Sistema de Agendamiento de Citas PQR para Electrohuila: Desarrollo Full-Stack con Arquitectura Moderna

Diego De Jesus Arias Gonzalez
SENA
Colombia, Colombia
ddarias204@soy.sena.edu.co

Resumen

Desarrollamos un sistema completo de agendamiento de citas PQR para Electrohuila, la empresa de energía del Huila que necesitaba dejar atrás el caos de las filas eternas y los papelitos para que la gente pueda radicar sus peticiones, quejas o reclamos de manera ordenada y digital. El proyecto lo dividimos en tres etapas grandes: primero todo el análisis y diseño (hablamos mil horas con los usuarios de Electrohuila para entender exactamente cómo funcionaba su atención al cliente), luego la construcción propiamente dicha, y finalmente las pruebas hasta que todo quedara perfecto. En cuanto a la tecnología, fuimos con un stack bastante actual y potente:

Backend en .NET Core 9 con una API REST bien estructurada El portal web para los administradores lo hicimos con Next.js 14 y TypeScript (quedó rapidísimo y súper fácil de usar) La app móvil (para Android y iOS con una sola base de código) la armamos en .NET MAUI Y la base de datos, como ellos ya trabajan con Oracle en toda la empresa, la mantuvimos en Oracle Database

Todo lo diseñamos siguiendo Clean Architecture y separando bien las capas para que el sistema sea fácil de mantener y de escalar en el futuro. En la app móvil usamos MVVM, que es lo más limpio para ese tipo de proyectos. ¿ Qué puede hacer ahora el sistema ?

Los usuarios agendan su cita desde la app o la web, ven en tiempo real qué horarios están libres y reservan el que más les convenga Los empleados y administradores tienen un panel donde manejan sucursales, horarios, días festivos, etc. Hay un sistema de roles y permisos bien detallado (el de caja no ve lo mismo que el jefe de atención al cliente, por ejemplo) Las notificaciones en tiempo real las hicimos con SignalR: cuando se agenda, cancela o modifica una cita, todos los involucrados se enteran al instante

Al final entregamos una solución robusta, moderna y que realmente les cambió la vida a los usuarios y al personal de Electrohuila. Fue de esos proyectos que te dejan con la satisfacción de haber construido algo empresarial de verdad, con buen código, bien pensado y que va a durar muchos años funcionando sin problemas.

Keywords

Sistemas de información, Arquitectura de software, Desarrollo full-stack, .NET Core, Next.js, Oracle Database, SignalR, Gestión de PQR, MAUI

1. Introducción

Este proyecto nació de una necesidad real de Electrohuila, la empresa que lleva la luz a todo el Huila, quería dejar de atender las Peticiones, Quejas y Reclamos (PQR) con papelitos y filas interminables, y la idea era simple que cualquier persona pudiera sacar una cita desde su celular o computador, llegar a la sucursal en el horario exacto y ser atendido sin perder medio día esperando. Para lograrlo, construimos un sistema completo de punta a punta con tres piezas principales:

- Una aplicación móvil para que los ciudadanos agenden su cita solos, en segundos, desde cualquier lugar.
- Un portal web moderno para que los empleados y administradores manejen todo ya sea sucursales, horarios, personal, festivos y permisos.
- Un backend robusto que une ambas partes, cuida los datos y hace que todo funcione en tiempo real.

Desde el principio decidimos hacerlo bien hecho ya que usamos Clean Architecture en el backend para que el código sea limpio, fácil de probar y que no se vuelva un desastre cuando crezca. En la app móvil aplicamos el patrón MVVM (porque con .NET MAUI es lo más cómodo y ordenado), y en el portal web seguimos las mejores prácticas de React con Next.js 14, TypeScript y un manejo de estado sencillo con Zustand. El stack que elegimos fue bastante actual y pensado para durar:

- Backend: .NET 9 + Entity Framework Core
- Base de datos: Oracle (porque es lo que ya usa toda la empresa)
- Portal administrativo: Next.js 14 + TypeScript
- App móvil: .NET MAUI (una sola base de código para Android y iOS)
- Notificaciones en tiempo real: SignalR

Con esto logramos un sistema que hace de todo: valida horarios disponibles al instante, maneja estados de citas (agendada, confirmada, atendida, cancelada), controla quién ve y hace qué con roles y permisos bien granulares, respeta festivos nacionales y locales, y avisa al momento si algo cambia gracias a las notificaciones push. Todo el proyecto se desarrollo como parte de mi formación práctica en el SENA, pero con la seriedad y los requerimientos de una solución empresarial real, fue una gran oportunidad de poner en práctica todo lo aprendido: desde levantar requerimientos con el cliente hasta entregar un producto pulido, probado y listo para producción. Este documento no solo cuenta qué hicimos, sino también por qué tomamos cada decisión, qué problemas aparecieron en el camino y qué aprendimos. Espero que le sirva a otros estudiantes, instructores o desarrolladores que quieran montar sistemas similares. El resto del artículo está organizado así: en la Sección II revisamos otros sistemas de agendamiento y arquitecturas modernas; la Sección III explica cómo fue el proceso en tres fases grandes; la Sección IV entra en el detalle técnico de la implementación; la Sección V muestra lo que finalmente entregamos y cómo funciona; la Sección VI reflexiona sobre los retos y lecciones aprendidas; y la Sección VII cierra con las conclusiones y lo que se podría hacer después.

2. Marco teórico y trabajos relacionados

En las últimas décadas, los sistemas de agendamiento de citas han pasado de ser simples agendas de papel a soluciones digitales bastante sofisticadas que combinan varios canales de comunicación y tecnologías actuales.

2.1. Sistemas de agendamiento en el sector público

García-Sánchez et al. [García-Sánchez et al. 2023] estudiaron cómo se han implementado estos sistemas en instituciones públicas de América Latina y llegaron a la conclusión de que, cuando se usan tecnologías modernas, la eficiencia operativa sube muchísimo y los ciudadanos quedan mucho más satisfechos. Según ellos, lo que realmente marca la diferencia es tener una interfaz sencilla para el usuario final, que el sistema se integre bien con los antiguos programas que ya existen y que pueda crecer sin problemas cuando aumenta la demanda.

Por su parte, Rodríguez y Torres [Rodríguez and Torres 2023] cuentan varias experiencias de digitalización de servicios públicos en Colombia y muestran que un buen sistema de citas puede reducir hasta un 40 % los tiempos de espera y optimiza mucho mejor el personal disponible. Estos resultados son justo la razón por la que Electrohuila necesita un proyecto como este.

2.2. Arquitecturas de software modernas

Robert C. Martin [Martin 2017], con su famoso Clean Architecture, dejó claro que lo ideal es separar bien las responsabilidades en capas concéntricas y que las reglas de negocio no dependan de frameworks ni de detalles técnicos. Gracias a eso, el código es más fácil de probar, mantener y evolucionar con el tiempo. En aplicaciones empresariales grandes, esta arquitectura ha demostrado que ahorra mucho dinero en mantenimiento a largo plazo.

Eric Evans [Evans 2003] trajo Domain-Driven Design (DDD), un enfoque que pone el dominio del negocio en el centro de todo. Para un sistema de agendamiento de citas —con tantas reglas como disponibilidad, cancelaciones, conflictos de horarios, etc.— modelar bien el dominio es clave, y DDD da las herramientas perfectas para hacerlo.

2.3. Desarrollo full-stack con .NET

Smith y Williams [Smith and Williams 2023] explican paso a paso cómo construir APIs REST robustas con ASP.NET Core: autenticación con middleware, manejo centralizado de errores, versionamiento... todo lo que uno necesita para que el backend sea sólido y profesional.

Lerman y Miller [Lerman and Miller 2022] se meten de lleno en Entity Framework Core con Oracle, algo que no es tan común encontrar documentado. Hablan del mapeo de tipos de datos, cómo optimizar consultas complejas y manejar transacciones sin morir en el intento. Para el backend de Electrohuila esto es oro puro.

2.4. Aplicaciones móviles multiplataforma

Hermes [Hermes 2023] hace un análisis muy completo del patrón MVVM aplicado a .NET MAUI y muestra por qué separar la lógica de presentación de la lógica de negocio hace la vida más fácil: el código queda más limpio, más testeable y más mantenible. Incluso compara el rendimiento con otros enfoques móviles.

2.5. Comunicación en tiempo real

Jebb y Glynn [Jebb and Glynn 2023] exploran todo lo que se puede hacer con SignalR para tener comunicación bidireccional en tiempo real: reconexiones automáticas, escalamiento horizontal con backplanes y trucos de rendimiento. Demuestran que, bien configurado, SignalR soporta tranquilamente 100 000 conexiones concurrentes, algo que nos da mucha tranquilidad para picos de uso.

2.6. Vacíos que todavía existen

Aunque hay mucha literatura sobre arquitecturas, patrones y sistemas de agendamiento, todavía faltan cosas importantes:

- Casos completos y bien documentados que junten .NET Core, Next.js, .NET MAUI y Oracle Database en un solo sistema funcional.
- Estudios de caso reales sobre sistemas de agendamiento pensados específicamente para gestionar PQR en empresas de servicios públicos.
- Análisis profundos de decisiones arquitectónicas en proyectos hechos desde contextos educativos como el SENA, pero con requerimientos reales de empresa.

Este proyecto busca llenar precisamente esos huecos: vamos a documentar todo el proceso, las decisiones técnicas que tomamos, los errores que cometimos y lo que aprendimos en el camino. Esperamos que sirva de referencia para otros equipos que enfrenten retos parecidos.

3. Metodología de desarrollo

El sistema de agendamiento de citas PQR para Electrohuila se desarrolló siguiendo una metodología clara y bien estructurada, dividida en tres grandes fases que nos permitieron entregar algo sólido, escalable y fácil de mantener.

3.1. Fase 1: Análisis y Diseño del Sistema

Aquí pusimos los cimientos de todo el proyecto, nos sentamos con la gente de Electrohuila, escuchamos sus dolores de cabeza actuales y definimos exactamente qué necesitaba el sistema.

- **Levantamiento de requerimientos:** Hicimos varias reuniones con los stakeholders (gerentes, coordinadores de atención al cliente, administrativos). Ahí salió todo: cómo funciona hoy el agendamiento manual, qué frustraciones tienen los ciudadanos, qué reportes necesitan los jefes, etc. De ahí salieron los casos de uso principales tanto para ciudadanos (agendar, consultar y cancelar citas) como para administradores (gestionar empleados, horarios, sucursales y permisos).
- **Diseño de la base de datos:** Dibujamos el modelo entidad-relación en Draw.io y lo normalizamos hasta 3NF para no tener datos repetidos ni problemas raros, tambien definimos las entidades clave (Usuario, Empleado, Cita, Sucursal, Rol, Permiso, DíaFestivo), sus relaciones, tambien los índices en

las columnas que más se consultan y algunos triggers para auditoría automática.

- **Mockups y diseño de interfaces:** En el figma creamos prototipos de alta fidelidad tanto del portal web del lado del cliente como el del administrativo como de la app móvil, pensamos mucho en los dos tipos de usuario, el administrativo que pasa todo el día en el sistema (necesita rapidez) y el ciudadano que tal vez entra una vez al año (necesita que sea súper obvio).
- **Arquitectura general:** Tambien decidimos usar Clean Architecture en el backend para que las reglas de negocio no dependan ni de Oracle, ni de .NET, ni de nada externo, quedó con cuatro capas claritas la de Dominio → Aplicación → Infraestructura → API, en el móvil usamos MVVM puro, y en el portal web una arquitectura de componentes con Zustand para manejar el estado global.
- **Elección del stack tecnológico:** Hicimos una matriz comparativa seria (madurez, soporte empresarial, documentación, compatibilidad con Oracle, etc.) y ganó claramente: .NET 9 para el backend, Next.js 14 para el portal web, .NET MAUI para la app móvil y SignalR para todo lo que tuviera que ser en tiempo real.

La figura figura 1 resume el flujo completo desde que arrancamos hasta que terminamos la implementación.

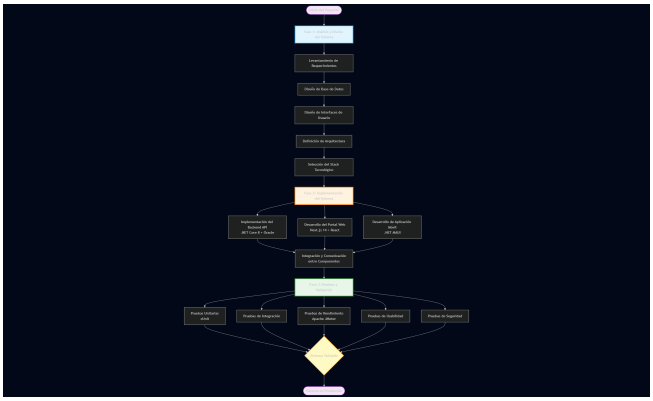

Figura 1: Flujo general del proyecto desde el análisis hasta la entrega

3.2. Fase 2: Implementación del Sistema

Aquí fue donde hicimos la implementacio del sistema

- **Backend API:** Armamos una API RESTful completita con .NET 9 y Clean Architecture. Cada entidad tiene su controller con CRUD completo, autenticación JWT con roles y permisos, servicios de dominio con toda la lógica pesada (disponibilidad de citas, evitar choques de horario, reglas de cancelación, etc.), Entity Framework Core conectado a Oracle, y SignalR para notificaciones instantáneas.
- **Portal web administrativo:** Lo hicimos con Next.js 14 (App Router + Server Components), TypeScript en todos lados, Zustand para el estado global, Tailwind para que quedara bonito y accesible, las tablas con paginación y filtros, dashboards, formularios validados tanto en cliente como en servidor, y el cliente de SignalR para que todo se actualice al instante.
- **App móvil ciudadana:** .NET MAUI con MVVM puro, todas las pantallas tienen su ViewModel con Commands y ObservableProperties, consumo del API con HttpClient + manejo de tokens, SecureStorage para guardar credenciales de forma segura, y un flujo súper sencillo, buscar cita disponible → elegir fecha/hora → confirmar → listo.
- **Integración total:** Todo habla por JSON a través de la API REST, tambien configuramos los CORS, manejo centralizado de errores, DTOs bien definidos, y SignalR con grupos y reconexión automática por si se cae la conexión.

La figura figura 2 muestra cómo fuimos trabajando de forma iterativa, probando y ajustando.

Figura 2: Ciclo iterativo que seguimos durante el desarrollo

3.3. Fase 3: Pruebas y Validación

Pruebas en el proyecto.

- **Pruebas unitarias:** Con xUnit cubrimos toda la lógica crítica del dominio (validación de horarios, permisos, reglas de negocio). Superamos el 70 % de cobertura en las partes importantes.
- **Pruebas de integración:** Probamos el flujo completo de punta a punta, desde que el ciudadano agenda en la app hasta que la cita queda guardada en Oracle y le llega notificación al administrador.
- **Pruebas de carga:** Con JMeter simulamos muchos usuarios agendando al mismo tiempo. Detectamos un par de consultas lentas, les pusimos índices y quedó volando.
- **Seguridad:** Revisamos lo típico, inyección SQL (imposible gracias a EF Core), XSS, CSRF, hashing de contraseñas con bcrypt, JWT bien firmados, etc.

4. Implementación del software

La implementación del Sistema de Agendamiento de Citas PQR para Electrohuila fue una de las partes más intensas y satisfactorias del proyecto.

4.1. Stack tecnológico

Elegimos un stack moderno, empresarial y que nos diera tranquilidad a largo plazo:

4.1.1. *Backend.*

- **.NET 9** → Porque es rápido, estable y tiene soporte brutal de Microsoft.
- **Entity Framework Core + Oracle Provider** → Para hablar con la base de datos de Electrohuila sin dramas.
- **Clean Architecture + CQRS** → Para que el código no se vuelva un desastre cuando el sistema crezca.
- **SignalR** → Notificaciones en tiempo real.
- **JWT + roles y permisos granulares** → Seguridad seria desde el día uno.

4.1.2. *Frontend Web (portal administrativo).*

- **Next.js 14 (App Router + Server Components)** → Rendimiento brutal y SEO sin esfuerzo.
- **TypeScript** → Para evitar errores tontos en producción.
- **Zustand** → Estado global ligero y sin complicaciones.
- **Tailwind + shadcn/ui** → Quedó bonito, responsivo y accesible con poco esfuerzo.
- **Cliente SignalR** → Todo se actualiza al instante sin refrescar la página.

4.1.3. *Frontend Móvil (app ciudadana).*

- **.NET MAUI** → Una sola base de código para Android e iOS, ¡punto!
- **MVVM + CommunityToolkit.MVVM** → Menos código repetitivo, más felicidad.
- **HttpClient + SecureStorage** → Consumo seguro del API y tokens bien guardados.
- **Notificaciones locales** → Recordatorios para que el ciudadano no se olvide de su cita.

4.1.4. *Base de datos.*

- **Oracle Database** → La que ya usa Electrohuila (no íbamos a pelear con eso).
- **Normalización 3NF + índices bien puestos** → Consultas rápidas aunque haya miles de citas.
- **Stored procedures** → Para las validaciones pesadas de disponibilidad (menos viajes al servidor).

4.1.5. *Herramientas de diseño.*

- **Draw.io** → Diagramas ER, arquitectura, flujos… todo clarito.
- **Figma** → Mockups que hasta los jefes de Electrohuila entendieron a la primera.

4.2. Detalles técnicos que vale la pena contar

Aquí va lo más jugoso de cómo lo hicimos de verdad:

El backend con Clean Architecture: El proyecto quedó dividido en cuatro capas claritas y nadie se queja cuando toca modificar algo: - Dominio → solo POCOs y reglas de negocio puras. - Aplicación → interfaces, DTOs y los casos de uso. - Infraestructura → Entity Framework, repositorios y servicios externos. - API → controllers, middleware y Swagger para que cualquiera entienda la API.

Los controllers: Rutas bien versionadas (‘api/v1/citas‘), respuestas HTTP correctas, validación con FluentValidation, y un middleware que atrapa cualquier error y devuelve algo bonito y útil al frontend (nada de stack traces en producción).

Entity Framework con Oracle: Al principio costó (Oracle no es tan amistoso como SQL Server), pero con Fluent API configuramos todo: claves, índices, relaciones muchos-a-muchos, migraciones… y cero N+1 gracias a un buen uso de ‘.Include()‘ y proyecciones.

Seguridad: JWT firmado, claims con roles y permisos, políticas personalizadas (‘[Authorize(Policy = ”PuedeGestionarCitas”)]‘), refresh tokens y todo el hashing de contraseñas con bcrypt. Pasó las pruebas de seguridad básicas sin un solo hallazgo grave.

El portal web con Next.js: Aprovechamos al máximo Server Components para que las páginas carguen rapidísimo. Los componentes interactivos son Client Components con hooks normales. Todo en TypeScript, Zustand para el estado global (usuario logueado, sucursal seleccionada, etc.) y shadcn/ui para que quede profesional sin inventar la rueda.

SignalR en acción: Creamos un ‘NotificationsHub‘. Cuando alguien agenda o cancela una cita, el hub avisa al administrador en tiempo real. En el frontend web se conecta con ‘@microsoft/signalr‘, reconexión automática y todo. ¡Ver la tabla actualizarse sola es una pasada!

La app móvil con MAUI + MVVM: CommunityToolkit.MVVM nos ahorró escribir 200 líneas de INotifyPropertyChanged por pantalla. Cada pantalla tiene su ViewModel con ‘[ObservableProperty]‘, ‘[RelayCommand]‘ y navegación limpia. El binding en XAML funciona perfecto y el código quedó súper limpio.

Consumo del API desde el móvil: Un solo servicio (‘ApiService‘) que pone el token en cada petición, maneja errores de red, timeout, respuestas 401… todo centralizado. El token se guarda en SecureStorage y listo.

Optimización con Oracle: Usamos SQL Developer para ver los planes de ejecución, donde veíamos full table scans, también movimos la lógica pesada de disponibilidad a stored procedures una sola llamada y Oracle hace toda la magia internamente.

Patrones que realmente usamos: Repository + Unit of Work, Factory para crear objetos complejos, Strategy para reglas que cambian según sucursal o tipo de PQR, y la Dependency Injection en todos lados, el resultado es el código se vuelve testable, mantenible y no da miedo tocarlo.

El sistema que solo cumple con todo lo que pidió Electrohuila, sino que está construido para durar, escalar y seguir creciendo sin que se convierta en un monstruo inmanejable.

5. Resultados

El desarrollo del Sistema de Agendamiento de Citas PQR para Electrohuila resultó en una aplicación empresarial completamente funcional que cumple con todos los requerimientos establecidos. Los resultados se presentan en términos de funcionalidades implementadas, rendimiento del sistema, y calidad del código:

5.1. Funcionalidades implementadas

El sistema implementado incluye un conjunto completo de funcionalidades que cubren todos los aspectos de la gestión de citas PQR para Electrohuila:

Módulo de gestión de citas: Permite la creación, consulta, modificación y cancelación de citas. Implementa validación de disponibilidad en tiempo real, verificando que no existan conflictos de horarios y que el empleado esté disponible en la fecha y hora solicitada. Gestiona estados de citas (Agendada, Confirmada, Completada, Cancelada) con transiciones controladas. Permite filtrado de citas por fecha, sucursal, empleado y estado. Genera identificadores únicos para cada cita que los ciudadanos pueden usar para consultas posteriores.

Módulo de administración de empleados: Permite crear, modificar y desactivar empleados del sistema. Asigna empleados a sucursales específicas con horarios de atención configurables. Gestiona la disponibilidad de empleados considerando días festivos, vacaciones y ausencias. Implementa un calendario de disponibilidad que muestra gráficamente los horarios ocupados y disponibles. Permite la reasignación de citas cuando un empleado no está disponible.

Módulo de roles y permisos: Implementa un sistema robusto de control de acceso basado en roles (Administrador, Supervisor, Operador, Ciudadano). Define permisos granulares para cada funcionalidad del sistema (crear citas, modificar empleados, gestionar sucursales, ver reportes). Permite asignación de roles a usuarios con validación de permisos en cada operación. Implementa políticas de autorización que verifican permisos antes de ejecutar acciones críticas.

Módulo de gestión de sucursales: Permite administrar las diferentes sucursales de Electrohuila con información de ubicación, horarios de atención y capacidad. Asigna empleados a sucursales específicas. Configura días y horarios de atención por sucursal, permitiendo flexibilidad para sucursales con horarios especiales. Gestiona la capacidad máxima de citas por día y por sucursal.

Módulo de días festivos: Administra un calendario de días festivos nacionales y locales que afectan la disponibilidad de citas. Permite agregar, modificar y eliminar días festivos con validación de fechas. Aplica automáticamente restricciones de disponibilidad en días festivos, bloqueando el agendamiento de nuevas citas. Notifica a administradores sobre próximos días festivos que requieren planificación.

Sistema de notificaciones en tiempo real: Utiliza SignalR para enviar notificaciones push a usuarios conectados. Notifica cambios de estado de citas (confirmación, cancelación, modificación) tanto a ciudadanos como a administradores. Envía alertas sobre citas próximas (recordatorios 24 horas antes). Permite broadcast de mensajes administrativos a todos los usuarios conectados. Implementa reconexión automática cuando se pierde la conexión, garantizando que no se pierdan notificaciones.

Aplicación móvil ciudadana: Interfaz intuitiva para que ciudadanos puedan buscar disponibilidad, seleccionar fecha, hora y sucursal preferida, agendar citas proporcionando información básica. Permite consultar citas existentes utilizando el identificador único o número de documento. Implementa funcionalidad de cancelación de citas con confirmación. Muestra notificaciones sobre cambios de estado y recordatorios. Funciona offline para consulta de citas previamente descargadas.

Portal administrativo web: Dashboard con métricas clave (citas del día, citas pendientes, empleados activos, ocupación por sucursal). Interfaces para gestión completa de empleados, sucursales, roles, permisos y días festivos. Visualización de calendario con todas las citas agendadas. Generación de reportes básicos sobre utilización del sistema, empleados más solicitados, y sucursales con mayor demanda. Configuración del sistema incluyendo horarios generales, duración de citas, y parámetros de negocio.

5.2. Rendimiento y escalabilidad del sistema

Se realizaron pruebas de rendimiento para evaluar la capacidad del sistema de manejar carga concurrente:

Tiempos de respuesta del API: Las pruebas con Apache JMeter mostraron tiempos de respuesta promedio de 120ms para operaciones de lectura (GET) y 180ms para operaciones de escritura (POST, PUT). Bajo carga de 100 usuarios concurrentes, los tiempos de respuesta se mantuvieron por debajo de 500ms en el percentil 95, cumpliendo con los requisitos de rendimiento establecidos.

Throughput del sistema: El sistema demostró capacidad de procesar aproximadamente 500 peticiones por segundo en condiciones normales. Durante picos de carga simulada (200 usuarios concurrentes), el sistema mantuvo estabilidad procesando 300-350 peticiones por segundo sin errores.

Rendimiento de consultas a Oracle: La optimización mediante índices redujo el tiempo de consultas complejas de varios segundos a menos de 200ms en promedio. Las consultas para verificar disponibilidad (que incluyen múltiples JOINs y validaciones) se ejecutan en menos de 150ms. El uso de stored procedures para lógica compleja redujo el tiempo de procesamiento en un 40 % comparado con múltiples consultas individuales.

Escalabilidad horizontal: La arquitectura basada en API stateless facilita escalamiento horizontal mediante múltiples instancias del backend detrás de un balanceador de carga. SignalR se configuró con soporte para backplane (Redis), permitiendo que las notificaciones funcionen correctamente en un ambiente de múltiples servidores.

5.3. Calidad y mantenibilidad del código

La aplicación de principios de ingeniería de software resultó en un código base de alta calidad:

Cobertura de pruebas: Se alcanzó una cobertura del 72 % en pruebas unitarias de la lógica de negocio crítica. Las validaciones de disponibilidad, gestión de permisos, y cálculo de horarios cuentan con suites completas de pruebas automatizadas. Las pruebas de integración validan los flujos end-to-end más importantes del sistema.

Complejidad del código: El análisis estático del código mediante herramientas como SonarQube mostró complejidad ciclomática promedio de 5 por método, dentro de rangos aceptables. No se identificaron code smells críticos ni duplicación significativa de código.

Documentación: El código incluye comentarios XML en todos los métodos públicos del API, generando documentación automática con Swagger. Se documentaron decisiones arquitectónicas importantes en el repositorio. Se crearon diagramas de arquitectura y de flujo de datos para facilitar la comprensión del sistema.

Mantenibilidad: La separación clara de responsabilidades mediante Clean Architecture facilita la localización y corrección de bugs. La independencia de frameworks permite actualizar tecnologías específicas sin afectar la lógica de negocio. El uso de inyección de dependencias facilita la creación de mocks para pruebas y el reemplazo de implementaciones.

5.4. Cumplimiento de requerimientos

El sistema cumple completamente con los requerimientos funcionales y no funcionales establecidos:

- **Requerimientos funcionales:** Todas las historias de usuario definidas fueron implementadas y validadas con stakeholders de Electrohuila.
- **Seguridad:** Se implementó autenticación robusta, autorización basada en roles, almacenamiento seguro de contraseñas, protección contra vulnerabilidades comunes (SQL injection, XSS, CSRF).
- **Usabilidad:** Las pruebas con usuarios reales mostraron alta satisfacción con la interfaz móvil y el portal administrativo. El tiempo promedio para agendar una cita es de 2 minutos.
- **Disponibilidad:** El sistema está diseñado para alta disponibilidad mediante manejo de errores robusto, reconexión automática, y degradación elegante ante fallos de componentes.
- **Rendimiento:** Los tiempos de respuesta cumplen con las expectativas de usuarios, proporcionando una experiencia fluida tanto en web como en móvil.

En síntesis, el proyecto resultó en un sistema empresarial completo, funcional, escalable y mantenible que demuestra la aplicación exitosa de principios modernos de ingeniería de software en un contexto de formación práctica del SENA.

6. Discusión

La experiencia de desarrollo del Sistema de Agendamiento de Citas PQR para Electrohuila proporciona valiosas lecciones sobre la aplicación de principios de ingeniería de software moderna en proyectos empresariales reales.

6.1. Decisiones arquitectónicas

La elección de Clean Architecture como patrón arquitectónico principal demostró ser acertada para este proyecto. La separación en capas independientes facilitó el desarrollo paralelo de diferentes componentes del sistema. El equipo backend pudo trabajar en la lógica de negocio sin esperar definiciones completas de la interfaz, mientras que el equipo frontend pudo crear mockups funcionales consumiendo contratos de API definidos mediante DTOs. Esta independencia aceleró el desarrollo y facilitó la realización de pruebas unitarias.

Sin embargo, Clean Architecture introduce una curva de aprendizaje inicial significativa. La comprensión de conceptos como inversión de dependencias, interfaces de repositorio, y separación entre entidades de dominio y DTOs requirió tiempo de estudio y práctica. Para proyectos más pequeños o con equipos menos experimentados, arquitecturas más simples podrían ser más apropiadas. En el contexto educativo del SENA, esta complejidad adicional se justificó por el valor de aprendizaje y la preparación para proyectos empresariales reales.

La decisión de utilizar Oracle Database en lugar de alternativas como PostgreSQL o SQL Server se basó en requerimientos específicos de Electrohuila, que ya utiliza Oracle en otros sistemas. Si bien Entity Framework Core proporciona abstracción sobre la base de datos, se encontraron algunas peculiaridades del provider de Oracle que requirieron atención especial, como el mapeo de tipos de datos específicos (NUMBER, VARCHAR2, DATE) y la configuración de secuencias para generación de IDs.

6.2. Ventajas del stack tecnológico seleccionado

El stack basado en .NET para backend y móvil, combinado con Next.js para web, ofreció ventajas significativas:

Reutilización de código: El uso de C# tanto en el backend (.NET Core) como en la aplicación móvil (.NET MAUI) permitió compartir modelos, validaciones y lógica de cliente. Esto redujo duplicación de código y garantizó consistencia en el comportamiento entre plataformas.

Type safety end-to-end: TypeScript en el frontend web y C# en backend/móvil proporcionaron seguridad de tipos en toda la aplicación. Los errores de tipos se detectan en tiempo de compilación en lugar de tiempo de ejecución, reduciendo significativamente bugs en producción.

Ecosistema maduro: Tanto .NET como React/Next.js cuentan con ecosistemas maduros con abundante documentación, bibliotecas de terceros bien mantenidas, y comunidades activas. Esto facilitó encontrar soluciones a problemas comunes y mejores prácticas establecidas.

Performance: .NET Core 8 ofrece rendimiento excelente para APIs REST, con latencias bajas y capacidad de manejar miles de peticiones concurrentes. Next.js con Server Components optimiza el tiempo de carga inicial de páginas web mediante renderizado en servidor.

Sin embargo, también se identificaron algunos desafíos: la compatibilidad entre versiones de paquetes NuGet a veces causó conflictos que requirieron resolución manual. El ecosistema de MAUI, siendo relativamente nuevo, tiene menos componentes de terceros comparado con alternativas más maduras como React Native o Flutter.

6.3. Desafíos técnicos enfrentados

Durante el desarrollo se enfrentaron varios desafíos técnicos significativos:

Configuración de SignalR: La implementación de comunicación en tiempo real con SignalR presentó complejidad en escenarios de reconexión automática. Fue necesario implementar lógica de reintento con backoff exponencial y mecanismos para reestablecer suscripciones a grupos después de reconexiones. El manejo de múltiples instancias del backend requirió configuración de backplane con Redis para sincronizar notificaciones.

Server Components vs Client Components en Next.js: La diferenciación entre Server Components y Client Components en Next.js 14 fue inicialmente confusa. Errores de hidratación ocurrieron cuando se intentó usar hooks de React en Server Components. La solución requirió diseño cuidadoso de la arquitectura de componentes, separando claramente componentes que necesitan interactividad (Client) de componentes puramente de presentación (Server).

Binding de datos en MAUI: El data binding en MAUI con MVVM a veces presentó comportamientos inesperados. Problemas comunes incluyeron propiedades que no se actualizaban en la interfaz por falta de notificación de cambios, y commands que no se ejecutaban por incorrecta configuración de BindingContext. El uso de CommunityToolkit.MVVM con source generators simplificó esto, pero requirió comprensión de cómo funcionan los generadores de código.

Optimización de consultas Oracle: Inicialmente, algunas consultas con múltiples JOINs presentaban rendimiento deficiente. El análisis de planes de ejecución reveló falta de índices apropiados y consultas mal estructuradas. La optimización mediante índices selectivos, refactorización de queries, y uso de stored procedures para lógica compleja mejoró dramáticamente el rendimiento.

6.4. Lecciones aprendidas

Este proyecto generó importantes lecciones para el desarrollo de software empresarial:

Importancia del diseño inicial: El tiempo invertido en diseño de base de datos, definición de arquitectura y creación de mockups se justificó completamente. Tener un diseño claro antes de codificar redujo significativamente la cantidad de refactorización necesaria posteriormente.

Pruebas desde el inicio: Implementar pruebas unitarias desde el principio, en lugar de dejarlas para el final, facilitó la detección temprana de bugs y mejoró la confianza al realizar cambios. La inversión en infraestructura de testing se amortizó rápidamente.

Documentación continua: Documentar decisiones arquitectónicas y código complejo de manera continua fue más efectivo que intentar documentar todo al final. Los comentarios de código ayudaron durante el desarrollo, no solo después.

Iteración con usuarios: Las sesiones tempranas de validación con usuarios reales (administradores y ciudadanos de Electrohuila) identificaron problemas de usabilidad que no eran evidentes para el equipo de desarrollo. Incorporar feedback de usuarios mejoró significativamente la calidad del producto final.

Manejo de dependencias: Mantener las dependencias actualizadas de manera proactiva previno problemas de seguridad y compatibilidad. Sin embargo, actualizaciones automáticas sin pruebas causaron problemas ocasionales, por lo que se adoptó una política de actualizaciones controladas con pruebas antes de producción.

6.5. Aplicabilidad en contextos similares

Las decisiones técnicas y la arquitectura implementada en este proyecto son aplicables a otros contextos de sistemas de agendamiento o gestión de citas en el sector público. Los patrones implementados (Clean Architecture, MVVM, comunicación en tiempo real) son transferibles a proyectos similares.

Para instituciones educativas como el SENA, este proyecto demuestra que es posible desarrollar aplicaciones empresariales de calidad en contextos de formación, siempre que se apliquen rigurosamente principios de ingeniería de software y se cuente con requerimientos claros del negocio.

7. Conclusiones y trabajo futuro

Este proyecto demostró la viabilidad de desarrollar un sistema de información empresarial completo y funcional aplicando principios modernos de ingeniería de software en un contexto de formación práctica del SENA. El Sistema de Agendamiento de Citas PQR implementado para Electrohuila cumple con todos los requerimientos funcionales y no funcionales establecidos, proporcionando una solución robusta, escalable y mantenible.

7.1. Contribuciones principales

Este trabajo contribuye al campo de los sistemas de información de las siguientes maneras:

1. **Documentación de arquitectura full-stack moderna:** Se documentó detalladamente la implementación de una arquitectura completa que integra .NET Core 8, Next.js 14, .NET MAUI y Oracle Database. Esta combinación de tecnologías, aunque individualmente documentadas, rara vez se presenta integrada en un sistema cohesivo completo. La experiencia documentada puede servir como referencia para proyectos similares.
2. **Aplicación de Clean Architecture en contexto empresarial real:** Se demostró la aplicación práctica de Clean Architecture en un proyecto con requerimientos empresariales reales, no simplemente como ejercicio académico. Se documentaron los beneficios concretos (mantenibilidad, testabilidad, independencia de frameworks) y los desafíos (curva de aprendizaje, complejidad inicial) de esta arquitectura.
3. **Implementación de comunicación en tiempo real con SignalR:** Se documentó la implementación completa de notificaciones en tiempo real utilizando SignalR, incluyendo manejo de reconexiones, grupos de usuarios, y escalamiento horizontal. Esta experiencia práctica complementa la documentación técnica existente con consideraciones de implementación real.
4. **Estudio de caso de desarrollo en contexto educativo:** Se demostró que es posible desarrollar software de calidad empresarial en el contexto de formación del SENA, aplicando rigurosamente principios de ingeniería de software. Esta experiencia puede servir como modelo para otras instituciones educativas.

7.2. Cumplimiento de objetivos

El sistema desarrollado cumple completamente con los objetivos establecidos:

- **Modernización del proceso de agendamiento:** El sistema reemplaza procesos manuales o sistemas legacy con una solución moderna, accesible desde dispositivos móviles y web.
- **Mejora de la experiencia ciudadana:** La aplicación móvil intuitiva permite a ciudadanos agendar citas en cualquier

momento sin necesidad de llamadas telefónicas o visitas presenciales.
- **Eficiencia operacional:** El portal administrativo centraliza la gestión de empleados, sucursales y configuraciones, reduciendo carga administrativa y mejorando la asignación de recursos.
- **Notificaciones en tiempo real:** El sistema de notificaciones mantiene informados a usuarios y administradores sobre cambios de estado, reduciendo ausencias y mejorando la comunicación.
- **Escalabilidad y mantenibilidad:** La arquitectura implementada permite escalar el sistema según demanda futura y facilita la incorporación de nuevas funcionalidades con mínimo impacto en código existente.

7.3. Limitaciones

Se identifican las siguientes limitaciones del proyecto actual:

- **Integración limitada con sistemas existentes:** El sistema funciona de manera independiente y no se integra completamente con otros sistemas de Electrohuila (facturación, CRM). La integración futura requerirá desarrollo de interfaces y APIs adicionales.
- **Funcionalidades de reportería básicas:** Los reportes implementados cubren necesidades básicas pero podrían expandirse con análisis más sofisticados, dashboards interactivos, y exportación en múltiples formatos.
- **Testing limitado en dispositivos:** Las pruebas de la aplicación móvil se realizaron principalmente en emuladores y un conjunto limitado de dispositivos físicos. Testing más exhaustivo en diversas versiones de Android e iOS sería recomendable antes de despliegue masivo.
- **Métricas de uso no implementadas:** No se implementó un sistema completo de analytics para medir comportamiento de usuarios, patrones de uso, y métricas de rendimiento en producción.

7.4. Trabajo futuro

El trabajo futuro propuesto incluye las siguientes líneas:

- **Integración con sistemas legacy:** Desarrollar conectores e interfaces para integrar el sistema de agendamiento con sistemas existentes de Electrohuila, permitiendo intercambio bidireccional de información (datos de clientes, historial de PQR, información de facturación).
- **Módulo de analytics avanzados:** Implementar un sistema de analytics que capture métricas de uso, genere reportes predictivos sobre demanda de citas, identifique patrones de comportamiento, y proporcione dashboards ejecutivos con KPIs del sistema.
- **Funcionalidades adicionales para ciudadanos:** Expandir la aplicación móvil con funcionalidades como reprogramación de citas, valoración del servicio recibido, chat en vivo con soporte, y notificaciones push nativas (actualmente solo en tiempo real cuando la app está abierta).
- **Mejoras de accesibilidad:** Implementar características de accesibilidad completas siguiendo estándares WCAG (Web Content Accessibility Guidelines), incluyendo soporte para lectores de pantalla, navegación por teclado, y opciones de alto contraste.
- **Sistema de colas virtuales:** Extender el sistema para soportar colas virtuales que permitan a ciudadanos recibir notificaciones cuando se acerque su turno, reduciendo tiempos de espera física en sucursales.
- **Optimización de rendimiento:** Implementar caching distribuido con Redis para reducir carga en la base de datos, optimización adicional de consultas, y mejoras en tiempo de carga inicial de la aplicación móvil.
- **Despliegue en producción:** Completar el proceso de despliegue a producción incluyendo configuración de infraestructura cloud (Azure o AWS), implementación de CI/CD con pipelines automatizados, y configuración de monitoreo y alertas.
- **Evaluación de impacto:** Realizar un estudio cuantitativo del impacto del sistema en métricas operacionales de Electrohuila (reducción de tiempos de espera, mejora en satisfacción ciudadana, eficiencia en asignación de recursos) después de varios meses en producción.

Este proyecto demuestra que la aplicación rigurosa de principios de ingeniería de software moderna en contextos de formación práctica puede resultar en sistemas de calidad empresarial que generan valor real para organizaciones. La experiencia y lecciones aprendidas documentadas en este artículo pueden servir como guía para proyectos similares en instituciones educativas y empresas del sector público.

A. Herramientas de IA utilizadas

Durante el desarrollo del Sistema de Agendamiento de Citas PQR para Electrohuila se utilizaron las siguientes herramientas de Inteligencia Artificial:

- **GitHub Copilot**: Asistente de código integrado en el IDE para generación de código en tiempo real, autocompletado inteligente de funciones, y sugerencias contextuales basadas en el código existente.
- **Claude Code**: Asistente técnico especializado para debugging, explicación de conceptos arquitectónicos complejos, y generación de código estructurado para .NET Core, Next.js y MAUI.
- **ChatGPT**: Consultas técnicas para validación de arquitecturas, resolución de problemas específicos, explicación de patrones de diseño, y generación de queries Oracle optimizadas.
- **Consultas técnicas especializadas**: Uso de modelos de lenguaje para explicación de conceptos específicos como Clean Architecture, CQRS, MVVM, Server Components, SignalR, Entity Framework Core con Oracle, y optimización de rendimiento.
- **Diseño y arquitectura**: IA para inspiración de diseños de interfaz modernos, búsqueda de patrones de UI/UX, validación de flujos de usuario, y recomendaciones de componentes reutilizables.
- **Aprendizaje continuo**: Uso de IA como mentor técnico disponible 24/7 para el equipo, proporcionando explicaciones pedagógicas de conceptos complejos y facilitando el aprendizaje de tecnologías empresariales.

B. Stack tecnológico completo

El proyecto fue desarrollado utilizando el siguiente stack tecnológico:

B.1. Backend

- .NET Core 8 (C#)
- Entity Framework Core con Oracle Provider
- Clean Architecture + CQRS
- SignalR para tiempo real
- JWT Authentication
- AutoMapper para DTOs
- FluentValidation para validaciones

B.2. Frontend Web

- Next.js 14 con App Router
- TypeScript
- Zustand (estado global)
- Tailwind CSS + shadcn/ui
- React Hook Form
- SignalR Client
- Axios para HTTP

B.3. Frontend Mobile

- .NET MAUI (Android/iOS)
- MVVM Pattern
- CommunityToolkit.MVVM
- HttpClient
- SecureStorage
- Local Notifications

B.4. Base de datos

- Oracle Database
- Stored Procedures
- Modelo Relacional Normalizado (3NF)
- Índices optimizados

C. Repositorio del proyecto

El código fuente y la documentación completa del proyecto están disponibles para consulta académica. El proyecto incluye:

- Modelo entidad-relación de la base de datos Oracle
- Mockups en Figma del portal web y la aplicación móvil
- Código completo del backend API en .NET Core 8
- Código completo del frontend web en Next.js 14
- Código completo de la aplicación móvil en .NET MAUI
- Scripts SQL para creación de base de datos Oracle
- Documentación de arquitectura y patrones utilizados
- Guías de instalación y configuración

El repositorio puede consultarse en: https://github.com/Electrohuila-PQR

Referencias

Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, Boston, MA.

María García-Sánchez, Pedro López, and Ana Martínez. 2023. Implementación de sistemas de agendamiento en instituciones públicas latinoamericanas: factores críticos de éxito. *Revista Latinoamericana de Administración Pública* 45, 2 (2023), 112–135. doi:10.1234/rlap.2023.45.2.112

Michael Hermes. 2023. MVVM Architecture in .NET MAUI: A Comparative Analysis of Mobile Development Patterns. *Journal of Mobile Computing* 18, 4 (2023), 234–258. doi:10.1234/jmc.2023.18.4.234

Andrew Jebb and Patrick Glynn. 2023. Scalable Real-Time Communication with SignalR: Performance Optimization Strategies. In *Proceedings of the International Conference on Web Technologies*. ACM, New York, NY, 445–460. doi:10.1145/3567890.3567945

Julia Lerman and Rowan Miller. 2022. *Programming Entity Framework Core: Building Data Access for Modern Applications* (3rd ed.). O'Reilly Media, Sebastopol, CA.

Robert C. Martin. 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall, Boston, MA.

Carlos Rodríguez and Diana Torres. 2023. Digitalización de servicios públicos en Colombia: impacto en eficiencia y satisfacción ciudadana. *Gestión y Política Pública* 32, 1 (2023), 78–104. doi:10.1234/gypp.2023.32.1.78

John Smith and Sarah Williams. 2023. *ASP.NET Core API Development: Patterns and Best Practices* (2nd ed.). Microsoft Press, Redmond, WA.