UNIVERSIDAD NACIONAL AUTÓNOMA DE NICARAGUA

UNAN-LEÓN

ÁREA DE CONOCIMIENTO CIENCIAS Y TECNOLOGÍA

ÁREA ESPECÍFICA INGENIERÍA EN SISTEMA DE INFORMACIÓN

INGENIERÍA EN SISTEMA DE INFORMACIÓN

# Desarrollo de una Aplicación Multiplataforma para la Gestión Eficiente de Citas Médicas.

Tesis para optar al título de Ingeniero en Sistemas de Información


AUTORES:

Br. KATHERINNE JULISSA CASTRO GONZALEZ.

Br. FELIX SAMIR MEJIA MENDOZA.

Br. MARIO RAMON SANCHEZ QUIROZ.

TUTOR:

Licda. DAVINIA ALEJANDRA QUIROZ ROQUE.


León, Nicaragua 13 de junio del 2025

UNIVERSIDAD NACIONAL AUTÓNOMA DE NICARAGUA

UNAN-LEÓN

Área de Conocimiento Ciencias y Tecnología

Área Específica Ingeniería En Sistemas de Información

INGENIERÍA EN SISTEMA DE INFORMACIÓN

# Desarrollo de una Aplicación Multiplataforma para la Gestión Eficiente de Citas Médicas.

Tesis para optar al título de Ingeniero en Sistema de Información


AUTORES:

BR: KATHERINNE JULISSA CASTRO GONZALEZ _____

BR: FELIX SAMIR MEJIA MENDOZA _____

BR: MARIO RAMON SANCHEZ QUIROZ _____

TUTOR(A):

Licda. DAVINIA ALEJANDRA QUIROZ ROQUE _____


León, Nicaragua 13 de junio del 2025

## Resumen

El presente proyecto desarrolla un sistema integral para la gestión de citas médicas, combinando una aplicación web construida con **ASP.NET Core MVC** y una aplicación móvil desarrollada en **Flutter**. Ambas plataformas trabajan de forma conjunta bajo una arquitectura **cliente-servidor**, permitiendo a pacientes, doctores y administradores interactuar con el sistema de manera eficiente y segura.

La aplicación web, diseñada para su uso por personal médico y administrativo, sigue el patrón de diseño **Modelo-Vista-Controlador (MVC)** y utiliza **Entity Framework Core** para la interacción con una base de datos **MySQL**. Incluye módulos para la gestión de doctores, especialidades, citas médicas y envío de confirmaciones por correo electrónico mediante **SMTP**.

Por otro lado, la aplicación móvil está dirigida principalmente a los pacientes y fue desarrollada en **Flutter**, usando **Dart** como lenguaje de programación. Implementa servicios de **Firebase** como backend, incluyendo autenticación, base de datos en la nube (**Cloud Firestore**), almacenamiento local con **sqflite**, y notificaciones con **Firebase Cloud Messaging**. El patrón **MVVM** y la separación de lógica permiten una arquitectura clara y mantenible.

El sistema permite a los usuarios programar y gestionar citas médicas, visualizar expedientes, comunicarse con médicos y recibir notificaciones. Los doctores pueden gestionar sus agendas y editar expedientes, mientras que el director tiene control administrativo total. La integración entre ambas plataformas se realiza mediante **API REST**, garantizando sincronización y escalabilidad.

Este proyecto demuestra la viabilidad técnica de un sistema de salud digital accesible desde múltiples plataformas, promoviendo la eficiencia en la atención médica.

## Dedicatoria

Primeramente, agradezco a Dios, fuente de sabiduría, fortaleza y esperanza, por haberme guiado en cada etapa de este camino. Sin Su presencia constante, nada de esto habría sido posible.

A mi madre Johanna González, mi pilar incondicional, gracias por tu amor, tus sacrificios silenciosos y tu fe en mí incluso cuando yo dudaba. Tu apoyo constante, tus palabras de aliento y tu ejemplo de perseverancia han sido la base sobre la cual se construye este logro. Esta meta es tan tuya como mía.

A mi familia y amigos, gracias por estar cerca, por su comprensión en los momentos difíciles y por compartir conmigo alegrías y desafíos.

A todos quienes, de una forma u otra, contribuyeron con su tiempo, su conocimiento o su cariño, les estoy profundamente agradecido.

Katherinne González

## Dedicatoria

Con todo mi amor y gratitud, dedico este logro a tres mujeres fundamentales en mi vida:

A mi querida mamita Lidia Aguilar, por ser mi refugio, por sus oraciones, su amor incondicional y por enseñarme con su ejemplo el valor de la perseverancia y la humildad. Su fortaleza silenciosa ha sido un motor constante en mi camino.

A mi mamá Lisbett, por cada sacrificio, por cada palabra de aliento y por creer en mí incluso cuando yo misma lo dudaba. Gracias por ser mi apoyo inquebrantable, por impulsarme a seguir adelante y por sostenerme con su amor en los momentos difíciles.

A mi tía Jasary, por estar siempre presente con su cariño, consejos y compañía. Su apoyo ha sido una luz constante a lo largo de esta carrera, y sus palabras han sido guía en los momentos de incertidumbre.

A ustedes, que con su amor y apoyo han sido el pilar de mis logros, les dedico este triunfo con todo mi corazón.

Félix Mejía

## Dedicatoria

Agradezco principalmente a Dios, por haberme dado la vida, la salud y la infinita sabiduría para iluminar cada paso en este camino. Su presencia ha sido mi fortaleza, mi refugio y mi fuente de inspiración para seguir adelante.

A mis amados padres, quienes, con su amor incondicional, su sacrificio constante y su fe inquebrantable en mí, fueron el motor de mis sueños y el pilar fundamental de mi formación. Esta tesis es un reflejo de su esfuerzo y dedicación.

A mis hermanos, por su alegría, su apoyo incondicional y por los momentos compartidos que hicieron más ligera la carga.

A mi familia y amigos, por cada palabra de aliento, por su comprensión y por ser la compañía perfecta en este trayecto. Sin su apoyo, este logro no hubiera sido posible.

Mario Sánchez

# Índice

**Índice de imágenes**

## Introducción

La gestión eficiente de servicios médicos representa un reto constante en los sistemas de salud de países en desarrollo como Nicaragua, donde aún prevalece el uso de métodos tradicionales para la programación de citas. Esta situación genera ineficiencias administrativas, tiempos de espera prolongados y una experiencia deficiente para el paciente.

En respuesta a esta problemática, el presente proyecto de tesis propone el desarrollo de un sistema multiplataforma para la gestión de citas médicas, compuesto por una aplicación web implementada con ASP.NET Core MVC y una aplicación móvil desarrollada en Flutter. Ambas plataformas se comunican mediante servicios REST y emplean Firebase como backend para autenticación, almacenamiento en la nube, notificaciones y sincronización de datos.

El objetivo principal es optimizar el proceso de agendamiento y seguimiento de citas, mejorar la comunicación entre los distintos roles del sistema (paciente, doctor y director) y reducir la carga operativa en los centros de salud. Esta solución tecnológica busca aportar una herramienta escalable, moderna y adaptada al contexto local, contribuyendo a la transformación digital del sector salud en Nicaragua.

## Agradecimiento

Dedicamos este proyecto a **nuestras familias**, quienes han sido nuestro pilar fundamental durante todo este proceso académico. Su apoyo incondicional, comprensión y palabras de aliento han sido claves para alcanzar esta meta.

Agradecemos también a nuestros **docentes y asesores**, por guiarnos con compromiso, paciencia y dedicación, y por contribuir significativamente a nuestro crecimiento profesional y personal.

Asimismo, dedicamos este trabajo a todas las personas que creen en el poder de la tecnología para transformar realidades y mejorar la calidad de vida. Este proyecto es una muestra de nuestro esfuerzo conjunto, con la convicción de aportar soluciones útiles a la sociedad.

## Antecedentes:

En Nicaragua, la transformación digital ha comenzado a impactar diversos sectores, incluida la salud. La necesidad creciente de optimizar el acceso, la eficiencia y la calidad de los servicios médicos ha impulsado el desarrollo de soluciones digitales orientadas a modernizar procesos tradicionales, como la gestión de citas médicas.

En este contexto, instituciones educativas han apostado por la integración de la tecnología en propuestas innovadoras que abordan problemas reales del entorno. Estas iniciativas no solo fortalecen la formación profesional, sino que también contribuyen al desarrollo de herramientas útiles para la sociedad.

Uno de estos proyectos es la aplicación web “Medicall”, desarrollada en Juigalpa utilizando React, Redux y Firebase, bajo la metodología ágil SCRUM. Diseñada para facilitar la gestión de citas médicas y consultas en línea, Medicall busca reducir la automedicación, evitar aglomeraciones y modernizar la atención primaria. Presentado en el documento institucional de la UNAN-Managua (2020) por estudiantes de la Dirección de Docencia de Grado, este proyecto forma parte de una propuesta pedagógica que promueve la formación integral, el pensamiento crítico, la responsabilidad social y el compromiso con el desarrollo humano sostenible.

Paralelamente, en la Universidad Nacional de Ingeniería (UNI-Managua), se desarrolló en 2021 un sistema web para la gestión de citas y expedientes médicos en la red de sucursales de la Clínica San Benito. Implementado con PHP, MySQL, HTML5 y Bootstrap, bajo el modelo de desarrollo en cascada, su principal aporte fue la automatización de procesos que anteriormente se realizaban manualmente, reduciendo tiempos de espera y mejorando la continuidad de la atención médica.

## Planteamiento del Problema

En Nicaragua, una parte significativa de los centros de salud aún utiliza métodos tradicionales para la programación de citas médicas, como listas manuales, llamadas telefónicas y atención presencial directa. Esta situación ha generado diversas problemáticas que afectan tanto la eficiencia del sistema de salud como la calidad del servicio brindado a los pacientes.

Entre los principales inconvenientes se encuentran los largos tiempos de espera, la saturación de las instalaciones médicas, el ausentismo por falta de recordatorios, y la desorganización en la gestión de turnos. Estas dificultades no solo repercuten en la experiencia del paciente, sino que también afectan la planificación interna de los centros médicos y limitan el aprovechamiento óptimo de los recursos disponibles.

A pesar de que existen avances internacionales en la digitalización de estos procesos, en Nicaragua aún es limitado el uso de soluciones tecnológicas que permitan automatizar y optimizar la gestión de citas médicas. Esto evidencia una brecha importante entre las necesidades reales del sistema de salud y la oferta tecnológica existente.

Ante este panorama, surge la necesidad de desarrollar una herramienta digital accesible, intuitiva y eficiente que permita transformar la manera en que se organizan las citas médicas. Tal solución debe responder a las condiciones tecnológicas del país y atender tanto a los pacientes como al personal médico y administrativo.

## Preguntas de investigación:

**Pregunta general:**

¿Cómo puede una aplicación multiplataforma optimizar la gestión de citas médicas, para facilitar la interacción con los usuarios, reducir el ausentismo y garantizar una arquitectura técnica eficiente?

**Preguntas específicas:**

- ¿Qué características debe tener una interfaz de usuario para ser considerada intuitiva y adaptable tanto en web como en dispositivos móviles?
- ¿Qué tipo de notificaciones (SMS, correo electrónico) resultaría más efectivo para recordar las citas médicas a los pacientes?
- ¿Cuáles serían los beneficios técnicos y funcionales de centralizar la lógica del sistema en una API para una aplicación de citas médicas?

## Justificación:

El acceso oportuno y eficiente a los servicios de salud en Nicaragua continúa siendo un desafío, particularmente en aquellos centros médicos que aún dependen de métodos manuales para la programación de citas. Estas limitaciones generan problemas recurrentes como largas filas, demoras en la atención, desorganización administrativa y pérdida de información clínica relevante.

Considerando este panorama se genera la oportunidad de mejorar con el siguiente proyecto propone el desarrollo de una aplicación multiplataforma (web y Android) que permita gestionar de forma eficiente el proceso de agendamiento de citas médicas. Esta solución digital busca mejorar la experiencia del paciente, fortalecer la operatividad de los centros de salud y reducir la carga de trabajo administrativo.

La implementación de este sistema permitirá automatizar procesos, centralizar la información y brindar notificaciones en tiempo real, contribuyendo a la modernización de los servicios médicos. Además, al integrar tecnologías accesibles y adaptadas al entorno local, se promueve la inclusión digital en el sector salud.

Este proyecto responde no solo a una necesidad tecnológica concreta, sino también al compromiso académico con el desarrollo de soluciones innovadoras orientadas al bienestar social. Su ejecución fortalecerá las competencias profesionales de los autores, promoverá el uso de tecnologías emergentes y aportará una herramienta útil y escalable para el sistema sanitario nicaragüense.

El proyecto tiene un impacto significativo tanto en la experiencia del paciente como en la gestión médica. Para los pacientes, mejora al reducir tiempos de espera, eliminar filas innecesarias, facilitar la gestión de citas y disminuir el ausentismo mediante recordatorios automatizados. En cuanto a la gestión médica, optimiza la organización interna con una agenda digital actualizada, reduce la carga administrativa del personal y mejora la eficiencia en la asignación de turnos, evitando conflictos de disponibilidad.

## Objetivos:

### Objetivo general:

Desarrollar una aplicación multiplataforma que permita gestionar de manera eficaz el proceso de agendamiento de citas médicas.

### Objetivos específicos:

1. Diseñar una interfaz de usuario intuitiva y adaptable que permita a pacientes y personal médico interactuar fácilmente con la plataforma, tanto en su versión web como móvil.
2. Implementar un sistema automatizado de notificaciones y recordatorios para disminuir el ausentismo de pacientes y mejorar la eficiencia del servicio médico.
3. Desarrollar una API RESTful que centralice la lógica del sistema y permita la comunicación eficiente entre la base de datos y las interfaces de usuario.

## Marco Teórico:

### Arquitectura y Organización del Sistema

Este proyecto está basado en una arquitectura cliente-servidor moderna, donde la interacción entre el frontend (aplicaciones web y móvil) y el backend (servidores y servicios en la nube) se realiza de manera eficiente a través de APIs REST y servicios de Firebase.

### Patrones de Diseño Utilizados

Web – MVC (Modelo-Vista-Controlador):

- Separa claramente la lógica de negocio, la presentación y el acceso a datos.
- Mejora la organización del código y facilita el mantenimiento y escalabilidad.

Móvil – MVVM (Modelo-Vista-ViewModel) y separación de responsabilidades:

- En Flutter se promueve la división de UI, lógica de presentación.
- Se aplican técnicas como Provider o Riverpod para la gestión de estado.

### Flujo General de Información

1. **Paciente móvil:**
   - Inicia sesión con Firebase Authentication.
   - Realiza una cita, que se almacena en Firestore.
   - Recibe notificaciones push mediante FCM.
   - Puede enviar mensajes al doctor y consultar su expediente médico.
2. **Doctor móvil:**
   - Visualiza sus citas mediante Firestore.
   - Modifica citas; los cambios se sincronizan en tiempo real.
   - Accede y edita expedientes clínicos de sus pacientes.

3. **Director móvil:**
    - Administra doctores, horarios y especialidades desde su app.
    - Tiene acceso completo a la base de datos del sistema.
4. **Sistema Web (ASP.NET Core):**
    - Administra todos los datos del sistema (citas, doctores, especialidades).
    - Se conecta con MySQL a través de Entity Framework Core.
    - Envía confirmaciones de cita por correo mediante SMTP.
    - Expone APIs REST para consultas desde aplicaciones móviles si se requiere sincronización cruzada.

**Comunicación entre Aplicaciones y Servicios**

**API REST (Web):**

La parte web de la aplicación ya expone endpoints RESTful que permiten compartir datos estructurados con otros módulos del sistema, facilitando, por ejemplo, la sincronización cruzada de información entre diferentes componentes. Para lograrlo, se utilizó ASP.NET Core, lo que permitió desarrollar APIs robustas y seguras de manera eficiente. Gracias a esta tecnología, se implementaron mecanismos de autenticación, enrutamiento y manejo de solicitudes que garantizan un intercambio de datos confiable y escalable, asegurando una integración fluida dentro del ecosistema de la aplicación.

**Firebase Services (Móvil):**

- Firestore actúa como base de datos en tiempo real para la app móvil.
- FCM permite la mensajería push entre servidor y dispositivos.
- Authentication gestiona el acceso seguro de los usuarios.

**Arquitectura General del Sistema**

Arquitectura cliente-servidor: Web (ASP.NET Core) + Móvil (Flutter) + Backend en la nube (Firebase).

La arquitectura cliente-servidor es un modelo ampliamente adoptado en el desarrollo de sistemas distribuidos, donde los clientes solicitan servicios o recursos, y un servidor central los procesa y responde. En el presente proyecto, se ha implementado una arquitectura cliente-servidor híbrida que integra aplicaciones web y móviles, junto con servicios en la nube para el almacenamiento y gestión de datos.

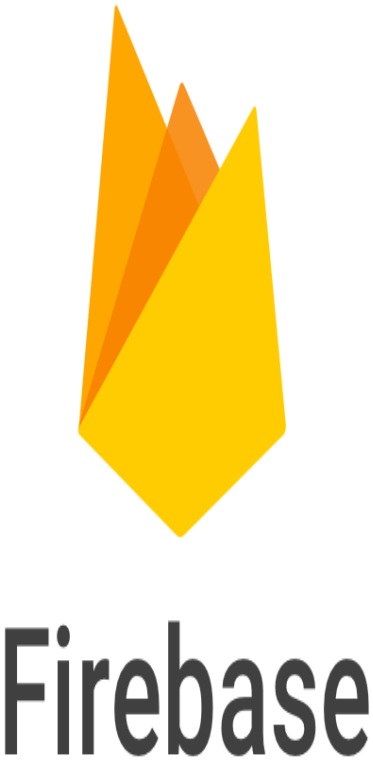
## Componentes de la arquitectura

### Cliente Web (ASP.NET Core MVC)

El cliente web está desarrollado utilizando el framework ASP.NET Core MVC, que sigue el patrón Modelo-Vista-Controlador. Esta capa permite a usuarios con perfil administrativo (por ejemplo, directores médicos) gestionar el sistema desde una interfaz accesible vía navegador. Las vistas están construidas con Razor Pages, y los controladores manejan la lógica de negocio y las interacciones con la base de datos mediante Entity Framework Core.

### Cliente Móvil (Flutter + Dart)

La aplicación móvil está construida con Flutter, un framework multiplataforma que utiliza el lenguaje Dart. Esta aplicación está orientada a pacientes y doctores, permitiendo funcionalidades como agendamiento de citas, consultas de expediente médico, mensajería directa y recepción de notificaciones. Flutter permite una experiencia de usuario fluida y nativa tanto en Android como en iOS.




### Backend en la Nube (Firebase)

Firebase actúa como backend en la nube y provee diversos servicios esenciales para la aplicación móvil:

**Firebase Authentication:** Manejo de autenticación de usuarios mediante correo, Google, entre otros.

**Cloud Firestore:** Base de datos NoSQL en tiempo real, donde se almacenan citas, mensajes, historiales y otra información relevante.

**Firebase Cloud Messaging (FCM):** Servicio de notificaciones push que permite enviar alertas instantáneas sobre cambios en citas o mensajes nuevos.

**Firebase Storage:** Almacenamiento de archivos, útil para resguardar informes médicos o imágenes relacionadas al expediente del paciente.

### Interacción entre componentes

La comunicación entre las aplicaciones cliente (web y móvil) y el backend se realiza mediante API RESTful. ASP.NET Core expone servicios web que permiten a la aplicación móvil acceder o modificar información almacenada en la base de datos MySQL del servidor. Paralelamente, la aplicación móvil interactúa con Firebase para operaciones que requieren sincronización en tiempo real o almacenamiento temporal.

Este enfoque dual permite aprovechar lo mejor de ambos mundos: una base de datos relacional robusta (MySQL) para procesos administrativos, y una plataforma flexible en la nube (Firebase) para mejorar la experiencia del usuario móvil, especialmente en aspectos como tiempo real, notificaciones y accesibilidad desde cualquier lugar.

## Patrones de diseño utilizados (MVC en Web, MVVM y separación de lógica en Flutter)

Los patrones de diseño son esquemas reutilizables de arquitectura que permiten organizar el código de manera eficiente, facilitar el mantenimiento y fomentar la escalabilidad de las aplicaciones. En este proyecto se han implementado diferentes patrones de diseño tanto en el desarrollo web como en el desarrollo móvil, adaptados a las características específicas de cada entorno tecnológico.

### Patrón MVC en ASP.NET Core Web

El patrón Modelo-Vista-Controlador (MVC) es el núcleo del desarrollo de aplicaciones web con ASP.NET Core. Este patrón promueve la separación de responsabilidades en tres componentes principales:

- **Modelo:** Representa la lógica de acceso a datos y las estructuras del negocio. En este caso, se definen entidades como Cita, Doctor, Especialidad, entre otras, que se gestionan mediante Entity Framework Core y se persisten en una base de datos MySQL.
- **Vista:** Define la presentación de la información al usuario. Utiliza Razor Pages para generar contenido HTML dinámico basado en los datos del modelo.
- **Controlador:** Actúa como intermediario entre el modelo y la vista. Recibe las solicitudes del usuario, procesa la lógica y determina qué vista debe mostrarse.

  Este patrón facilita la organización del código, la reutilización de componentes y una interfaz limpia para los usuarios administradores y directores del sistema.

### Patrón MVVM y Separación de Lógica en Flutter

En el desarrollo móvil con Flutter se ha adoptado el enfoque del patrón MVVM (Modelo-Vista-ViewModel) de forma adaptada, combinando buenas prácticas de arquitectura limpia y separación de responsabilidades para mantener un código modular y mantenible.

- **Modelo:** Define las clases de datos, como usuarios, citas o mensajes. Estas clases son utilizadas por otras capas de la aplicación.
- **Vista:** Está compuesta por los widgets de Flutter, que representan la interfaz gráfica del usuario (UI). Estas vistas son reactivas y responden a cambios en el estado del ViewModel.
- **ViewModel:** Encapsula la lógica de presentación, contiene los controladores y los estados necesarios para que la vista funcione correctamente. Interactúa con los modelos para obtener o modificar datos y notifica a la vista de los cambios mediante mecanismos de notificación como ChangeNotifier.

  Además del MVVM, se aplica una **arquitectura limpia** al dividir el código en capas como:

- **Servicios:** Encargados de interactuar con Firebase o con APIs externas (por ejemplo, llamadas HTTP al backend ASP.NET).
- **Repositorios:** Que abstraen el origen de los datos (pueden provenir de Firebase, SQLite o memoria).
- **Controladores o Providers:** Que manejan el estado de la aplicación usando librerías como provider, riverpod o bloc.

Esta separación permite testear de forma independiente cada componente, facilita la extensión de funcionalidades, y promueve un desarrollo basado en principios SOLID.

## Flujo general de información entre frontend, backend y base de datos

El flujo de información en una aplicación cliente-servidor moderna es fundamental para garantizar la coherencia de los datos, la interacción fluida del usuario y la seguridad de los procesos. En este proyecto, el flujo de datos se organiza en función de dos canales principales de interacción: la plataforma web (ASP.NET Core MVC) y la aplicación móvil (Flutter), ambas comunicándose con un backend y distintas bases de datos (MySQL y Firebase).

## Flujo en la Plataforma Web (ASP.NET Core MVC)

El flujo de información para la plataforma web sigue una arquitectura tradicional cliente-servidor con persistencia de datos relacional:

**Interacción del Usuario:** El usuario accede a través de un navegador y realiza acciones como crear una cita o registrar un doctor.

**Petición HTTP:** Las acciones del usuario son enviadas como solicitudes HTTP (GET, POST, PUT, DELETE) a los controladores de ASP.NET Core.

**Lógica de Controlador:** El controlador procesa la petición, valida los datos y se comunica con el modelo correspondiente.

**Acceso a Datos:** A través de Entity Framework Core, el modelo accede o modifica la base de datos MySQL utilizando DbContext.

**Respuesta:** El controlador retorna una vista (HTML dinámico) con los datos actualizados, que se muestran en el navegador del usuario.

Este flujo es síncrono y centrado en el modelo relacional, ideal para operaciones administrativas y gestión estructurada de datos.

## Flujo en la Aplicación Móvil (Flutter + Firebase)

El flujo de información en la app móvil es más flexible y orientado a servicios en la nube y sincronización en tiempo real:

**Inicio de Sesión:** El usuario inicia sesión mediante Firebase Authentication.

**Interacción del Usuario:** El paciente o doctor navega por la app, visualiza información, agenda citas.

**Servicios y ViewModel:** La capa lógica en Flutter (ViewModel o provider) interpreta las acciones del usuario y llama a los servicios correspondientes.

**Comunicación con Firebase:**

1. **Cloud Firestore:** Se consulta, crea o actualiza información médica o de citas.
2. **Cloud Messaging:** Se envían notificaciones push cuando se modifica una cita.

3. **Firebase Storage:** Se suben o descargan documentos médicos, si aplica.

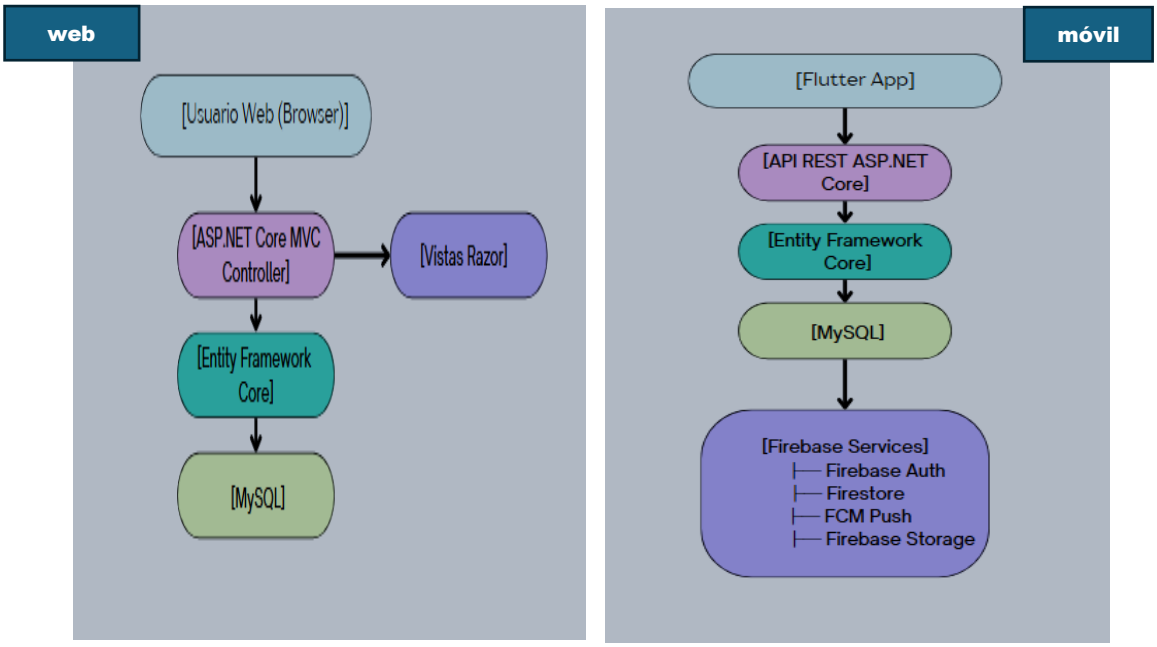
**Respuesta en Tiempo Real:** Firebase responde automáticamente a los cambios con flujos reactivos, actualizando la interfaz del usuario sin necesidad de recargar manualmente.

Este flujo permite experiencias reactivas y asincrónicas, ideales para usuarios que requieren movilidad y notificaciones inmediatas.

**Flujo Híbrido: Comunicación con API REST**

Algunas de las operaciones clave de la aplicación móvil requieren acceso a datos que residen exclusivamente en el backend del sistema web, como la verificación de la disponibilidad de doctores, la gestión de horarios o la sincronización de citas médicas con una base de datos relacional. Para satisfacer estos requerimientos, la app desarrollada en Flutter establece comunicación con el backend mediante peticiones HTTP dirigidas a servicios RESTful implementados en ASP.NET Core. Cada solicitud enviada desde Flutter es procesada por controladores definidos en el backend, los cuales interactúan con la base de datos MySQL utilizando Entity Framework Core como ORM (Object-Relational Mapping). Esta arquitectura permite que el backend consulte, procese y estructure la información necesaria, devolviéndola en formato JSON para que Flutter pueda interpretarla y presentarla adecuadamente al usuario final. Este flujo de datos, que combina la interfaz móvil con un backend robusto y centralizado, permite mantener una lógica de negocio coherente y segura, mientras se aprovechan los beneficios de la computación en la nube, como la escalabilidad, el acceso en tiempo real y la integración multiplataforma. Además, este enfoque facilita el mantenimiento y la evolución del sistema, ya que la lógica crítica permanece en el backend, reduciendo la complejidad en el cliente móvil.

## Diagrama Resumido del Flujo de Información




El diagrama representa el flujo de información entre la app web y móvil, donde ambas acceden a MySQL mediante ASP.NET Core, y la app Flutter también intercambia datos con servicios Firebase.

### Comunicación entre aplicaciones mediante API REST y servicios Firebase

El presente proyecto contempla un enfoque multiplataforma en el que coexisten una aplicación web desarrollada con ASP.NET Core MVC y una aplicación móvil construida con Flutter. La sincronización entre ambas plataformas se logra a través de dos mecanismos clave: API RESTful para la interacción directa con el backend y servicios de Firebase para funcionalidades en tiempo real, autenticación y persistencia en la nube.

Comunicación mediante API REST (Web ↔ Móvil)

La API REST actúa como un puente entre el frontend móvil y el backend web, permitiendo que Flutter consuma servicios expuestos por el servidor ASP.NET Core.

### Estructura de la API REST

Se define en el backend web con controladores específicos ([ApiController]) que reciben solicitudes HTTP (GET, POST, PUT, DELETE).

Las rutas están organizadas por recursos, por ejemplo:

/api/citas – para gestionar las citas.

/api/doctores – para obtener la lista de doctores disponibles.

/api/especialidades – para consultar especialidades médicas.

## Flujo de comunicación

El usuario en la app móvil ejecuta una acción (crear cita, consultar doctores, etc.).

Flutter envía una solicitud HTTP al backend usando http, dio o alguna librería similar.

ASP.NET Core procesa la solicitud, accede a la base de datos MySQL mediante Entity Framework Core y responde con un objeto JSON.

Flutter recibe y deserializa la respuesta para presentarla en la interfaz.

## Seguridad y Control de Acceso

Para proteger la comunicación:

Se utiliza autenticación basada en tokens (JWT o Firebase ID Token).

Se implementan políticas de autorización en el backend para restringir el acceso a usuarios no autorizados.

### Comunicación mediante Servicios Firebase

Firebase proporciona una suite de herramientas que simplifican el desarrollo móvil. En este proyecto, se han implementado los siguientes servicios:

### Firebase Authentication


Permite a los pacientes y doctores autenticarse mediante correo electrónico y contraseña.

Gestiona sesiones de forma segura y persistente.

Los tokens de sesión pueden ser reutilizados para autenticación en peticiones a la API REST.

**Cloud Firestore**

Base de datos NoSQL en tiempo real para almacenar datos dinámicos como mensajes, historial de citas y notificaciones. Ofrece sincronización automática entre múltiples dispositivos.

**Cloud Messaging (FCM)** 


Utilizado para el envío de notificaciones push.

Se informa al paciente cuando su cita ha sido modificada, confirmada o cancelada.

También puede notificar al doctor sobre nuevas consultas.

**Firebase Storage** 


Permite almacenar archivos como resultados de laboratorio o documentos clínicos asociados al expediente del paciente.

Los archivos pueden ser accedidos desde la app móvil por usuarios autenticados.

**Integración Coordinada**

La combinación de API REST y Firebase permite una arquitectura híbrida en la que:

La lógica centralizada del sistema (validaciones complejas o integridad de datos) se maneja desde el backend ASP.NET Core.

La experiencia del usuario móvil (como notificaciones, autenticación y datos en tiempo real) se gestiona con Firebase para maximizar la velocidad y respuesta inmediata. Esto permite escalar el sistema fácilmente, mantener una experiencia unificada entre plataformas y reutilizar servicios comunes.

## Desarrollo del Módulo Web (ASP.NET Core MVC)

El módulo web del sistema fue construido utilizando **ASP.NET Core MVC**, un framework moderno y multiplataforma que permite desarrollar aplicaciones web robustas, escalables y bien estructuradas mediante el patrón arquitectónico **Modelo-Vista-Controlador (MVC)**. Este enfoque garantiza una clara separación de responsabilidades, facilita el mantenimiento del código y permite una escalabilidad eficiente.

**Estructura General del Proyecto**

El sistema web se encuentra dividido en múltiples áreas funcionales, cada una controlada por su respectivo controlador y modelo. Esta estructura promueve el modularidad, el desacoplamiento del código y la reutilización de componentes.

## Componentes principales:

- **Modelos**: Representan las entidades del dominio (Cita, Doctor, Especialidad, Usuario).
- **Vistas**: Plantillas Razor que permiten generar interfaces dinámicas.
- **Controladores**: Gestionan las peticiones HTTP y vinculan los modelos con las vistas.

**Controlador Cita**

Responsable de gestionar todo lo relacionado con las citas médicas:

- Crear nuevas citas.
- Listar, editar y cancelar citas.
- Validaciones de disponibilidad por fecha, hora y especialidad.
- Comunicación con el modelo Cita y su correspondiente vista para mostrar información dinámica.

**Controlador Doctor**

Administra la información del personal médico:

- Alta, modificación y baja de doctores.
- Consulta de información detallada: nombre, especialidad, horario laboral.
- Enlace con la vista para mostrar los datos a los usuarios y administradores.

**Controlador Especialidad**

Permite mantener actualizadas las especialidades médicas que se ofrecen:

- Gestión CRUD de especialidades.
- Uso de este controlador como filtro previo a la asignación de un doctor.
- Integración con el modelo Especialidad y sus vistas asociad

**Controlador ConfirmaciónCita**

Su principal propósito es la notificación por correo electrónico al paciente cuando una cita es creada o modificada.

- Integración con un servidor SMTP para el envío de correos automáticos.
- Generación dinámica del contenido del correo (fecha, hora, doctor, especialidad).
- Facilita la comunicación entre el sistema y el paciente.

## Conexión a la Base de Datos

Se utilizó **Entity Framework Core** como ORM para facilitar la interacción con **MySQL**. Todas las operaciones de persistencia se gestionan mediante el contexto AppDbContext, el cual contiene los DbSet correspondientes a cada entidad del sistema.

**Ventajas del uso de EF Core:**

- Permite consultas LINQ para facilitar el manejo de datos.
- Automatiza la creación de tablas y relaciones mediante migraciones.
- Abstracción de SQL puro, mejorando la productividad y seguridad del desarrollador.

**Seguridad y Gestión de Accesos**

El sistema web contempla distintos niveles de usuario:

- **Pacientes**: Acceso limitado a creación y consulta de citas.
- **Doctores**: Visualización y edición del expediente del paciente.
- **Administrador (Director)**: Acceso completo a la gestión del sistema.

Se implementan mecanismos de autenticación y autorización basados en roles para restringir el acceso a funcionalidades críticas.

## Integración con Módulo Móvil

El módulo web también actúa como backend para la aplicación móvil, sirviendo datos mediante una **API REST** que expone los controladores necesarios. De esta manera, los datos se mantienen sincronizados entre la plataforma web y móvil.

### Estructura del Proyecto Web

### Organización General del Proyecto

El proyecto está organizado en las siguientes carpetas principales:

- **Controllers**
  Contiene los controladores que manejan las solicitudes HTTP, procesan la lógica de negocio y devuelven respuestas a las vistas o a la API.
- **Models**
  Incluye las clases que representan las entidades del dominio (como Cita, Doctor, Especialidad, Usuario). Estas clases también definen las relaciones entre las entidades y son utilizadas por Entity Framework Core para mapear a la base de datos.
- **Views**
  Carpeta que contiene las plantillas de interfaz de usuario escritas con Razor. Está subdividida por controlador, lo que permite mantener organizada la representación visual del sistema.
- **Data**
  Contiene la clase AppDbContext, la cual representa el contexto de la base de datos y se utiliza para interactuar con MySQL a través de Entity Framework Core.
- **wwwroot**
  Almacena archivos estáticos como CSS, JavaScript, imágenes y bibliotecas de frontend utilizadas por la aplicación.

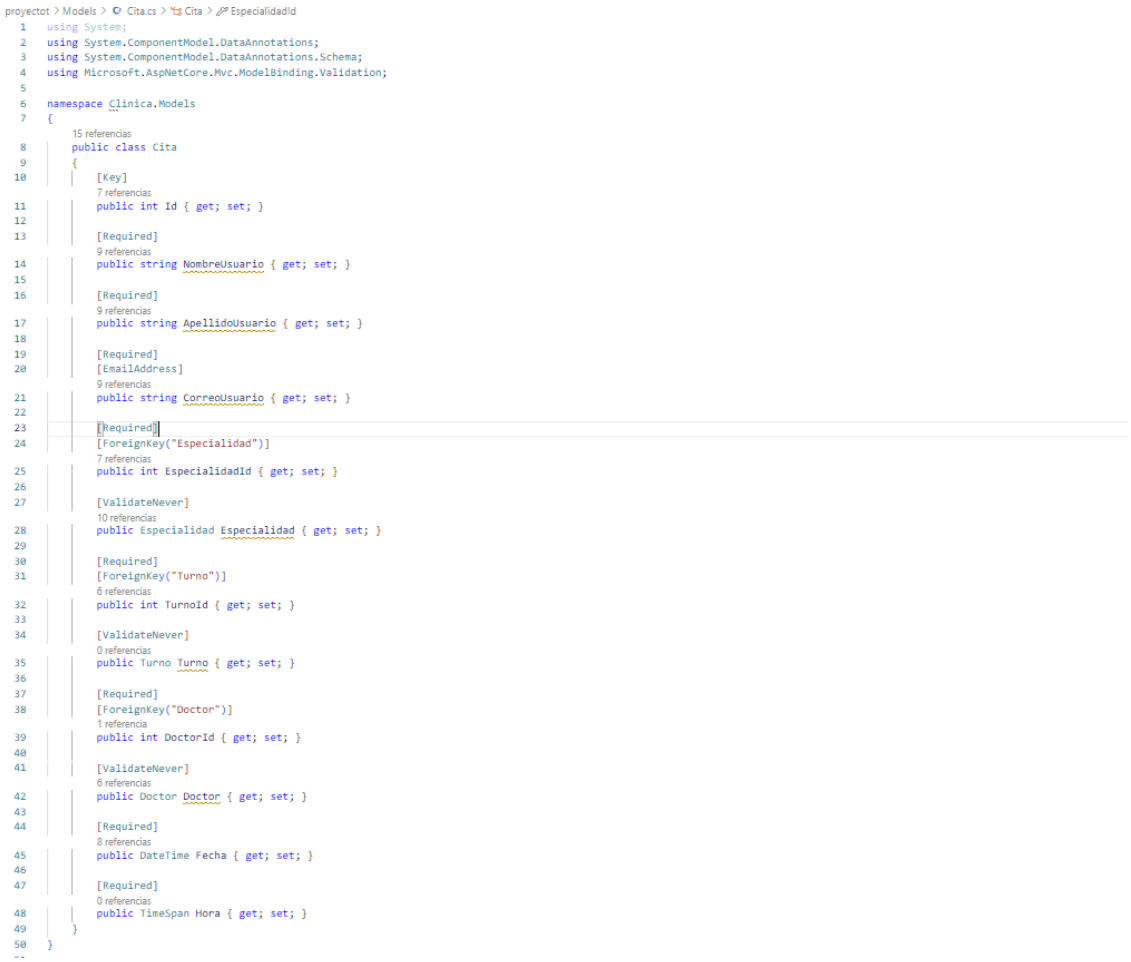
- **Migrations**
  Carpeta que almacena los archivos generados por las migraciones de Entity Framework Core para la gestión de cambios en el esquema de base de datos.
- **appsettings. json**
  Archivo de configuración donde se especifican parámetros del sistema, como la cadena de conexión a MySQL, configuración de correo SMTP, entre otros.

**Controladores Principales:**

- **CitaController**:
  Gestiona la creación, edición, eliminación y visualización de citas médicas.
- **DoctorController**:
  Administra los datos de los doctores, incluyendo horarios y especialidades.
- **EspecialidadController**:
  Permite la gestión de las especialidades médicas registradas en el sistema.
- **ConfirmacionCitaController:**
  Encargado de generar y enviar correos electrónicos al paciente cuando una cita es confirmada o modificada.

## Modelo de Datos (Entity Framework Core)

Las entidades están representadas como clases con propiedades que se traducen directamente a tablas y columnas en MySQL. Por ejemplo:



```
proyectot > Models > Cita.cs > Cita > EspecialidadId
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;

namespace Clinica.Models
{
    15 referencias
    public class Cita
    {
        [Key]
        7 referencias
        public int Id { get; set; }

        [Required]
        9 referencias
        public string NombreUsuario { get; set; }

        [Required]
        9 referencias
        public string ApellidoUsuario { get; set; }

        [Required]
        [EmailAddress]
        9 referencias
        public string CorreoUsuario { get; set; }

        [Required]
        [ForeignKey("Especialidad")]
        7 referencias
        public int EspecialidadId { get; set; }

        [ValidateNever]
        10 referencias
        public Especialidad Especialidad { get; set; }

        [Required]
        [ForeignKey("Turno")]
        6 referencias
        public int TurnoId { get; set; }

        [ValidateNever]
        0 referencias
        public Turno Turno { get; set; }

        [Required]
        [ForeignKey("Doctor")]
        1 referencia
        public int DoctorId { get; set; }

        [ValidateNever]
        6 referencias
        public Doctor Doctor { get; set; }

        [Required]
        8 referencias
        public DateTime Fecha { get; set; }

        [Required]
        0 referencias
        public TimeSpan Hora { get; set; }
    }
}
```

*Ilustración 1: Define la estructura de una cita médica con sus propiedades clave para el almacenamiento de datos.*

### Flujo de Trabajo MVC

1. El usuario realiza una solicitud en el navegador (por ejemplo, crea una cita).
2. El controlador correspondiente recibe la solicitud y procesa la lógica de negocio.
3. El modelo se comunica con la base de datos para recuperar o almacenar información.
4. Los resultados se devuelven a una vista, que se encarga de renderizar la respuesta HTML y presentarla al usuario.

## Modelo-Vista-Controlador (MVC)

El patrón **Modelo-Vista-Controlador (MVC)** es un enfoque arquitectónico ampliamente utilizado en el desarrollo de aplicaciones web, y ha sido implementado

en este proyecto mediante el framework **ASP.NET Core MVC**. Este patrón promueve la separación de responsabilidades, permitiendo una mejor organización del código, una mayor mantenibilidad y facilidad para futuras expansiones o integraciones.

## Descripción del Patrón MVC

El patrón MVC divide una aplicación en tres componentes principales:

- **Modelo (Model)**

  Representa los datos y la lógica. En este proyecto, los modelos son clases que definen las entidades principales del sistema como Cita, Doctor, Paciente, y Especialidad. También se conectan con la base de datos mediante **Entity Framework Core**.

- **Vista (View)**

  Es la parte de la aplicación encargada de la presentación visual al usuario. Las vistas en ASP.NET Core se construyen con **Razor**, una sintaxis que combina HTML con C#. Cada vista está asociada a una acción del controlador y permite mostrar datos dinámicos que provienen del modelo.

- **Controlador (Controller)**

  Maneja las solicitudes del usuario, procesa la lógica de la aplicación y devuelve una respuesta. Los controladores actúan como intermediarios entre los modelos y las vistas. Cada controlador en el sistema (como CitaController, DoctorController, etc.) contiene métodos que responden a acciones específicas como crear, editar o eliminar datos.

## Aplicación del MVC en el Proyecto

Ejemplo aplicado al flujo de gestión de una cita médica:

**Vista**: El usuario llena un formulario para agendar una cita.

**Controlador**: El método Create del CitaController recibe los datos, valida la información y llama al modelo.

**Modelo**: El objeto Cita es guardado en la base de datos mediante el contexto AppDbContext.

**Controlador**: Devuelve una vista de confirmación.

**Vista**: Se muestra al usuario una página que confirma la creación exitosa de la cita.

Este flujo garantiza que cada capa tenga una única responsabilidad, reduciendo la complejidad del sistema y facilitando la depuración o ampliación de este.

La implementación del patrón Modelo-Vista-Controlador (MVC) en el módulo web garantiza una base sólida para el desarrollo estructurado de funcionalidades. Su adopción no solo mejora la claridad del proyecto, sino que también permite un desarrollo colaborativo más efectivo y una integración más ordenada con otros módulos como el sistema móvil.

## Organización de Controladores: Cita, Doctor, Especialidad, Confirmación

El sistema está estructurado alrededor de cuatro controladores fundamentales que se alinean con las entidades principales del dominio médico:

### CitaController

La responsabilidad principal del puesto consiste en administrar todos los aspectos relacionados con la gestión de citas médicas. Entre sus funciones clave se encuentran la creación de nuevas citas, la edición o reprogramación de citas ya existentes, la eliminación de citas cuando sea necesario y la organización de las citas programadas, ya sea por paciente o por fecha, con el fin de mantener un control ordenado y eficiente del calendario médico.

**Relación con otros componentes**:

Además, esta función hace uso del modelo Cita para gestionar la información relacionada con cada reserva médica. Interactúa directamente con los modelos Doctor y Paciente, lo cual permite validar la disponibilidad de los profesionales de la salud y asegurar la correcta asignación de las citas. Asimismo, está integrada con las vistas del sistema, tales como Create.cshtml, Edit.cshtml e Index.cshtml, facilitando la creación, edición y visualización de las citas desde la interfaz del usuario.

**DoctorController**

La responsabilidad principal de este rol es gestionar la información del personal médico. Entre sus funciones clave se incluyen el registro de nuevos doctores en el sistema, la edición de su información personal y profesional, como nombre, especialidad y horario de atención, así como la eliminación de aquellos doctores que se encuentren inactivos. Además, permite visualizar la lista completa de doctores registrados y acceder a los detalles individuales de cada uno para una administración más eficiente y actualizada del equipo médico.

**Relación con otros componentes**:

Este componente utiliza el modelo Doctor como base para gestionar la información del personal médico y se vincula con el modelo Especialidad, permitiendo asignar una o varias especialidades a cada doctor según su formación y experiencia. Además, se conecta con diversas vistas del sistema, como Details.cshtml, Create.cshtml y Edit.cshtml, lo que facilita la visualización de información detallada, así como la creación y edición de registros médicos desde la interfaz del usuario.

**EspecialidadController**

La responsabilidad principal de este componente es administrar las especialidades médicas disponibles en el sistema. Sus funciones clave incluyen la creación y registro de nuevas especialidades, la edición de sus nombres o descripciones para mantener la información actualizada, la eliminación de aquellas especialidades que ya no se utilicen, y la visualización de todas las especialidades disponibles, lo cual

facilita su correcta asignación al personal médico y mejora la organización del sistema de salud.

**Relación con otros componentes**:

Este componente mantiene una relación directa con el modelo Doctor, permitiendo clasificar a los médicos según su área médica correspondiente. Gracias a esta relación, es posible asignar una o varias especialidades a cada doctor, lo que mejora la organización y búsqueda de profesionales según sus competencias. Además, se apoya en vistas asociadas, como formularios y listados, que facilitan la gestión visual de las especialidades dentro del sistema.

**ConfirmacionCitaController**

El ConfirmacionCitaController tiene como responsabilidad principal enviar confirmaciones automáticas de citas médicas por correo electrónico al paciente. Entre sus funciones clave se encuentra la generación de correos electrónicos que incluyen los datos relevantes de la cita, como la fecha, la hora, el doctor asignado y la especialidad correspondiente. Utiliza la configuración SMTP para realizar el envío automático de estos mensajes, y se encarga de asegurar que las notificaciones sean enviadas correctamente tanto al momento de crear una cita como cuando esta es modificada, garantizando así una comunicación oportuna y eficiente con el paciente.

**Relación con otros componentes**:

El ConfirmacionCitaController se relaciona directamente con el modelo Cita y sus asociaciones, de las cuales obtiene la información necesaria para generar los correos de confirmación, como datos del paciente, doctor, fecha y hora. No cuenta con vistas propias, ya que actúa como un servicio de notificación en segundo plano, sin requerir interacción directa con el usuario a través de la interfaz. Para el envío de correos, utiliza los servicios SMTP previamente configurados en el archivo

appsettings.json, lo que permite automatizar la entrega de notificaciones de manera eficiente y centralizada.

## Envío de Notificaciones vía Correo con SMTP

Una funcionalidad clave del sistema web desarrollado con ASP.NET Core MVC es el envío automático de correos electrónicos como mecanismo de notificación para los usuarios. Esta capacidad está implementada en el ConfirmacionCitaController y tiene como finalidad mantener informado al paciente sobre el estado de sus citas médicas. El sistema envía correos electrónicos para confirmar la creación exitosa de una cita, notificar cambios en la fecha u hora de una cita existente, o informar sobre cancelaciones u otras modificaciones importantes. Este proceso no solo asegura una comunicación efectiva, sino que también mejora la experiencia del usuario al brindarle mayor confianza mediante recordatorios y confirmaciones automatizadas.

### Tecnología Utilizada

- **Protocolo**: SMTP (Simple Mail Transfer Protocol).
- **Lenguaje**: C# con ASP.NET Core.
- **Configuración**: Parámetros SMTP definidos en el archivo appsettings. json.
- **Librería**: Uso de SmtpClient y MailMessage del espacio de nombres System.Net.Mail.

## Proceso de Envío de Correo

1. **Captura de datos**: Cuando un usuario agenda una cita, el sistema recopila información como:
    - Nombre del paciente.
    - Especialidad médica.
    - Nombre del doctor.
    - Fecha y hora de la cita.
2. **Envío a través de SMTP**:
    - Se configura un SmtpClient con servidor, puerto, credenciales y opciones de seguridad.
    - Se ejecuta el método Send () para entregar el mensaje.
3. **Resultado**:

- Si el envío es exitoso, se muestra un mensaje de confirmación.
- En caso de fallo (por ejemplo, error de red o credenciales inválidas), se captura una excepción para manejo de errores.

## Configuración Básica en appsettings.json

```
"EmailSettings": {
  "SmtpServer": "smtp.gmail.com",
  "Port": "587",
  "Username": "fsmm0404xm@gmail.com",
  "Password": "04022002faS12.",
  "FromAddress": "fsmm0404xm@gmail.com"
}
```

*Ilustración 2: Muestra la lógica para enviar correos desde Gmail mediante SMTP, incluyendo servidor, puerto, usuario y contraseña.*

## Código Simplificado de Ejemplo

```
proyectot > Controllers > EmailService.cs > EmailService
// Services/EmailService.cs
using System.Net;
using System.Net.Mail;
using System.Threading.Tasks;
using Microsoft.Extensions.Options;
using proyectot.Models; // Asegúrate de que el namespace sea el correcto

1 referencia
public class EmailService
{
    6 referencias
    private readonly EmailSettings _emailSettings;

    Tabnine | Edit | Test | Explain | Document | 0 referencias
    public EmailService(IOptions<EmailSettings> emailSettings)
    {
        _emailSettings = emailSettings.Value;
    }

    Tabnine | Edit | Test | Explain | Document | 0 referencias
    public async Task EnviarCorreoConfirmacionAsync(string emailDestino, string asunto, string mensaje)
    {
        var mail = new MailMessage(_emailSettings.FromAddress, emailDestino, asunto, mensaje)
        {
            IsBodyHtml = true
        };

        using var smtp = new SmtpClient(_emailSettings.SmtpServer, _emailSettings.Port)
        {
            Credentials = new NetworkCredential(_emailSettings.Username, _emailSettings.Password),
            EnableSsl = true
        };

        await smtp.SendMailAsync(mail);
    }
}
```

*Ilustración 3: Muestra la lógica del controlador EmailService*

**Persistencia de Datos en la Web**

La persistencia de datos es uno de los pilares fundamentales de cualquier aplicación web. En este proyecto, se ha implementado utilizando **Entity Framework Core** como ORM (Object-Relational Mapping), en combinación con **MySQL** como sistema gestor de bases de datos. Esta integración permite almacenar, consultar, modificar y eliminar datos de manera eficiente, segura y mantenible.

## ¿Qué es Entity Framework Core?


Entity Framework Core (EF Core) es un framework de acceso a datos desarrollado por Microsoft que permite a los desarrolladores interactuar con bases de datos relacionales mediante objetos C#, evitando la escritura directa de SQL. EF Core traduce las operaciones del lenguaje C# a comandos SQL que ejecuta sobre la base de datos.

**Configuración de EF Core con MySQL**

La conexión entre la aplicación ASP.NET Core y MySQL se configura desde el archivo appsettings. json y el DbContext de la aplicación.

Ejemplo de configuración:

**appsettings. json**

```
{
  "ConnectionStrings": {
    "DefaultConnection": "server=localhost;database=ClinicaDB;user=root;password=tu_contraseña"
  }
}
```

*Ilustración 4: Muestra la lógica del appsettings.jsom para conectar con la base de datos.*

**Startup.cs (o Program.cs en versiones recientes)**

```
var builder = WebApplication.CreateBuilder(args);

// Registro del contexto con MySQL
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseMySql(
        builder.Configuration.GetConnectionString("DefaultConnection"),
        ServerVersion.AutoDetect(builder.Configuration.GetConnectionString("DefaultConnection"))
    )
);

var app = builder.Build();
```

*Ilustración 5: Muestra ejemplo de lógica en program.cs para el AppDbContext*

### Clase **AppDbContext**

Es el puente entre los modelos del sistema y la base de datos. Define los DbSet para cada entidad.

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options) { }

    public DbSet<Cita> Citas { get; set; }
    public DbSet<Doctor> Doctores { get; set; }
    public DbSet<Especialidad> Especialidades { get; set; }
    // Agrega tus demás entidades aquí
}
```

*Ilustración 6: Muestra ejemplo de lógica del AppDbContext para conexión entre modelos.*

### Ciclo de Vida de una Operación CRUD con EF Core

| Fase | Descripción | Métodos/Funciones Comunes |
|---|---|---|
| **1. Creación del Contexto** | EF Core necesita un `DbContext` para acceder a la base de datos. Se configura la conexión y los DbSet. | `new ApplicationDbContext()` |
| **2. Consulta (Read)** | Se recuperan datos desde la base de datos. Puede ser por ID, por filtros o con relaciones. | `context.Entity.Find(id)context.Entity.ToList()context.Entity.Include(...)` |
| **3. Creación (Create)** | Se crea una nueva entidad y se agrega al contexto, pero aún no se guarda en la base de datos. | `context.Add(entity)context.SaveChanges()` |
| **4. Actualización (Update)** | Se modifica una entidad existente. Primero se recupera, luego se alteran sus propiedades y finalmente se guarda. | `context.Update(entity)context.SaveChanges()` |
| **5. Eliminación (Delete)** | Se elimina una entidad del contexto y luego se guarda el cambio en la base de datos. | `context.Remove(entity)context.SaveChanges()` |
| **6. Seguimiento (Tracking)** | EF Core realiza seguimiento (tracking) de los cambios en las entidades. Esto permite detectar qué propiedades se han modificado. | Automático con el `ChangeTracker` del DbContext |
| **7. Guardado (Save)** | Todos los cambios en el contexto (agregados, modificados, eliminados) se confirman en la base de datos con `SaveChanges()`. | `context.SaveChanges()` |

Tabla 1: Ciclo de vida de una operación CRUD con EF Core

### Migraciones

Las migraciones permiten crear o actualizar la estructura de la base de datos desde el código fuente.

dotnet ef migrations add InitialCreate
dotnet ef database update

### Uso de Entity Framework Core

Entity Framework Core (EF Core) es el **ORM (Object-Relational Mapper)** utilizado en este proyecto para facilitar el acceso a la base de datos MySQL desde la aplicación ASP.NET Core MVC. Esta herramienta permite trabajar con datos

relacionales utilizando objetos C# fuertemente tipados, reduciendo significativamente la necesidad de escribir consultas SQL manuales.

### ¿Qué es EF Core?

EF Core es una versión moderna, ligera, multiplataforma y de alto rendimiento de Entity Framework, diseñada para aplicaciones .NET Core. Su propósito es mapear automáticamente clases .NET a tablas de una base de datos, permitiendo así desarrollar aplicaciones basadas en datos de forma más productiva y segura.

## Componentes Clave de EF Core en el Proyecto

### a) Entidades (Modelos)

Son clases C# que representan las tablas de la base de datos.

**Ejemplo:**

```
public class Paciente
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public string Correo { get; set; }
    public DateTime FechaNacimiento { get; set; }

    // Relaciones
    public ICollection<Cita> Citas { get; set; }
}
```

*Ilustración 7: Muestra ejemplo de la lógica de un modelo paciente.*

### b) DbContext

El DbContext es el **puente principal** entre tus clases (entidades) y la base de datos. Es responsable de:

- Realizar consultas a la base de datos

- Guardar cambios
- Configurar las relaciones entre las entidades

  **Ejemplo:**

```
using Microsoft.EntityFrameworkCore;

public class ClinicaContext : DbContext
{
    public ClinicaContext(DbContextOptions<ClinicaContext> options) : base(options) { }

    public DbSet<Paciente> Pacientes { get; set; }
    public DbSet<Cita> Citas { get; set; }
    public DbSet<Doctor> Doctores { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Configuración adicional si es necesaria (relaciones, restricciones, etc.)
        base.OnModelCreating(modelBuilder);
    }
}
```

*Ilustración 8: Muestra ejemplo de lógica del código AppDbContext*

### c) Configuración del Contexto

Para que EF Core sepa cómo conectarse a tu base de datos **MySQL**, debes configurar el DbContext en **Program.cs.**

```
var builder = WebApplication.CreateBuilder(args);

// Agregar el contexto de la base de datos y configuración de conexión
builder.Services.AddDbContext<ClinicaContext>(options =>
    options.UseMySql(builder.Configuration.GetConnectionString("DefaultConnection"),
    new MySqlServerVersion(new Version(8, 0, 23)) // Asegúrate de usar tu versión de MySQL
));

var app = builder.Build();
```

*Ilustración 9: muestra lógica de configuración en Program.cs*

## Migraciones

EF Core permite aplicar cambios en el esquema de la base de datos a través de comandos de migración.

dotnet ef migrations add InitialCreate

dotnet ef database update

Esto crea automáticamente las tablas y relaciones según los modelos definidos.

## Consultas con LINQ

Ejemplo de consulta para obtener citas filtradas por especialidad:

```
var citas = context.Citas
    .Where(c => c.Especialidad == "Pediatría")
    .Include(c => c.Doctor)
    .ToList();
```

*Ilustración 10: Muestra cómo hacer una consulta con código SQL.*

## Operaciones CRUD Básicas

**Crear:**

```
context.Citas.Add(nuevaCita);
var nuevaCita = new Cita
{
    Fecha = DateTime.Now.AddDays(1),
    Especialidad = "Pediatría",
    DoctorId = 1,
};

context.Citas.Add(nuevaCita);
context.SaveChanges();
```

*Ilustración 11: Muestra ejemplo de lógica básica de operación CRUD créate.*

**Leer:**

```
var cita = context.Citas.Find(id);
```

```
var cita = context.Citas.Find(5);
if (cita != null)
{
    Console.WriteLine($"Cita con el doctor {cita.DoctorId} en {cita.Fecha}");
}
```

*Ilustración 12: Muestra ejemplo de lógica básica de operación CRUD Leer.*

## Actualizar:

```
cita.Fecha = nuevaFecha;
context.SaveChanges();
var cita = context.Citas.Find(5);
if (cita != null)
{
    cita.Fecha = DateTime.Now.AddDays(3);
    context.SaveChanges();
}
```

*Ilustración 13: Muestra ejemplo de lógica básica de operación CRUD actualizar.*

## Eliminar:

```
context.Citas.Remove(cita);
context.SaveChanges();
var cita = context.Citas.Find(5);
if (cita != null)
{
    context.Citas.Remove(cita);
    context.SaveChanges();
```

*Ilustración 14: Muestra ejemplo de lógica básica de operación CRUD Eliminar.*

## MySQL como motor de base de datos.


En este proyecto, se ha optado por utilizar **MySQL** como el sistema de gestión de base de datos (DBMS) principal para el módulo web desarrollado con ASP.NET Core MVC. MySQL es una base de datos relacional de código abierto ampliamente utilizada en aplicaciones web por su rendimiento, estabilidad y compatibilidad con diversos entornos de desarrollo.

### Características de MySQL

MySQL es un sistema de gestión de bases de datos relacional que destaca por ser de código abierto y compatible con múltiples sistemas operativos como Windows, Linux y macOS, lo que facilita su implementación en distintos entornos. Está diseñado para ofrecer un alto rendimiento, capaz de manejar grandes volúmenes de datos y procesar múltiples transacciones simultáneamente sin afectar su velocidad ni estabilidad. Gracias a su escalabilidad, MySQL se adapta tanto a proyectos pequeños como a sistemas empresariales complejos, permitiendo crecer sin necesidad de cambiar de plataforma. Además, cuenta con soporte para integridad referencial mediante el uso de claves foráneas, lo que garantiza que las relaciones entre tablas sean consistentes y que los datos sean confiables. Finalmente, MySQL utiliza el lenguaje SQL, un estándar ampliamente reconocido para la gestión y consulta de bases de datos, lo que facilita su aprendizaje y uso en diversas aplicaciones.

### Justificación del uso de MySQL

Se eligió MySQL por las siguientes razones:

- Su integración nativa y estable con **Entity Framework Core**, el ORM utilizado en ASP.NET Core.
- Su compatibilidad con **herramientas de desarrollo populares** como MySQL Workbench.
- El soporte a largo plazo de la comunidad y de empresas como Oracle.
- Su **rendimiento eficiente en operaciones CRUD** (crear, leer, actualizar y eliminar), que es clave para el funcionamiento del sistema de citas médicas.

## Diseño de la base de datos

La estructura de la base de datos incluye varias entidades clave relacionadas entre sí:

- **Cita**: contiene la información sobre cada cita médica.
- **Doctor**: almacena los datos personales y profesionales de los médicos.
- **Especialidad**: define las áreas médicas disponibles en el sistema.
- **Paciente**: mantiene los registros de los usuarios/pacientes del sistema.
- **Usuario**: incluye credenciales de autenticación y datos básicos.

Relaciones principales:

- Un **Doctor** puede tener muchas **Citas**.
- Una **Especialidad** puede estar asignada a muchos **Doctores**.
- Un **Paciente** puede tener múltiples **Citas**.

## Integración con ASP.NET Core MVC

La conexión entre MySQL y la aplicación web se realiza mediante **Entity Framework Core**, que traduce las operaciones sobre objetos C# en comandos SQL que MySQL ejecuta.


La cadena de conexión se define en el archivo appsettings.json:

```
"ConnectionStrings": {
  "DefaultConnection": "server=localhost;database=ClinicaDB;user=root;password=0404;"
},
"AllowedHosts": "*",
```

*Ilustración 15: Muestra lógica de código para la conexión de la base de datos.*

## Configuración del contexto AppDbContext

En una aplicación ASP.NET Core que utiliza Entity Framework Core como ORM, el contexto de base de datos es el componente fundamental que actúa como un

puente entre el modelo de datos (clases C#) y la base de datos relacional (en este caso, MySQL). En este proyecto, el contexto se define en la clase AppDbContext.

**El AppDbContext**:

- Administra las conexiones a la base de datos.
- Realiza el mapeo entre las clases del modelo y las tablas de la base de datos.
- Permite consultar, insertar, actualizar y eliminar datos utilizando LINQ.
- Gestiona las relaciones entre entidades, validaciones y configuraciones adicionales.

**Estructura de la clase AppDbContext**


*Ilustración 16: Muestra lógica de código completo AppDbContext.*

### Relaciones entre entidades

Las relaciones están definidas usando **navegación entre objetos** y **llaves foráneas**. Algunas relaciones clave son:

- **Uno a muchos** entre Doctor y Cita:
  - Un doctor puede tener muchas citas.
  - Cada cita pertenece a un solo doctor.
- **Uno a muchos** entre Paciente y Cita:
  - Un paciente puede tener varias citas.
  - Cada cita pertenece a un paciente.
- **Uno a muchos** entre Especialidad y Doctor:
  - Una especialidad médica puede ser compartida por varios doctores.
  - Cada doctor tiene una especialidad.

Estas relaciones se configuran explícitamente en el método OnModelCreating () dentro de AppDbContext:

```
modelBuilder.Entity<Doctor>()
    .HasOne(d => d.Turno)
    .WithMany(e => e.Doctores)
    .HasForeignKey(d => d.TurnoId)
    .OnDelete(DeleteBehavior.Restrict); // Cambiar a Restrict o SetNull

// Cita - Especialidad (una cita tiene una especialidad)
modelBuilder.Entity<Cita>()
    .HasOne(c => c.Especialidad)
    .WithMany(e => e.Citas)
    .HasForeignKey(c => c.EspecialidadId)
    .OnDelete(DeleteBehavior.Restrict);
```

*Ilustración 17: Muestra pastes de la lógica del AppDbContext*

### Ejecución del flujo de migraciones

Para aplicar todos los cambios al crear o actualizar la base de datos:

1. **Agregar una migración:**

   dotnet ef migrations add NombreMigracion

2. **Aplicar migraciones:**

   dotnet ef database update

3. **Verificar que las tablas y relaciones se reflejan correctamente en el motor MySQL.**


*Ilustración 18: Muestra las tablas de la base de datos.*

## Seguridad Web

La seguridad es un aspecto esencial en el desarrollo de aplicaciones web, especialmente cuando se trata de un sistema que maneja información confidencial como datos médicos de pacientes, citas clínicas y accesos diferenciados por roles (paciente, doctor y director). En este proyecto, se han implementado diversas medidas de seguridad a nivel de autenticación, autorización, protección contra ataques y cifrado de datos. Autenticación de usuarios

ASP.NET Core proporciona un sistema de autenticación robusto mediante cookies, tokens JWT o Identity. En este sistema, se ha optado por **ASP.NET Core Identity**, que permite:

- Registro y login de usuarios.
- Gestión de contraseñas seguras mediante hashing (por defecto, utiliza el algoritmo PBKDF2).
- Almacenamiento de roles y claims.
- Validación automática de credenciales.

**Ejemplo de configuración en Program.cs:**

```
builder.Services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<AppDbContext>();
```

*Ilustración 19: Muestra la lógica para trabajar los roles.*

**Autorización por roles**

La autorización garantiza que solo los usuarios autorizados accedan a ciertas funcionalidades. En este sistema, los roles definidos son:

- **Paciente:** acceso a gestión de sus citas y expediente.
- **Doctor:** acceso a su agenda y expedientes de sus pacientes.
- **Director:** acceso completo para administrar usuarios, doctores y especialidades.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

[Authorize(Roles = "Doctor")]
public class ExpedientesController : Controller
{
    public IActionResult VerExpediente(int id)
    {
        // Buscar expediente por ID y retornarlo a la vista
        var expediente = ObtenerExpedientePorId(id);
        if (expediente == null)
            return NotFound();

        return View(expediente);
    }

    private object ObtenerExpedientePorId(int id)
    {
        // Lógica de obtención desde la base de datos
        return new { Id = id, Nombre = "Paciente Ejemplo" }; // Simulado
```

*Ilustración 20: Muestra controlador ASP.NET Core que permite a usuarios con rol "Doctor" ver expedientes simulados por ID.*

**Protección contra ataques comunes**

ASP.NET Core incorpora protección integrada contra los ataques web más frecuentes:

ASP.NET Core incorpora varias medidas de seguridad para proteger las aplicaciones web contra los ataques más comunes. Para prevenir **CSRF (Cross-Site Request Forgery)**, utiliza tokens antifalsificación que se generan automáticamente y se insertan en los formularios mediante la directiva @Html.AntiForgeryToken(), garantizando que las solicitudes provengan de usuarios legítimos y no de fuentes externas maliciosas. En cuanto a la protección contra **XSS (Cross-Site Scripting)**, el motor de vistas Razor codifica automáticamente el contenido HTML generado dinámicamente, evitando que scripts maliciosos se inyecten y ejecuten en el navegador del usuario. Finalmente, para evitar **SQL Injection**, ASP.NET Core recomienda el uso de Entity Framework Core, que parametriza todas las consultas a la base de datos, lo que elimina la posibilidad de que un atacante inserte código SQL dañino a través de entradas maliciosas. Estas protecciones integradas facilitan el desarrollo seguro y robusto de aplicaciones web en ASP.NET Core.

### Cifrado y almacenamiento seguro

- Las **contraseñas** de los usuarios se almacenan cifradas y nunca en texto plano.
- Los **tokens de autenticación** se transmiten de forma segura mediante HTTPS.
- Se recomienda implementar políticas de complejidad de contraseña y expiración de sesiones.

### HTTPS y certificados SSL

Para proteger la comunicación entre el navegador y el servidor, se ha habilitado el uso obligatorio de HTTPS en la aplicación:

```
app.UseHttpsRedirection(); // 👉 Fuerza redirección a HTTPS
```

*Ilustración 21: Muestra app.UseHttpsRedirection();que redirige las solicitudes HTTP a HTTPS para proteger la comunicación.*

Además, al implementar el proyecto en un servidor real (por ejemplo, Azure, AWS o un VPS), se debe asegurar la instalación de un **certificado SSL válido**

## Autenticación y Autorización en la Aplicación Móvil (Flutter + Firebase)

La autenticación y autorización son componentes críticos en la arquitectura del sistema móvil, ya que permiten identificar a los usuarios, asignar roles y restringir el acceso a funcionalidades según el tipo de usuario (paciente, doctor o director). Para este proyecto, se utiliza **Firebase Authentication** como proveedor de autenticación centralizado, lo que simplifica la implementación y garantiza un alto nivel de seguridad.

### Autenticación con Firebase

**Firebase Authentication** permite autenticar a los usuarios mediante múltiples métodos. En esta aplicación se utiliza principalmente:

- **Correo electrónico y contraseña**: El usuario se registra y luego inicia sesión mediante sus credenciales.
- **Autenticación persistente**: El estado de sesión se mantiene incluso si la app se reinicia, hasta que el usuario cierre sesión explícitamente.

**Ejemplo en Dart (Flutter):**

```dart
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';

Qodo Gen: Options | Test this class
class LoginScreen extends StatefulWidget {
  const LoginScreen({super.key});

  @override
  LoginScreenState createState() => LoginScreenState();
}

Qodo Gen: Options | Test this class
class LoginScreenState extends State<LoginScreen> {
  final TextEditingController emailController = TextEditingController();
  final TextEditingController passwordController = TextEditingController();

  String? selectedRole;
  bool rememberUser = false;

  @override
  Qodo Gen: Options | Test this method
  void initState() {
    super.initState();
    _loadSavedUser();
  }

  Qodo Gen: Options | Test this method
  Future<void> _loadSavedUser() async {
    final prefs = await SharedPreferences.getInstance();
    final savedEmail = prefs.getString('email');
    final savedRole = prefs.getString('role');
    final remember = prefs.getBool('remember') ?? false;

    if (remember && savedEmail != null && savedRole != null) {
      setState(() {
        emailController.text = savedEmail;
        selectedRole = savedRole;
        rememberUser = true;
      });
    }
  }
```

*Ilustración 22: muestra la lógica de validación de roles al iniciar sesión.*

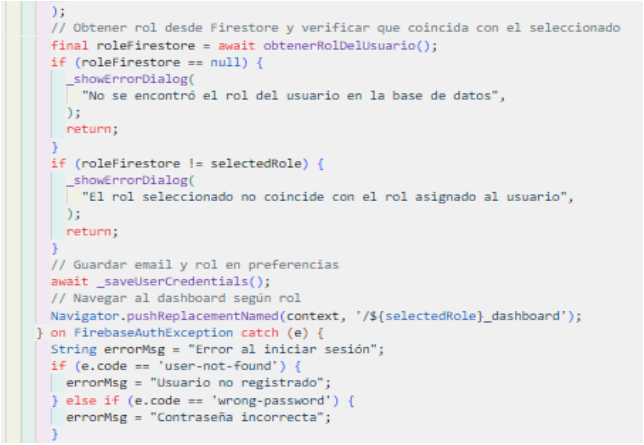
**Firebase también maneja automáticamente:**

- Recuperación de contraseñas.
- Verificación de correo electrónico.
- Control de sesión.

**Autorización basada en roles**

Después de que un usuario se autentica en el sistema, es fundamental determinar su tipo o rol específico (como paciente, doctor o director) para otorgarle los permisos y accesos adecuados dentro de la aplicación. Para lograr esto, el rol de cada usuario se almacena previamente en una base de datos, ya sea en Firebase Firestore o en Firebase Realtime Database. Cuando el usuario inicia sesión, el sistema consulta esta información para identificar su rol exacto. Con base en el rol recuperado, el usuario es redirigido automáticamente a la interfaz o panel de control que corresponde a su perfil, asegurando así que solo pueda acceder a las funcionalidades y recursos que le están permitidos. Este mecanismo de autorización basada en roles garantiza un control de acceso organizado, seguro y personalizado, mejorando la experiencia del usuario y la gestión interna de la aplicación.

**Ejemplo de consulta de rol:**



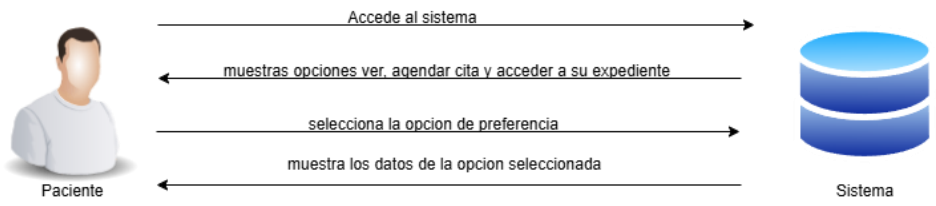
```
      );
      // Obtener rol desde Firestore y verificar que coincida con el seleccionado
      final roleFirestore = await obtenerRolDelUsuario();
      if (roleFirestore == null) {
        _showErrorDialog(
          "No se encontró el rol del usuario en la base de datos",
        );
        return;
      }
      if (roleFirestore != selectedRole) {
        _showErrorDialog(
          "El rol seleccionado no coincide con el rol asignado al usuario",
        );
        return;
      }
      // Guardar email y rol en preferencias
      await _saveUserCredentials();
      // Navegar al dashboard según rol
      Navigator.pushReplacementNamed(context, '/${selectedRole}_dashboard');
    } on FirebaseAuthException catch (e) {
      String errorMsg = "Error al iniciar sesión";
      if (e.code == 'user-not-found') {
        errorMsg = "Usuario no registrado";
      } else if (e.code == 'wrong-password') {
        errorMsg = "Contraseña incorrecta";
      }
```

```dart
Qodo Gen: Options | Test this method
Future<void> handleLogin() async {
  if (selectedRole == null) {
    _showErrorDialog("Selecciona un rol antes de continuar");
    return;
  }

  final email = emailController.text.trim();
  final password = passwordController.text.trim();

  if (email.isEmpty || password.isEmpty) {
    _showErrorDialog("Por favor, ingresa todos los campos");
    return;
  }
  try {
    // Iniciar sesión con Firebase Auth
    await FirebaseAuth.instance.signInWithEmailAndPassword(
      email: email,
      password: password,
```

*Ilustración 23: muestra la lógica de autorización de rol basada en firebase*

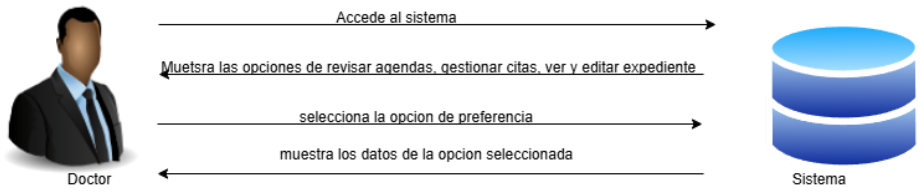
**Casos de uso: Según el valor de rol, se navega a la vista específica para:**

- **Paciente:** ver y agendar citas, acceder a su expediente.

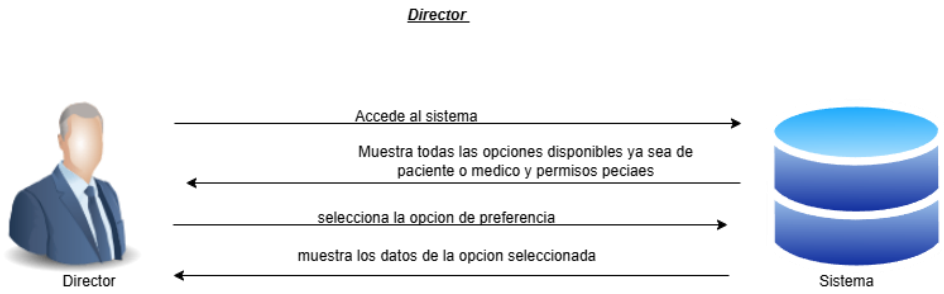



- **Doctor:** revisar agenda, gestionar citas, ver y editar expedientes.

***DOCTOR***




- **Director:** administrar doctores, horarios, especialidades, y acceder a la vista de administración.




## Seguridad adicional con Firebase Rules

Para evitar accesos no autorizados desde el frontend, se configuran **Firebase Security Rules**, las cuales restringen el acceso a colecciones o documentos según el rol y la autenticación del usuario.

## Ejemplo básico:

```
  // Iniciar sesión con Firebase Auth
  await FirebaseAuth.instance.signInWithEmailAndPassword(
    email: email,
    password: password,
  );
  // Obtener rol desde Firestore y verificar que coincida con el seleccionado
  final roleFirestore = await obtenerRolDelUsuario();
  if (roleFirestore == null) {
    _showErrorDialog(
      "No se encontró el rol del usuario en la base de datos",
    );
    return;
  }
  if (roleFirestore != selectedRole) {
    _showErrorDialog(
      "El rol seleccionado no coincide con el rol asignado al usuario",
    );
    return;
  }
  // Guardar email y rol en preferencias
  await _saveUserCredentials();
  // Navegar al dashboard según rol
  Navigator.pushReplacementNamed(context, '/${selectedRole}_dashboard');
} on FirebaseAuthException catch (e) {
  String errorMsg = "Error al iniciar sesión";
  if (e.code == 'user-not-found') {
    errorMsg = "Usuario no registrado";
  } else if (e.code == 'wrong-password') {
    errorMsg = "Contraseña incorrecta";
  }
  _showErrorDialog(errorMsg);
}
```

*Ilustración 24: muestra la lógica para evitar el acceso usuarios no autorizados*

## Manejo de sesiones y cierre de sesión

- Firebase mantiene automáticamente la sesión activa del usuario.
    - Se proporciona un botón de "Cerrar sesión" en la interfaz, que ejecuta

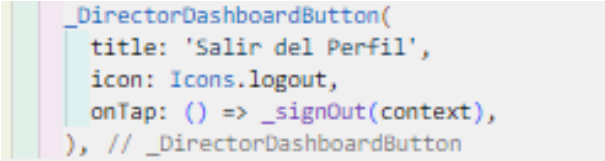



*Ilustración 25: muestra la función para cerrar el perfil del usuario*

### Control de Acceso Basado en Roles (Paciente, Doctor, Director)

En un sistema médico digital donde interactúan múltiples tipos de usuarios, es fundamental implementar un mecanismo de control de acceso robusto basado en **roles**. Este mecanismo asegura que cada usuario pueda acceder únicamente a las funcionalidades que le corresponden según su perfil, protegiendo así la confidencialidad de los datos y la integridad de las operaciones.

## Control de Acceso en la Aplicación Web (ASP.NET Core MVC)

Gestión de Roles con Identity

ASP.NET Core Identity permite crear y asignar roles fácilmente a los usuarios autenticados. En este sistema, los roles se definen al momento del registro o mediante herramientas de administración disponibles para el director.

```
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (!ModelState.IsValid)
        return View(model);

    var user = new IdentityUser { UserName = model.Email, Email = model.Email };
    var result = await _userManager.CreateAsync(user, model.Password);

    if (result.Succeeded)
    {
        // Crear rol si no existe
        if (!await _roleManager.RoleExistsAsync(model.Role))
        {
            await _roleManager.CreateAsync(new IdentityRole(model.Role));
        }

        // Asignar rol
        await _userManager.AddToRoleAsync(user, model.Role);

        // Iniciar sesión
        await _signInManager.SignInAsync(user, isPersistent: false);

        // Redirigir al dashboard correspondiente
        return RedirectToAction("Dashboard", model.Role);
    }

    foreach (var error in result.Errors)
        ModelState.AddModelError("", error.Description);

    return View(model);
}
```

*Ilustración 26: muestra la lógica de creación y asignación de roles específicos*

### Autorización por Roles

Las vistas y controladores están protegidos con el atributo [Authorize], el cual restringe el acceso según el rol del usuario.

### Ejemplo:

```
0 referencias | Qodo Gen: Options | Test this class
public class DirectorController : Controller
{
    Qodo Gen: Options | Test this method
    [Authorize(Roles = "Director")]
    0 referencias
    public IActionResult GestionarDoctores()
    {
        return View();
    }
}

0 referencias | Qodo Gen: Options | Test this class
public class DoctorController : Controller
{
    Qodo Gen: Options | Test this method
    [Authorize(Roles = "Doctor")]
    0 referencias
    public IActionResult VerAgenda()
    {
        return View();
    }
}

0 referencias | Qodo Gen: Options | Test this class
public class PacienteController : Controller
{
    Qodo Gen: Options | Test this method
    [Authorize(Roles = "Paciente")]
    0 referencias
    public IActionResult VerCitas()
    {
        return View();
    }
}
Qodo Gen: Options | Test this method
[Authorize]
```

*Ilustración 27: muestra la lógica de autorización de roles*

### Interfaz Personalizada por Rol

Cada usuario ve una interfaz diferente según su rol. Esto se controla mediante condiciones en las vistas Razor:

```
@if (User.Identity.IsAuthenticated)
{
    @* Mostrar enlaces según rol *@

    @if (User.IsInRole("Paciente"))
    {
        <li><a asp-controller="Home" asp-action="AgendarCita">Agendar Cita</a></li>
    }

    @if (User.IsInRole("Doctor"))
    {
        <li><a asp-controller="Home" asp-action="VerAgenda">Ver Agenda</a></li>
    }

    @if (User.IsInRole("Director"))
    {
        <li><a asp-controller="Home" asp-action="GestionarDoctores">Gestionar Doctores</a></li>
    }

    <li><form asp-controller="Account" asp-action="Logout" method="post" id="logoutForm">
        <button type="submit">Cerrar sesión</button>
    </form>/#logoutForm</li>
}
else
{
    <li><a asp-controller="Account" asp-action="Login">Iniciar sesión</a></li>
    <li><a asp-controller="Account" asp-action="Register">Registrarse</a></li>
}
```

*Ilustración 28: muestra la implementación de menú, según la asignación de roles*

Control de Acceso en la Aplicación Móvil (Flutter + Firebase) Almacenamiento de Roles en Firestore.

Después del registro, a cada usuario se le asigna un rol en una colección de usuarios:

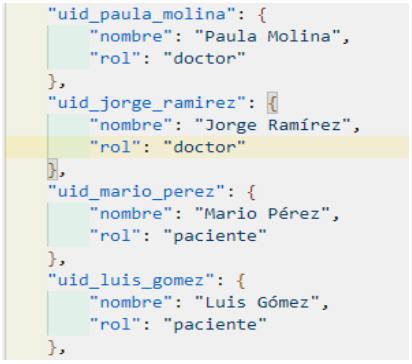



*Ilustración 29: muestra el registro de los usuarios registrados*

Restricción de Funcionalidades en Flutter.

Al iniciar sesión, la app consulta el rol del usuario y lo redirige a la interfaz adecuada:

```
Qodo Gen: Options | Test this method
Widget _buildLoginForm() {
  return Card(
    elevation: 8,
    shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(16)),
    margin: const EdgeInsets.symmetric(horizontal: 24),
    child: Padding(
      padding: const EdgeInsets.all(24.0),
      child: Column(
        mainAxisSize: MainAxisSize.min,
        children: [
          Text(
            'Iniciar sesión como ${selectedRole![0].toUpperCase()}${selectedRole!.substring(1)}',
            style: const TextStyle( // TextStyle ···
          ), // Text
          const SizedBox(height: 20),
          TextField( // TextField ···
          const SizedBox(height: 16),
          TextField( // TextField ···
          const SizedBox(height: 10),
          Row( // Row ···
          const SizedBox(height: 10),
          ElevatedButton( // ElevatedButton ···
          TextButton(
            onPressed:
                () => setState(() {
                  selectedRole = null;
                  emailController.clear();
                  passwordController.clear();
                  rememberUser = false;
                }),
            child: const Text('Cambiar rol'),
          ), // TextButton
          TextButton(
            onPressed: () {
              Navigator.pushNamed(context, '/register_${selectedRole ?? ""}');
            },
            child: const Text('¿No tienes cuenta? Crea una aquí'),
          ), // TextButton
        ],
```

*Ilustración 30: muestra la lógica para consultar datos y redireccionamiento de perfil al usuario asignado.*

## Protección con Reglas de Seguridad en Firebase

Se aplican reglas que restringen el acceso a los datos desde el backend:

```
service cloud.firestore {
  match /databases/{database}/documents {

    match /usuarios/{userId} {
      allow read, write: if request.auth != null && request.auth.uid == userId;
    }

    match /citas/{citaId} {
      allow read, write: if request.auth != null &&
        get(/databases/$(database)/documents/usuarios/$(request.auth.uid)).data.rol in ['doctor', 'director', 'paciente'];
    }
  }
```

*Ilustración 31: muestra la implementación de reglas de acceso*

Resumen de Permisos por Rol

| Funcionalidad | Paciente | Doctor | Director |
|---|---|---|---|
| Agendar y cancelar citas | ✅ | ✅ | ✅ |
| Ver expediente propio | ✅ | ✅ | ✅ |
| Ver y modificar expedientes | 🚫 | ✅ | ✅ |
| Gestionar citas | 🚫 | ✅ | ✅ |
| Administrar doctores | 🚫 | 🚫 | ✅ |
| Asignar especialidades | 🚫 | 🚫 | ✅ |
| Configurar horarios | 🚫 | ✅ | ✅ |
| Acceso completo | 🚫 | 🚫 | ✅ |

Tabla 2: Permisos por Rol

### Desarrollo del Módulo Móvil (Flutter)

El módulo móvil fue desarrollado utilizando **Flutter**, un framework multiplataforma de código abierto creado por Google, que permite compilar una única base de código en aplicaciones nativas para Android e iOS. En este proyecto, la aplicación móvil está orientada principalmente a pacientes, doctores y el director médico, cada uno con permisos específicos según su rol.

### Arquitectura y Patrón MVVM

Se utilizó el patrón **MVVM (Model-View-ViewModel)** para separar la lógica de negocios de la interfaz de usuario, facilitando así el mantenimiento del código, la escalabilidad y la reutilización de componentes:

- **Model**: Define la estructura de los datos (como citas, usuarios, expediente).
- **View**: Interfaz gráfica que el usuario ve e interactúa.
- **ViewModel**: Gestiona la lógica de negocios y se comunica con Firebase y SQLite.

## Funcionalidades Implementadas

### Rol Paciente

- Registro y autenticación vía Firebase.
- Agendar, modificar y cancelar citas médicas.
- Ver historial de citas y expediente clínico resumido.
- Chat con el doctor asignado.
- Recibir notificaciones cuando el doctor modifica una cita.

### Rol Doctor

- Ver agenda diaria de citas.
- Consultar y modificar el expediente completo de sus pacientes.
- Cambiar fecha y hora de una cita si es necesario.
- Comunicarse con pacientes a través del sistema de mensajería.

### Rol Director

- Gestionar el alta, edición y eliminación de doctores.
- Asignar especialidades y horarios de trabajo.
- Supervisar toda la plataforma desde el móvil (funcionalidad administrativa).

## Flujo de Datos

### Local y en la Nube

En la aplicación, Firebase se utiliza como la principal plataforma para almacenar y sincronizar la mayoría de los datos en la nube, asegurando que la información esté siempre actualizada y accesible desde cualquier dispositivo. Para mejorar la experiencia del usuario y permitir el funcionamiento incluso sin conexión a internet, se emplea Sqflite para almacenar datos en caché localmente en el dispositivo. Esto posibilita que ciertas funcionalidades sigan operando de manera offline. Una vez que el dispositivo recupera la conexión, todos los cambios realizados localmente se sincronizan automáticamente con Firestore, manteniendo la coherencia y actualización de los datos entre el almacenamiento local y la nube.

### Sincronización

Al iniciar sesión, la aplicación descarga los datos del usuario desde la nube y los guarda localmente en el dispositivo para un acceso rápido y eficiente. Cuando el usuario realiza modificaciones mientras está offline, estos cambios se almacenan temporalmente en SQLite, garantizando que la información no se pierda. Una vez que el dispositivo recupera la conexión a internet, la aplicación sincroniza automáticamente todas las modificaciones almacenadas localmente con la base de datos en la nube, asegurando que los datos estén actualizados y consistentes en ambos entornos.

### Comunicación con el Backend

Para operaciones específicas como agendar o cancelar citas, la aplicación se comunica con una API REST desarrollada en ASP.NET Core, lo que garantiza la integridad y consistencia de los datos al interactuar directamente con la base de datos MySQL utilizada por el sistema web. Paralelamente, Firebase funciona como un backend complementario, encargándose de la autenticación de usuarios, el

envío de notificaciones y el almacenamiento de datos en tiempo real, ofreciendo así una experiencia más ágil y sincronizada para los usuarios.

**Seguridad y Control de Acceso**

La autenticación de usuarios se realiza mediante Firebase Auth utilizando el método de Email y Contraseña, garantizando un acceso seguro y sencillo. Al momento de iniciar sesión, se valida el rol del usuario para determinar sus permisos y el tipo de interfaz que debe mostrar. Además, se implementan reglas de seguridad en Firestore que restringen el acceso no autorizado a los datos, protegiendo la información sensible. Finalmente, el sistema controla la navegación según el rol, asegurando que cada tipo de usuario —ya sea paciente, doctor o director— acceda únicamente a las pantallas y funcionalidades que le corresponden.

## Tecnologías Utilizadas

### Lenguaje: Dart


Dart es el lenguaje de programación utilizado para desarrollar la aplicación móvil con Flutter. Creado por Google, Dart está diseñado para ser eficiente, moderno y fácil de usar, siendo el lenguaje nativo de Flutter, lo que garantiza una integración perfecta con todos sus componentes. Entre sus características principales, Dart es un lenguaje orientado a objetos que facilita la reutilización del código y una organización modular clara. Su sintaxis es limpia y moderna, similar a lenguajes como JavaScript o Java, lo que reduce la curva de aprendizaje para los desarrolladores. Además, ofrece soporte completo para programación asíncrona mediante palabras clave como async, await y Future, lo que facilita la gestión de operaciones como llamadas a APIs o acceso a bases de datos. Dart utiliza compilación Just-In-Time (JIT) durante el desarrollo para permitir recargas rápidas (hot reload) que aceleran la iteración, y Ahead-Of-Time (AOT) en producción para optimizar el rendimiento de la aplicación. También cuenta con un recolector de basura eficiente que gestiona la memoria automáticamente, optimizando el uso de recursos.

En el contexto del proyecto, Dart presenta varias ventajas importantes. Facilita el desarrollo rápido de interfaces reactivas con Flutter, mejorando la experiencia de usuario y la productividad del equipo. Además, permite mantener una sola base de código que funciona en múltiples plataformas móviles, como Android e iOS, reduciendo costos y tiempos de desarrollo. Finalmente, Dart ofrece una excelente integración con herramientas y servicios clave como Firebase para autenticación y backend, así como con plugins como sqflite para almacenamiento local, brindando un entorno completo y eficiente para crear aplicaciones móviles modernas y robustas.

***Framework: Flutter***


Flutter es un framework de desarrollo multiplataforma que permite crear aplicaciones para Android e iOS a partir de una única base de código, lo que reduce significativamente los costos y tiempos de desarrollo. Una de sus características más destacadas es el **Hot Reload**, que permite ver los cambios en la interfaz de usuario de forma inmediata, acelerando el ciclo de desarrollo y facilitando la experimentación. Flutter ofrece un amplio catálogo de widgets personalizables, que facilitan la construcción de interfaces modernas, responsivas y atractivas. Además, cuenta con un motor gráfico propio que no depende de componentes nativos para renderizar la interfaz, garantizando un aspecto uniforme en todas las plataformas. Gracias a su compilación Ahead-Of-Time (AOT), Flutter proporciona un alto rendimiento y una experiencia fluida para el usuario final.

En el contexto del proyecto, Flutter presenta múltiples ventajas. Permite acelerar el desarrollo de interfaces adaptadas a distintos perfiles de usuario, como pacientes, doctores y directores, facilitando la creación de experiencias personalizadas. También facilita la integración con Firebase como backend en la nube, aprovechando servicios como autenticación, base de datos en tiempo real y notificaciones. Además, mejora la experiencia del usuario final con interfaces dinámicas, responsivas y altamente personalizables. Finalmente, Flutter soporta el uso de paquetes adicionales importantes para el proyecto, como firebase_core, firebase_auth, cloud_firestore y sqflite, brindando un ecosistema robusto para el desarrollo de aplicaciones móviles completas.

***IDE: Visual Studio Code***

Visual Studio Code (VS Code) es el entorno de desarrollo integrado (IDE) utilizado para la creación de la aplicación móvil en Flutter. Es una herramienta ligera, rápida y altamente extensible, desarrollada por Microsoft, que se ha convertido en uno de los entornos más populares para el desarrollo de aplicaciones multiplataforma.

**Características de Visual Studio Code:**

Visual Studio Code es un editor de código ligero y rápido que consume pocos recursos del sistema, lo que lo hace ideal para trabajar en equipos con especificaciones modestas. Cuenta con extensiones especializadas para Flutter y Dart que proporcionan funcionalidades como autocompletado, depuración, testing y análisis de código. Además, incluye una terminal integrada que permite ejecutar comandos directamente sin salir del entorno de desarrollo, soporte para control de versiones mediante integración con Git, y una interfaz altamente personalizable que puede adaptarse a las preferencias del usuario mediante temas, atajos de teclado y configuraciones específicas.

**Ventajas en el contexto del proyecto:**

En el contexto del proyecto, Visual Studio Code ofrece un soporte completo para Flutter y todas sus herramientas asociadas, lo que facilita el desarrollo eficiente de la aplicación móvil. Su depurador y consola integrada permiten identificar y corregir errores rápidamente, mejorando la calidad del código. Asimismo, facilita una integración fluida con Firebase y otras dependencias importantes gracias a sus terminales y extensiones, lo que contribuye a un entorno de trabajo más robusto y productivo para el equipo de desarrollo.

## Base de Datos Local: Sqflite

Para la persistencia de datos local en la aplicación móvil, se ha utilizado **Sqflite**, un plugin de Flutter que permite utilizar una base de datos **SQLite** de forma local. Esta herramienta es fundamental cuando se requiere que ciertos datos estén disponibles sin conexión a internet, mejorando la experiencia del usuario en entornos de conectividad limitada.


**Características de Sqflite:**

Sqflite es un paquete basado en SQLite que utiliza un motor de base de datos ligero y ampliamente reconocido por su eficiencia y confiabilidad. Entre sus características principales se encuentra el soporte completo para operaciones CRUD (crear, leer, actualizar y eliminar), lo que permite una gestión integral de los datos. Además, ofrece la posibilidad de realizar consultas personalizadas mediante sentencias SQL tradicionales, brindando flexibilidad en el manejo de la información. Sqflite también garantiza la persistencia de datos en modo offline, asegurando que la información esté disponible incluso cuando no hay conexión a la red. Por último, es compatible con múltiples plataformas, funcionando tanto en dispositivos Android como iOS, lo que lo convierte en una opción ideal para aplicaciones móviles multiplataforma.

**Ventajas en el contexto del proyecto:**

En el contexto del proyecto, Sqflite ofrece varias ventajas importantes. Al almacenar datos localmente, mejora la eficiencia de la aplicación al reducir la cantidad de llamadas innecesarias a servicios en línea, lo que también contribuye a disminuir el consumo de datos y la dependencia de la conexión a internet. Esta capacidad de persistencia offline brinda a los usuarios una experiencia más fluida y confiable, permitiéndoles acceder y modificar información incluso sin estar conectados. Además, Sqflite proporciona un acceso rápido y directo a datos esenciales almacenados localmente, lo que optimiza el rendimiento general de la aplicación y mejora la satisfacción del usuario.

## Backend en la Nube: Firebase

**Firebase** es la plataforma de desarrollo de aplicaciones móviles y web proporcionada por Google, y se ha utilizado como backend en la nube para la aplicación móvil del proyecto. Firebase ofrece una amplia gama de servicios que simplifican la gestión del backend, permitiendo que los desarrolladores se concentren en la lógica del negocio y la experiencia del usuario.


**Servicios de Firebase utilizados en el proyecto:**

En el proyecto se utilizan varios servicios clave de Firebase para garantizar una gestión eficiente y segura de los datos y usuarios. Firebase Authentication se encarga de manejar el registro e inicio de sesión de los distintos tipos de usuarios, como pacientes, doctores y directores, proporcionando un sistema seguro y confiable para la autenticación. Por otro lado, Cloud Firestore funciona como la base de datos NoSQL en tiempo real, almacenando información vital como expedientes médicos, conversaciones de chat y la programación de citas. Gracias a esta combinación, el proyecto puede ofrecer una experiencia fluida, segura y sincronizada en tiempo real para todos los usuarios involucrados.

**Ventajas de Firebase en el contexto del proyecto:**

Firebase aporta numerosas ventajas que potencian el rendimiento y la confiabilidad de la aplicación. Su escalabilidad automática permite que la plataforma se adapte sin esfuerzo a un número creciente de usuarios, sin necesidad de configurar manualmente los servidores, lo que facilita el crecimiento del sistema. La capacidad de sincronización en tiempo real garantiza que los datos, como mensajes de chat y actualizaciones de citas, se actualicen instantáneamente entre usuarios, mejorando la comunicación y la coordinación. Además, al estar alojado en la nube de Google, Firebase ofrece alta disponibilidad y un servicio estable, asegurando que la aplicación esté siempre accesible. Finalmente, Firebase proporciona reglas de

seguridad personalizadas que protegen la información médica sensible, garantizando la privacidad y cumplimiento de normativas de seguridad.

**Uso estratégico en el proyecto:**

Firebase actúa como el intermediario entre el frontend móvil desarrollado en Flutter y la lógica del negocio, ofreciendo funcionalidades críticas como autenticación, almacenamiento seguro, mensajería instantánea y gestión de datos estructurados y no estructurados en la nube.

Firestore (Base de Datos en la Nube)

Para la gestión de datos en la nube en la aplicación móvil, se ha utilizado **Cloud Firestore**, un servicio de base de datos NoSQL proporcionado por Firebase. Firestore es una base de datos flexible, escalable y en tiempo real, ideal para aplicaciones móviles modernas como la desarrollada en este proyecto.

**Características principales de Firestore:**

Firestore es una base de datos NoSQL que organiza la información mediante una estructura basada en documentos y colecciones, donde las colecciones funcionan como tablas y los documentos como filas, lo que facilita el almacenamiento tanto de datos estructurados como semiestructurados. Una de sus principales características es la sincronización en tiempo real, que permite que cualquier cambio en la base de datos se refleje automáticamente en la interfaz del usuario sin necesidad de recargar la aplicación. Además, Firestore ofrece alta disponibilidad al ser un servicio completamente administrado y alojado en la nube de Google, garantizando estabilidad y confiabilidad. También soporta consultas complejas, incluyendo filtros, ordenamientos, paginación y combinaciones de condiciones, lo que brinda flexibilidad para acceder y manipular los datos de manera eficiente.

**Uso de Firestore en el proyecto:**

- **Gestión de citas médicas**: Creación, modificación y cancelación de citas en tiempo real.
- **Almacenamiento de expedientes médicos**: Documentos clínicos asociados a cada paciente, accesibles solo por el usuario autorizado.

- **Sistema de chat en tiempo real**: Comunicación directa entre pacientes y doctores, con mensajes almacenados en colecciones específicas por conversación.
- **Control de usuarios y roles**: Información sobre cada usuario autenticado, incluyendo su rol (paciente, doctor, director) y configuración de perfil.

**Ejemplo de estructura de datos:**

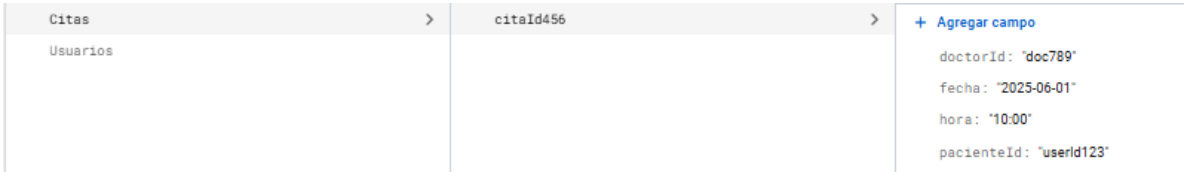



*Ilustración 32: visualización de la estructura de datos registrados para agendar citas*

**Firebase Cloud Messaging (Notificaciones Push)**

**Firebase Cloud Messaging (FCM)** es el servicio utilizado en la aplicación móvil para el envío de notificaciones push a los usuarios. Este componente mejora significativamente la interacción entre el sistema y sus usuarios al mantenerlos informados en tiempo real sobre eventos relevantes, como la confirmación, reprogramación o cancelación de citas médicas.

**Características de FCM:**

Firebase Cloud Messaging (FCM) permite el envío dirigido de mensajes a dispositivos individuales, grupos de usuarios o suscriptores a temas específicos, garantizando que las notificaciones lleguen al público adecuado. Las notificaciones pueden mostrarse tanto en primer plano como en segundo plano, siendo visibles incluso cuando la aplicación está cerrada, lo que asegura que los usuarios estén siempre informados. Respaldado por la infraestructura de Google, FCM ofrece una alta tasa de entrega y eficiencia en el envío de mensajes. Además, su integración con otros servicios de Firebase como Authentication y Firestore facilita su implementación y coordinación con el resto del sistema.

En el proyecto, FCM se utiliza para varios casos de uso clave que mejoran la comunicación con los usuarios. Por ejemplo, cuando un paciente programa una cita, recibe una notificación con los detalles relevantes como fecha, hora y doctor asignado. Si el doctor reprograma una cita, el paciente es notificado al instante, y en caso de cancelación de citas, ya sea por parte del paciente o del personal médico, se envían alertas automáticas para mantener a todos informados. También

se emplea para enviar mensajes importantes del sistema o del director, tales como recordatorios de citas o comunicaciones urgentes, asegurando que la información crítica llegue oportunamente a los usuarios.

**Uso de FCM en el proyecto:**

- **Confirmación de citas**: Cuando un paciente programa una cita, recibe una notificación con los detalles (fecha, hora y doctor asignado).
- **Reprogramación por parte del doctor**: Si el doctor cambia la cita, el paciente es notificado al instante.
- **Cancelación de citas**: Notificación automática cuando una cita es cancelada, ya sea por el paciente o por el personal médico.
- **Mensajes importantes del sistema o del director**: Alertas administrativas, como recordatorios de cita o mensajes urgentes.

**Implementación técnica:**

La implementación técnica de Firebase Cloud Messaging (FCM) en Flutter se realiza principalmente mediante los paquetes firebase_messaging y flutter_local_notifications. Estos paquetes permiten gestionar tanto la recepción como la visualización de notificaciones en diferentes estados de la aplicación, ya sea en primer plano, segundo plano o cuando la app está cerrada. Cada dispositivo registra un token único con FCM, el cual se asocia al usuario autenticado para dirigir las notificaciones de manera personalizada. Para enviar las notificaciones, se puede utilizar directamente Firestore o implementar funciones en la nube (Cloud Functions) que reaccionen a eventos específicos, como la creación de un nuevo documento relacionado con una cita, desencadenando así notificaciones programadas o automáticas para mantener a los usuarios informados en tiempo real.

Ventajas de usar FCM:

- **Gratuito y altamente escalable**.
- **Bajo consumo de batería y datos**.
- **Personalización del contenido y acciones de las notificaciones**.
- **Compatibilidad multiplataforma** (Android, iOS y Web).

## Uso de Servicios REST para Integración con el Sistema Web

La arquitectura del proyecto permite la integración entre los distintos módulos del sistema (web y móvil) mediante **servicios web RESTful**, lo que facilita la interoperabilidad, la escalabilidad y la independencia de plataformas.

## ¿Qué es una API REST?


Una API REST (Representational State Transfer) es una interfaz que permite la comunicación entre sistemas utilizando los métodos estándar del protocolo HTTP (GET, POST, PUT, DELETE). Las APIs REST son ampliamente utilizadas por su simplicidad, eficiencia y compatibilidad multiplataforma.

Implementación de Servicios REST en el Proyecto Web

El sistema web, desarrollado con **ASP.NET Core MVC**, puede actuar como proveedor de servicios REST a través de controladores tipo ApiController, exponiendo datos almacenados en MySQL mediante endpoints que pueden ser consumidos por la aplicación móvil u otros sistemas externos.

Ejemplos de Endpoints REST Web:

- GET /api/doctores → Lista todos los doctores registrados.
- GET /api/especialidades → Muestra todas las especialidades médicas disponibles.
- POST /api/citas → Permite registrar una nueva cita médica.
- PUT /api/citas/{id} → Actualiza una cita médica.
- DELETE /api/citas/{id} → Elimina una cita específica.

Estos servicios hacen uso del ORM **Entity Framework Core** para interactuar con la base de datos MySQL, asegurando un acceso estructurado y seguro a los datos.

Integración con la Aplicación Móvil

La app móvil desarrollada en **Flutter** puede consumir estos servicios REST mediante librerías como http, dio o integraciones con Retrofit para Dart, permitiendo que los módulos web y móvil compartan información cuando sea necesario.

Uso de integración:

- Sincronización de información de doctores y especialidades en la app móvil.
- Visualización de citas programadas creadas desde la plataforma web.
- Administración centralizada de usuarios y control de acceso desde el BackOffice web.

**Funcionalidades por Rol: Paciente**

El rol de **Paciente** está diseñado para brindar a los usuarios acceso sencillo, seguro y eficiente a los servicios médicos ofrecidos a través de la plataforma. Las funcionalidades disponibles para este perfil están orientadas a mejorar la experiencia del usuario, permitiendo una gestión autónoma de sus citas y la consulta de su información médica.

**Registro/Login (Firebase Authentication).**

Firebase Authentication permite que los pacientes se registren y autentiquen de manera sencilla y segura utilizando correo electrónico y contraseña. Además, ofrece la posibilidad de ampliar los métodos de autenticación incluyendo proveedores externos como Google o Facebook, facilitando el acceso a la aplicación mediante cuentas ya existentes. Este servicio no solo garantiza una gestión eficiente de las sesiones de usuario, sino que también proporciona una capa robusta de seguridad, siendo escalable para manejar desde pocos hasta miles de usuarios sin comprometer el rendimiento ni la protección de la información.

**Solicitud y Cancelación de Citas**

En el sistema, el paciente tiene la capacidad de solicitar y cancelar citas de manera sencilla desde su dispositivo móvil. Primero, puede navegar por las diferentes

especialidades médicas disponibles y, al seleccionar una, se despliega una lista de doctores junto con sus horarios disponibles. El paciente entonces elige la fecha y hora que mejor le convenga para crear la cita, la cual se almacena de forma inmediata en Firestore, garantizando que la información se sincronice en tiempo real con el sistema central. Asimismo, el paciente puede cancelar una cita directamente desde la aplicación, y dicha acción se registra y actualiza al instante, manteniendo la base de datos actualizada y permitiendo una gestión eficiente y transparente de las agendas médicas.

**Visualización de Expediente Médico**

La aplicación móvil permite al paciente acceder a una visualización resumida de su expediente médico de manera rápida y segura. Este expediente incluye información general del paciente, un historial detallado de citas anteriores, así como diagnósticos o notas médicas registradas por los doctores. Los datos mostrados se sincronizan constantemente desde Firestore o, alternativamente, desde una API que conecta con la base de datos del sistema web, garantizando que la información esté siempre actualizada y disponible para el paciente en tiempo real. Esta funcionalidad facilita el seguimiento de su salud y mejora la comunicación entre paciente y personal médico.

**Cambio de Fecha/Hora de Cita**

El sistema permite al paciente modificar una cita previamente programada cuando surgen imprevistos, brindándole flexibilidad para elegir una nueva fecha y hora disponible según el calendario actualizado del doctor. Antes de confirmar el cambio, el sistema realiza una validación para asegurar que el nuevo horario esté libre y no genere conflictos en la agenda médica. Una vez confirmado el ajuste, el doctor recibe una notificación automática mediante Firebase Cloud Messaging, garantizando que esté informado en tiempo real sobre cualquier modificación en su agenda, lo que mejora la coordinación y comunicación entre pacientes y profesionales de salud.

### Funcionalidades por Rol: Doctor

El rol de **Doctor** está orientado a proporcionar herramientas que faciliten la gestión de sus pacientes y citas, permitiendo un control más eficiente y personalizado del proceso de atención médica. Este perfil cuenta con acceso extendido a la información clínica y funcionalidad para mantener actualizado el expediente de sus pacientes.

### Gestión de Citas Programadas

El sistema ofrece al doctor la capacidad de gestionar sus citas programadas de manera eficiente desde su panel en la aplicación móvil. Puede consultar todas las citas agendadas organizadas por fecha y hora, lo que facilita la planificación de su jornada. Además, tiene acceso a información relevante como los datos del paciente, el motivo de consulta cuando ha sido ingresado, y la especialidad asignada a cada cita. Esta funcionalidad permite al doctor organizar su agenda diaria de forma clara y anticipada, optimizando su tiempo y mejorando la preparación para las atenciones médicas.

### Edición de Expediente Clínico

El doctor cuenta con acceso completo al expediente clínico de cada paciente que tiene asignado, lo que le permite gestionar de manera integral la información médica. Desde la aplicación, puede añadir, modificar o eliminar datos clínicos importantes, tales como diagnósticos, observaciones médicas y tratamientos indicados, asegurando que el historial médico esté siempre actualizado y refleje el estado real del paciente. Toda esta información se almacena en Firebase Firestore o se sincroniza con el sistema web a través de una API REST, garantizando la persistencia de los datos y su actualización en tiempo real para que tanto el personal médico como el paciente tengan acceso a información precisa y oportuna.

**Notificación al Paciente en Caso de Cambio**

Cuando el doctor necesita modificar una cita por motivos de disponibilidad o emergencias, tiene la opción de cambiar la fecha y/o la hora programada. Al realizar esta modificación, el sistema actualiza inmediatamente el estado de la cita en Firestore y/o en la base de datos MySQL, asegurando que la información esté sincronizada en todas las plataformas. Además, se envía automáticamente una notificación al paciente a través de Firebase Cloud Messaging (FCM), informándole del nuevo horario para mantenerlo al tanto de los cambios. Finalmente, la modificación queda registrada en el historial del sistema, facilitando el control administrativo y el seguimiento de todas las alteraciones realizadas sobre las citas.

**Administración de Doctores**

El director cuenta con un panel de control administrativo que le permite gestionar de manera eficiente el personal médico del sistema. Desde esta interfaz, puede agregar nuevos doctores registrando su información básica como nombre, correo electrónico, credenciales y otros datos relevantes para su identificación y contacto. Además, tiene la capacidad de consultar el listado completo de doctores registrados, lo que facilita la supervisión y organización del equipo médico. En caso de actualizaciones o cambios en la información de los doctores, el director puede modificar los datos directamente desde el panel, asegurando que el sistema siempre mantenga información precisa y actualizada sobre el personal.

**Asignación de Especialidades y Horarios**

Al crear o editar el perfil de un doctor, el director tiene la capacidad de asignarle una o varias especialidades médicas acorde a su formación y certificaciones, asegurando que cada profesional esté correctamente categorizado dentro del sistema. Además, puede establecer y configurar el horario de trabajo semanal para cada médico, definiendo sus disponibilidades para la atención a pacientes. También es posible gestionar rotaciones o realizar cambios en los horarios según las necesidades del centro médico, brindando flexibilidad en la organización del

personal. Toda esta información queda registrada y sincronizada, siendo accesible tanto desde el sistema web como desde la aplicación móvil, lo que facilita su uso efectivo en el proceso de agendamiento de citas.

**Eliminación de Personal Médico**

- En caso de retiro, licencia prolongada o baja del personal, el director puede:
- **Eliminar doctores** del sistema, deshabilitando su acceso y disponibilidad en la plataforma.
- Esta acción también actualiza automáticamente los horarios y disponibilidad visibles para los pacientes al momento de agendar citas.

**Persistencia de Datos en el Módulo Web (ASP.NET Core MVC)**

**Entity Framework Core**

Entity Framework Core es un ORM (Object-Relational Mapping) que se utiliza para gestionar de manera eficiente las operaciones con la base de datos relacional MySQL en el proyecto. Gracias a esta herramienta, las clases definidas en C# se mapean automáticamente a las tablas correspondientes en la base de datos, lo que simplifica significativamente la manipulación y consulta de datos. Además, EF Core permite trabajar con LINQ, proporcionando una forma intuitiva y potente de realizar consultas, actualizaciones y otras operaciones sobre la base de datos sin necesidad de escribir código SQL directamente.

**MySQL como motor de base de datos**

La base de datos relacional utilizada en el proyecto almacena información clave como los datos de usuarios (doctores, pacientes y director), citas médicas, especialidades y los historiales médicos que se sincronizan desde la aplicación móvil cuando corresponde. Esta estructura organizada permite gestionar eficientemente grandes volúmenes de datos relacionados, garantizando la coherencia y el acceso rápido a la información necesaria. Entre sus principales ventajas destacan la escalabilidad, que facilita el crecimiento del sistema sin perder

rendimiento; la fiabilidad, que asegura la integridad y disponibilidad de los datos; y el soporte para integridad referencial, que mantiene la consistencia entre las diferentes tablas y relaciones dentro de la base de datos.

**Contexto AppDbContext**

- Clase que actúa como punto de conexión entre la aplicación y la base de datos.
- Configurado para manejar las entidades del sistema y sus relaciones (citas, doctores, especialidades, etc.).
- Utiliza migraciones para crear y actualizar el esquema de la base de datos automáticamente.

**Persistencia de Datos en el Módulo Móvil (Flutter + Firebase)**

**Firebase Firestore (Base de datos en la nube)**

Firebase Firestore es una base de datos en la nube utilizada para almacenar información dinámica y sincronizada del sistema, como las citas agendadas por los pacientes, los cambios realizados en los expedientes médicos por los doctores, y los datos actualizados del perfil de usuario. Esta base de datos NoSQL destaca por ofrecer sincronización en tiempo real, lo que permite que cualquier modificación se refleje instantáneamente en los dispositivos móviles de los usuarios. Además, cuenta con escalabilidad automática, facilitando que la aplicación pueda crecer y manejar un número creciente de usuarios sin necesidad de ajustes manuales en la infraestructura, garantizando así un servicio fluido y confiable.

**sqflite (Base de datos local en Flutter)**

Utilizada en Flutter para el almacenamiento temporal de datos cuando el dispositivo no cuenta con conexión a Internet. Esta base de datos permite guardar citas y mantener copias de los expedientes médicos directamente en el dispositivo, asegurando que la aplicación continúe funcionando sin interrupciones en modo offline. Una vez que la conectividad se restablece, Sqflite sincroniza

automáticamente la información almacenada localmente con Firebase Firestore, garantizando que todos los datos estén actualizados y consistentes tanto en el dispositivo como en la nube.

### Sincronización

- Se implementa lógica en Flutter para sincronizar datos entre:
    - sqflite (local).
    - Firestore (nube).
    - API REST del sistema web (cuando se requiere compartir datos).

## Firebase Firestore para Almacenamiento en la Nube

**Firebase Firestore** es el servicio de base de datos NoSQL en tiempo real ofrecido por Google como parte de la plataforma Firebase. En este proyecto, se utiliza Firestore como la principal solución de almacenamiento en la nube para la aplicación móvil desarrollada en Flutter.

### Características Clave

- **Modelo de documentos y colecciones**: Firestore organiza los datos en documentos que se agrupan dentro de colecciones, permitiendo una estructura flexible y escalable.
- **Sincronización en tiempo real**: Cualquier cambio en los datos se refleja automáticamente en los dispositivos conectados, lo que permite una experiencia de usuario fluida e interactiva.
- **Escalabilidad automática**: Firestore ajusta automáticamente la capacidad según la demanda, ideal para aplicaciones con crecimiento de usuarios.
- **Alta disponibilidad y replicación**: Los datos se replican en múltiples centros de datos para garantizar redundancia y confiabilidad.

**Uso en el Proyecto**

En el contexto de esta aplicación, Firestore se emplea para almacenar y gestionar la información relevante a cada uno de los roles del sistema:

**Paciente**

- Citas médicas registradas.
- Resumen de su expediente clínico.
- Mensajes enviados al doctor.

**Doctor**

- Lista de citas programadas.
- Información médica editable de cada paciente.
- Historial de interacciones con pacientes.

**Director**

- Datos de los doctores agregados o modificados desde la app.
- Cambios de horarios y asignación de especialidades (si aplica desde móvil).

## Integración con Flutter

- Se utiliza el plugin cloud_firestore para conectarse y operar sobre Firestore desde Flutter.
- Las operaciones CRUD (crear, leer, actualizar, eliminar) se implementan de forma asíncrona para garantizar fluidez en la interfaz de usuario.
- La aplicación escucha en tiempo real los cambios en los documentos relevantes (por ejemplo, notificaciones sobre citas), permitiendo una actualización inmediata sin necesidad de recargar vistas.

## Seguridad y Reglas

Las reglas de seguridad de Firestore se configuran cuidadosamente para garantizar que el acceso a los datos esté restringido según el rol del usuario

autenticado. De esta manera, los pacientes únicamente pueden leer y modificar su propia información, asegurando su privacidad. Los doctores, por su parte, tienen permiso para acceder exclusivamente a los datos de los pacientes que les han sido asignados, lo que permite una gestión clínica segura y controlada. En cambio, el director cuenta con privilegios más amplios para supervisar y administrar la información dentro del sistema. Estas reglas se implementan directamente desde el panel de Firebase Console, utilizando Firebase Authentication como el mecanismo de identidad que valida y determina los permisos de cada usuario, garantizando así un acceso seguro y personalizado a los recursos.

## sqflite para Almacenamiento Local Offline

Para garantizar la funcionalidad de la aplicación móvil incluso sin conexión a Internet, se ha implementado una capa de persistencia local utilizando la biblioteca sqflite, que permite el uso de SQLite en aplicaciones desarrolladas con Flutter.

### Rol de sqflite en el Proyecto

El sistema cuenta con un almacenamiento temporal que guarda datos críticos como las citas agendadas por el paciente, la información resumida del expediente médico y los datos del perfil del usuario autenticado. Esto asegura que la información esencial esté disponible de manera rápida y eficiente para su consulta y actualización.

Además, se garantiza la operatividad offline, permitiendo que el paciente pueda consultar sus citas y su expediente médico sin necesidad de estar conectado a Internet. Los cambios realizados localmente se almacenan y se sincronizan automáticamente con Firestore cuando la conexión a la red se restablece, manteniendo la información actualizada y consistente.

Este enfoque también soporta funcionalidades clave de la aplicación, como mostrar las citas programadas en modo offline, editar y guardar temporalmente los registros clínicos hasta que puedan ser subidos a la nube, y reducir el tiempo de carga inicial mediante el uso de caché local, mejorando así la experiencia del usuario y la eficiencia del sistema.

### Ventajas del Uso de sqflite

- **Velocidad**: Acceso rápido a datos locales sin latencia de red.
- **Persistencia**: Los datos se conservan, aunque se cierre o reinicie la app.
- **Compatibilidad**: Disponible tanto para Android como para iOS.
- **Simplicidad**: Interfaz directa para realizar operaciones SQL (CRUD).

### Integración con Firestore

Se ha implementado una lógica de sincronización entre la base de datos local (sqflite) y Firestore (nube):

Se implementa una lógica de sincronización automática con Firestore al detectar conexión a Internet, garantizando coherencia entre dispositivos. Se usan DAOs personalizados para gestionar el acceso a datos, y se manejan los estados mediante Provider o Riverpod. Además, se aplican políticas para resolver conflictos de sincronización, incluyendo actualizaciones y eliminaciones locales.

## Sincronización de Datos Local/Nube

Una funcionalidad crítica en aplicaciones móviles modernas es la capacidad de operar sin conexión a Internet y mantener la coherencia de los datos una vez que se restablece la conectividad. En este proyecto, se implementó un mecanismo de **sincronización bidireccional entre sqflite (almacenamiento local)** y **Firebase Firestore (almacenamiento en la nube)**, permitiendo al usuario una experiencia fluida tanto online como offline.

### Objetivos de la Sincronización

La sincronización en el sistema tiene como objetivo principal permitir una funcionalidad offline completa para pacientes, doctores y directores, asegurando que puedan realizar operaciones esenciales incluso sin conexión a Internet. Esto reduce la dependencia inmediata de la red para actividades básicas como visualizar citas o editar datos clínicos, mejorando la continuidad del servicio.

Además, se busca garantizar la coherencia de los datos entre la información almacenada localmente en el dispositivo y la que se encuentra persistida en

Firestore, evitando inconsistencias. Para ello, se implementan mecanismos de control de versiones y marcas de tiempo que evitan la generación de datos duplicados o la sobrescritura de información errónea, asegurando la integridad y precisión de los registros.

**Estrategia de Sincronización Implementada**

La estrategia de sincronización implementada en la aplicación es manual pero con un control automático de conflictos, diseñada para mantener la coherencia y actualización de los datos entre el dispositivo local y Firestore. Para ello, cada registro que se modifica localmente se marca con un campo isSynced = false junto con una marca de tiempo (timestamp), lo que permite identificar fácilmente los cambios pendientes de sincronización.

Para asegurar que la sincronización se realice en el momento adecuado, se utiliza un monitor de conectividad basado en el paquete connectivity_plus, que detecta cambios en el estado de la red. Cuando la aplicación identifica que el dispositivo ha recuperado la conexión a Internet, se activa un proceso en segundo plano que sube a Firestore todos los registros locales que aún no han sido sincronizados, y a su vez actualiza los datos locales con cualquier cambio que haya ocurrido en la nube desde la última sincronización.

En cuanto a la resolución de conflictos, se aplica una regla sencilla pero efectiva: se prioriza siempre el dato más reciente según la timestamp asociada. De esta manera, se minimizan las pérdidas de información y se mantiene la integridad y precisión de los datos en ambas fuentes.

**Componentes Técnicos**

- **Firebase Firestore**:
    - Base de datos central sincronizada entre dispositivos.
- **sqflite**:
    - Base de datos local para uso offline.
- **Provider / Riverpod (u otro gestor de estado)**:
    - Administra el flujo de datos entre la lógica de negocio y la interfaz.
- **Timestamps y flash**:
    - Cada entidad contiene un lastUpdated y un isSynced.

### Casos de Uso Comunes

| Acción | Estado Offline | Sincronización Posterior |
|---|---|---|
| Crear cita | Guardada en sqflite con isSynced = false | Se sube a Firestore cuando hay conexión |
| Editar expediente | Cambios se almacenan localmente | Se actualiza Firestore al reconectarse |
| Eliminar cita | Se marca como eliminada en local | Se borra de Firestore si el estado es sincronizable |

Tabla 3: Casos de uso comunes

### Beneficios Obtenidos

- **Experiencia de usuario sin interrupciones**.
- **Tolerancia a fallos de red**.
- **Reducción de llamadas a la nube**, lo cual mejora el rendimiento y reduce costos.
- **Consistencia eventual** asegurada por la estrategia de sincronización diferida.

## Diseño de UI Responsiva con Flutter (Material Design)

El diseño de interfaces en aplicaciones móviles no solo debe ser estéticamente agradable, sino también funcional, accesible y adaptable a distintos dispositivos. En este proyecto, se utilizó **Flutter** como framework de desarrollo, aprovechando su soporte nativo para **Material Design** y sus capacidades integradas para construir interfaces **responsivas**.

### Fundamentos de Material Design

Material Design es un sistema de diseño desarrollado por Google que proporciona directrices visuales y de interacción para construir interfaces coherentes, accesibles y modernas. Sus principios clave son:

- **Jerarquía visual clara** (tipografía, color, iconografía).
- **Transiciones y animaciones suaves**.
- **Componentes reutilizables** (botones, tarjetas, listas).
- **Enfoque en la accesibilidad y usabilidad.**

Flutter implementa Material Design de forma nativa mediante widgets como Scaffold, AppBar, FloatingActionButton, TextField, Card, entre otros.

## Estructura Visual de la Aplicación

La aplicación móvil se estructura visualmente en base a:

- **Layouts responsivos con MediaQuery, LayoutBuilder y Flexible**, que ajustan el contenido según el tamaño de pantalla.
- **Navegación modular** utilizando Navigator, BottomNavigationBar y Drawer.
- **Estilos centralizados** con ThemeData para asegurar coherencia visual.

### Widgets y Componentes Personalizados

Se desarrollaron widgets personalizados para mantener la consistencia visual y mejorar la reutilización del código:

- CustomAppointmentCard: muestra citas médicas de forma compacta.
- DoctorProfileTile: presenta detalles del doctor con su especialidad.
- ResponsiveScaffold: estructura adaptable que cambia entre diseño de una sola columna y múltiples columnas según el tamaño del dispositivo.

### Adaptabilidad y Escalabilidad

Para que la app funcione bien en teléfonos, tablets y distintas resoluciones, se usaron diseños fluidos con Expanded y Flexible, que permiten adaptar los elementos al espacio disponible. Además, se ajustan automáticamente el tamaño de fuentes e íconos usando MediaQuery.textScaleFactorOf(context) e IconTheme, garantizando una interfaz escalable y cómoda en cualquier dispositivo.

### Accesibilidad

Se prestó especial atención a las buenas prácticas de accesibilidad:

- Contraste adecuado de colores.
- Soporte para lectores de pantalla (Semantics).
- Tamaños de texto escalables.
- Botones e interacciones táctiles amigables.

### Ejemplo de Código

```dart
import 'package:flutter/material.dart';

Qodo Gen: Options | Test this class
class PatientDashboard extends StatelessWidget {
  const PatientDashboard({super.key});

  @override
  Qodo Gen: Options | Test this method
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        decoration: BoxDecoration(
          gradient: LinearGradient(
            colors: [Colors.teal.shade100, Colors.white],
            begin: Alignment.topCenter,
            end: Alignment.bottomCenter,
          ), // LinearGradient
        ), // BoxDecoration
        child: Padding(
          padding: const EdgeInsets.all(16.0),
          child: Column(
            children: [
              const SizedBox(height: 40),
              Center(
                child: Image.asset(
                  'lib/img/logo2.png',
                  height: 80,
                ), // Image.asset
              ), // Center
              const SizedBox(height: 16),
              Text(
                "Dashboard del Paciente",
                style: TextStyle(
                  fontSize: 24,
                  fontWeight: FontWeight.bold,
                  color: Colors.teal[800],
                ), // TextStyle
                textAlign: TextAlign.center,
              ), // Text
              const SizedBox(height: 24),
```

*Ilustración 33: muestra la lógica de personalización de estilos aplicados en los widgets.*

**Beneficios Obtenidos**

- **Interfaz intuitiva y atractiva para todos los roles** (paciente, doctor, director).
- **Experiencia de usuario consistente** en diferentes dispositivos.
- **Reducción del tiempo de desarrollo** gracias al enfoque declarativo de Flutter.
- **Facilidad de mantenimiento y escalabilidad**.

## Formularios Dinámicos y Navegación Intuitiva

La interacción del usuario con la aplicación móvil se basa en gran parte en la gestión eficiente de formularios y una navegación clara. Para ello, se aplicaron prácticas modernas de diseño y desarrollo con **Flutter**, aprovechando su capacidad para construir formularios dinámicos y flujos de navegación fluidos e intuitivos.

**Formularios Dinámicos**

Los formularios permiten capturar datos clave del usuario, como el registro de una cita, actualización del expediente médico o edición de perfil. En este proyecto se utilizaron formularios **dinámicos y validados en tiempo real**, construidos mediante el widget Form y TextFormField, junto con lógica condicional.

**Características clave:**

- **Campos adaptativos:** se muestran u ocultan según la selección del usuario (por ejemplo, especialidad seleccionada filtra los doctores disponibles).
- **Validaciones automáticas:** formatos de correo, fechas válidas, campos requeridos, etc.
- **Integración con backend:** al enviar los formularios, los datos son almacenados directamente en **Firebase Firestore** o enviados al **backend web** mediante API REST.

Ejemplo de formulario dinámico:

```
Qodo Gen: Options | Test this method
void registerDirector(BuildContext context) async {
  final name = nameController.text.trim();
  final email = emailController.text.trim();
  final password = passwordController.text.trim();
  final confirmPassword = confirmPasswordController.text.trim();

  if (name.isEmpty || email.isEmpty || password.isEmpty || confirmPassword.isEmpty) {
    _showError("Todos los campos son obligatorios.");
    return;
  }

  if (password != confirmPassword) {
    _showError("Las contraseñas no coinciden.");
    return;
  }

  try {
    // Registrar en Firebase Auth
    UserCredential userCredential = await FirebaseAuth.instance
        .createUserWithEmailAndPassword(email: email, password: password);

    String uid = userCredential.user!.uid;

    // Guardar en Firestore
    await FirebaseFirestore.instance.collection('usuarios').doc(uid).set({
      'nombre': name,
      'rol': 'Director',
      'correo': email,
    });

    Navigator.pop(context); // Puedes redirigir a una pantalla principal si deseas
  } on FirebaseAuthException catch (e) {
    String error = "Error al registrar.";
    if (e.code == 'email-already-in-use') {
      error = "El correo ya está en uso.";
    } else if (e.code == 'invalid-email') {
      error = "Correo inválido.";
    } else if (e.code == 'weak-password') {
      error = "La contraseña es muy débil.";
    }
    _showError(error);
  } catch (e) {
    _showError("Ocurrió un error inesperado.");
  }
```

*Ilustración 34: muestra la lógica de los formularios utilizados para registrar usuarios*

### Navegación Intuitiva

Una buena experiencia de usuario depende de una navegación fluida y clara. Para lograrlo, se empleó el sistema de rutas de Flutter y se estructuraron las vistas según el rol del usuario (Paciente, Doctor, Director), con un enfoque de navegación **modular y centralizada**.

**Herramientas y componentes utilizados:**

- Navigator y Named Routes: para controlar el flujo entre pantallas.
- BottomNavigationBar para pacientes y doctores.
- Drawer (Menú lateral) para el rol de director.
- WillPopScope para manejar correctamente la navegación hacia atrás.

**Organización por rol:**

- **Paciente:** navegación sencilla entre cita, expediente, chat y perfil.
- **Doctor:** acceso directo a citas programadas, expediente clínico y notificaciones.
- **Director:** panel administrativo con navegación lateral.

**Ejemplo de navegación:**

```
Qodo Gen: Options | Test this class
class InitialScreenState extends State<InitialScreen> {
  Qodo Gen: Options | Test this method
  Future<void> checkRoleAndNavigate() async {
    final prefs = await SharedPreferences.getInstance();
    final role = prefs.getString('role');

    if (!mounted) return;

    if (role == 'director') {
      Navigator.pushReplacementNamed(context, '/director_dashboard');
    } else if (role == 'doctor') {
      Navigator.pushReplacementNamed(context, '/doctor_dashboard');
    } else if (role == 'patient') {
      Navigator.pushReplacementNamed(context, '/patient_dashboard');
    } else {
      Navigator.pushReplacementNamed(context, '/login');
    }
  }
```

*Ilustración 35: muestra la lógica para verificar el rol del usuario almacenado en los registros y redirigirlo a su perfil correspondiente.*

**Beneficios obtenidos**

Los beneficios obtenidos incluyen una experiencia de usuario optimizada, que evita recargas innecesarias y ofrece interacciones guiadas y validadas para reducir errores. Además, la modularidad y mantenibilidad del código mejoran gracias a la separación por rutas y el uso de formularios reutilizables. Todo esto contribuye a una mayor productividad del usuario final, al proporcionar flujos de trabajo claros, lógicos y sencillos.

## Accesibilidad y Adaptabilidad

La accesibilidad y adaptabilidad son componentes clave para garantizar que la aplicación móvil pueda ser utilizada por una amplia gama de usuarios, incluyendo personas con diferentes capacidades y dispositivos con diversas características de pantalla. En el desarrollo de este proyecto en Flutter, se implementaron diversas prácticas enfocadas en hacer la aplicación inclusiva, funcional y adaptable.

### Accesibilidad

Flutter ofrece soporte integrado para accesibilidad, lo cual facilita el desarrollo de interfaces que cumplen con estándares como WCAG (Web Content Accessibility Guidelines). En este proyecto se consideraron los siguientes aspectos:

**Ejemplo:**

```
Row(
  children: [
    Expanded(
      child: Text(
        'Fecha y hora: ${dateTime.toString().substring(0, 16)}',
        style: const TextStyle(fontSize: 16),
      ), // Text
    ), // Expanded
    TextButton(
      onPressed: selectDateTime,
      child: const Text('Seleccionar'),
    ), // TextButton
  ],
), // Row
const SizedBox(height: 24),
ElevatedButton(
  onPressed: createAppointment,
  child: const Text('Crear Cita'),
), // ElevatedButton
```

*Ilustración 36: muestra la lógica de personalización de textos, colores y controles accesibles.*

### Adaptabilidad

La adaptabilidad asegura que la aplicación se visualice correctamente en una amplia gama de dispositivos (smartphones, tablets) y en distintas orientaciones de pantalla. En este proyecto se aplicaron los principios de **responsive design**:

**Implementaciones clave:**

Estas permiten que los widgets se ajusten dinámicamente según las dimensiones del dispositivo, como el ancho, alto y orientación de la pantalla, lo que garantiza que la experiencia del usuario sea óptima tanto en teléfonos móviles pequeños como en tablets o dispositivos con pantallas más grandes. Este tipo de diseño es fundamental para mantener la coherencia visual y funcionalidad sin importar el dispositivo que se utilice.

Además, se incorporaron widgets flexibles y escalables, como Expanded, Flexible, Wrap, GridView y ListView, que ofrecen una gran versatilidad para organizar el contenido de forma eficiente y adaptable. Estos widgets permiten que los elementos dentro de la interfaz crezcan, se contraigan o se redistribuyan según el espacio disponible, lo que ayuda a evitar problemas de desbordamiento o espacios vacíos en la pantalla.

Finalmente, se incorporó soporte para ambas orientaciones de pantalla, vertical y horizontal, lo que permite que la aplicación funcione correctamente y mantenga su estructura y usabilidad sin importar cómo el usuario sostenga el dispositivo. Esta flexibilidad es especialmente útil en tablets y teléfonos, donde cambiar la orientación puede ser común, garantizando así una experiencia fluida y sin interrupciones en cualquier situación.

**Ejemplo de layout adaptable:**

```
@override
Qodo Gen: Options | Test this method
Widget build(BuildContext context) {
  return Scaffold(
    body: LayoutBuilder(
      builder: (context, constraints) {
        return SingleChildScrollView(
          padding: const EdgeInsets.all(16),
          child: Center(
            child: ConstrainedBox(
              constraints: const BoxConstraints(maxWidth: 600),
              child: Column(
                children: [
                  const SizedBox(height: 30),
                  Image.asset('lib/img/logo2.png', height: 100),
                  const SizedBox(height: 20),
                  Card(
                    elevation: 6,
                    shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(16)),
                    child: Padding(
                      padding: const EdgeInsets.all(24),
                      child: Form(
                        key: _formKey,
                        child: Column(
```

*Ilustración 37: muestra la implementación de herramientas responsive*

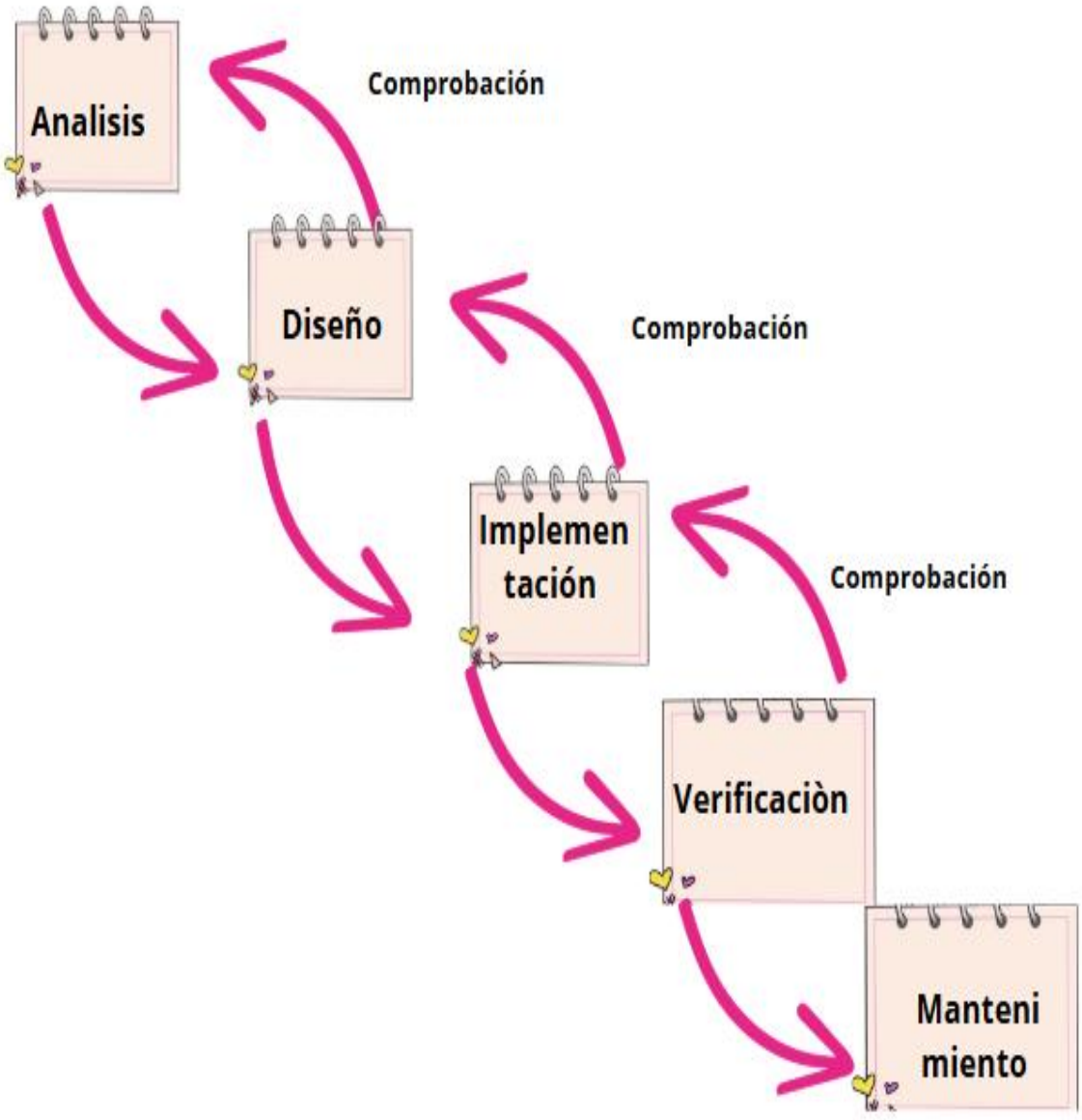
**Beneficios obtenidos**

Los beneficios obtenidos en el proyecto son notables en varios aspectos clave. En primer lugar, se logró una mejora significativa en la experiencia de usuario para personas con discapacidades visuales o dificultades motrices, gracias a la implementación de técnicas de accesibilidad y adaptabilidad que facilitan la interacción con la aplicación. Esto no solo hace que la app sea más inclusiva, sino que también amplía su potencial alcance y utilidad para un público más diverso.

Además, la aplicación se adapta correctamente a diferentes dispositivos y resoluciones, lo que permite un mayor alcance y una experiencia consistente sin importar si se utiliza en teléfonos, tablets u otros dispositivos. Esta adaptabilidad asegura que más usuarios puedan acceder y utilizar la app de forma cómoda y eficiente, independientemente de las características de su dispositivo.

Por último, el uso de estructuras escalables y componentes reutilizables facilita el mantenimiento y la evolución del proyecto a largo plazo.

**Diseño Metodológico: Aplicación Web y Android para la Gestión de Citas Médicas utilizando la Metodología Ágil SCRUM Estándar.**



## ANEXOS:

### Aplicación web:

Esta es la página de inicio donde los usuarios pueden iniciar sesión o registrarse si aún no tienen una cuenta.


*Ilustración 38: Muestra la página de inicio de la aplicación web.*

En esta imagen se muestra el proceso de registro de un nuevo usuario, específicamente un paciente.


*Ilustración 39: Muestra cómo se registra un usuario (Paciente) por primera vez.*

En esta imagen, el usuario ya se ha registrado previamente y está iniciando sesión en el sistema.

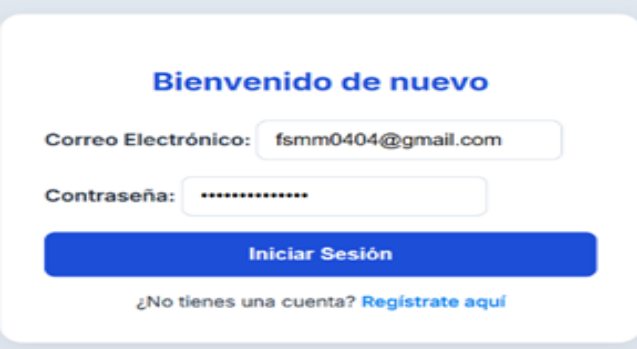



*Ilustración 40: Muestra el inicio sesión del usuario creado*

Acá se inició sesión y está en su perfil

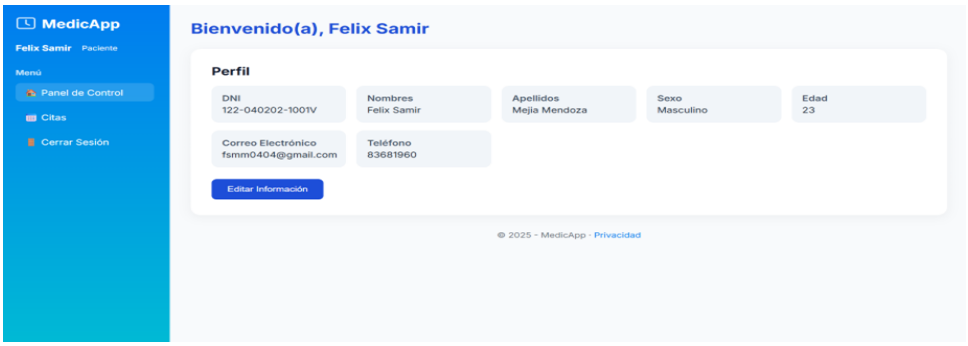



*Ilustración 41: Muestra la interfaz de inicio de sesión de un usuario (Paciente)*

En esta imagen, el usuario está agendando una cita desde el apartado de citas. Se puede observar que las horas marcadas en rojo indican los horarios que ya tienen citas previamente agendadas.


*Ilustración 42: Muestra cómo se agenda.*

En esta imagen se muestran las especialidades médicas disponibles para seleccionar al momento de agendar una cita.


*Ilustración 43: Muestra las opciones de especialidades para agendar cita.*

En esta imagen se muestra una cita que ya ha sido agendada. También se puede ver un botón que permite acceder a los detalles de la cita.

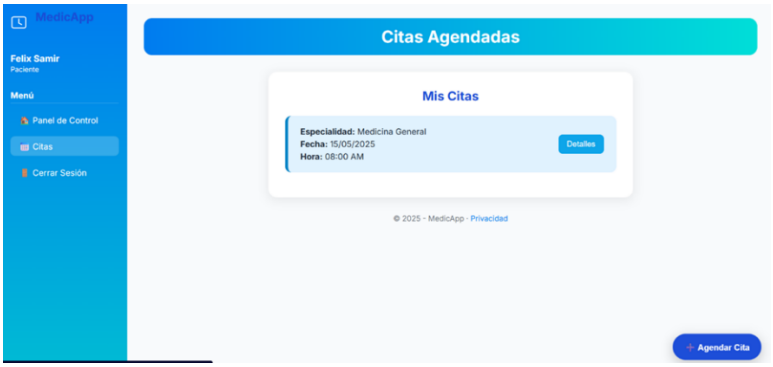



*Ilustración 44: Muestra que ya se agendo una cita:*

Esta imagen corresponde al perfil del administrador, donde puede gestionar y supervisar las diferentes funciones del sistema.

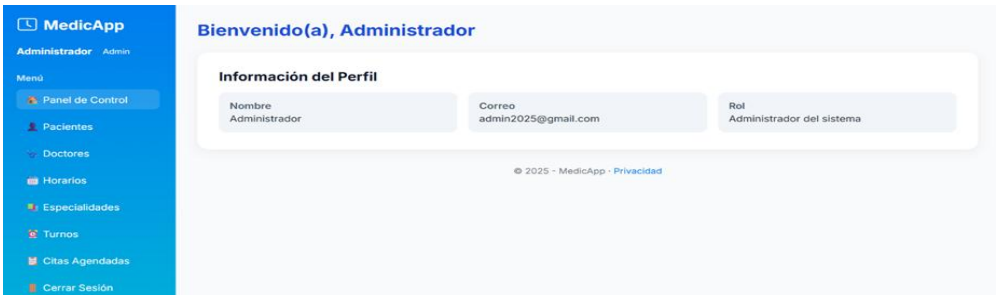



*Ilustración 45: Muestra el perfil de usuario con rol de administrador.*

Esta imagen muestra la lista de pacientes registrados en el sistema, junto con un botón para ver los detalles de cada uno.

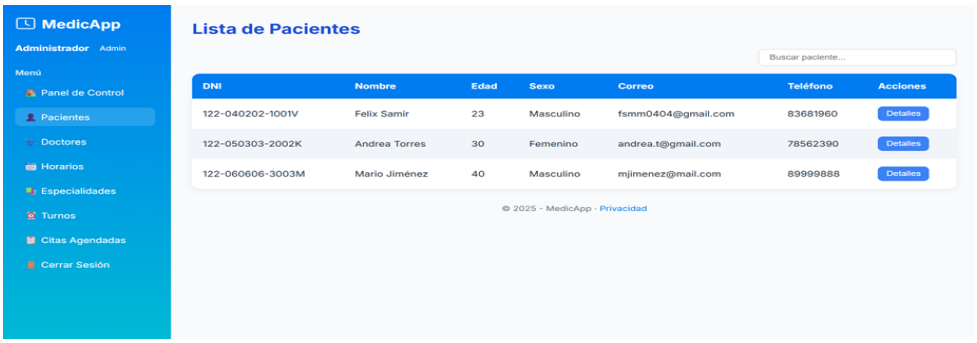



*Ilustración 46: Muestra la lista de pacientes registrados en la página.*

Aquí se visualizan los detalles de un paciente que tiene una cita médica confirmada.


*Ilustración 47: Muestra los detalles de la cita de un paciente que tiene agendada una.*

En esta imagen se muestran los detalles de un paciente que tiene una cita médica pendiente de confirmación.

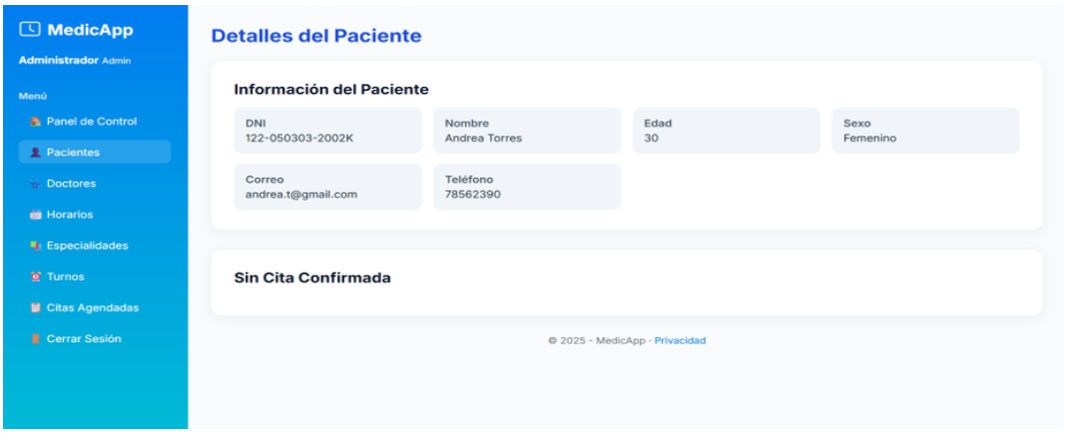



*Ilustración 48: Muestra los detalles de la cita de un paciente con cita sin agendar o confirmar.*

Esta imagen muestra la lista de doctores registrados en el sistema. Desde esta sección es posible agregar un nuevo doctor, así como editar o eliminar sus horarios y turnos asignados.

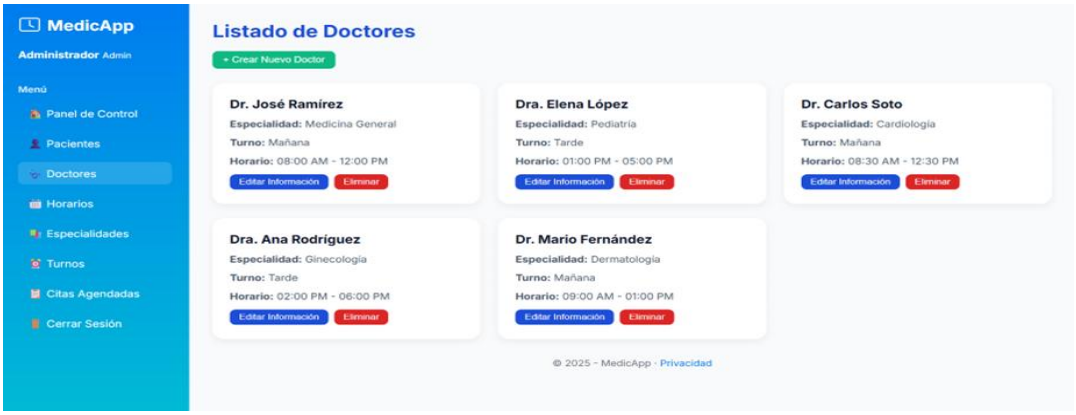



*Ilustración 49: Muestra la lista de doctores con los que se pueden agendar citas.*

En esta imagen se muestra el proceso de creación de un nuevo doctor en el sistema.


*Ilustración 50: Muestra cómo se agrega un doctor desde el perfil administrador.*

En esta sección se muestran las especialidades médicas disponibles. Es posible crear nuevas especialidades o eliminar las existentes.


*Ilustración 51: Muestra las especialidades que están disponibles para agendar citas:*

## APLICACIÓN ANDROID

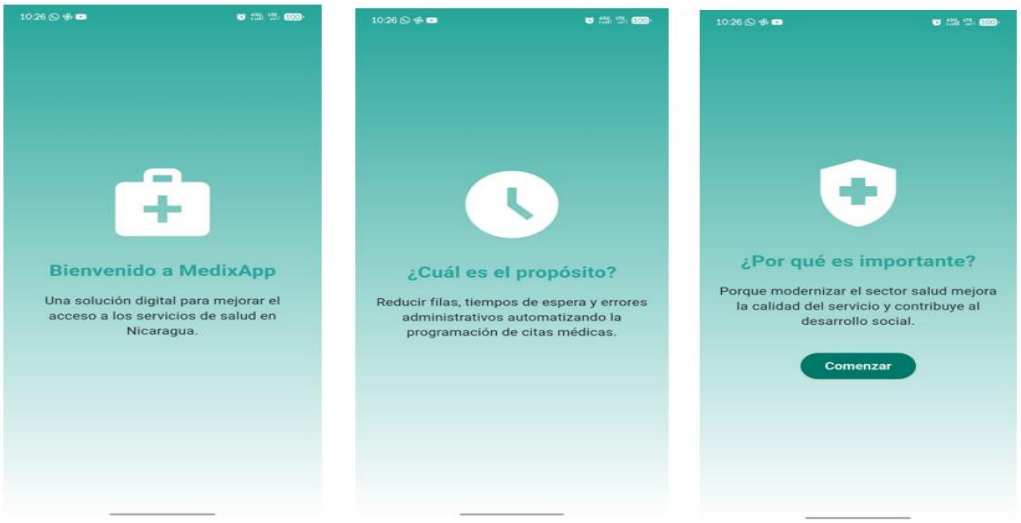



*Ilustración 52: Pantallas de introducción a la aplicación*

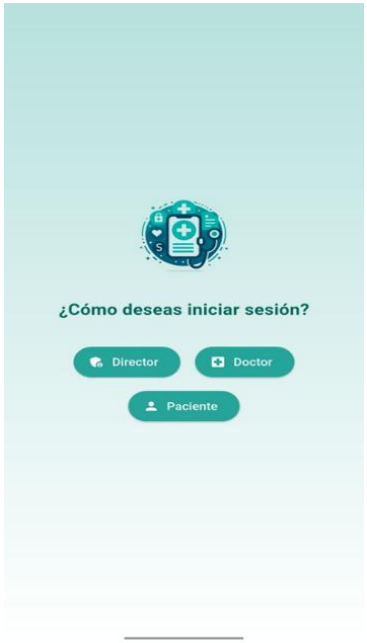



*Ilustración 53: selección de rol del usuario*

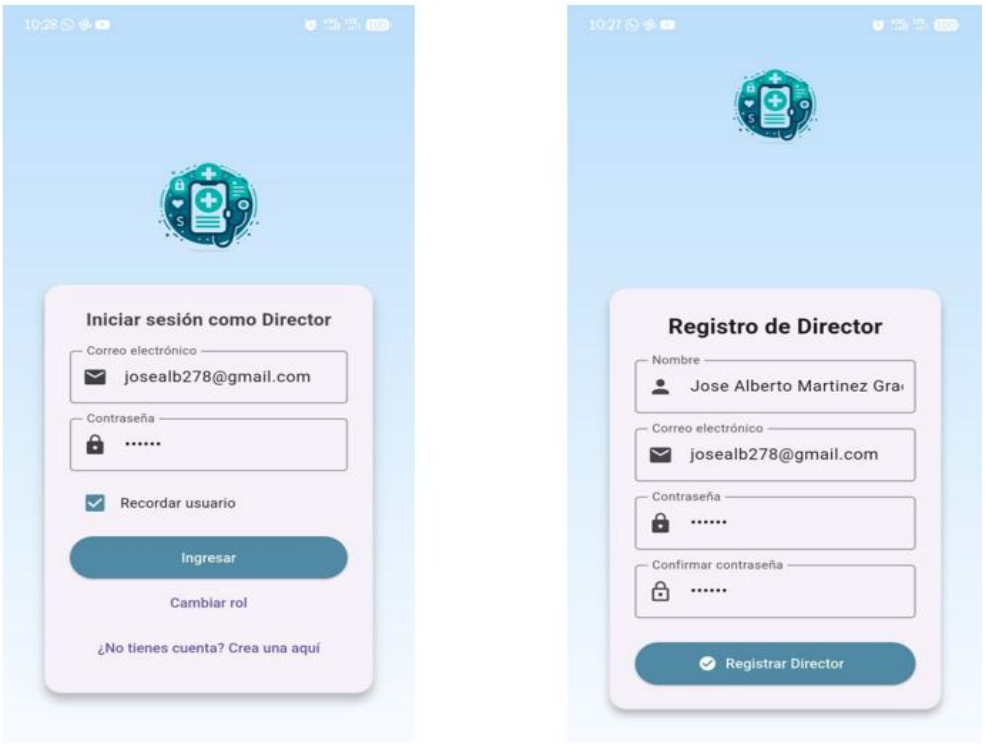



*Ilustración 55: vista de inicio de sesión y registro.*


*Ilustración 54: vista del perfil del director*

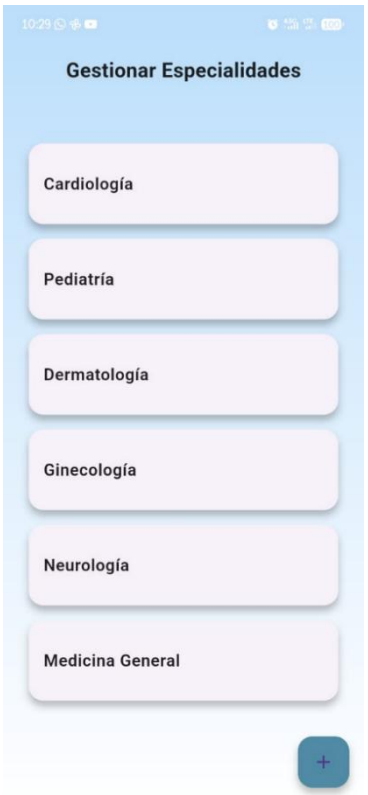



*Ilustración 57:Pantalla de función de especialidades*

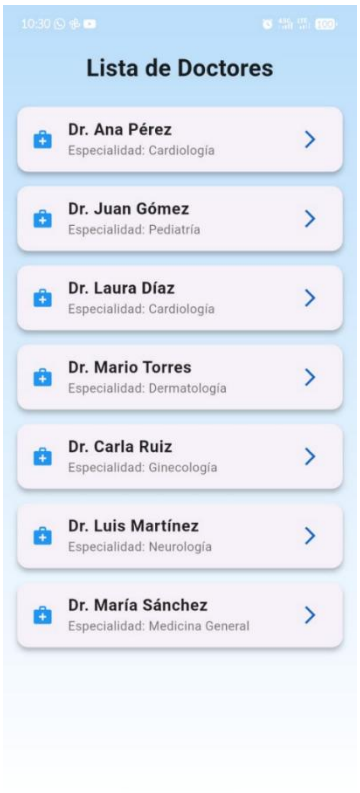



*Ilustración 56: Pantalla de lista de doctores*

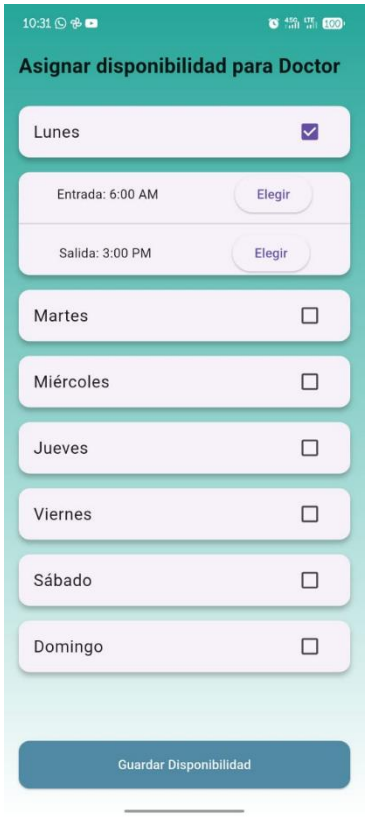



*Ilustración 59: Pantalla de asignación de horarios*

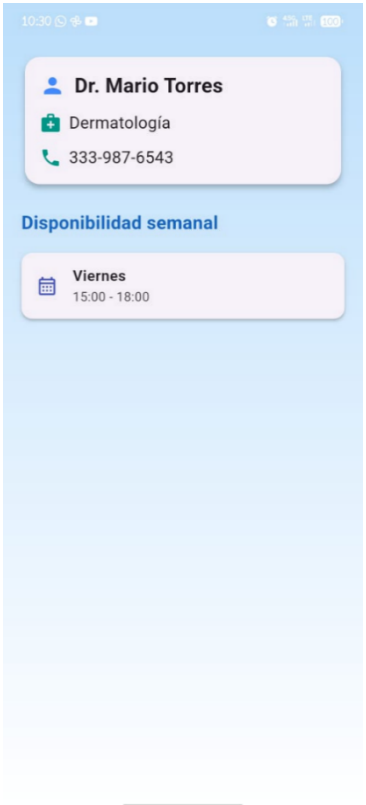



*Ilustración 58: vista citas registradas*

## Conclusión

El desarrollo del sistema integral de gestión de citas médicas ha demostrado la viabilidad y eficacia de una arquitectura cliente-servidor distribuida que integra tecnologías modernas tanto en el ámbito web como móvil. A través del uso de ASP.NET Core MVC en la parte web, se logró construir un sistema robusto, escalable y seguro que permite la gestión eficiente de doctores, especialidades y citas médicas, manteniendo una arquitectura clara mediante el patrón Modelo-Vista-Controlador (MVC) y una correcta persistencia de datos con Entity Framework Core y MySQL.

En el módulo móvil, desarrollado con Flutter y Dart, se priorizó la experiencia del usuario final, brindando una aplicación intuitiva y adaptable que permite a los pacientes y doctores interactuar fácilmente con el sistema. La utilización de Firebase como backend en la nube proporcionó servicios clave como autenticación, base de datos en tiempo real (Firestore), notificaciones (Firebase Cloud Messaging) y almacenamiento, complementando así la funcionalidad local ofrecida por sqflite.

La integración entre ambas plataformas mediante API REST permitió una comunicación fluida y segura entre el cliente móvil y el servidor web, garantizando consistencia en los datos y sincronización eficiente. Además, la implementación de control de acceso basado en roles (Paciente, Doctor, Director) contribuyó significativamente a la seguridad del sistema y a la personalización de las funcionalidades según el tipo de usuario.

En conjunto, el proyecto no solo ha cumplido con los objetivos planteados al inicio, sino que también sienta las bases para futuras ampliaciones, tales como la incorporación de historiales clínicos más completos, video consultas, inteligencia artificial para asignación de médicos, y una interfaz de administración más avanzada. De esta manera, se contribuye al mejoramiento de los procesos de atención médica y se fortalece la transformación digital en el sector salud.

## Recomendaciones

Ampliar funcionalidades, considerando la integración de videollamadas para consultas remotas y herramientas que faciliten la interacción entre paciente y doctor.

Implementar pruebas y monitoreo, mediante herramientas que permitan detectar fallos, asegurar el rendimiento y facilitar el mantenimiento del sistema.

Desarrollar sesiones de capacitación para los usuarios finales a fin de facilitar la adopción del sistema.

Garantizar la escalabilidad del sistema, considerando infraestructuras en la nube de mayor rendimiento si aumenta el número de usuarios.

## Referencias bibliográficas:


Duarte Rocha, C. G. (2022). *Desarrollo de una aplicación web de gestión de citas médicas y asistencia previa a pacientes llamada "Medicall" en clínicas locales de la ciudad de Juigalpa-Chontales utilizando la biblioteca React, durante el I semestre del año 2022* [Proyecto de graduación, Universidad Nacional Autónoma de Nicaragua, UNAN-Managua]. Repositorio Institucional UNAN-Managua. http://repositorio.unan.edu.ni/id/eprint/20269

López Jara, K. S., & Valle Cárcamo, K. A. (202). *Desarrollo de una aplicación web para el control de citas y expediente médico de los pacientes de la cadena de sucursales de Clínica San Benito* [Monografía de licenciatura, Universidad Nacional de Ingeniería]. Repositorio Institucional de la UNI. https://ribuni.uni.edu.ni/view/creators/Valle_C%3DE1rcamo%3D3AKeyner_Asiel%3D3A%3D3A.default.html

Fierro Mariño, J. S., & Rodríguez Espejo, L. M. (2024). *Implementación de un software para la gestión del historial clínico de un consultorio médico* [Trabajo integrador curricular, Universidad Tecnológica Ecotec]. Repositorio ECOTEC. https://repositorio.ecotec.edu.ec/bitstream/123456789/1477/1/FIERRO%20MARI%C3%91O%20JUAN%20STEEVEN%20%26%20RODRIGUEZ%20ESPEJO%20LUIS%20MATEO.pdf

Revista Médica. (s.f.). *Gestión de citas médicas: optimizar el tiempo de los pacientes y médicos*. https://revistamedica.com/gestion-citas-medicas-optimizar-tiempo/

Organización Panamericana de la Salud. (s.f.). *Sistemas de Información para la Salud (IS4H)*. Organización Panamericana de la Salud. https://www3.paho.org/ish/index.php/es/

Freeman, A., & Sanderson, P. (2021). *Pro ASP.NET Core MVC 2*. Apress. https://doi.org/10.1007/978-1-4842-3191-4

Esposito, D. (2020). *Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure: Add Identity and Security to your App*. Microsoft Press. https://dotnet.microsoft.com

Rahimi, A., & Reynolds, C. (2021). *Entity Framework Core in Action*. Manning Publications.

Date, C. J. (2019). *An Introduction to Database Systems* (8th ed.). Pearson Education.

Microsoft Docs. (2025). *ASP.NET Core MVC overview*. https://learn.microsoft.com/en-us/aspnet/core/mvc/overview

Microsoft Docs. (2025). *Entity Framework Core documentation*. https://learn.microsoft.com/en-us/ef/core/

Microsoft Docs. (2025). *ASP.NET Core Web API*. https://learn.microsoft.com/en-us/aspnet/core/web-api/

Google. (2025). *Get started with Firebase Authentication on Flutter*. Firebase Documentation. https://firebase.google.com/docs/auth/flutter/start

Martin, R. C. (2018). *Clean Architecture: A Craftsman's Guide to Software - Structure and Design*. Prentice Hall.

Sommerville, I. (2020). *Software Engineering* (10th ed.). Pearson.

## Cronograma de actividades

| | | | Mes 1 | | | | Mes2 | | | | Mes 3 | | | | Mes 4 | | | | Mes 5 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Fase** | **Actividad** | **Duración Estimada** | 1 SE | 2 SE | 3 SE | 4 SE | 1 SE | 2 SE | 3 SE | 4 SE | 1 SE | 2 SE | 3 SE | 4 SE | 1 SE | 2 SE | 3 SE | 4 SE | 1 SE | 2 SE | 3 SE | 4 SE |
| **1. Planeación** | Investigación de antecedentes y estudios previos | 2 semanas | | | | | | | | | | | | | | | | | | | | |
| | Creación del cronograma detallado | 1 semanas | | | | | | | | | | | | | | | | | | | | |
| **2. Diseño Conceptual** | Diseño del prototipo inicial del sistema | 3 semanas | | | | | | | | | | | | | | | | | | | | |
| **3. Desarrollo Técnico** | Configuración del ent.de desarrollo (frameworks y bases de datos) | 3 semanas | | | | | | | | | | | | | | | | | | | | |
| | Imp. de funcionalidades principales (gestión de citas, autenticación) | 3 semanas | | | | | | | | | | | | | | | | | | | | |
| | Desarrollo de la interfaz móvil (Android) | 2 semanas | | | | | | | | | | | | | | | | | | | | |
| **4. Pruebas** | Pruebas funcionales y de seguridad | 2 semanas | | | | | | | | | | | | | | | | | | | | |
| | Corrección de errores detectados en pruebas | 2 semanas | | | | | | | | | | | | | | | | | | | | |
| **5. Implementación** | Configuración del sistema en el entorno de producción | 1 semanas | | | | | | | | | | | | | | | | | | | | |
| | Lanzamiento oficial del sistema | 1 semanas | | | | | | | | | | | | | | | | | | | | |