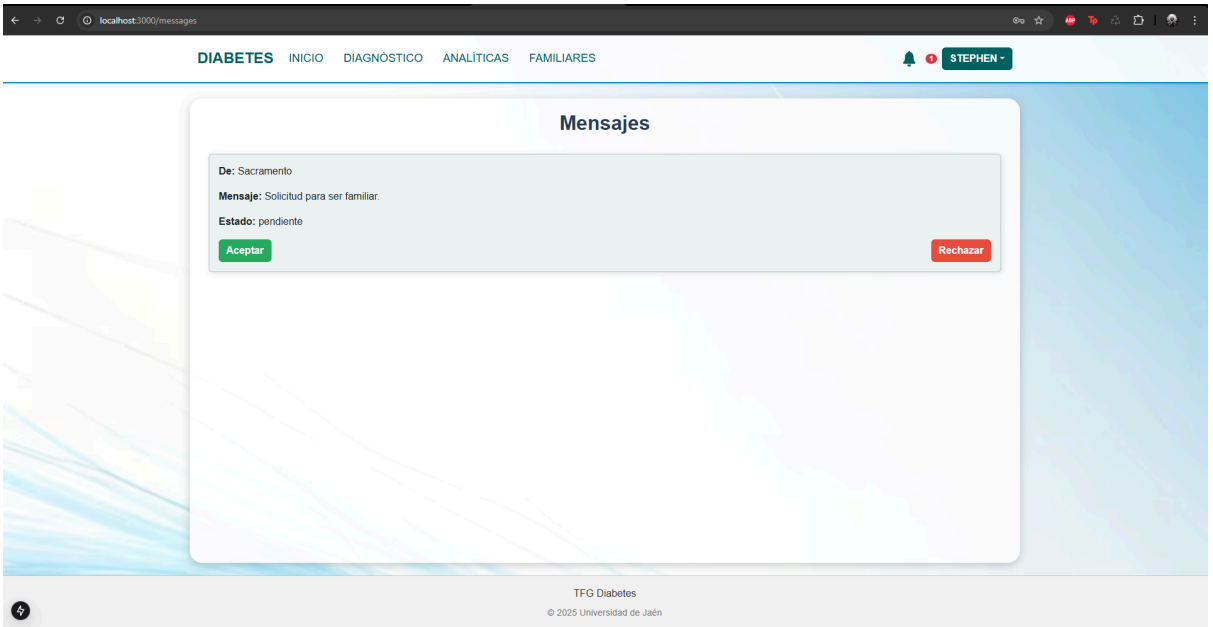


Figura 119: Vista mensajes paciente

Puede aceptar o rechazar la solicitud. Al elegir “Aceptar”, se muestra una alerta “Solicitud aceptada correctamente” (Figura 120).

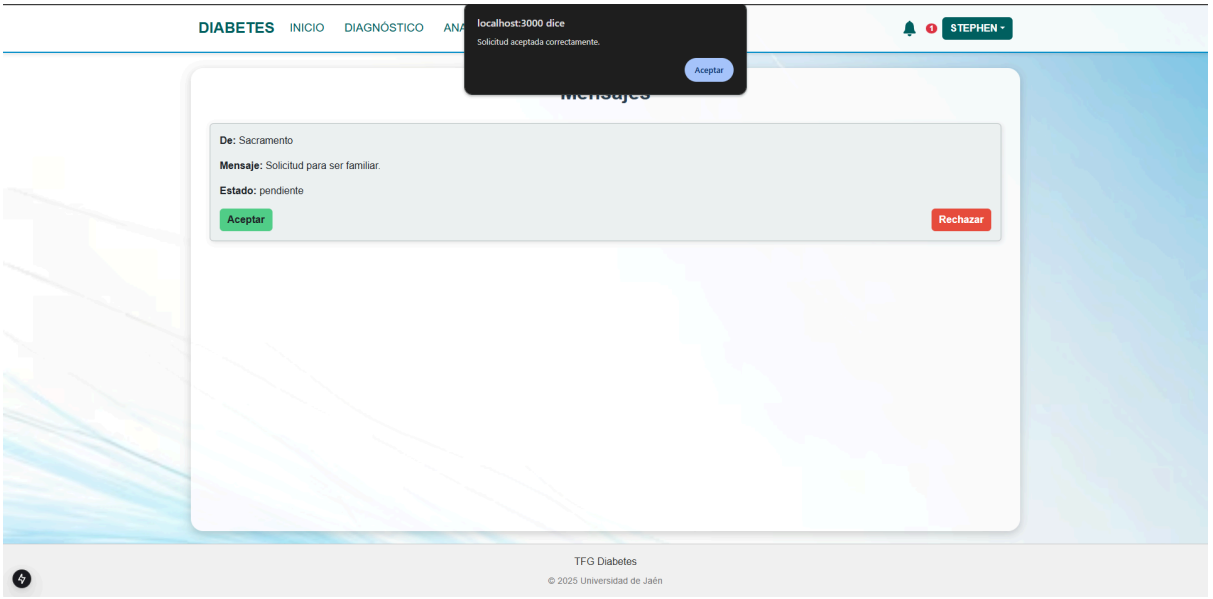


Figura 120: Vista mensaje aceptado paciente

Inmediatamente, el familiar dispone de un enlace para ver el perfil del paciente (Figura 121). Al acceder, se muestra un resumen de los datos del paciente (Figura 122).

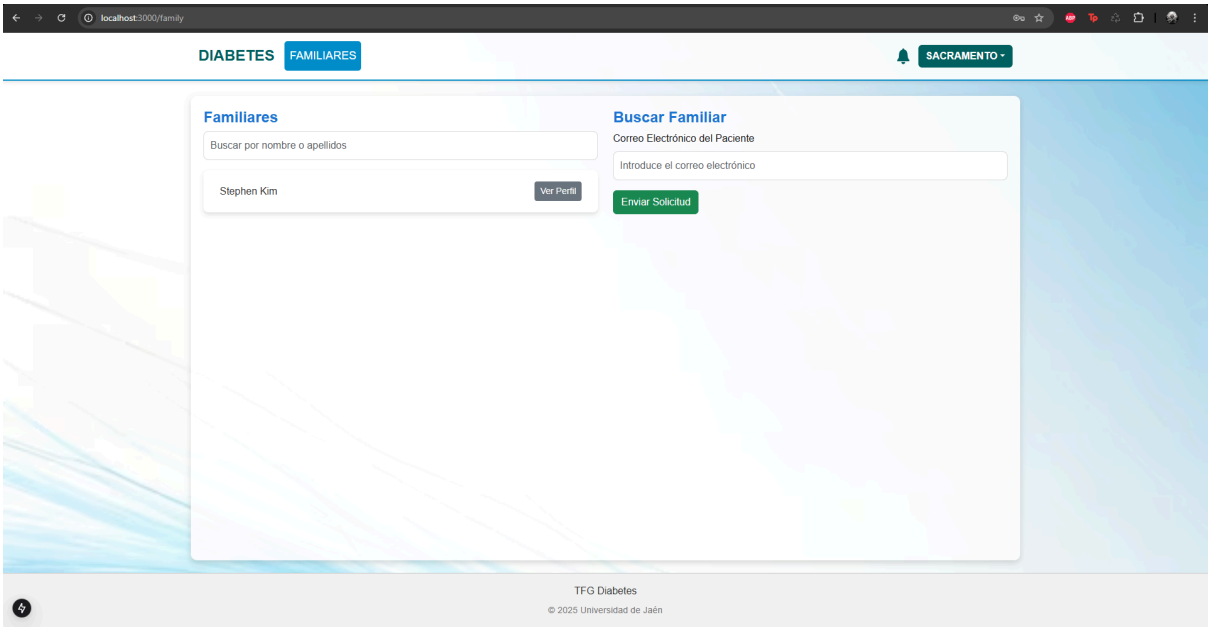


Figura 121: Vista familiar con familiar

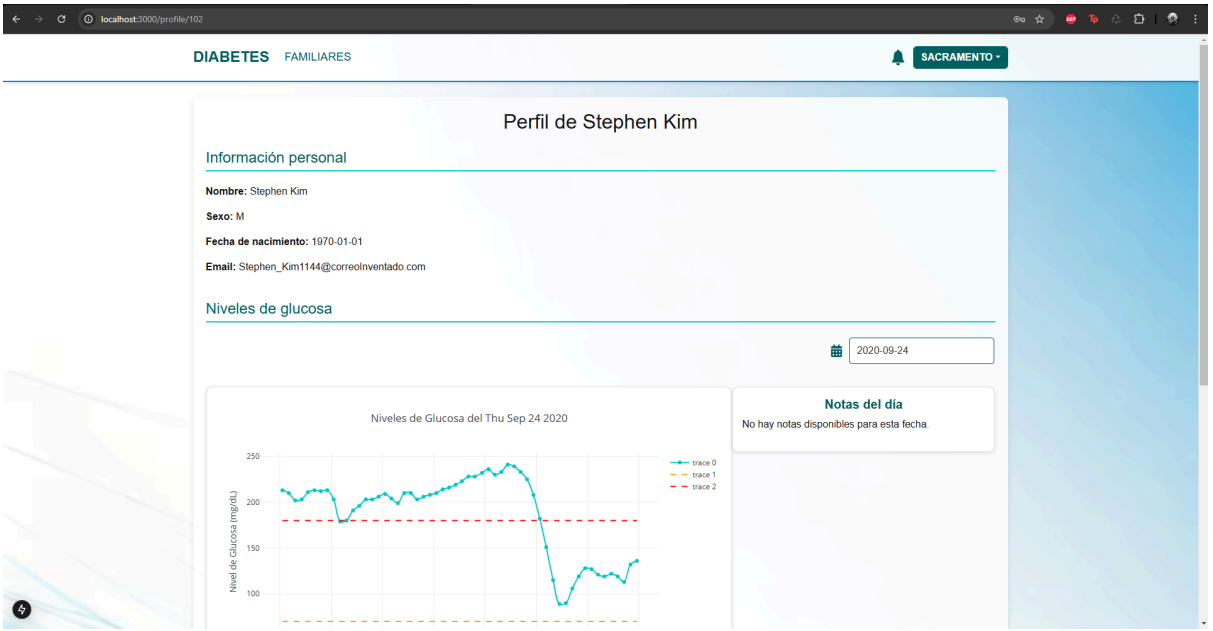


Figura 122: Vista perfil paciente desde familiar

- Rol médico

En la siguiente secuencia, el médico añade un diagnóstico a un paciente. Se ilustra cómo primero se ingresa al perfil del paciente (Figura 123), se desciende hasta la sección de diagnósticos y se pulsa “Añadir Diagnóstico”.

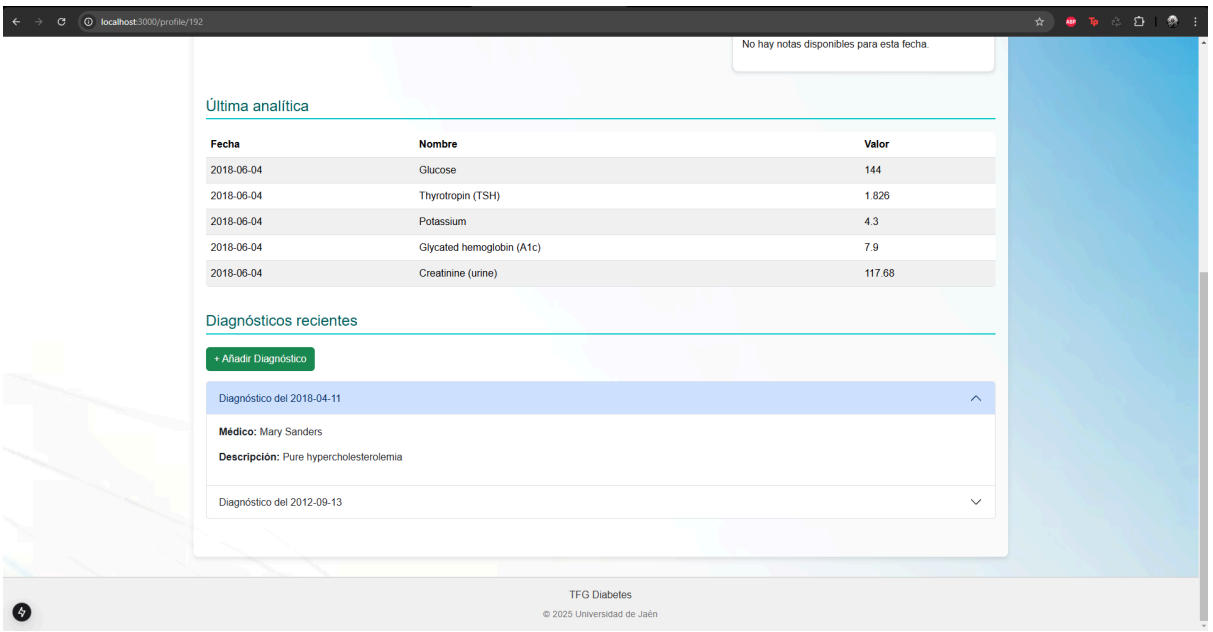


Figura 123: Vista perfil paciente desde médico

Un modal emerge (Figura 124) para completar la descripción y la fecha, guardándolo posteriormente. Dicho diagnóstico aparece inmediatamente en la lista (Figura 125).

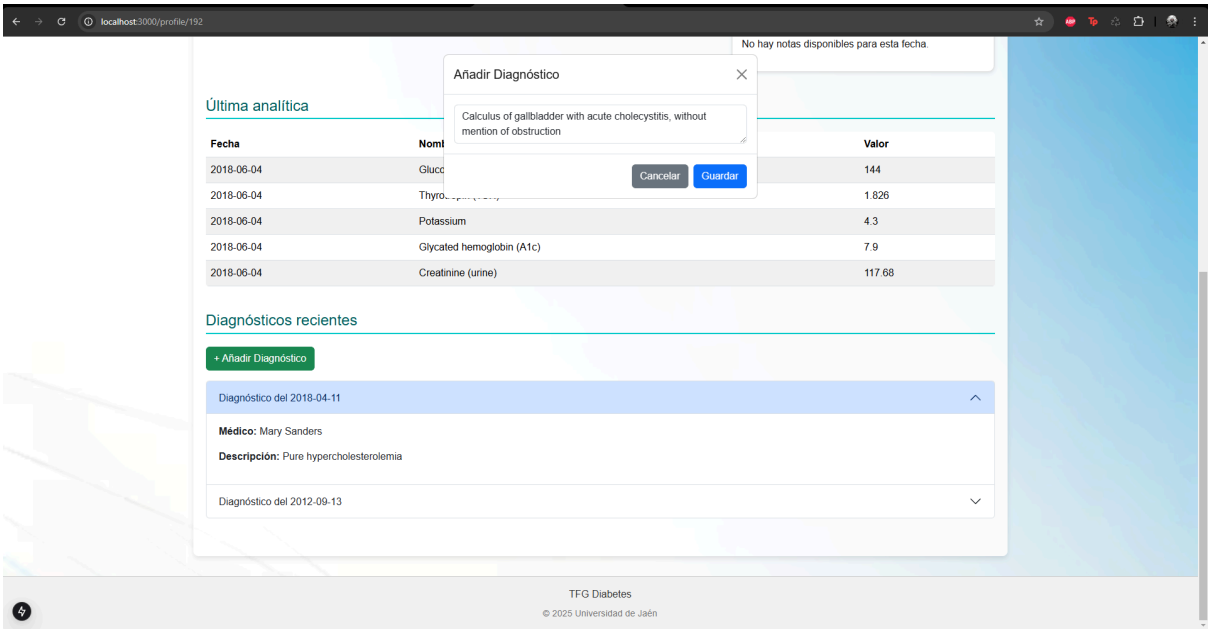


Figura 124: Vista perfil paciente diagnóstico

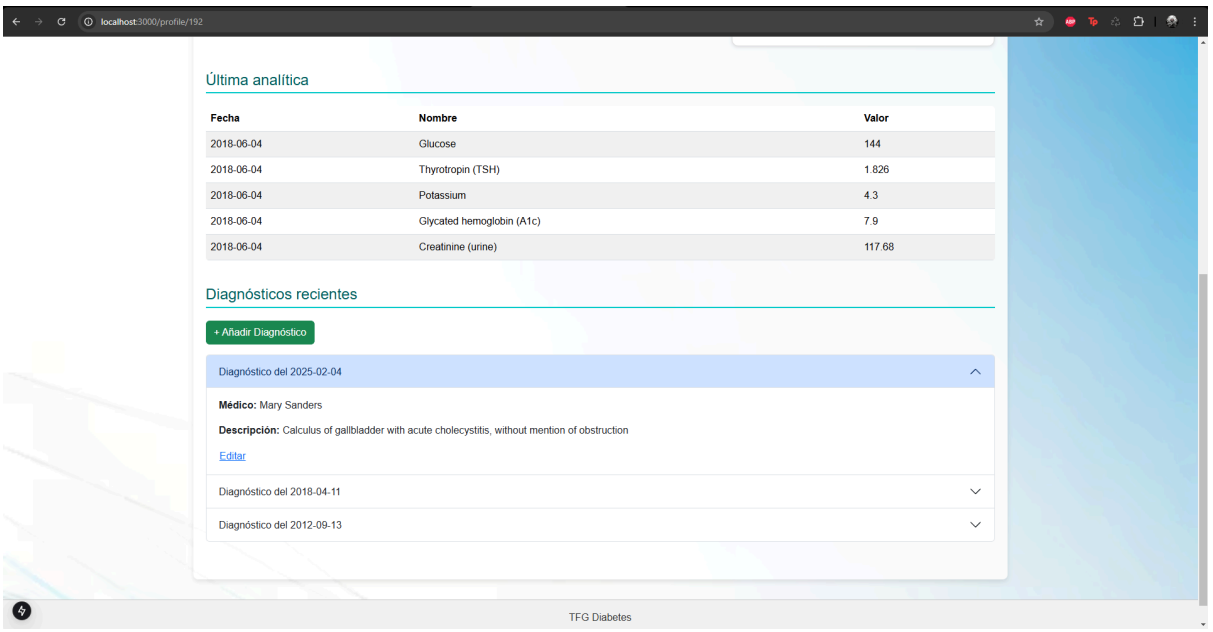


Figura 125: Vista perfil paciente diagnóstico creado

- Rol administrador

Por último, se muestra el flujo en que un paciente solicita el cambio de médico y el administrador lo aprueba.

En la vista de configuración (Figura 126), el paciente pulsa “Cambiar Médico”. Se notifica “Solicitud enviada correctamente” (Figura 127).

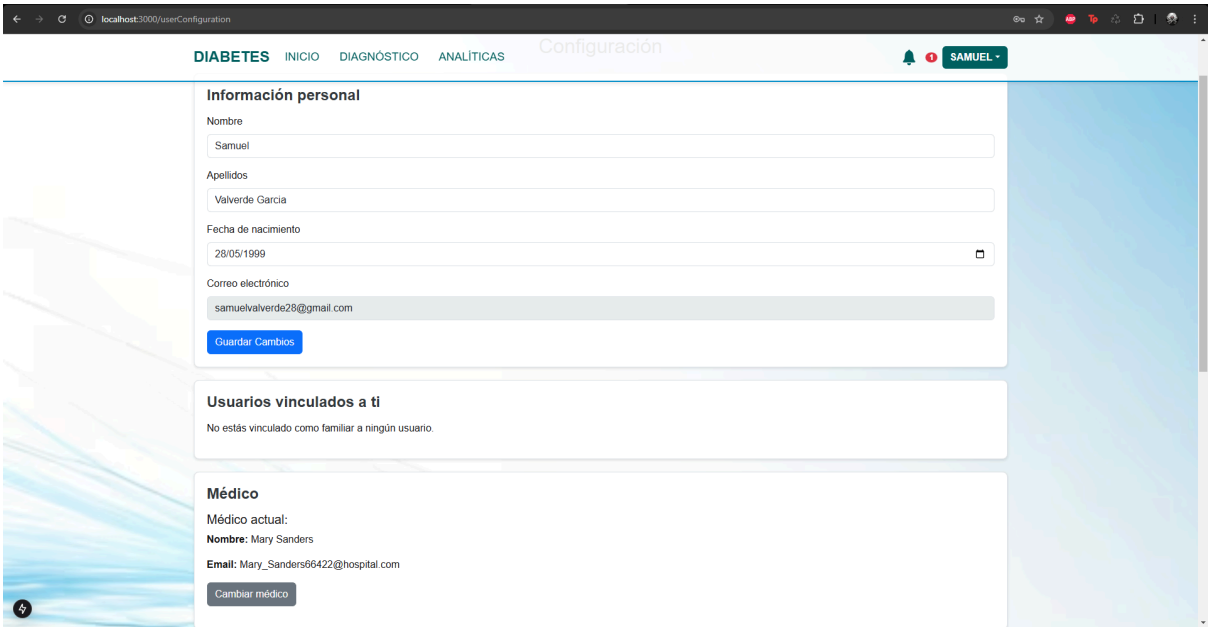


Figura 126: Vista configuración paciente

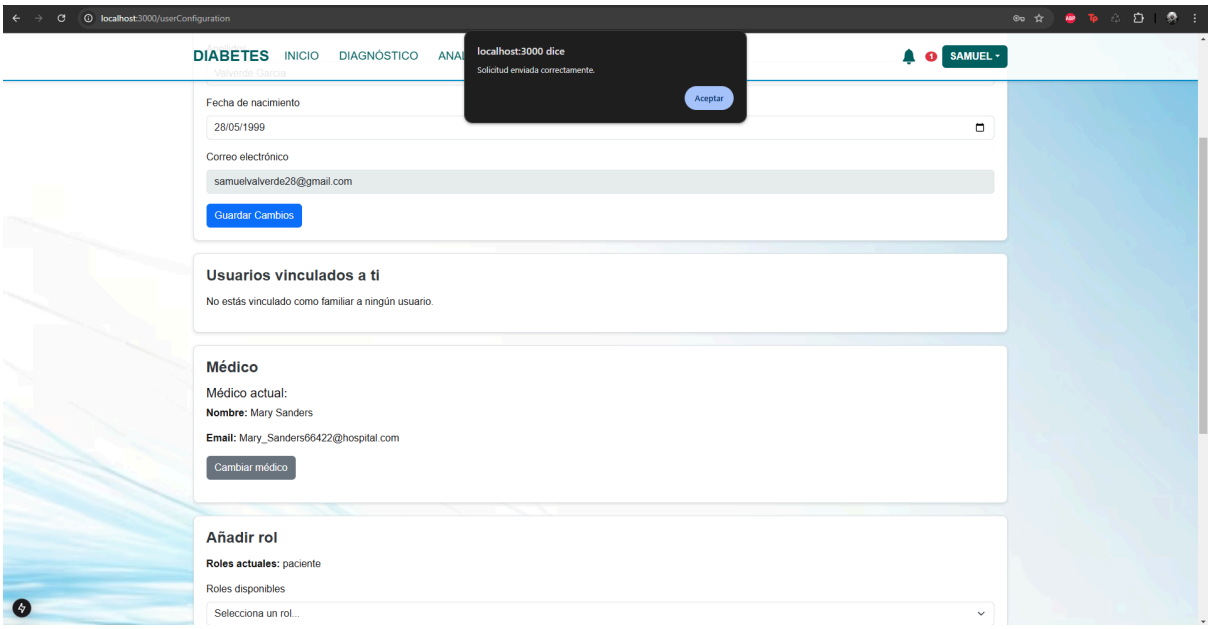


Figura 127: Vista configuración paciente cambio médico

El administrador visualiza la solicitud en su panel (Figura 128). Tiene opción de aceptar o rechazar.

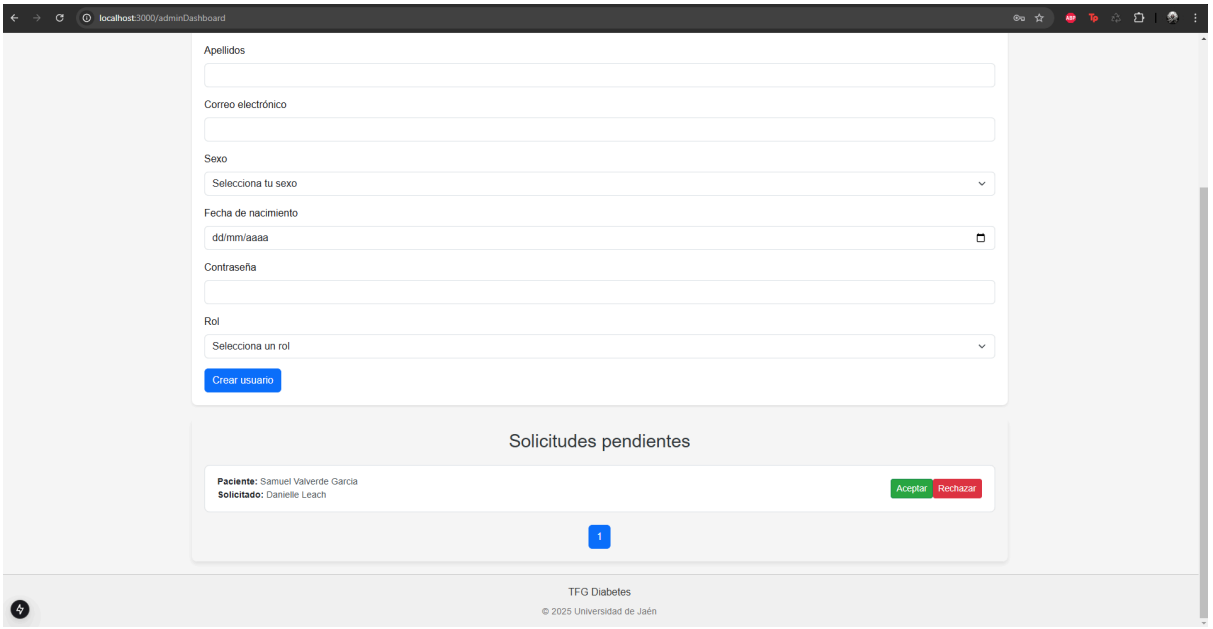


Figura 128: Vista administrador

Al aceptar, se muestra un mensaje de confirmación (Figura 129).

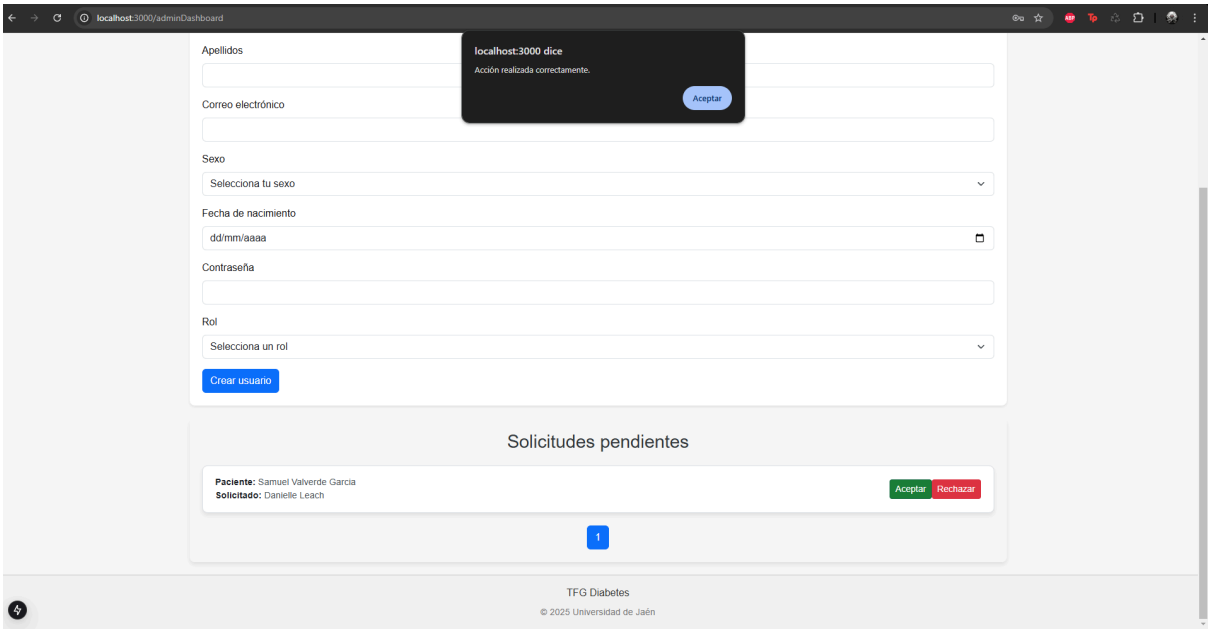


Figura 129: Vista administrador solicitud

Al regresar a la configuración del paciente, se constata que el médico asignado ha cambiado exitosamente (Figura 130).

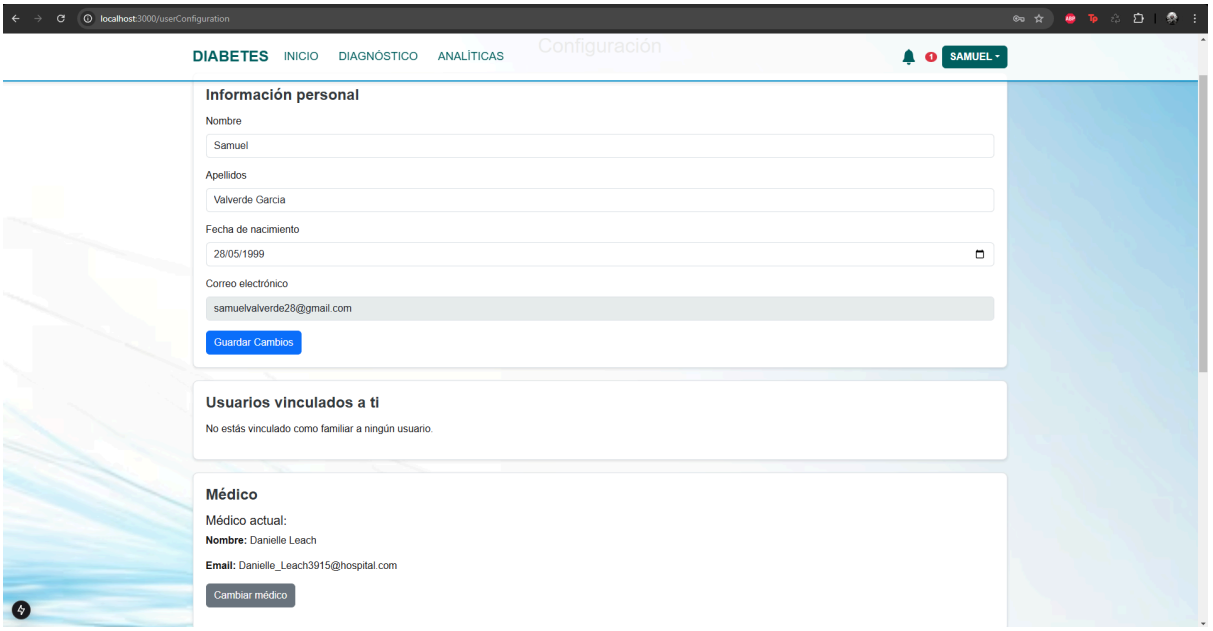
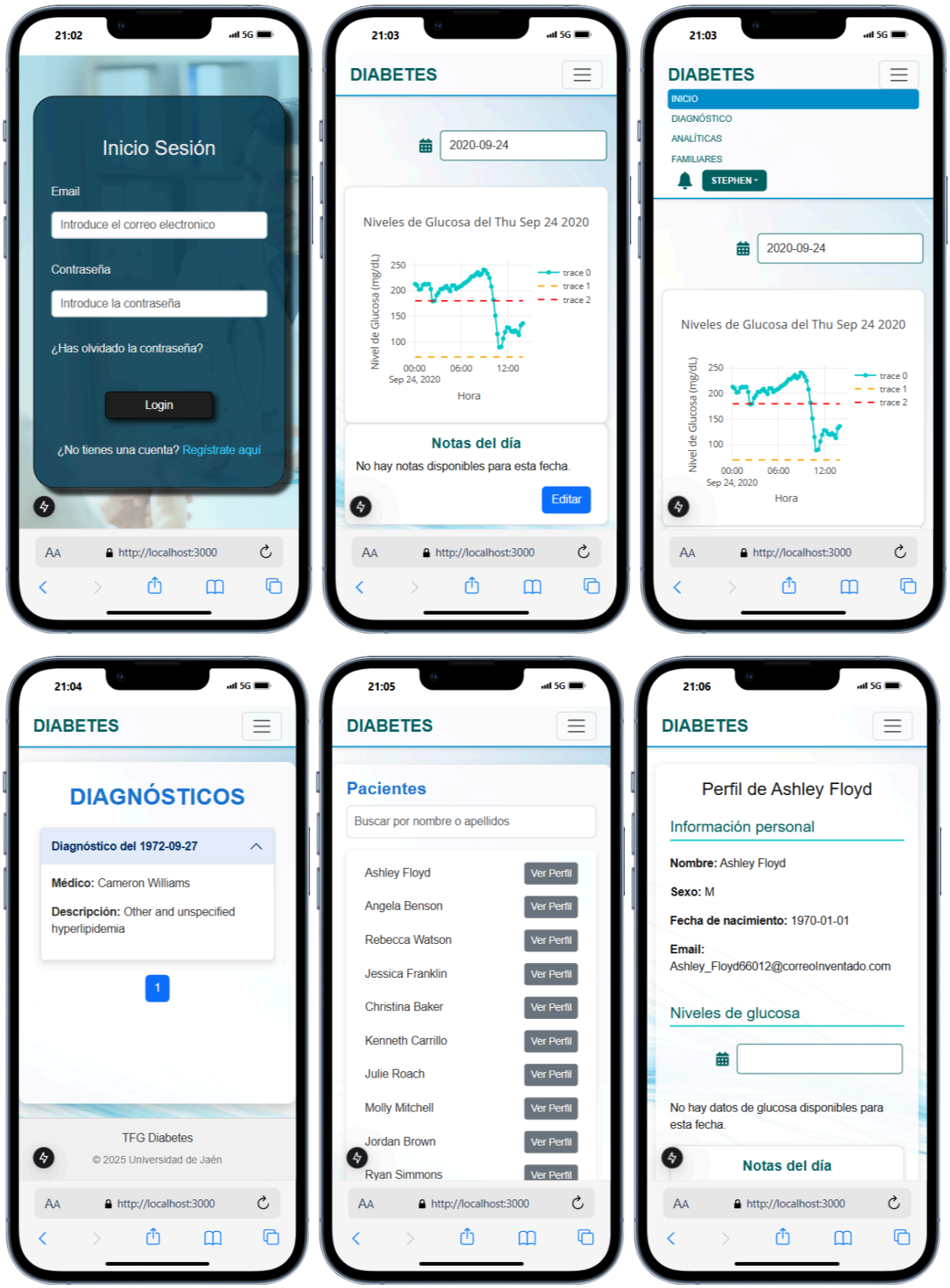
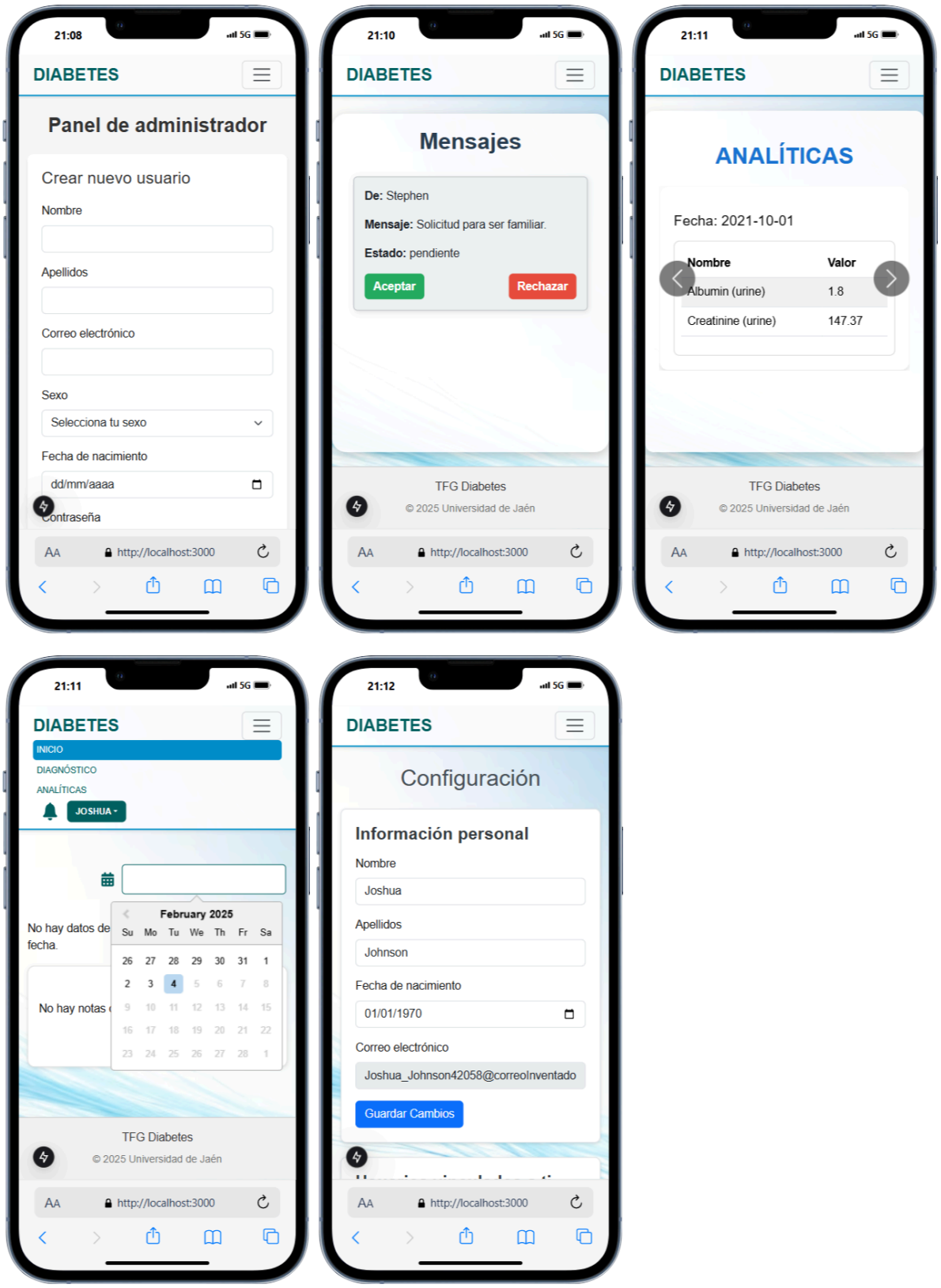


Figura 130: Vista paciente nuevo médico

- Responsividad


21:02
Inicio Sesión
Email
Introduce el correo electronico
Contraseña
Introduce la contraseña
¿Has olvidado la contraseña?
Login
¿No tienes una cuenta? Regístrate aquí
21:03
DIABETES
2020-09-24
Niveles de Glucosa del Thu Sep 24 2020
Nivel de Glucosa (mg/dL)
Hora
Notas del día
No hay notas disponibles para esta fecha.
Editar
INICIO
DIAGNÓSTICO
ANALÍTICAS
FAMILIARES
STEPHEN
21:04
DIAGNÓSTICOS
Diagnóstico del 1972-09-27
Médico: Cameron Williams
Descripción: Other and unspecified hyperlipidemia
TFG Diabetes
© 2025 Universidad de Jaén
21:05
Pacientes
Buscar por nombre o apellidos
Ashley Floyd
Angela Benson
Rebecca Watson
Jessica Franklin
Christina Baker
Kenneth Carrillo
Julie Roach
Molly Mitchell
Jordan Brown
Ryan Simmons
Ver Perfil
21:06
Perfil de Ashley Floyd
Información personal
Nombre: Ashley Floyd
Sexo: M
Fecha de nacimiento: 1970-01-01
Email:
Ashley_Floyd66012@correoInventado.com
Niveles de glucosa
No hay datos de glucosa disponibles para esta fecha.
Notas del día
http://localhost:3000



Figuras 131: Vistas responsives

7. Demostración

Se han añadido cuatros vídeos divididos mediante el rol asociado, en el cual se va a ver unos vídeos cortos de como es la demostración práctica de la aplicación.

7.1. Rol paciente

rol_paciente.mp4

En este vídeo se muestra la interfaz destinada al paciente, desde el momento de inicio de sesión hasta la gestión de las funciones más frecuentes. Se observa cómo el paciente puede registrar sus datos diarios como las notas, consultar gráficas de seguimiento, ver diagnósticos emitidos por el médico y gestionar su configuración personal. El foco principal está en la usabilidad para el autocuidado y la visualización clara de la información clínica.

7.2. Rol familiar

rol_familiar.mp4

Aquí se presenta la vista que se ofrece a un usuario con rol de familiar. El vídeo ilustra cómo el familiar puede buscar y vincularse con un paciente, revisar la información médica compartida y recibir notificaciones de solicitud. El objetivo es subrayar la colaboración y el control de acceso a la información del paciente.

7.3. Rol médico

rol_medico.mp4

Este apartado muestra la perspectiva del profesional médico, desde la consulta de la lista de pacientes hasta el registro de nuevos diagnósticos o la revisión de la información de cada caso. El vídeo refleja las herramientas que facilitan la toma de decisiones clínicas (gráficas de glucosa, notas del paciente, diagnósticos emitidos anteriormente y analíticas), la emisión de nuevas prescripciones y la comunicación con el paciente. Enfatiza el ahorro de tiempo al disponer de un panel integral con datos actualizados.

7.4. Rol administrador

rol_administrador.mp4

Finalmente, el vídeo presenta las funciones dirigidas al administrador, responsable de crear nuevos usuarios o asignar roles (paciente, médico, familiar), aceptar o rechazar solicitudes de cambio de médico. Se aprecia la flexibilidad en la gestión de cuentas y la seguridad al mantener un control global de las solicitudes y permisos en la plataforma.

8. Conclusiones

La realización de este proyecto ha permitido construir una plataforma web centrada en la gestión de pacientes con diabetes de tipo I y en la colaboración entre diferentes roles. A lo largo del desarrollo, se ha podido comprobar la efectividad de tecnologías como FastAPI en el back-end y Next.js en el front-end, combinadas con SQLAlchemy para la persistencia de datos y un enfoque responsivo y modular en el diseño de la interfaz.

- Cumplimiento de objetivos

Se ha implementado una estructura de entidades clara, y un sistema de autenticación y control de roles, mostrando a cada rol únicamente las vistas que necesita, dotándolo con una usabilidad que abarcan las funciones esenciales..

Se ha garantizado la comunicación entre front-end y back-end con peticiones HTTP y tokens JWT, protegiendo la información sensible.

- Experiencia de desarrollo

El uso de Kanban y la modularización del proyecto, han contribuido a un flujo de trabajo ordenado y flexible.

Además la adopción de tecnologías modernas ha facilitado la escalabilidad de la interfaz y la integración con la API.

Además una parte más relevante fue el aprendizaje de los entornos ya que son nuevos en comparación con el conocimiento que ya poseía de otros frameworks.

Al crear los componentes desde cero, sin depender de muchas librerías externas, lo que me ha ayudado a comprender los fundamentos del estilo y la funcionalidad de la interfaz.

- Verificación del sistema

Mediante las pruebas manuales con Swagger en el back-end y recorridos funcionales en el front-end, se han identificado y corregido inconvenientes de usabilidad y validación de datos.

Las pruebas han mostrado que los flujos principales se realizan sin conflictos y ofrecen al usuario retroalimentación adecuada con los mensajes de error y éxito.

En conjunto, la implementación resulta satisfactoria, cumpliendo con los requisitos establecidos y sentando las bases para su potencial despliegue en un entorno real. Este trabajo no solo ilustra la capacidad de unir componentes front-end, back-end, base de datos con tecnologías actuales, sino que demuestra cómo estos servicios pueden interactuar para proporcionar una herramienta de valor en la gestión de datos clínicos y la coordinación entre distintos roles.

8.1. Posibles desarrollos futuros

Ya que esté proyecto es una base para una aplicación mayor en el cual se podrían añadir los siguientes factores para su mejora:

- Refinamiento de la experiencia del usuario

Se podría hacer una personalización de la pantalla principal en que cada rol puede personalizar qué módulos se muestran en su vista inicial y en qué orden, facilitando una interacción más eficiente según las necesidades de cada usuario.

También relacionado con la interfaz, se puede implementar un modo oscuro para usuarios que prefieren menos brillo, garantizando altos contrastes.

Además de ofrecer traducciones a varios idiomas, pudiendo detectar la preferencia según el navegador y la posibilidad de modificación en la configuración.

Ya que no todos los usuarios tienen una agilidad con entornos web, se puede incorporar un tutorial inicial la primera vez que un usuario accede al sistema, indicando los pasos básicos y marcando visualmente las zonas clave de la interfaz, además de añadir ventanas emergentes en botones con conejos y descripciones de la funcionalidad.

Para el paciente y roles relacionados se puede añadir un sistema de notificaciones en el cual aparezca un aviso en tiempo real para aspectos urgentes como un cambio repentino de glucosa.

- Automatización de pruebas

Añadir pruebas automatizadas que ejecuten pruebas unitarias y de integración tanto en el back-end como en el front-end tras cada commit, asegurando la calidad continua.

- Funcionalidades colaborativas

Añadir un foro donde se crea un espacio colectivo donde pacientes con el mismo tipo de diabetes compartan experiencias y consejos, moderado por personal médico para que no haya inconvenientes.

O la implementación de un sistema de gestión de citas para facilitar la reserva o actualización de consultas, innegable con agendas del médico.

- Ampliación de roles y permisos

Añadir roles para enfermería que pueda supervisar pacientes de forma intermedia entre el paciente y el médico, con ciertas funcionalidades pero sin privilegios totales.

O la implementación de un rol investigador donde tenga acceso a los datos pero anónimamente, a los datos de los usuarios que den su consentimiento. Para fomentar estudios y estadísticas con fines de investigación.

- Funciones de IA avanzadas

Usar algoritmos de machine learning para analizar la evolución de la glucosa y predecir potenciales hipoglucemia o hiperglucemias, enviando alertas preventivas.

- Despliegue escalable

Pasar a un despliegue en contenedores que permita escalar el back-end y front-end de forma automática, manejando picos de usuarios o cargas intensivas.

9. Bibliografía

References

[1]. (2024, November 14). Diabetes. Retrieved February 5, 2025, from https://www.who.int/es/news-room/fact-sheets/detail/diabetes

[2]. (n.d.). ¿Qué es un ORM? Desarrolladores Páginas Web. Retrieved February 5, 2025, from https://www.dreams.es/transformacion-digital/desarrolladores-paginas-web/que-es-un-orm

[3]. (n.d.). App mySugr. Retrieved February 5, 2025, from https://www.accu-chek.es/productos/mysugr

[4]. (n.d.). Glooko: Connected Care for Diabetes and Related Chronic Conditions. Retrieved February 5, 2025, from https://glooko.com/es_EU/

[5]. (2024, August 30). Retrieved February 5, 2025, from https://canaldiabetes.com/la-app-mysugr-podria-estar-proporcionando-recomendaciones-incorrectas-de-dosis-de-insulina/

[6]. (n.d.). LibreView. Retrieved February 5, 2025, from https://www.libreview.com/

[7]. (n.d.). HL7 Wikipedia. Retrieved February 5, 2025, from https://es.wikipedia.org/wiki/HL7

[8]. Krug, S. (2006). *Don't Make Me Think! A Common Sense Approach to Web Usability*. New Riders.

[9]. (n.d.). React. Retrieved February 5, 2025, from https://es.react.dev/learn

[10]. (n.d.). Angular • What is Angular? Retrieved February 5, 2025, from https://angular.dev/overview

[11]. (n.d.). Nuxt: The Intuitive Vue Framework · Nuxt. Retrieved February 5, 2025, from https://nuxt.com/

[12]. (2021, May 10). ¿Qué son la SSR y la SSG? Retrieved February 5, 2025, from https://www.azion.com/es/blog/ssr-o-ssg-para-la-renderizacion-de-paginas/

[13]. (n.d.). Virtual DOM. Retrieved February 5, 2025, from https://es.legacy.reactjs.org/docs/faq-internals.html

[14]. (n.d.). Retrieved February 5, 2025, from https://rafarjonilla.com/que-es/backend/

[15]. (n.d.). Next.js. Retrieved February 5, 2025, from https://nextjs.org/docs

[16]. (n.d.). FastAPI. Retrieved February 5, 2025, from https://fastapi.tiangolo.com/es/

[17]. (n.d.). SQLAlchemy - The Database Toolkit for Python. Retrieved February 5, 2025, from https://www.sqlalchemy.org/

[18]. (n.d.). Swagger REST API Documentation Tool. Retrieved February 5, 2025, from https://swagger.io/tools/swagger-ui/

[19]. (n.d.). Visual Studio Code - Code Editing. Redefined. Retrieved February 5, 2025, from *https://code.visualstudio.com/*

[20]. (n.d.). Qué es la metodología waterfall y cuándo utilizarla [2024] • Asana. Retrieved February 5, 2025, from https://asana.com/es/resources/waterfall-project-management-methodology

[21]. Subra, J.-P., & Vannieuwenhuyse, A. (2018). *Scrum: un método ágil para sus proyectos*. Ediciones Eni.

[22]. Edge, J. (2018). *Kanban: La guía definitiva de la metodología Kanban para el desarrollo de software ágil*. Amazon Digital Services LLC - Kdp.

[23] Beck, K., & Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Pearson Education.

[24]. (2005). In *Ingeniería del software*. Pearson Educación.

[25]. Kendall, K. E., & Kendall, J. E. (2005). *Análisis y diseño de sistemas* (A. Núñez Ramos, Trans.). Pearson Educación.

[26]. (n.d.). Lucidchart - Qué es un wireframe para un sitio web. Retrieved February 5, 2025, from https://www.lucidchart.com/pages/es/que-es-un-wireframe-para-un-sitio-web

[27]. (n.d.). IONOS. Retrieved February 5, 2025, from https://www.ionos.es/digitalguide/servidores/know-how/modelo-cliente-servidor/

[28]. Schulz, R. G. (2009). *Diseño web con CSS*. Marcombo.

[29]. (n.d.). Bootstrap · The most popular HTML, CSS, and JS library in the world. Retrieved February 5, 2025, from https://getbootstrap.com/

[30]. *Introducción a JSON*. (n.d.). JSON. Retrieved February 5, 2025, from https://www.json.org/json-es.html