UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

DEPARTAMENTO DE ELECTROTECNIA E INFORMÁTICA
INGENIERÍA EN INFORMÁTICA

Desarrollo del Back-End para un sistema de gestión de visitas para un recinto hospitalario

Christián Nicolas Rojas San Martín

christian.rojass@usm.cl

Dagoberto Cabrera
Profesor Guía

Diego Caceres
Profesor Correferente

Resumen: Este proyecto forma parte de VisitaConnect, una solución web desarrollada para poder optimizar y digitalizar todo el proceso de gestión de visitas en el Hospital Gustavo Fricke. Esta solución permite agendar visitas, generar QR únicos que facilitan el acceso y la verificación de identidad por parte del personal de seguridad. El sistema fue desarrollado con ASP.NET Core y utiliza una base de datos relacional. El back-end, que corresponde al foco principal de esta tesina, actúa como un eje central para que se puedan comunicar los módulos del sistema. Las acciones realizadas por los usuarios se registran en el sistema, lo que permite mejorar la trazabilidad de la información. Utilizando una metodología ágil Scrum, se busca mejorar la organización interna del hospital, con ello, reducir tiempos administrativos y brindar una experiencia más eficiente y segura tanto para el personal como para los visitantes, todo esto dentro del proceso de las visitas.

Palabras Clave: ASP.NET Core, gestión de visitas, eficiencia, Base de datos, seguridad

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: Desarrollo del Back-End para un sistema de gestión de visitas para un recinto hospitalario

Nombre del candidato(a): Christian Nicolas Rojas San Martín

Carrera / Grado: Ingeniería en Informática

Campus: Sede Viña del Mar **Departamento:** Departamento de electrónica e informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Dagoberto Cabrera Tapia, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

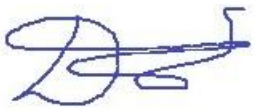
☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por **(marcar una opción)**:

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 28-05-2026 **Firma:** 

Estudiante o Candidato(a):

Fecha: 28-05-2026 **Firma:** 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

1 Introducción.

1.1 Contexto

Actualmente, la gestión de visitas en el Hospital Gustavo Fricke presenta diferentes dificultades. Al momento de ingresar al centro hospitalario con la intención de visitar a un paciente, la persona debe dirigirse al mesón administrativo que se encuentra ubicado en la entrada. En ese lugar, el personal administrativo, que se encuentra realizando diversas labores, debe interrumpir sus funciones para poder rellenar junto al visitante una ficha en papel con los datos necesarios para realizar la visita, todo esto realizado a mano.

Una vez finalizado el registro, el funcionario debe explicar al visitante indicaciones generales relacionadas a la visita. La información contenida en el formulario en papel debe ser comunicada a las distintas áreas involucradas (personal clínico, personal de seguridad y administrativo), lo que genera un proceso lento, poco eficiente y propenso a errores. Todo ese proceso dificulta la trazabilidad de la información e incrementa la carga de trabajo del personal administrativo y clínico.

Por último, también está el factor de la seguridad, ya que, no hay control de identidad al momento de autorizar el ingreso de visitantes, lo que hace todo este proceso inseguro.

1.2 Definición del problema

El proyecto aborda diversas problemáticas presentes en la gestión de visitas, desde la solicitud de agendamiento hasta la realización efectiva de la visita. Los problemas abordados en este proyecto son principalmente los siguientes:

1. **Deficiencia en la seguridad:** La seguridad en el ámbito del proceso de visitas carece de un control de acceso con validación de identidad, lo que permite que cualquier persona pueda ingresar como visitante, sin mayor dificultad, lo que representa un riesgo para los pacientes y funcionarios del hospital.

2. **Desinformación:** Actualmente, las personas no se enteran si se realizan cambios respecto a las visitas, por ello el personal administrativo debe informar cada vez que entra una persona, cómo funciona el sistema o si ha cambiado algo como, por ejemplo, los horarios de visitas, cantidad de personas que pueden entrar, etc.

3. **Sobrecarga de trabajo:** Tanto el personal administrativo como el personal clínico cumplen diversas funciones dentro del hospital. Entre ellas se encuentra la gestión de visitas, actividad que implica registrar datos, coordinar información y comunicarla a las áreas correspondientes, esto incrementa la carga operativa y el tiempo de atención, especialmente en momentos de alta afluencia de visitantes, y en cuanto al clínico deben de recibir la información de esto y hacer llegar de igual manera a todos los que necesiten saber acerca de la visita.

4. **Deficiente en la trazabilidad de la información:** El uso del papel físico como ficha para rellenar datos, hace que la información tenga un flujo lento, desde el momento que se termina de llenar el formulario, se debe hacer llegar a bastantes personas para continuar con el proceso de la visita, y el problema sigue persistiendo mientras avance el proceso de la visita, ya que, se debe hacer un seguimiento de la persona y el personal administrativo debe estar asegurándose

constantemente, preguntando, si la persona llegó con el paciente, si la persona sigue con el paciente o si la visita se ha ido.

1.3 Propuesta de solución

La solución consiste en una plataforma web donde se realice el proceso de visita, esta plataforma está diseñada para ser accesible desde diferentes dispositivos.

Desde el punto de vista del desarrollo, este debe implementar las funcionalidades y la lógica detrás de todo el proceso de la visita de manera eficaz, lo que incluye un formulario virtual donde se debe rellenar los datos de la visita y al paciente que quiere visitar. Asimismo, esta plataforma tendrá un calendario junto a bloques horarios disponibles, para poder organizar y reservar visitas de forma eficiente.

Una vez ya confirmada la solicitud, se enviará de forma automática un correo electrónico de confirmación que incluirá las indicaciones generales de la visita y un código QR adjunto. Este código también se mostrará en la página web y es lo que se usará para poder ingresar a la visita, ya que esto permitirá apoyar el control de acceso del visitante junto a una verificación del documento de identidad por parte del guardia.

La plataforma también tendrá funcionalidades específicas según roles, que son los siguientes:

1. **Clínico:** Podrá autenticarse con sus credenciales y gestionar el estado de los pacientes

2. **Guardia:** Podrá autenticarse con sus credenciales, contará con acceso a un lector de códigos QR para poder verificar la identidad de los visitantes y autorizar el ingreso.

1.4 Objetivos generales y específicos de la tesina

El objetivo general es desarrollar una plataforma web que optimice la gestión de visitas en el Hospital Gustavo Fricke, mediante la digitalización del registro, la mejora de la trazabilidad de la información, la reducción de la carga administrativa y clínica, y el apoyo al control de acceso mediante códigos QR.

Para poder alcanzar el objetivo general, se definieron los siguientes objetivos específicos:

- Digitalizar el registro de las visitas a través de un formulario virtual
- Implementar un calendario junto a bloques de horarios
- Implementar una generación y validación de códigos QR
- Desarrollar los roles para el personal clínico y guardia
- Implementar el cambio de estado del paciente para el personal clínico
- Diseñar e implementar la base de datos relacional
- Automatizar el envío de correos electrónicos de confirmación
- Implementar la cancelación de una hora agendada

1.5 Justificación del proyecto

Entre los beneficios esperados para los usuarios se encuentran, la automatización del control de visitas, mayor seguridad para el acceso a la visita, y, en general, una mejor experiencia tanto para el personal que trabaja dentro del hospital como para los visitantes.

Desde el punto de vista de la relevancia técnica, este proyecto es importante porque implica varios conceptos distintos para el desarrollo back-end, que se puede llevar a cabo con diversas tecnologías y herramientas modernas que facilitan la creación de sistemas web robustos. Su implementación en el contexto del proyecto tiene como fin optimizar la gestión de visitas, mejorando la eficiencia del proceso y lograr que sea más segura.

En el contexto del trabajo las tecnologías usadas ponen en práctica los conceptos como el diseño de base de datos, conexión con esta, la autenticación de usuarios, generación y lectura de códigos QR, etc. Por ello, es una gran oportunidad para desarrollar una solución funcional con un impacto real. Por todo lo mencionado anteriormente, el desarrollo de este proyecto, y por ello también el del back-end, aporta un valor real para el visitante, como el de optimizar su tiempo y de igual manera para los funcionarios del hospital, mejorando la eficiencia y el control dentro del establecimiento.

1.6 Metodología

Para el desarrollo del proyecto se decidió utilizar la metodología ágil Scrum, debido a que tiene un enfoque flexible, iterativo e incremental, lo que la hace ideal para proyectos de software robustos y con una gran cantidad de funcionalidades, como el sistema de gestión de visitas que estamos desarrollando.

A diferencia de metodologías tradicionales, rígidas, como el modelo en cascada, Scrum no se basa en una secuencia lineal de fases cerradas. En el modelo en cascada, existen las etapas de análisis, diseño, implementación y pruebas donde se deben completarse totalmente antes de avanzar a la siguiente, lo que genera una rigidez considerable. Si durante el proceso surgen cambios en los requerimientos, se debe retroceder a fases anteriores, aumentando significativamente el tiempo requerido para realizar correcciones

Scrum, en cambio, propone un enfoque iterativo e incremental, en el cual el desarrollo se organiza en ciclos cortos y repetitivos llamados sprints. Cada sprint permite desarrollar, revisar y entregar un incremento funcional del producto, el cual puede ser analizado y ajustado en función de la retroalimentación recibida. Esto proporciona una mayor flexibilidad ante cambios de requerimientos.

En cuanto a la comparación con otras metodologías ágiles, como Kanban, la elección de Scrum se justifica porque Kanban funciona con un flujo de trabajo continuo mediante tableros que muestran el estado de las tareas (pendiente, en progreso, finalizado). Esto lo hace útil para procesos con tareas pequeñas o de mantenimiento, no establece ciclos de trabajo ni objetivos con plazos definidos, lo que dificulta en gran medida la planificación y el control del progreso en proyectos de mayor complejidad, como lo es el presente proyecto. Por el contrario, Scrum estructura el trabajo en sprints con objetivos y tiempos delimitados, lo que permite una mejor organización de las funcionalidades, entregables claros y un control constante del avance. Además, hay validación continua

mediante reuniones periódicas y revisiones, para poder confirmar que el proceso de desarrollo se está realizando de manera correcta.

- Sprint 1: El objetivo del primer sprint fue realizar toda la lógica y funcionalidades del agendamiento de visitas, este objetivo tenía un tiempo límite, pero para poder completarlo se debía realizar varias tareas las cuáles fueron, diseñar e implementar la base de datos relacional, con esa base poder implementar el formulario donde se registra la visita, que serán guardadas en la base de datos donde también estarán los datos de los pacientes. Luego poder implementar el calendario junto a los bloques horarios, los cuales se almacenan en la base de datos para gestionar su disponibilidad y evitar duplicidad en las reservas. Después se necesita que se muestre un resumen con todos los datos que fueron ingresados, esto para que pueda verificar si hay algo erróneo. Para finalizar, la última tarea, se debía implementar el envío de correo automático una vez ya confirmada la visita.

- Sprint 2: El objetivo del segundo sprint fue implementar toda la lógica y funcionalidades para el personal clínico y de seguridad. Para poder llevarlo a cabo se puso un tiempo límite y se dividió en diferentes tareas, las les fueron, implementar una autenticación para el personal clínico y de seguridad con credenciales distintas para ambos, para ello se tuvo que implementar dos roles, una vez hecho eso las siguientes tareas fueron, implementar el cambio de estado para el paciente, esto a través del personal clínico, el cual consiste en que si el paciente está apto para recibir visitas, no recibir visitas, recibir visitas de ciertas personas o recibir visitas, pero a excepción de ciertas personas en específico. Por último, las tareas relacionadas al guardia, donde fue poder implementar la lectura de códigos QR y una vez leído el código este retorne los datos del visitante y algunos del paciente, como la sala donde se encuentra.

- Sprint 3: Tuvo como objetivo realizar ajustes, correcciones y mejoras sobre lo desarrollado en las etapas anteriores. En este sprint se pulieron detalles de funcionalidad y diseño, se optimizaron procesos existentes y se incorporaron validaciones adicionales, como la verificación del RUT del visitante, con el fin de asegurar la integridad de los datos registrados.

1.7 Breve descripción de la organización del informe en capítulos

Este informe se divide en capítulos, los cuales cada uno de estos se centra en algún aspecto específico del proyecto, por lo que ahora, veremos una breve descripción del contenido de cada uno de estos capítulos:

- Capítulo 1: introducción
  En este capítulo se presenta todo lo necesario para comprender el problema y como lo abordamos con nuestra propuesta de solución, mencionando los objetivos generales y específicos del proyecto. Acompañado de la justificación de este y qué metodología fue utilizada y el porqué.

- Capítulo 2: Marco teórico
  Este capítulo se enfoca en los fundamentos teóricos que sustentan el desarrollo del back-end del sistema. Donde se explica qué es el back-end, cuál es su rol dentro de la arquitectura del software y, por último, tecnologías y frameworks que fueron utilizados.

- Capítulo 3: Diseño e implementación
  Este capítulo incluye la definición del diseño, arquitectura y estructura de los componentes, además de las decisiones técnicas que guiaron el desarrollo. Donde se justifica el uso de tecnologías, diseño de la interfaz, detalles de la codificación y desarrollo y, por último, la explicación de cómo se probó y validó

- Capítulo 4: Conclusiones
  En este capítulo se presentan las conclusiones del proyecto, y hacer una retroalimentación personal, identificando cuáles fueron las dificultades en el desarrollo, logros importantes, etc.

- Capítulo 5: Agradecimientos
  Este capítulo se enfoca en agradecer a quienes colaboraron con la realización de este trabajo.

- Capítulo 6: Referencias
  En este capítulo se adjuntan las fuentes y documentos citados en el informe en formato IEEE.

- Capítulo 7: Anexos
  Este capítulo corresponde al material complementario, que no forma parte del contenido de los capítulos, pero ayudan a documentar mejor el trabajo realizado

2 Marco teórico

2.1 Fundamentos del desarrollo back-end

El desarrollo back-end es la parte del software que se encarga de gestionar la lógica interna, el procesamiento de datos, la comunicación con la base de datos y la interacción con los servicios externos. A diferencia del front-end, que se encarga de la interfaz visible para el usuario, el back-end se enfoca en todo aquello que ocurre "detrás de escena", permitiendo que las acciones del usuario, con el software, se traduzcan en resultados funcionales dentro del sistema [1] por ello, es esencial en la construcción del software moderno, porque, actúa como el núcleo operativo que garantiza la correcta ejecución de las funciones del sistema, la seguridad de los datos, la eficiencia en el procesamiento de la información y la escalabilidad. Sin la capa de back-end sólida, las aplicaciones carecerían de la integridad de los datos, persistencia, autenticación y la capacidad de poder interactuar con otros servicios o dispositivos. [2]

El concepto de back-end surgió junto con las primeras arquitecturas llamadas cliente-servidor en los años 1970 y 1980, cuando los sistemas comenzaron a dividir sus componentes entre un cliente y un servidor. En el momento en el que llega Internet en los años 90, el back-end debido a esto tomó un papel fundamental en las aplicaciones web dinámicas, con tecnologías como PHP, Java EE y .NET Framework, que permitieron poder generar contenido en tiempo real [3]. Actualmente, el desarrollo back-end ha evolucionado a entornos más flexibles y modulares, esto gracias a tecnologías como Node.js, Python, Ruby on Rails, Go, y ASP.NET Core, las cuales permiten construir API RESTful. [4]

El back-end se implementa en una gran variedad de sistemas, como en nuestro caso, aplicaciones web, pero también en aplicaciones móviles, sistemas híbridos, plataformas de escritorio con conexión a la nube, servicios IoT, y sistemas empresariales de gran

escala. En todos estos casos mencionados, como se mencionó anteriormente, el back-end cumple funciones esenciales. [1]

Entre las principales características del desarrollo back-end se destacan:

- Persistencia de datos, mediante bases de datos relacionales (como MySQL, PostgreSQL, SQL Server) o no relacionales (como MongoDB o Redis).
- Seguridad y autenticación, protegiendo el acceso y la integridad de los datos mediante protocolos como JWT, OAuth2 o HTTPS [2]. Para el caso de este proyecto, esto se materializa gracias ASP.NET Core Identity con autenticación por cookies y cifrado HTTPS.
- Eficiencia en el procesamiento, garantizando respuestas rápidas ante solicitudes concurrentes.
- Escalabilidad, permitiendo que el sistema crezca sin perder rendimiento.
- Modularidad, gracias a la separación de componentes en servicios independientes o microservicios. [3]

El desarrollo back-end tiene varias ventajas, pero las principales son la seguridad, la organización estructurada del sistema, la capacidad de poder integrarse múltiples plataformas, y la escalabilidad a largo plazo, pero también tiene sus desventajas que son la complejidad técnica, la necesidad de servidores y el continuo mantenimiento, además depende de configuraciones adecuadas para poder garantizar un rendimiento óptimo. [1]

Actualmente, dentro de una arquitectura de software moderna, el back-end funciona como el intermediario entre el usuario y los recursos del sistema. Su función principal es recibir todas las solicitudes del usuario a través de la interfaz, procesarlas, interactuar con la base de datos o servicios externos, y devolver una respuesta en un formato interpretable como lo son el formato JSON o XML. [4]

En el contexto de este proyecto, el back-end concentra la lógica de reservas, validaciones de visita, autenticación por roles, persistencia de datos, generación de códigos QR y envío de correos de confirmación.

2.2 Arquitectura de software

El sistema se diseñó con una arquitectura MVC (Model-View-Controller), la cual organiza la aplicación en distintas capas independientes, cada una tiene su función y son las siguientes:

- Vistas: Esta capa es la encargada de mostrar toda la información a los usuarios (interfaz), y así el sistema poder recibir sus interacciones.
- Controladores: Esta capa se encarga de gestionar las solicitudes HTTP y de coordinar las acciones según las interacciones del usuario. En este proyecto, una parte importante de la lógica del flujo de reservas se mantuvo centralizada en los controladores, especialmente en el archivo llamado, ReservasController

- Modelos: Esta capa está compuesta por las distintas tablas como lo son la tabla Paciente, Reserva, Usuario, Sala, HorarioSala, entre otras. Estas clases se gestionan por Entity Framework Core a través de ApplicationDbContext, que actúa como capa de acceso de datos.

Además, es importante aclarar que, el sistema incorpora componentes de apoyo para ciertas responsabilidades específicas, como lo son VisitaPolicyService, SlotHelper y el servicio de envío de correos. Estos nos permiten desacoplar ciertas validaciones y tareas auxiliares sin construir una capa de servicios completamente separada para toda la lógica de negocio.

Por ello, esta arquitectura implementada se puede describir como una solución MVC pero, con separación parcial de responsabilidades, donde los controladores coordinan el flujo principal y algunos servicios, como los mencionados anteriormente, apoyan a tareas específicas.

Se decidió este patrón de diseño como arquitectura, ya que, nos permite organizar el software, de forma que, nos permite poder separar las distintas responsabilidades y así facilitar que el equipo pueda trabajar a la vez sin interferir entre sí.

2.3 Tecnologías y frameworks utilizados

El desarrollo del sistema fue mediante un conjunto de tecnologías las cuáles fueron las siguientes:

- ASP.NET Core 8: Es el framework principal del back-end, este fue seleccionado por su seguridad, rendimiento, hecho para soportar sistemas robustos, está enfocado en el desarrollo de aplicaciones web. También tiene su integración nativa con identity, Entity Framework Core y SQL server y así poder facilitar la implementación de autenticación por roles, persistencia relacional y migraciones, además de su documentación en materia de autenticación, autorización y protección de datos.

  Por lo mencionado anteriormente es que, si bien hay otros frameworks, como lo es Django, que también ofrecen un enfoque integrado para el desarrollo web, la elección de ASP.NET, se justificó dado por su mejor ajuste al ecosistema utilizado en el proyecto.

  Como se ha dicho, la seguridad del framework es bastante importante y más con el contexto de nuestro proyecto, por ello, es importante mencionar por qué es seguro ASP.NET.

  El framework consta de una tecnología integrada llamada “Identity”, la cual se encarga de poder crear usuarios y roles, pudiendo gestionarlos. [5] En cuanto a la seguridad que mencionábamos, es importante para esta tecnología, ya que Identity, guarda usuarios, contraseñas y roles, también almacena las contraseñas mediante mecanismos de hashing seguro con algoritmos fuertes como, PBKDF2 (Password-Based Key Derivation Function 2), la cual es un estándar criptográfico, que se encarga de proteger contraseñas creando hashes lentos, por lo que lo protege de ataques de fuerza bruta o ataques de diccionario, si el atacante desea descifrar las contraseñas al probar millones de combinaciones, la lentitud

deliberada hace que el proceso se ralentice de forma masiva. Además, tiene integrado un manejo de bloqueo por intentos fallidos. [6]

Por último, en cuanto a Identity, se usan políticas para decir qué acciones puede ejecutar cada rol, por ejemplo, "Solo el rol de Guardia puede escanear los QR", "Solo el personal médico puede cambiar el estado del paciente". De esta manera con sus credenciales correspondientes realizar lo que le corresponda hacer dependiendo el rol que tenga. [7][8]

Siguiendo con la seguridad, contamos con middleware de seguridad, la cual ASP.NET ejecuta una cadena de componentes antes de entregar una respuesta, eso quiere decir que un Middleware es un componente que se ejecuta entre la solicitud del cliente y respuesta del servidor, esto es código que intercepta cuando hay un request antes de llegar al controlador, y cuando hay un response antes de volver a al cliente. [9] La cadena mencionada anteriormente está conformada por las siguientes:

- Cookies: Cuando un funcionario se autentica correctamente, el framework genera una cookie que es firmada digitalmente con una clave de protección de datos. Esta contiene claims del usuario, como lo son el id y roles, los cuales son codificados y protegidos ante posible manipulaciones. Ante cada solicitud generada, el middleware (UseAuthentication()) intercepta la cookie, verifica su integridad criptográfica y restaura el contexto de identidad antes de que esta petición llegue al controlador. Esta cookie tiene una vida útil, la cual está establecida en 60 minutos con una renovación deslizante, lo que hace extender la sesión de forma automática ante la actividad del usuario, gestionando así los permisos y accesos del usuario según su rol.[7]

- Protección contra XSS (Cross-Site Scripting): El scripting entre sitios (XSS) es una vulnerabilidad que permite a un ciberdelincuente colocar scripts en el lado cliente (normalmente a través de JavaScript) en páginas web. Los scripts del ciberdelincuente se ejecutan cuando otros usuarios cargan páginas afectadas, lo que permite al ciberdelincuente robar cookies y tokens de sesión y cambiar el contenido de la página web a través de la manipulación de DOM o redirigir el explorador a otra página. [10] ASP.NET reduce el riesgo de XSS mediante el encoding automático aplicado por Razor (Motor de plantillas que mezcla HTML y el poder ejecutar código C#), convierte caracteres peligrosos a su forma segura, es decir, al momento de que un ciberdelincuente coloque algún script malicioso, ese script se quedará inutilizado ya que se transformara solo como texto, no se ejecutará ese script, además, dentro de estas vistas Razor no hay usos de Html.Raw, ni inserciones directas de datos que no son confiables como HTML dinámico. Por todo esto, puede afirmarse que el sistema reduce el riesgo de XSS.

- Protección contra CSRF (Cross-Site Request Forgery): La falsificación de solicitud entre sitios es un ataque contra aplicaciones hospedadas en web, en el que una aplicación web malintencionada puede influir en la interacción entre un explorador

cliente y una aplicación web que confía en ese explorador. Estos ataques son posibles porque los exploradores web envían automáticamente algunos tipos de token de autenticación con cada solicitud en un sitio web. Esta forma de exploit también se conoce como *ataque con un clic* o *secuestro de sesión* porque el ataque aprovecha la sesión autenticada previamente del usuario. La falsificación de solicitudes entre sitios también se conoce como XSRF o CSRF. [11] Esto implica que, si un usuario se encuentra autenticado en el sistema y visita alguna página maliciosa, esto podría intentar enviar algún post que sea, por ejemplo, para eliminar una cuenta. Para ello ASP.NET reduce este riesgo a través de token anti-falsificación, en los formularios, para ello se implementaron mecanismos antiforgery como lo es, @Html.AntiForgeryToken() o a través de FormTagHelper en los formularios con método POST, lo cual genera un token único y temporal, de esta manera cuando el sistema vea que un usuario envía un formulario, el servidor compara el token que venía en la cookie con el token enviado por el formulario, si no coinciden, esta petición se rechaza, y esto es importante para evitar ataques donde el usuario realiza acciones que no quiere hacer como cambios o eliminaciones.

- Protección contra inyecciones SQL: La inyección de código SQL es un ataque en el que se inserta código malintencionado en cadenas que posteriormente se pasan a una instancia del motor de base de datos de SQL Server para su análisis y ejecución. Todos los procedimientos que generan instrucciones SQL deben revisarse en busca de vulnerabilidades de inyección de código, ya que el motor de base de datos ejecutará todas las consultas recibidas que sean válidas desde el punto de vista sintáctico. Un atacante cualificado y con determinación puede manipular incluso los datos con parámetros. La forma principal de inyección de código SQL consiste en la inserción directa de código en variables especificadas por el usuario que se concatenan con comandos SQL y se ejecutan. [12]Pero el sistema reduce el riesgo de esta vulnerabilidad gracias a Entity Framework (EF Core) que al usarse como el canal principal de acceso a datos, las operaciones sobre entidades de la base de datos se realizan principalmente mediante consultas LINQ sobre DbSet, lo que permite generar consultas parametrizadas. Gracias a esto, cualquier entrada maliciosa será tratada como texto literal y no como una instrucción SQL, también es importante el no usar consultas SQL manuales dentro del proyecto.

Se mencionó anteriormente una tecnología llamada “Entity Framework” (EF Core), por el lado que ayuda a la seguridad del sistema, ¿pero por qué lo usamos y qué es?

Entity Framework Core (EF Core): es un ORM (*Object–Relational Mapper)* desarrollado por Microsoft que permite trabajar con bases de datos usando clases y objetos en lugar de escribir consultas SQL manuales. Según la documentación oficial, EF Core actúa como una capa intermedia que convierte automáticamente las consultas LINQ en SQL parametrizado, lo que mejora la seguridad y evita ataques como inyección SQL, como lo mencionamos anteriormente. También incluye mecanismos como el seguimiento de

entidades, migraciones para versionar el esquema de datos, y el uso del patrón DbContext para administrar conexiones, transacciones y persistencia. [13]

Principalmente lo usamos por su integración natural con ASP.NET Core, que permite gestionar datos de forma eficiente, segura y mantenible, especialmente cuando se trabaja con SQL Server. [13]

Para toda la administración de la base de datos se utilizó la siguiente tecnología:

SQL Server Management Studio (SSMS): Esta es una herramienta de gestión oficial para entornos SQL Server. SSMS proporciona un entorno gráfico que permite crear, modificar y administrar bases de datos de manera eficiente durante el desarrollo del proyecto. Esta herramienta facilita la realización de tareas como la definición de tablas, índices y relaciones, la ejecución de consultas SQL, revisión del rendimiento del servidor, y la administración de permisos y seguridad.[14]

El uso de SSMS resulta especialmente adecuado en el contexto del sistema implementado, ya que complementa el funcionamiento de Entity Framework Core, encargado de la generación de migraciones y del manejo del acceso a datos mediante consultas parametrizadas y por ello fue por lo que utilizamos SSMS principalmente. [14]

Como otra tecnología utilizamos una biblioteca para el manejo de correo electrónico que es la siguiente:

MailKit: Es una biblioteca moderna, segura y multiplataforma para el manejo de correo electrónico en aplicaciones .NET. Según su documentación oficial, MailKit soporta los protocolos SMTP, IMAP y POP3, también soporta diversos servidores correo como Outlook, Gmail, etc. y permite enviar correos usando conexiones seguras SSL o STARTTLS. Es especialmente útil para aplicaciones ASP.NET Core porque permite construir mensajes MIME complejos, incluyendo HTML, adjuntos e imágenes incrustadas mediante CID, lo que es fundamental para la funcionalidad del envío del código QR, que envía una copia del código al correo de confirmación de la cita. [15]

La biblioteca es recomendada por Microsoft como alternativa a System.Net.Mail debido a su mayor seguridad, compatibilidad y mantenimiento activo, ya que System.Net.Mail está marcada como tecnología sin evolución futura. [15]

Para finalizar utilizamos otra tecnología para poder testear con usuarios el sistema, que es la siguiente:

Ngrok: Es una herramienta que permite exponer servicios web que se están ejecutando en un servidor local a Internet mediante la creación de un túnel seguro. esto implica que una aplicación que corre en una máquina, tras activar Ngrok, obtiene una URL pública temporal a través de la cual puede ser accedida desde cualquier dispositivo conectado a la red mundial. [16]

Esto decidimos utilizarlo ya que necesitábamos probarlo de forma rápida con usuarios, ya que, solo fue con intención de utilizarlo como testing del sistema, por ello, esta herramienta es rápida de configurar, con solo un comando podemos iniciar el servidor.

3 Diseño e implementación

En este capítulo se va a describir el diseño y la implementación del back-end del sistema de gestión de visitas hospitalarias. Se detallan las decisiones arquitectónicas, la estructura del proyecto y los componentes implementados. Además del uso de buenas prácticas, patrones de diseño, control de versiones y metodologías de trabajo.

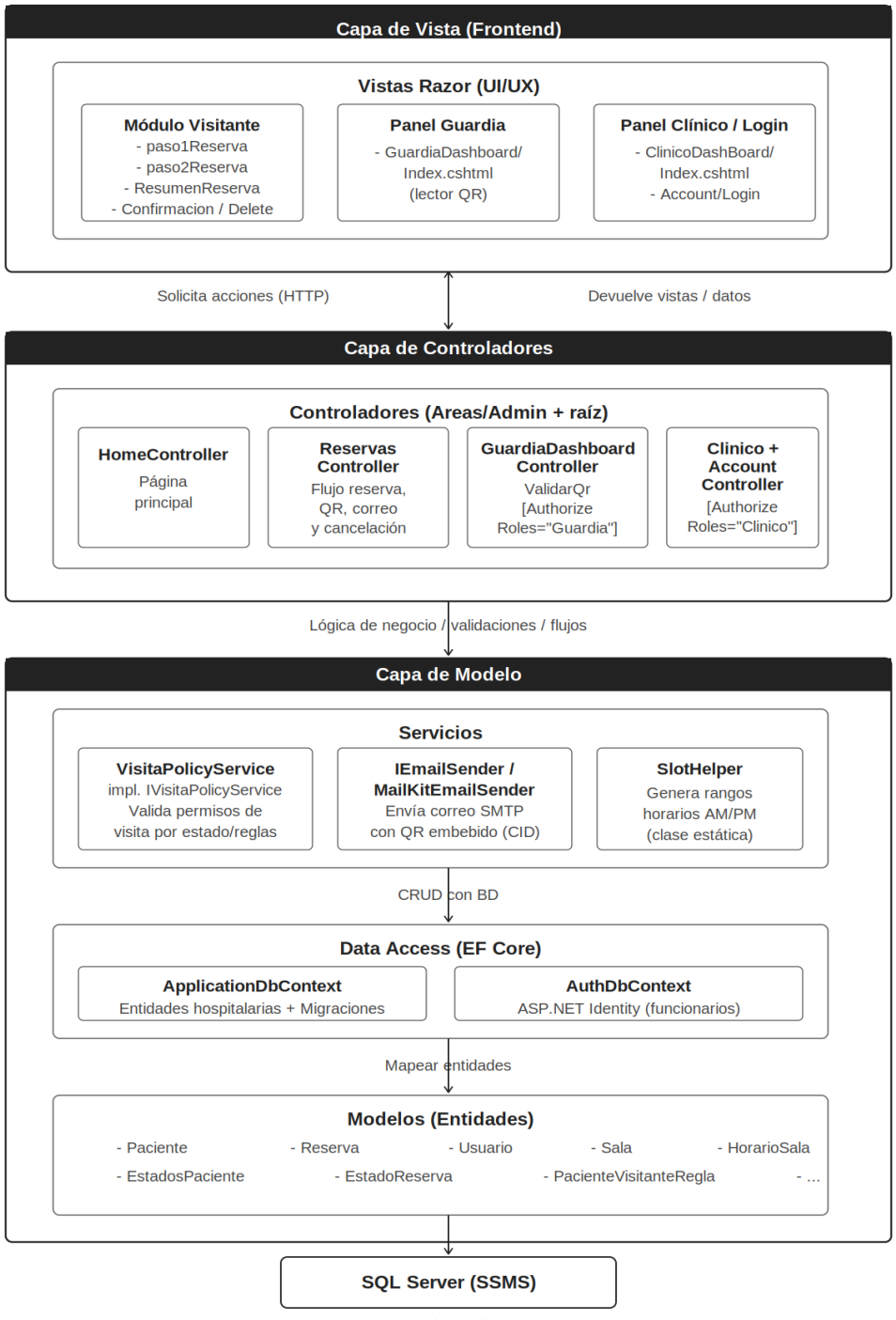
3.1 Diseño de Componentes

3.1.1 Arquitectura y estructura de los componentes.

Como vimos en el capítulo anterior, utilizaremos ASP.NET CORE MVC, donde se mencionó el porqué de la elección de esta arquitectura y framework, de todas formas a nivel de resumen, a comparación de otras arquitecturas MVC, nos brinda una facilidad a la hora de poder dividir las distintas tareas y responsabilidades dentro del equipo, debido a su separación por capas que nos brinda la arquitectura seleccionada, esto nos permite trabajar en estas distintas capas, sin afectar la de los demás, es decir, el encargado o encargada de UI/UX y Front-end pueden trabajar en la capa visual, y por otro lado poder trabajar en las otras 2 capas, que es la de los controladores y datos, sin afectar a los demás. De esta manera hace el sistema escalable, más facilidad para testear y un desarrollo más rápido.

Lo hace escalable debido a que se pueden integrar más funcionalidades, e ir testeando sin afectar a los demás por ello también hace que el desarrollo sea más rápido.

En la ilustración 3.1 se puede ver esto de formal visual, donde se ve como se comunica cada capa, que tiene cada una, a grandes rasgos, y como están separadas.



Fuente: Elaboración propia

Ilustración 3.1 Diagrama de estructura de las capas
Fuente: Elaboración propia

Para poder abordar todas las funcionalidades, tuvimos que dividir a los usuarios en 3, los cuáles fueron, personal de guardia, personal clínico y el visitante, de esta manera poder enfocarnos en las funcionalidades de cada uno.

Como hemos mencionado el personal clínico y el de guardia tienen roles dentro del sistema gracias a Identity, cada uno debe ingresar a través de este login con sus credenciales correspondientes, después de pasar por el autentificador, ingresaran a su vista asignada.

Una vez dentro sus funcionalidades son:

- Guardia: Validar códigos QR para autorizar o denegar el ingreso de visitas
- Clínico: Poder buscar pacientes por su RUT, que pueda visualizar sus datos, estado en el que está el paciente y poder modificarlo, habilitar y/o bloquear visitas.

Ambos son roles distintos y tienen controladores para cada uno por separado, para ver como es el flujo de este proceso dentro del sistema nos apoyaremos de la ilustración 3.2

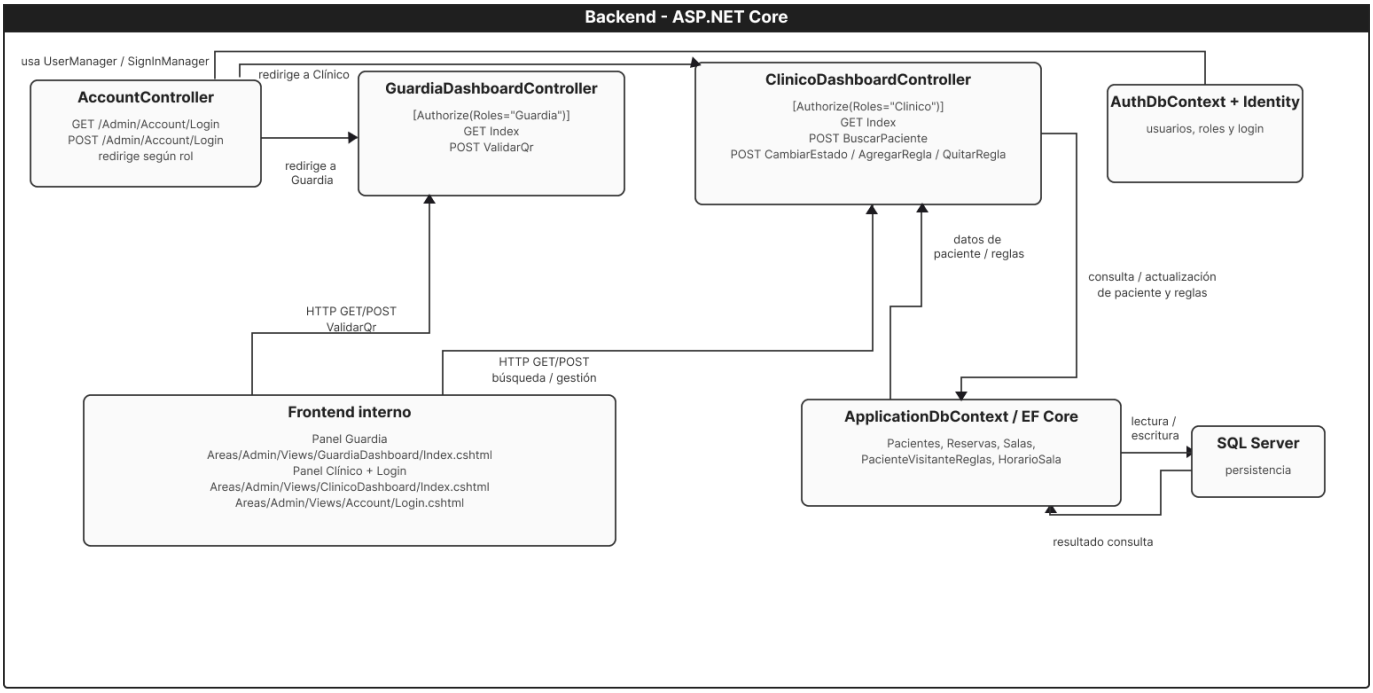


Ilustración 3.2 Diagrama de componentes (Guardia y Clínico)

Fuente: Elaboración propia

Ilustración 3.2 Diagrama de componentes (Guardia y Clínico)
Fuente: Elaboración propia

Por otro lado, tenemos al usuario visitante, el cual deberá a través de la página principal de la plataforma web, dirigirse al formulario para poder registrarse como visita, rellenando la información correspondiente a él y al paciente, después de completar todos los pasos para solicitar una visita, se guarda y se genera un QR, el cual es embebido para poder ser mostrado en el correo, el QR contiene los datos necesarios para que cuando sea escaneado por el guardia muestre la información que necesita para permitir o no el ingreso de la visita.

Por último, llega a la vista de confirmación de su visita donde abra un check, en caso de que todo se haya ejecutado bien, de lo contrario solo saldrá un mensaje de error, junto al check se mostrará el código QR que fue generado anteriormente y al momento de llegar a ese punto, el sistema a través del servicio de email, que como mencionamos antes, es el MailKit, envía el correo automático con los detalles de la visita y adjuntado una copia del código QR.

A continuación en la ilustración 3.3, veremos este flujo de manera visual para ayudar a comprender como funciona dentro del sistema.

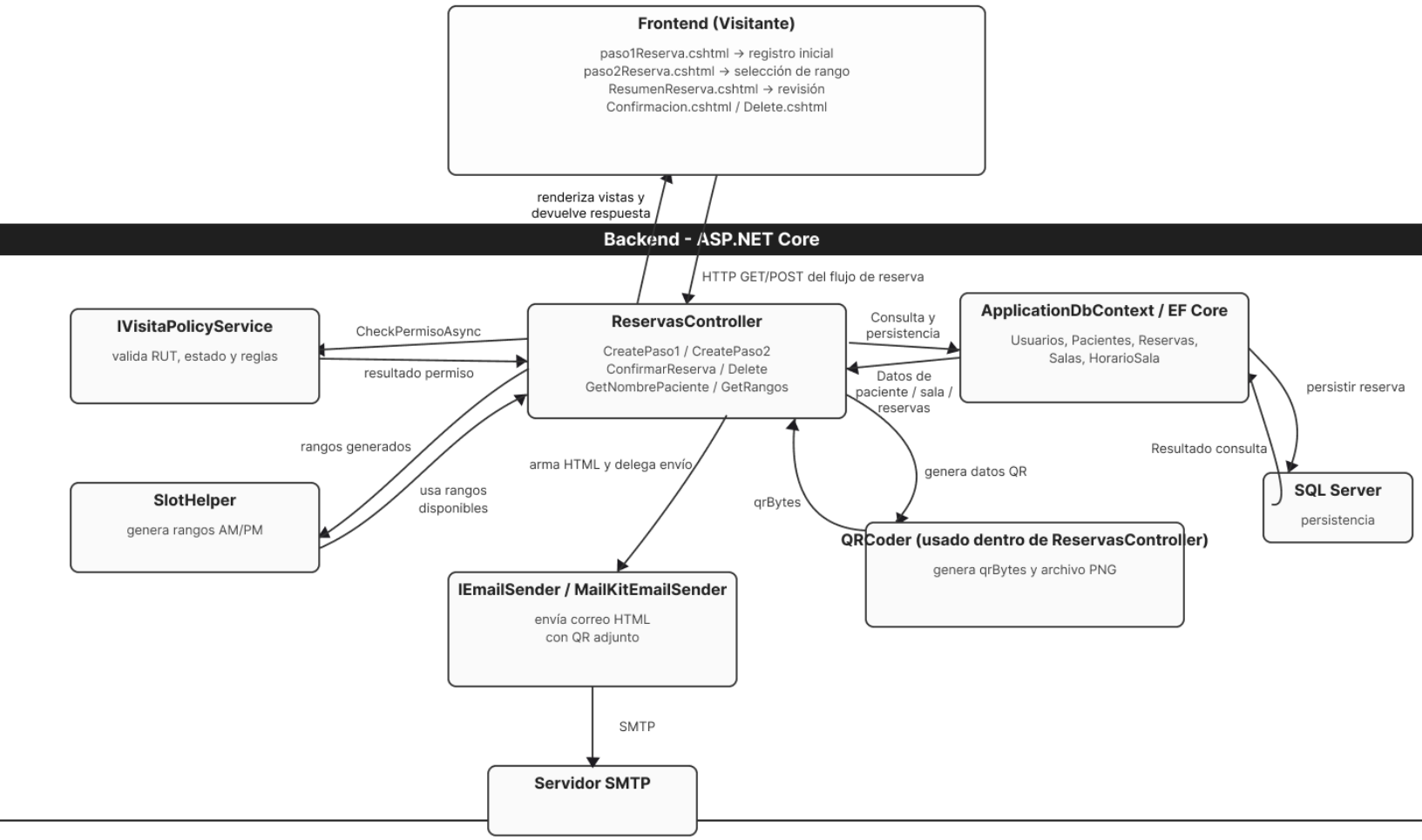


Ilustración 3.3 Diagrama de componentes (Visita)

Fuente: Elaboración propia

En cuanto a los modelos (base de datos), se decidió realizar un modelo relacional, como podemos ver a continuación en la ilustración 3.4:

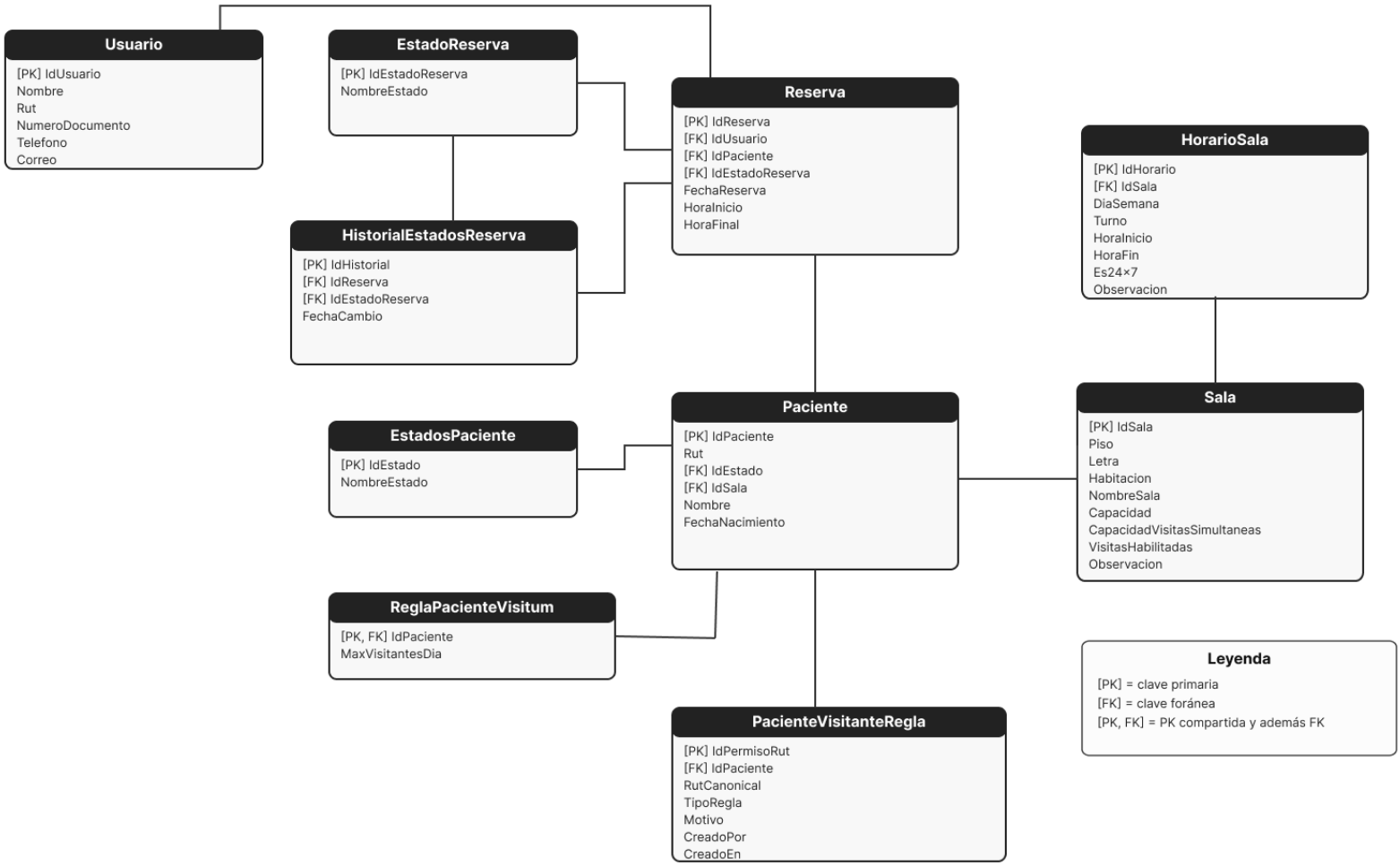


Ilustración 3.4 Diagrama de base de datos
Fuente: Elaboración propia

Como se observa en la ilustración, se utilizó un modelo relacional debido a que encaja correctamente para cada entidad ya que, cada una representa una pieza en específico del proceso, y mediante las relaciones podemos ver como entre ellas reflejan el flujo real entre los distintos actores del sistema, por ejemplo, quien solicita, a quien se visita,

en que horario, estado del paciente, etc. También es importante este tipo de estructura en el modelo porque nos garantiza integridad, esto implica que, una visita siempre va a pertenecer a un paciente, lo mismo para un guardia, siempre valida sobre una visita existente, como también cualquier bloqueo queda asociado a un registro verificable, entre otros. Ahora hablando más a nivel general, un diseño relacional nos aporta y facilita lo siguiente:

- Consistencia en los datos: Quiere decir que nos evita las duplicidad, entradas inválidas y estados contradictorios.

- Seguridad: Nos referimos a este punto ya que Identity se integra naturalmente con este tipo de tablas, como vimos en el capítulo anterior, lo importante que es Identity para la seguridad en el sistema.

- Escalabilidad funcional: Quiere decir que nos permite agregar nuevos roles, más reglas, etc. Sin necesidad de rediseñar todo

- Auditoría: Gracias a estas relaciones en el modelo, se puede utilizar estas para reconstruir lo que ocurrió, es decir, saber quién se registró, a quién se bloqueó, y más.

Ahora repasaremos cada una de tablas y su función:

- **Tabla pacientes:** Representa a cada paciente hospitalizado dentro del sistema, que como vemos en la ilustración 3.5, consta de campos id paciente, RUT, id estado la cual está relacionada con la tabla EstadosPaciente, id sala la cual está relacionada a la tabla Salas, nombre, etc.

Ilustración 3.5 Tabla pacientes
Fuente: Elaboración propia

  Su principal función es que es nuestra entidad central que se relaciona con las distintas tablas reservas, reglas de visita, estados y salas.

- **Tabla Usuarios (Visitantes):** Representa a los usuarios que realizan la reserva como visita, sus campos principales, como se ven la ilustración 3.6, son el id usuario, nombre, RUT, numero documento, teléfono, correo.

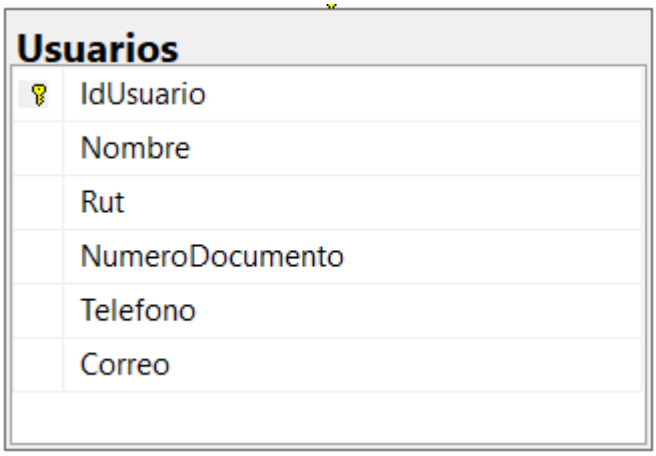


Ilustración 3.6 Tabla Usuarios (Visitante)
Fuente: Elaboración propia

Su principal función es ser la entidad utilizada para registrar solicitudes de visitas.

- Tabla Reservas: Esta entidad registra cada solicitud de visita realizada por el usuario hacia un paciente, sus campos principales son, como se ven la ilustración 3.7, id reserva, id usuario que es la visita, id paciente, id estado reserva, fecha reserva, hora inicio, hora fin.

Ilustración 3.7 Tabla Reservas
Fuente: Elaboración propia

Su principal función es ser la entidad base para todo el flujo de "reservar visita, generar QR y enviar el correo".

- Tabla Estado Reserva: Representa la lista de estados posibles de la reserva, es decir, que puede ser pendiente, confirmada, cancelada y finalizada. Como se ve la ilustración 3.8.

Ilustración 3.8 Tabla Estado Reservas
Fuente: Elaboración propia

Su función nos permite controlar el "ciclo de vida" de una visita.

- Tabla Historial Estados Reserva: Este registra cada cambio de estado de una reserva y sus campos principales, como se ven la ilustración 3.9, son el id historial, id reserva, id estado reserva, fecha cambio.

Ilustración 3.9 Tabla Historial Estado Reservas
Fuente: Elaboración propia

Su labor principal es, funcionar como una auditoría del flujo de la reserva

- Tabla Estados Paciente: Esta define el estado actual del paciente respecto a las visitas que quiere decir esto, que puede cambiar entre, disponible para visitas, no disponible, aislado, etc. Como se ve la ilustración 3.10.

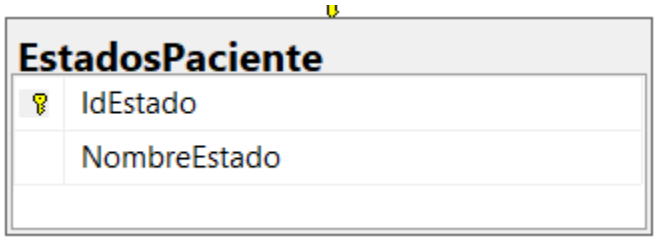


Ilustración 3.10 Tabla Estados Paciente
Fuente: Elaboración propia

Su función principal es controlar si un paciente puede ser visitado o no.

- Tabla Salas: Esta tiene información de las salas donde se encuentran hospitalizados los pacientes, con los siguientes campos, como se ve la ilustración 3.11, id sala, piso, letra, habitación, nombre sala, capacidad, etc.

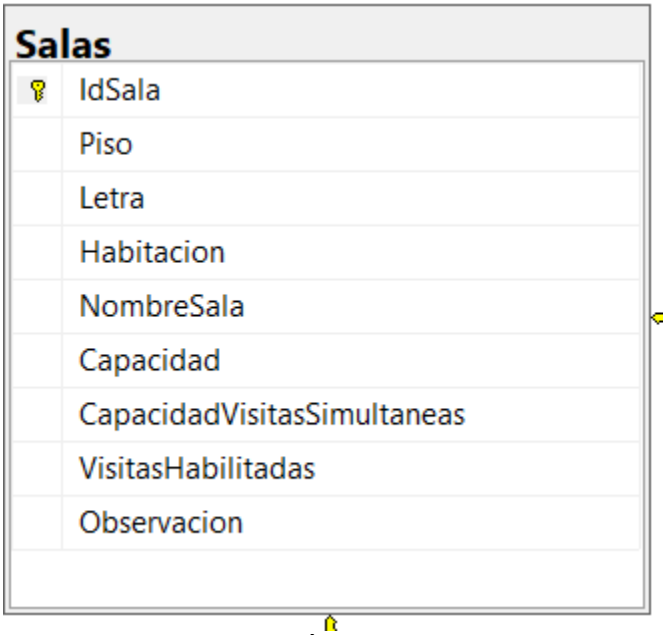


Ilustración 3.11 Tabla Salas
Fuente: Elaboración propia

Su función principal es la de poder organizar y tener un control de las salas.

- Tabla Horario Sala: En esta entidad se definen los horarios disponibles para poder agendar una visita con campos, como se ve la ilustración 3.12, día semana, turno, horario inicio, hora fin.

Ilustración 3.12 Tabla Horario Sala
Fuente: Elaboración propia

Su función principal es que es utilizada cuando el sistema valida si una reserva cae dentro del horario permitido.

- Tabla Regla Paciente Visita: Esta contiene las reglas específicas para un paciente, por ejemplo, tener una cantidad máxima de visitas, con campos id paciente y max visitante día, como se ve en la ilustración 3.13.

Ilustración 3.13 Tabla Regla paciente visita
Fuente: Elaboración propia

- Tabla Paciente Visitante Regla: Esta contiene reglas especiales relacionado entre paciente y visitante, contiene campos, como se ve en la ilustración 3.14, de id permiso RUT, id paciente, RUT canónico, tipo regla, motivo, etc.

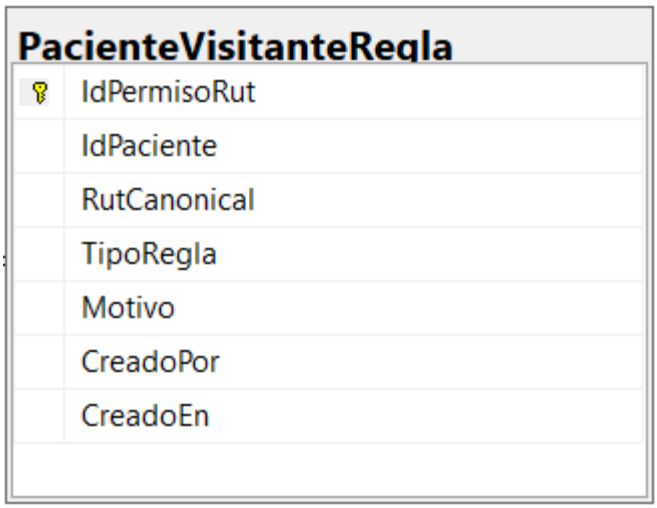


Ilustración 3.14 Tabla Regla paciente visita
Fuente: Elaboración propia

Su función principal es permitir al clínico gestionar en casos excepcionales, por ejemplo, en caso de que se deba bloquear algún ingreso de persona, por petición del paciente o por motivos legales.

3.1.2 Uso de buenas prácticas y patrones de diseño.

El sistema incorpora buenas prácticas y patrones de diseño orientados a favorecer una arquitectura más organizada, segura, escalable y mantenible, considerando el alcance definido para el MVP. Estas prácticas nos ayudan a un futuro crecimiento del proyecto para implementar más roles, reglas para las visitas, etc. Todo eso se puede realizar gracias a estas buenas prácticas.

Como se señaló previamente la separación por capas, gracias al MVC, establece una base clara entre Model, View y Controller. Además, se incorporaron componentes auxiliares para ciertas tareas específicas, como el cálculo de bloques de horario, envíos de correos electrónicos y reglas de visita.

No obstante, debido al carácter acotado del proyecto y su enfoque práctico, parte importante de la lógica de negocio se implementó directamente en los controladores, como lo fue principalmente en ReservasController. Por ello, si bien, hay una separación de ciertas tareas, esto no corresponde a una arquitectura completamente desacoplada en capas de servicios, sino que, a una solución MVC con apoyo de componentes auxiliares como VisitaPolicyService, SlotHelper y MailKitEmailSender.

Inyección de dependencias (DI): Es importante mencionar que ASP.NET Core utiliza inyección de dependencias de forma nativa, es decir, el sistema registra servicios como en nuestro caso lo es el IEmailSender.

Construcción progresiva de objetos**:** Inspirado en los principios de diseño paso a paso, este concepto permite construir estructuras complejas de forma secuencial. Es decir, en lugar de crear un objeto u operación con un constructor muy grande lleno de distintos parámetros, esta estrategia permite armarlo por partes. Esto asegura claridad en el código, evita errores por datos faltantes, otorga flexibilidad en caso de querer modificar la construcción sin cambiar quien lo usa, y permite una fácil extensión en caso de que se agreguen nuevos campos [17].

Esta estrategia se implementó directamente dentro del método ConfirmarReserva del ReservasController, y lo podemos ver representado en 2 procesos principales:

- Construcción del código QR: Debido a que para generar el código QR, el sistema debe reunir datos que provienen de distintas fuentes (datos del visitante, datos del paciente, fecha y hora), en lugar de armar la estructura de una sola vez, el proceso se hace de forma progresiva:
    - Primero, se extraen y recopilan los datos del visitante y del paciente directamente desde la base de datos.
    - Segundo, se construye la cadena de texto base organizando la información con el formato requerido (RUT|Nombre|Documento|NombreArchivo|Sala|HoraInicio|Fecha).
    - Tercero, se procesan estos datos utilizando la biblioteca QRCoder
    - Cuarto, se genera la imagen en formato PNG y se almacena físicamente en la carpeta wwwroot/qrcodes/.
- Construcción del correo HTML**:** Aquí ocurre lo mismo que en el caso del código QR, la conformación del correo de confirmación se crea en etapas ordenadas y dependientes entre sí utilizando la biblioteca MimeKit:
    - Primero, se arma la estructura del mensaje base.

- Segundo, se insertan los datos dinámicos correspondientes a la reserva confirmada.
- Tercero, se toma el arreglo de bytes de la imagen del código QR previamente generada y se embebe como recurso adjunto.
- Cuarto, se agrega el cuerpo HTML definitivo y se delega el envío al método SendWithQrAsync del servicio MailKitEmailSender.

- ViewModels y transferencia controlada de datos: En el sistema se utilizaron modelos de vista o *ViewModels* para transportar únicamente los datos necesarios entre las vistas y los controladores, evitando trabajar directamente con entidades completas de la base de datos en ciertos flujos del sistema. Este enfoque permite reducir el acoplamiento entre la interfaz y el modelo de datos, mejorar la claridad del flujo de información y limitar el uso de campos que no son necesarios para una operación específica.
  Conceptualmente, este enfoque se relaciona con el uso de DTO (*Data Transfer Object*), ya que ambos buscan transportar datos de manera acotada sin exponer entidades completas. Sin embargo, en este proyecto, debido al uso de ASP.NET Core MVC, este principio se materializa principalmente mediante *ViewModels*, los cuales están orientados a representar la información requerida por una vista específica. A diferencia de un DTO, que suele utilizarse para transferir datos entre capas, servicios o APIs, el *ViewModel* se enfoca en adaptar los datos al flujo de interacción del usuario dentro de la interfaz.
  Por ejemplo, en el flujo de agendamiento de visitas se utiliza un modelo de vista para manejar los datos necesarios del visitante, paciente, fecha y horario seleccionado, sin requerir que la vista trabaje directamente con las entidades completas de la base de datos. De esta forma, la interfaz recibe únicamente la información necesaria para completar el proceso de reserva, reduciendo la exposición de datos innecesarios y manteniendo una separación más clara entre la presentación y el modelo de datos.
  En el caso del correo automático de confirmación, el proceso ocurre directamente en el back-end. El sistema recopila únicamente los datos necesarios de la reserva, el visitante, el paciente y la sala para construir el contenido del mensaje y generar el código QR correspondiente. Aunque este flujo no utiliza un DTO formal como clase independiente, sí aplica el mismo principio de transferencia controlada de datos, ya que evita utilizar información que no es necesaria para la confirmación de la visita, como datos clínicos o información interna del paciente.

Ilustración 3.15 Imagen de correo enviado de forma automático
Fuente: Elaboración propia

Además, se aplicaron parcialmente algunos principios SOLID, principalmente en la separación de responsabilidades específicas y en el uso de interfaces para servicios como el envío de correos:

- (S) Principio de Responsabilidad Única (SRP): Este principio tiene como propósito que cada clase dentro del sistema cumpla un único propósito, es decir, en el caso de MailKitEmailSender, solo se encarga de envío de correos, no genera QR y viceversa, el ClinicoDashboardController, solo se encarga de gestionar tareas para su rol, lo mismo para el GuardiaDashboardController. [19]

- (O) Principio Abierto/Cerrado (OCP): Este principio indica que las clases están diseñadas de manera que permitan extensiones sin necesidad de ser modificadas, es decir, se puede añadir nuevos estados de visita sin que afecte las partes donde se necesite utilizar. [19]

- (L) Principio de Sustitución de Liskov (LSP): Este principio indica que clases derivadas pueden utilizarse en lugar de sus clases base sin afectar el funcionamiento del sistema. Esto se ve reflejado en el uso de modelos y servicios que implementen interfaces, de forma que el sistema pueda funcionar con distintas implementaciones. [19]

- (I) Principio de Segregación de Interfaces (ISP): Este principio se refiere a cuando el sistema define interfaces específicas y reducidas, evitando que una clase esté obligada a implementar métodos que no necesita. Es decir, al tener

un ClinicoDashboardController, solo expone métodos necesarios para él como lo es el buscar a través de RUT a los pacientes que estén hospitalizados. [19]

- (D) Principio de Inversión de Dependencias (DIP): Este principio se refiere a que las clases dependen de abstracciones, es decir de interfaces, no de implementaciones, es decir, el controlador de reservas depende del servicio de mail, pero no de clases concretas, de esta manera permite reemplazar el servicio de correo que utilizo por otro sin necesidad de tocar el controlador.[19]

3.1.3 Integración con el sistema general.

Como bien hemos mencionado, el proyecto tiene una arquitectura basada en MVC, el cuál consta de las 3 capas que hemos visto a detalle, pero debido a que el proyecto está diseñado bajo un enfoque práctico y acotado, la lógica de negocio se encuentra implementada principalmente en ReservasController. Además, el proyecto incorpora componentes auxiliares especializados, que son, VisitaPolicyService, SlotHelper y MailKitEmailSender, utilizados para encapsular validaciones y tareas específicas. Por ello, se dice que corresponde a una arquitectura MVC con apoyo parcial, como se dijo anteriormente, de servicios complementarios.

Con el contexto anterior quiere decir que existe "ReservasController" que cumple un rol fundamental en esta integración, ya que administra todo el proceso asociado a una visita desde la creación, verificación de disponibilidad, validaciones de reglas operativas, generación del código QR y envío del correo de confirmación. Este controlador interactúa directamente con Entity Framework Core (capa de datos), accediendo a entidades como Paciente, Visita, etc.

Por otro lado, el "GuardiaDashboardController" y el "ClinicoDashboardController" utilizan la misma estructura de integración, realizando consultas y actualizaciones relacionadas a la validación del código QR, visibilidad de registros, habilitación o bloqueo de visitas y la gestión de los pacientes. A causa de la integración nativa con ASP.NET Core Identity, los 2 controladores funcionan, permitiendo que cada perfil acceda únicamente a las funcionalidades correspondientes y esto se hace a través del "AccountController".

Esta implementación se puede ver visualizada dentro del proyecto a través de la organización de las carpetas como se mostrará a continuación en la ilustración 3.16:

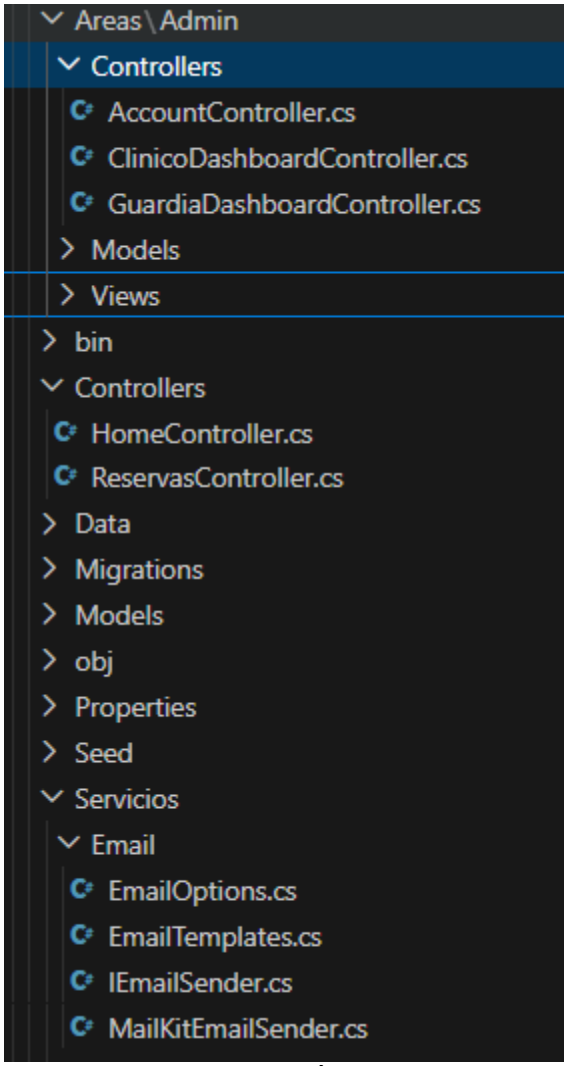


Ilustración 3.16 Imagen de organización de controladores y servicio de email
Fuente: Elaboración propia

Dentro del sistema cada interacción del usuario genera solicitudes HTTP que son procesadas por los controladores del sistema, estas solicitudes activan las validaciones, consultas a la base de datos SQL Server y la ejecución de toda la lógica de negocio para cumplir con las funcionalidades solicitadas.

A continuación, veremos la tabla 3.1, llamada tabla de especificaciones de Endpoints, donde explicaremos el método que se usa, el endpoint, su descripción, los parámetros de entrada y la respuesta esperada.

MÉTODO	ENDPOINT	DESCRIPCIÓN	PARÁMETROS DE ENTRADA	RESPUESTA ESPERADA
GET	/Admin/Account/Login	Despliega el formulario de inicio de sesión.	Ninguno.	Vista HTML del Login.
POST	/Admin/Account/Login	Procesa las credenciales para autenticar al funcionario.	Email, Password, RememberMe.	Redirección al Dashboard según rol (Guardia o Clínico).
GET	/Admin/Account/Logout	Cierra la sesión del usuario actual.	Ninguno.	Redirección al Login.
GET	/Admin/GuardiaDashboard	Carga el panel principal del guardia con el lector QR.	Ninguno (Requiere Rol Guardia).	Vista HTML con cámara activada.
POST	/Admin/GuardiaDashboard /ValidarQr	Servicio API para verificar si un código QR es válido para ingresar.	JSON: { "qrTexto": "cadena_del_qr" }.	JSON: { "valido": bool, "mensaje": "...", "rut": "..." }.
GET	/Admin/ClinicoDashboard	Carga el panel de gestión clínica.	Opcional: rut (para cargar datos).	Vista HTML de gestión de pacientes.
POST	/Admin/ClinicoDashboard /BuscarPaciente	Busca un paciente específico por RUT.	RutBusqueda (string).	Redirección a la misma vista con datos cargados.
POST	/Admin/ClinicoDashboard /CambiarEstado	Modifica el estado global del paciente (ej. recibir visitas).	idPaciente, idEstado.	Redirección con notificación de éxito.
POST	/Admin/ClinicoDashboard /AgregarRegla	Crea una excepción de visita (Bloquear/Habilitar persona).	idPaciente, tipo, rut, motivo.	Redirección actualizando la lista de reglas.
POST	/Admin/ClinicoDashboard /QuitarRegla	Elimina una regla de excepción existente.	idPermisoRut, idPaciente.	Redirección confirmando eliminación.
GET	/Reservas/CreatePaso1	Muestra el formulario inicial de solicitud de visita.	Ninguno.	Vista HTML (Paso 1).
POST	/Reservas/CreatePaso1	Valida existencia del paciente y reglas de visita.	RutPaciente, RutVisita, Email, etc.	Redirección al Paso 2 si es válido; retorno a vista si hay error.
GET	/Reservas/CreatePaso2	Muestra la selección de bloques horarios disponibles.	Datos en TempData (del Paso 1).	Vista HTML (Paso 2) con calendario.
POST	/Reservas/CreatePaso2	Valida y reserva temporalmente el bloque horario seleccionado.	IdPaciente, IdSala, Fecha, Rango.	Vista HTML de Resumen de Reserva.

MÉTODO	ENDPOINT	DESCRIPCIÓN	PARÁMETROS DE ENTRADA	RESPUESTA ESPERADA
POST	/Reservas/ConfirmarReserva	Finaliza el proceso: guarda en BD, genera QR y envía correo.	Datos consolidados de la reserva provenientes del flujo de agendamiento	Vista HTML de Confirmación con QR visible.
GET	/Reservas/GetNombrePaciente	Servicio API para autocompletar nombre al escribir RUT.	rut (string).	JSON: { "nombre": "Nombre del Paciente" }.
GET	/Reservas/GetRangos	Servicio API para obtener horas disponibles dinámicamente.	idSala, idPaciente, fecha.	JSON: { "am": [...], "pm": [...] }.
GET	/Reservas/Delete/{id}	Muestra la confirmación para cancelar una hora.	id (Identificador reserva).	Vista HTML con detalles de la reserva a cancelar.
POST	/Reservas/Delete	Ejecuta el proceso de cancelación de la reserva y redirige a la vista de confirmación.	id (Identificador reserva).	Redirección a vista de "Cita Cancelada".

Tabla 3.1 Tabla de especificaciones de Endpoints
Fuente: Elaboración propia

3.2 Detalles de la codificación y desarrollo.

El desarrollo del back-end se realizó utilizando metodología ágil SCRUM, la cual fue dividida en 3 sprints, cada uno enfocado en las principales funcionalidades para los tipos de usuario. Al trabajar en iteraciones, permite poder realizar entregas incrementales, y teniendo reuniones para ir mejorando el sistema, sin embargo, al ser un MVP, a través de nuestro patrocinador llegamos a acuerdos en reuniones sobre el alcance que debía tener, detallando la funcionalidad de cada usuario. A continuación, se detallará el cómo fue el desarrollo de parte del back-end.

En el Sprint 1 nos enfocamos en el usuario tipo visitante, el objetivo principal era implementar toda la lógica y funcionalidades necesarias para el agendamiento de visitas exitoso

1. **Historia 1:** "Como visitante, quiero acceder a un formulario virtual para registrar mi visita y los datos del paciente"

    1. Tareas back-end:
        1. Diseñar e implementar el modelo de datos relacional inicial (tablas Pacientes, Reservas, Usuarios)
        2. Crear endpoint POST/Reservas/CreatePaso1 para poder recibir y poder validar los datos del formulario de registro.
        3. Implementar la lógica de persistencia (insertar a la base de datos) a través de Entity Framework Core, usando el acceso directo al contexto de datos (ApplicationDBContext) para

optimizar la arquitectura del proyecto y evitar la sobrecarga de clases innecesarias

2. **Historia 2:** "Como visitante, quiero visualizar un calendario con bloques horarios disponibles para seleccionar la hora de la visita de forma correcta"

    2. Tareas back-end:
        1. Diseñar el esquema de la base de datos para poder almacenar HorarioSala y Reservas para poder marcar si un horario se encuentra ocupado
        2. Implementar la lógica de filtrado de los bloques horarios disponibles, la cual fue delegada al componente SlotHelper
        3. Crear endpoint GET/Reservas/CreatePaso2 que consuma el servicio de horario y devuelve los rangos correctos y disponibles

3. **Historia 3:** "Como visitante, quiero visualizar un resumen de la visita antes de confirmar, para verificar que no haya errores en los datos para mi visita"

    3. Tareas back-end:
        1. Implementar y utilizar Tempdata y serialización JSON para poder mantener el estado de la reserva entre los pasos del formulario.
        2. Implementar la lógica de negocio para establecer los datos de paciente, hora y visitante para la vista de resumen.

4. **Historia 4:** "Como visitante, quiero recibir un correo automático de confirmación que incluya un código QR, para validar mi identidad al momento de la visita"

    4. Tareas back-end:
        1. Integrar la biblioteca MailKit para el envío del correo automático.
        2. Implementar la lógica del EmailService para poder adjuntar el código QR al correo de confirmación
        3. Implementar la lógica de generación de datos y la imagen del QR al momento de confirmar la reserva.

Siguiendo con el sprint 2, el cual se enfocó en los usuarios clínicos y de seguridad, es decir, implementamos la lógica de ambos en este sprint.

1. **Historia 1:** "Como personal del hospital, quiero autenticarme con mis credenciales únicas, para acceder a las funcionalidades correspondientes a mi rol"

    5. Tareas back-end:
        1. Implementar ASP.NET Identity para poder gestionar a los usuarios y asignar los roles correspondientes al guardia como al clínico.
        2. Implementar el controlador llamado "AccountController" con lógica de redireccionamiento para los roles.

2. **Historia 2:** "Como Guardia, quiero escanear un código QR para verificar la identidad del visitante, a quien va a visitar y saber dónde se encuentra la sala del paciente"

    6. Tareas back-end:
        1. Crear endpoint POST/Admin/GuardiaDashboard/ValidarQr el cual recibe la cadena del código QR
        2. Implementar la lógica de descifrado del QR y la validación de que ese QR existe
        3. Implementar que retorne los datos del visitante y paciente en formato JSON.

3. **Historia 3:** "Como Clínico, quiero buscar pacientes y modificar su estado, para que las visitas se deshabiliten o se habiliten ante algún evento"

    7. Tareas back-end:
        1. Crear el controlador llamado "ClinicoDashboardController" para gestionar todo lo relacionado al clínico
        2. Diseñar el modelo para las reglas de excepción ("PacienteVisitanteRegla").
        3. Implementar los endpoints POST /CambiarEstado y POST /AgregarRegla para poder actualizar los datos en la base de datos a través de EF Core.

Y por último el sprint 3, el cual fue nuestro último sprint, enfocado en implementar detalles faltantes y corregir lo de los anteriores sprints.

1. **Historia 1:** "Como administrador del sistema, quiero garantizar la integridad de los datos de los visitantes mediante validaciones adicionales"

    8. Tareas back-end:
        1. Implementar RutUtils para la verificación y canonización del RUT del visitante y del paciente.

2. **Historia 2:** "Como visitante, quiero tener la opción de cancelar una hora agendada previamente"

    9. Tareas back-end:
        1. Diseñar el flujo de cancelación, esto a través de un enlace adjuntado en el correo de confirmación.
        2. Crear los endpoints de GET /Reservas/Delete/{id} y POST /Reservas/Delete para poder cancelar la visita.
        3. Implementar la lógica para eliminar la reserva de la base de datos.

El backlog general del proyecto incluye objetivos con plazos establecido, lo cual fue mencionado en el primer capítulo, ahora se explicó el desarrollo con tareas específicas del back-end explicadas a través de historias y las tareas que tuve que realizar.

En cuanto al control de versiones, esto se gestionó a través de un repositorio compartido en Google Drive, en el cual se iban subiendo avances de todos los que desarrollamos el proyecto, los dividimos por sprints, dentro del sprint se subía un avance el cual en el título se decía que se cambio a grandes rasgos, junto con un número, por ejemplo,

"versión 1.5 Se implementa formulario inicial para la visita". Esta decisión tiene limitaciones importantes frente a otras herramientas de control de versiones como lo es Git. Por ello, en caso de que el proyecto continúe desarrollándose, una tarea prioritaria debería ser migrar el repositorio a un entorno basado en Git.

3.3 Pruebas y Validación

En este apartado se describe cómo el sistema back-end fue testeado y validado para asegurar que se cumplan con los requerimientos establecidos, mediante pruebas manuales. Estas distintas pruebas que se hicieron a lo largo de todo el desarrollo del back-end, incluyen la realización de pruebas de flujo completo, pruebas de integración, y la validación de la experiencia de usuario (aunque este apartado será explicado en la tesina de Front-end, también es importante mencionarlo) esto mediante usuarios externos al equipo de trabajo, simulando a través de otros dispositivos un agendamiento de visitas.

3.3.1Estrategias de testing aplicadas a los componentes.

Durante el desarrollo del back-end, las pruebas (testing) que fueron realizadas, fueron manuales, es decir, no ocupamos ninguna herramienta dedicada para el testing. En cambio, se utilizó una estrategia iterativa y manual. Nos enfocamos en validar cada funcionalidad inmediatamente después de implementarla. De esta manera nos permitió detectar errores rápidamente y asegurar que cada uno de los distintos módulos funcionara correctamente antes seguir con el desarrollo.

Testing incremental por funcionalidad: Cada vez que debía añadir un nuevo controlador, endpoint, servicio o algún tipo de modificación en la base de datos, se realizaban pruebas manuales es decir, mientras desarrollaba funcionalidades, debía ir verificando que los requests HTTP se hiciera de manera correcta y esperada, lo mismo para los endpoints con sus respectivos parámetros y para la base de datos la cual llevaba un constante seguimiento en cuanto se debía comunicar con ella el sistema, o si se agregaban, actualizaban o si se eliminaban entidades.

Es decir, el proceso seguía una lógica iterativa de verificación, corrección y nueva validación, de la siguiente manera:

1. Crear o modificar una funcionalidad
2. Ejecutar esa funcionalidad de forma manual
3. Revisar si cumple con el comportamiento esperado
4. En caso de fallar, corregir inmediatamente
5. Repetir hasta que funciones sin errores para una integración limpia.

Para apoyar a entenderlo mejor, visualice la Ilustración 3.17:

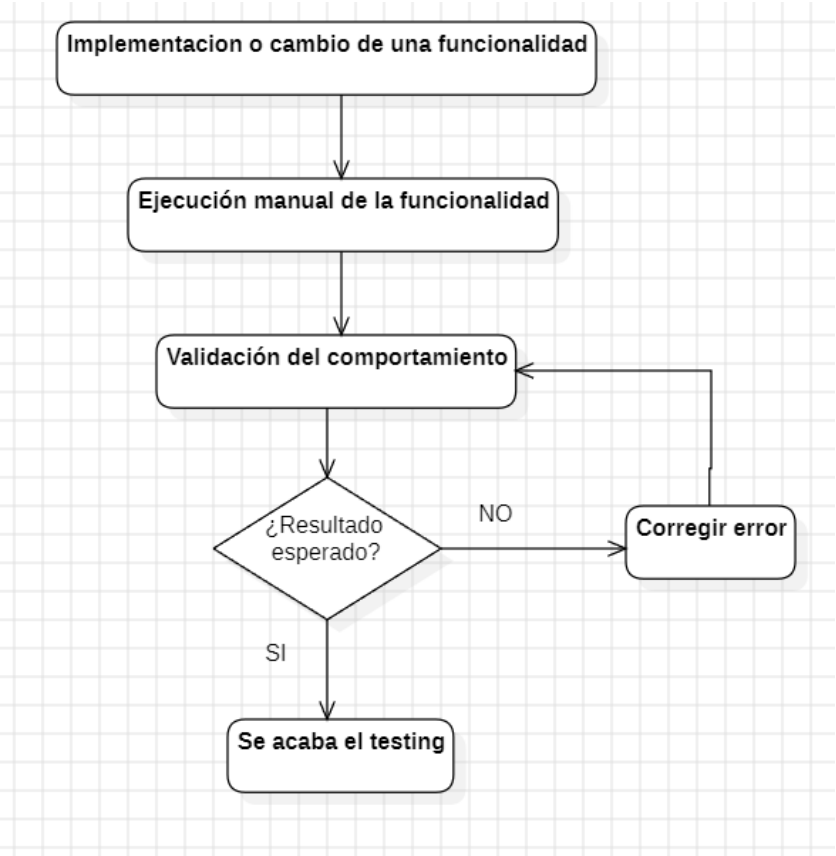


Ilustración 3.17 Diagrama de flujo de testing de una funcionalidad
Fuente: Elaboración propia

Pruebas asociadas a cambios en la base de datos: Como hemos visto anteriormente las entidades y relaciones fueron creciendo a medida que se avanzaba en el desarrollo, por ello con cada actualización hicimos lo siguiente:

1. Cada nueva migración se probó de forma inmediata
2. Se verificó que los controladores estuvieran consumiendo correctamente las nuevas propiedades o relaciones en la base de datos.
3. Se realizaron distintas verificaciones directas con consultas en el gestor de base de datos, es decir, consultas de tipo SELECT, INSERT, UPDATE.

De esta manera fue como nos permitimos asegurar que el modelo de nuestra base de datos estuviera alineado de forma correcta con la lógica del sistema.

Pruebas en flujo real del sistema: También a pesar de probar los endpoints de forma aislada como vimos antes, también se realizó validaciones de flujo completos, por ejemplo:

- Iniciar sesión y ser redirigido al rol cual corresponda con las credenciales
- Creación de una reserva, verificar la generación del QR, consultar la reserva.
- Escaneo del QR desde la vista del guardia, validar los datos y ver si la eliminación de la reserva este correcto

Y así con los distintos flujos que están dentro del sistema, esta validación nos ayudó a detectar diversos errores en la interacción entre los componentes como los controladores, servicios y base de datos.
Es importante aclarar hasta aquí, que todas las pruebas si bien se fueron haciendo durante el desarrollo del back-end, también debíamos testearlo entre cada sprint en sistema completo específicamente entre el sprint 1 y 2, y entre el sprint 2 y 3. Esto para saber si estábamos totalmente alineados con nuestro patrocinador, por ello, también tuvimos reuniones extra con el patrocinador y otros encargados de innovación o de usuarios, que son parte del hospital, los cuales nos dieron bastante feedback en cuanto al funcionamiento del sistema, esto es importante para saber si datos que se trabajan a nivel de back-end están bien tomados o bien enfocados, como se menciona, es importante, para poder testear dentro de un escenario más parecido a la realidad decidimos realizar una última prueba la cual es la siguiente:

Pruebas en entorno externo (Ngrok): Por último al finalizar decidimos probar el sistema dentro de un entorno que se pueda acceder desde dispositivos externos y para ello se utilizó la herramienta Ngrok, lo que gracias a ella nos permitió poder exponer temporalmente la aplicación web local a Internet, hacer pruebas desde distintos dispositivos móviles y equipos ajenos al local, validar la experiencia real de usuarios externos al equipo de trabajo, identificar diferencias entre entornos como lo son el local y acceso remoto, como por ejemplo tres personas a la vez solicitando hora, que el bloqueo de horas disponibles sea en tiempo real, para evitar duplicados y además si el programa soportaba este tipo de interacciones a la vez con múltiples usuarios, y verificar las respuestas desde redes móviles, diferentes navegadores y obviamente diferentes dispositivos.

Este último tipo de prueba fue clave para asegurarnos que el back-end pudiera responder correctamente a diferentes situaciones que se pueden dar en la vida real, como se mencionó antes, como reaccionan a distintos navegadores, con usuarios interactuando con el sistema a la vez y todo este tipo de cosas que son lo más cercanos con lo que pudimos testear, en cuanto a un escenario real.

3.3.2 Resolución de bugs y optimización

Como hemos visto anteriormente durante el desarrollo, la corrección de los distintos errores y la optimización fueron procesos continuos, siempre estuvieron en cada una de las diferentes etapas del desarrollo del back-end. Dado que nuestro proyecto tiene enfoque el cual se basa en el desarrollo iterativo, cada funcionalidad pasa por el ciclo que vimos en la ilustración 3.17. Esta estrategia permitió poder identificar fallas en fases tempranas del desarrollo o de la implementación de las funcionalidades y así evitar acumulación de errores que pudieran comprometer o poner en riesgo la estabilidad a nivel general del sistema.

Por ello la corrección de bugs fue bastante sencilla, pero de todas formas teníamos una lógica para actuar ante alguna falla:

1. Detección inmediata: Como mencionamos, cada vez que se implementaba alguna funcionalidad nueva, esta se probaba de forma manual directamente desde la interfaz o desde el flujo interno del sistema

2. Análisis del origen del error: Cuando aparecía un comportamiento inesperado, se analizaba en que capa se originaba el problema por ejemplo podían ser

validaciones incorrectas en los controladores, datos faltantes o mal mapeados en los modelos, consultas mal hechas o relaciones inconsistentes en EF Core o respuestas mal formadas o códigos HTTP incorrectos.

3. Corrección inmediata en el entorno local: Una vez ya identificado el origen de la falla, la corrección se aplica directamente en el módulo afectado, es decir, si había algún error en llave foráneas, se ajustaban las relaciones en el modelo y migraciones, también en datos no encontrados, se agregaban validaciones y mensajes adecuados o QR inválidos, lo cual ocurrió bastante, pero se ajustaba la forma en que se generaban o se leían ciertos campos.

4. Repetir pruebas de forma controlada: Después de cada corrección, se repetía este mismo proceso para asegurarnos que el error había sido resuelto y que no se hayan generado nuevos errores de forma colateral.

Este ciclo fue el principal y el que se repitió de manera constante a lo largo de todo el desarrollo, para que no vayan quedando pequeños errores para el futuro desarrollo y así ir evitando que se genere una acumulación progresiva de errores.

3.4 Integración con Otros Componentes

En cuanto a la integración de componentes adicionales, no contamos con ello. Desde las primeras etapas del diseño del sistema, a través de reuniones con nuestro patrocinador, se evaluó utilizar APIs externas. En particular, se analizó utilizar 2 APIs externas las cuales serán nombradas y explicadas, la razón de no utilizarlas:

API para confirmar RUT: Esta API se iba a encargar que, al momento de digitar el RUT del visitante, pudiera el sistema identificar si el RUT ingresado correspondía a una identificación válida existente en registros oficiales como lo es el registro civil, que esto no es lo mismo a lo que tenemos integrado actualmente, ya que, al llenar el campo, lo que hace el sistema actual es revisar si los dígitos que componen al RUT corresponden a su dígito verificador. Además implementar la API que mencionamos, ayudaría también en cuanto a que la visita posea, en este caso, su cédula de identidad al día, pero esto también resultaba contradictorio en cuanto al pensar, una persona que no tiene su documento al día no aceptar su entrada a una visita resultaba una restricción muy dura, en ciertos casos como los de emergencias, por ello solo decidimos implementar que se requiera su cédula de identidad y que esta coincida con los datos insertados en la visita, para que el guardia aplique control de identidad para hacer pasar o no a la visita.

Por eso, con estas consideraciones, se decidió no implementar esta API

API Paciente y cama hospital: Esta API se iba a encargar de poder acceder a una base de datos con Pacientes y disponibilidad de camas reales, con sus respectivos datos de ambas entidades como, nombre, sala, etc. Hubo distintos intentos para poder conseguir esto, para en algún futuro, en caso de seguir desarrollando este proyecto, hacer pruebas reales con pacientes existentes, pero a pesar de todos los intentos no hubo respuestas positivas para ello o algún tipo de acuerdo. Si bien sabíamos que iba a ser difícil debido a que son datos de personas reales por ello son sensibles y delicados.

En resumen, por razones de privacidad, protección de datos y por falta de disponibilidad institucional, la API no se pudo integrar al sistema.

Como se mencionó, no se implementaron ninguna de las APIs explicadas, aunque para nosotros, hubiera sido un gran aporte y haría nuestro proyecto más robusto gracias a que se integrarían APIs externas lo que nos hubiera dado un acercamiento más a un desarrollo real.

4 Conclusiones

El desarrollo de este proyecto permitió observar cómo el sistema fue creciendo y haciéndose más robusto a medida que se integraban nuevas funcionalidades. Este proceso fue complejo, ya que desde un comienzo había varias tecnologías que, en su mayoría, eran desconocidas para el equipo de trabajo.

A medida que transcurrió el tiempo y los distintos sprints, fue posible entender mejor el uso de las tecnologías que debíamos de implementar y de adaptar al contexto del proyecto. Hasta este momento, no se había trabajado en un proyecto de esta magnitud, en cuanto a la complejidad y el alcance que podría llegar a tener, ni con una organización tan definida como lo fue el equipo de trabajo y el patrocinador. En este sentido, la comunicación dentro del equipo fue fundamental, ya que las reuniones permitieron entender mejor el avance de cada uno, cómo se relacionaban sus partes con el sistema general y qué aspectos debían corregirse. Esto también fue acompañado de pruebas constantes, manuales, sobre cada integración antes de incorporarla formalmente al repositorio de trabajo, con el fin de evitar que los errores persistieran en el tiempo y se transformaran en un problema mayor.

Todo el proceso relacionado con la base de datos también representó un desafío relevante, ya que su diseño tuvo que ir ajustándose a medida que avanzaba el desarrollo. Se comenzó con una base de datos inicial, a la cual se fueron agregando nuevas entidades y relaciones según aparecían requerimientos más claros del sistema. Esto ocurrió porque en un comienzo no estaban completamente definidas ciertas lógicas de negocio, como las correspondientes al guardia y al clínico, las cuales se fueron aclarando en sus respectivos sprints.

Además, al tratarse de un MVP, el proyecto también dejó en evidencia ciertas limitaciones y aspectos a mejorar. Entre ellos, se encuentra el intento de integrar una API del hospital que entregaba información en tiempo real sobre las camillas, lo cual no pudo concretarse debido a procesos burocráticos y tiempos de respuesta incompatibles con el alcance del proyecto. También quedó en evidencia la necesidad de implementar metodologías de testing más robustas, ya que, si el sistema continúa desarrollándose, será indispensable avanzar hacia pruebas automatizadas. Del mismo modo, sería importante seguir mejorando la distribución de la lógica de negocio para lograr una arquitectura más desacoplada.

Para finalizar, a nivel personal, este proyecto permitió conocer nuevas tecnologías y tener un acercamiento más real al desarrollo de software. También permitió comprender mejor la importancia de la comunicación dentro del equipo, del cumplimiento de plazos por sprint y de la interacción con actores externos, como el patrocinador y trabajadores del hospital. En ese sentido, la comunicación con el entorno hospitalario fue positiva, ya que el patrocinador mantuvo una participación, mostrando preocupación por el avance del proyecto y disponibilidad cuando fue necesario.

En relación con los objetivos definidos, el sistema logró implementar el registro digital de visitas, la generación de códigos QR, la autenticación por roles para guardia y clínico, la gestión de estados del paciente, la persistencia de datos mediante una base relacional y el envío automático de correos de confirmación. Estos elementos permitieron construir una base funcional para digitalizar el proceso de visitas y mejorar la trazabilidad respecto al procedimiento manual inicial.

En conclusión, el proyecto cumplió con el objetivo de desarrollar una base funcional de back-end para la gestión de visitas en un recinto hospitalario, dejando además una base técnica que puede seguir mejorándose en futuras etapas de desarrollo.

5 Agradecimientos

Quisiera expresar mi más sincero agradecimiento a todos los que colaboraron para poder realizar este proyecto. Comenzando con Michael Araya, quien fue nuestro patrocinador en este proyecto, dándonos su total disposición para cualquier cosa que necesitáramos, desde el comienzo del proyecto, es decir, desde la fase inicial, hasta el final, que fue la feria de software, que nos brindó material del hospital para poder ambientar el stand. También agradecer a Nicolas Bugueño, quien fue nuestro Scrum Máster, que fue la persona con la que más colaboré en este proyecto, también se encargó de organizar las distintas reuniones con nuestro patrocinador, reunión del equipo para los sprint, etc. Agradecer a Alexander Alzamora que estuvo a cargo del Front-end, también agradecer a Melissa González quien fue la encargada de UI y UX, diseñando todas las vistas utilizando herramientas como el Figma y por último agradecer a Camilo Thenoux quien fue el encargado del testing, fue el integrante que también más colaboré, me ayudó a testear diferentes escenarios de las funcionalidades, los distintos flujos completos del sistema, la integración del sistema a Ngrok.

6 Referencias

[1] I. Sommerville, Ingeniería del Software, 9ª ed., Pearson Educación, 2011.

[2] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.

[3] E. Gamma, R. Helm, R. Johnson, y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[4] Microsoft, *ASP.NET Core Documentation*, Microsoft Learn, 2024. [En línea]. Disponible en: https://learn.microsoft.com/aspnet/core

[5] Microsoft, *Introduction to ASP.NET Core Identity*, Microsoft Learn, 2024. Disponible en: https://learn.microsoft.com/aspnet/core/security/authentication/identity

[6] Microsoft, *Password hashing in ASP.NET Core Identity*, Microsoft Learn, 2024. Disponible en: https://learn.microsoft.com/aspnet/core/security/authentication/identity#password-hashing

[7] Microsoft, *Role-based Authorization in ASP.NET Core*, Microsoft Learn, 2024. Disponible en: https://learn.microsoft.com/aspnet/core/security/authorization/roles

[8] Microsoft, *Policy-based Authorization in ASP.NET Core*, Microsoft Learn, 2024. Disponible en: https://learn.microsoft.com/aspnet/core/security/authorization/policies

[9] Microsoft, *Middleware en ASP.NET Core*, Microsoft Learn, 2024. [En línea]. Disponible en: https://learn.microsoft.com/es-es/aspnet/core/fundamentals/middleware/?view=aspnetcore-10.0

[10] Microsoft, *Prevent Cross-Site Scripting (XSS) in ASP.NET Core*, Microsoft Learn, 2024. Disponible en: https://learn.microsoft.com/aspnet/core/security/cross-site-scripting

[11] Microsoft, *Prevent Cross-Site Request Forgery (CSRF) attacks in ASP.NET Core*, Microsoft Learn, 2024. Disponible en: https://learn.microsoft.com/aspnet/core/security/anti-request-forgery

[12] Microsoft, *Protección contra inyección SQL en SQL Server*. [En línea]. 2024. Disponible en: https://learn.microsoft.com/es-es/sql/relational-databases/security/sql-injection?view=sql-server-ver17

[13] Microsoft, *Entity Framework Core Documentation*, Microsoft Learn, 2024. Disponible en: https://learn.microsoft.com/ef/core/

[14] [SSMS] Microsoft, *SQL Server Management Studio (SSMS) Documentation*, Microsoft Learn, 2024. Disponible en: https://learn.microsoft.com/sql/ssms/

[15] [MailKit] J. Stedfast, *MailKit – Cross-Platform Mail Client Library*, GitHub, 2024. Disponible en: https://github.com/jstedfast/MailKit

[16] [ngrok] ngrok, *Secure tunnels for HTTP/S and TCP services*, ngrok documentation, 2024. Disponible en: https://ngrok.com/docs/http

[17] Refactoring.Guru, *Builder (patrón de diseño)*, Refactoring Guru, 2025. Disponible en: https://refactoring.guru/es/design-patterns/builder

[18] [DTO] Universidad Politécnica de Catalunya, *Data Transfer Object (Patrón de diseño)*, Informe académico, 2025. Disponible en: https://upcommons.upc.edu/bitstream/handle/2117/329217/152705.pdf

[19] DigitalOcean, *SOLID design principles explained: building better software architecture*, DigitalOcean Community, 2025. Disponible en: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design