UNIVERSIDAD NACIONAL TECNOLÓGICA DE LIMA SUR

FACULTAD DE INGENIERÍA Y GESTIÓN

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

SISTEMA DE CONTROL DE RESERVAS Y ADMINISTRACIÓN PARA LA BARBERÍA DIAMOND BARBERSHOP

CÓDIGO DE BLOQUE: IS09R3

CURSO: ARQUITECTURA DE SOFTWARE

DOCENTE: AZABACHE ASMAT JAVIER ALEJANDRO

INTEGRANTES

HUERTAS GONZALES DAFNE AYELEN – 2123110473

CABALLERO VILLAZANA DIEGO JEREMY - 2123010302

FERNANDEZ HUAMAN ANDRE RAFAEL - 2123110296

SEVILLANO COLINA ARIAN ERICK - 212310074

PASCACIO SANCHEZ ROXANA GADITA - 2113010120

Villa El Salvador

2025

ÍNDICE

INTRODUCCIÓN

Hoy en día, como mencionan Espina-Romero et al. (2025) la transformación digital ya no es una tendencia de las grandes empresas, sino que se ha tornado en una condición obligatoria para que las pequeñas y medianas empresas (PYMES) puedan alcanzar el éxito en el sector de servicios. Tras el incremento de las transacciones y la llegada de la autogestión del consumidor de hoy en día, se han llegado a crear múltiples soluciones para este tipo de necesidades. En este marco, la importancia de los diseños de arquitecturas de software escalables forma parte de las bases que deben estar presentes para tener la oportunidad de no sólo apuntalar la resolución de problemas que vayan surgiendo, sino también preparar el terreno para un crecimiento futuro sostenible.

En Perú, el sector de la belleza y cuidado personal está en expansión, pero muchas barberías y salones de belleza en el formato tradicional todavía poseen sistemas de gestión de reservas obsoletos, en los que las reservas se gestionan a través de llamadas telefónicas, mensajería instantánea o bien una agenda en papel. Esta dependencia genera cuellos de botella operativos, como reservas erróneas, mal manejo de horarios, y nula capacidad para obtener datos sobre los usuarios. Como resultado, el crecimiento de la empresa queda limitado y su competitividad frente a cadenas ya digitalizadas queda igualmente reducida.

Diamond Barbershop, un negocio local en plena expansión, enfrenta precisamente estos desafíos. El aumento en su clientela, si bien es un indicador de éxito, ha puesto en evidencia las serias limitaciones de su modelo de gestión actual. La administración manual de citas consume un tiempo considerable del personal, es propensa a errores de agendamiento y no ofrece al cliente una forma autónoma y moderna de reservar sus servicios. Esta ineficiencia operativa no solo afecta la productividad interna, sino que también representa una barrera para la fidelización de clientes y la profesionalización de su servicio.

El objetivo de este trabajo es proponer una arquitectura de software para un sistema integral de control de reservas y administración para Diamond Barbershop, materializado en una aplicación web y una aplicación móvil. La solución busca no solo automatizar y optimizar el proceso de agendamiento de citas, sino también proporcionar a la administración herramientas eficientes para la gestión de su personal y clientes. Para lograrlo, se analizarán y aplicarán distintos estilos arquitectónicos, patrones de diseño y buenas prácticas de desarrollo, con el fin de construir una solución tecnológica de alta calidad que responda a las necesidades específicas del negocio y potencie su competitividad en el mercado actual.

1. DESCRIPCIÓN DEL CASO DE ESTUDIO

A nivel mundial, el sector de servicios de cuidado personal está experimentando una profunda transformación impulsada por la digitalización y las nuevas expectativas de los consumidores. Los clientes modernos valoran no solo la calidad del servicio, sino también una experiencia fluida y omnicanal, donde la autogestión de citas y la comunicación digital son clave. Para competir eficazmente, las empresas del sector deben implementar tecnologías que automaticen las reservas y personalicen la interacción con el cliente.

En el caso del Perú, el mercado de la belleza y el cuidado masculino muestra una tendencia de crecimiento sostenido, lo que aumenta la presión sobre los negocios locales para modernizar sus operaciones. Sin embargo, muchas barberías y salones peruanos aún enfrentan una brecha digital significativa, caracterizada por una alta dependencia de procesos manuales para la gestión de citas y una escasa adopción de herramientas de gestión empresarial. Estos problemas merman su eficiencia y limitan su capacidad para fidelizar a una clientela cada vez más digitalizada.

Diamond Barbershop, una barbería en crecimiento ubicada en Lima, se enfrenta directamente a estas dificultades. Su popularidad ha provocado que su sistema tradicional de reservas, basado en llamadas y mensajería instantánea, se vea completamente desbordado. La falta de automatización en negocios de servicio de alto contacto genera cuellos de botella operativos que impiden la escalabilidad. Por ello, se vuelve imperativo implementar un sistema de gestión centralizado que optimice los procesos y mejore la experiencia del cliente.

Por lo tanto, es evidente la necesidad de modernizar la gestión de Diamond Barbershop para mejorar su eficiencia operativa y fortalecer su posición en el mercado. La implementación de un sistema integral, compuesto por una aplicación web y una móvil, le permitiría a la barbería automatizar por completo su proceso de reservas, mejorar la gestión de su personal y clientes, y obtener datos valiosos para la toma de decisiones estratégicas, sentando así las bases para un crecimiento sostenible.

1.1 Problemas identificados

1.1.1 Gestión de Citas Manual y Propensa a Errores

La dependencia de llamadas telefónicas y WhatsApp para agendar citas genera una alta probabilidad de errores humanos, como el doble agendamiento, la pérdida de reservas o la comunicación incorrecta de horarios. Además que Espina-Romero et al. (2023) menciona que la preferencia de los usuarios actuales a tendencias digitales en cuanto a citas, es notoria. Por otro lado, esto afecta directamente la organización del trabajo y puede llevar a una notable disminución de la satisfacción del cliente y la productividad del personal.

1.1.2 Ausencia de un Historial Centralizado del Cliente

La barbería carece de una base de datos para registrar información de sus clientes, como su historial de servicios, preferencias o frecuencia de visitas. Según Nugroho, D. (2025), la ausencia de un sistema de gestión de relaciones con el cliente (CRM) impide aplicar estrategias de fidelización efectivas, perdiendo oportunidades de marketing personalizado y ventas recurrentes.

1.1.3 Falta de Visibilidad del Rendimiento del Negocio

La toma de decisiones se basa en la intuición en lugar de en datos concretos, ya que no existe un sistema que genere reportes sobre las horas de mayor demanda, los servicios más solicitados o el rendimiento individual de los barberos. Los negocios del sector que utilizan análisis de datos para guiar su estrategia experimentan tasas de crecimiento considerablemente más altas.

1.1.4 Alta Tasa de Inasistencias

La falta de un sistema de recordatorios automáticos contribuye a una elevada tasa de clientes que no se presentan a su cita, lo que genera pérdidas de ingresos y huecos improductivos en la agenda de los barberos. Se ha demostrado que la implementación de recordatorios automáticos reduce significativamente la tasa de inasistencias en negocios basados en citas.

1.1.5 Experiencia del Cliente Fragmentada y Limitada

Los clientes solo pueden reservar durante el horario comercial y dependen de la disponibilidad de una persona para responder. Esto crea una experiencia de usuario deficiente y una barrera para aquellos que prefieren la autogestión, ya que un alto porcentaje de consumidores modernos prefiere interactuar con empresas que ofrecen opciones de autoservicio disponibles 24/7.

1.1.6 Riesgos en la Escalabilidad del Negocio

El modelo de gestión actual es insostenible para el crecimiento. La idea de abrir una nueva sucursal, contratar más personal o manejar un volumen de clientes significativamente mayor es inviable con los procesos manuales existentes. Los modelos operativos no escalables son una de las principales barreras que impiden el crecimiento de las microempresas de servicios.

2. OBJETIVOS

2.1 Objetivo Principal:

Desarrollar e implementar una **arquitectura de software escalable, robusta y mantenible** para un sistema integral de gestión y reservas de la barbería *Diamond Barbershop*, que permita automatizar los procesos operativos clave, centralizar la información del negocio y mejorar la experiencia del cliente.

2.2. Objetivos Secundarios:

- **Diseñar una arquitectura multicapa y modular**, que separe de forma clara las capas de presentación, lógica de negocio y acceso a datos, facilitando la evolución, el mantenimiento y la incorporación de nuevas funcionalidades en el sistema de reservas.

- **Definir una infraestructura de despliegue escalable en la nube**, utilizando servicios como AWS Elastic Beanstalk y RDS, que garantice la disponibilidad, el rendimiento y la seguridad de la información de clientes, citas y servicios.

- **Automatizar la gestión de citas mediante componentes especializados**, diseñando servicios y APIs que permitan a las aplicaciones web y móvil gestionar reservas de forma consistente, reduciendo errores de agendamiento manual y la carga administrativa del personal.

- **Mejorar la experiencia y retención del cliente a nivel arquitectónico**, incorporando módulos y servicios para autogestión de reservas 24/7, manejo de historiales de servicios y envío de recordatorios automáticos (correo/notificaciones), integrados de manera segura con proveedores externos.

- **Implementar un módulo de reportes básicos sobre una base de datos centralizada**, que permita obtener indicadores clave del negocio (servicios más solicitados, volumen de citas atendidas), apoyando la toma de decisiones gerenciales.

- **Garantizar la seguridad y protección de los datos del sistema**, integrando mecanismos de autenticación y autorización basados en JWT, control de accesos por roles y encriptación de información sensible en los diferentes componentes de la arquitectura.

3. MARCO TEÓRICO

3.1. Estilos Arquitectónicos

Según Ford y Richards (2020), un estilo arquitectónico es una forma general y repetible de estructurar un sistema de software que define cómo se organizan sus componentes y cómo interactúan entre sí. Los estilos sirven como plantillas conceptuales que orientan el diseño de la arquitectura global.

Existen múltiples estilos arquitectónicos, cada uno con ventajas y desventajas según el contexto del sistema. Algunos de los más utilizados son:

Monolítico: Todos los módulos del sistema se integran en una sola aplicación. Es simple de desplegar, pero difícil de escalar y mantener.

En capas (Layered): Divide el sistema en niveles de responsabilidad, promoviendo la separación de preocupaciones y facilitando el mantenimiento.

Cliente-servidor: Separa las responsabilidades entre un cliente que solicita servicios y un servidor que los provee.

Microservicios: Fragmenta la aplicación en servicios independientes que se comunican entre sí mediante APIs.

Event-Driven (dirigido por eventos): Utiliza eventos como mecanismo de comunicación asíncrona entre componentes.

Para el sistema Diamond Barbershop, se adopta una arquitectura combinada en capas + cliente-servidor, ya que permite modularidad, escalabilidad y facilita la integración de aplicaciones web y móviles con un backend centralizado.

3.2. Patrones Arquitectónicos

Los patrones arquitectónicos son soluciones probadas a problemas recurrentes en el diseño de arquitecturas de software (Ford & Richards, 2020). Mientras que los estilos describen la estructura general, los patrones se enfocan en cómo resolver problemas específicos dentro de esa estructura.

Algunos patrones arquitectónicos relevantes son:

Modelo Vista Controlador (MVC): Divide la aplicación en tres componentes (modelo, vista y controlador) para separar la lógica de presentación y la lógica de negocio.

Model-View-ViewModel (MVVM): Variante moderna de MVC, que mejora la independencia entre la interfaz y la lógica de negocio mediante la introducción del ViewModel.

Microkernel (Plugin): Ideal para aplicaciones extensibles que necesitan integrar módulos adicionales sin alterar el núcleo.

Broker: Utilizado en sistemas distribuidos para coordinar la comunicación entre componentes cliente y servidor.

Event-Bus: Organiza la comunicación mediante eventos, permitiendo sistemas desacoplados y escalables.

Estos patrones son aplicados dependiendo del tipo de cliente y las necesidades del negocio. En este caso, el MVVM será implementado en la aplicación móvil y una arquitectura en capas en el backend para promover el desacoplamiento.

3.3. Patrones de Diseño de Software

Los patrones de diseño de software, a diferencia de los arquitectónicos, se aplican a un nivel más bajo, dentro de los módulos o clases del sistema. Son soluciones recurrentes para problemas de diseño de código, y fueron popularizados por el libro Design Patterns de Gamma et al. (1994).

Se clasifican en tres grupos:

Creacionales: Controlan la creación de objetos (Factory Method, Singleton, Builder).

Estructurales: Organizan clases y objetos para formar estructuras complejas (Adapter, Facade, Composite).

Comportamentales: Gestionan la comunicación entre objetos (Observer, Strategy, Command).

En el sistema propuesto se podrían aplicar, por ejemplo, Repository Pattern (para acceso a datos en Spring Boot), DTO Pattern (para transferir información entre capas) y Observer Pattern (para notificaciones o actualizaciones de estados en la app móvil).

3.4. Arquitectura en Capas

La arquitectura en capas (Layered Architecture) es uno de los estilos más utilizados en el desarrollo de software empresarial. Según Ford y Richards (2020), este estilo promueve la separación de responsabilidades, organizando el sistema en niveles donde cada capa ofrece servicios a la capa superior y depende de la inferior.

Las capas típicas son:

Capa de Presentación (UI): Interfaz con el usuario, implementada en la web y la app móvil.

Capa de Aplicación o Negocio: Contiene la lógica central del sistema (procesos de reservas, gestión de clientes, reportes).

Capa de Persistencia: Gestiona la interacción con la base de datos.

Capa de Datos: Responsable del almacenamiento físico de la información.

Este estilo es especialmente adecuado para sistemas como el de Diamond Barbershop, porque facilita la mantenibilidad, la prueba de componentes y la escalabilidad vertical (añadir nuevas funcionalidades sin alterar las capas existentes).

3.5. Patrón MVVM (Model–View–ViewModel)

El patrón MVVM, derivado de MVC, es ampliamente usado en desarrollo móvil moderno (Android, Flutter, etc.). Divide la aplicación en tres partes principales:

Model: Representa los datos y la lógica de negocio.

View: Interfaz visual que muestra los datos al usuario.

ViewModel: Intermediario que conecta la vista con el modelo, gestionando los datos de forma reactiva y evitando dependencias directas.

Este patrón mejora la prueba unitaria, el reuso de componentes y la separación de responsabilidades, características esenciales para una arquitectura móvil limpia y escalable. En este proyecto, el MVVM permitirá que la app móvil de reservas se comunique con el backend de forma eficiente mediante servicios RESTful.

3.6. Cloud Computing

Según Amazon Web Services (s. f.), la computación en la nube es la entrega bajo demanda de recursos informáticos a través de Internet con precios de pago por uso. En lugar de comprar, poseer y mantener centros de datos y servidores físicos, puede acceder a servicios tecnológicos, como potencia informática, almacenamiento y bases de datos, según sea necesario, a través de un proveedor de servicios en la nube como Amazon Web Services (AWS).

Según Mell & Grance (NIST, 2011), la nube se caracteriza por su escalabilidad, elasticidad y modelo de pago por uso.

Las ventajas principales incluyen:

Disponibilidad: Servicios accesibles desde cualquier lugar.

Escalabilidad: Ajuste dinámico de recursos.

Reducción de costos: Evita infraestructura física.

Mantenimiento centralizado: Las actualizaciones se gestionan desde el proveedor.

En el contexto del sistema para Diamond Barbershop, la nube permitirá alojar el backend, almacenar imágenes y gestionar notificaciones o respaldos automáticos.

3.7. Modelos de Servicio en la Nube: PaaS, IaaS y SaaS

Según Google Cloud (s.f), Cloud computing tiene tres modelos principales de servicios en la nube: IaaS (infraestructura como servicio), PaaS (plataforma como servicio) y SaaS (software como servicio). Además, es posible que conozcas los términos IaaS, PaaS y SaaS como "servicios de nube" o categorías de cloud computing, pero todos ellos hacen referencia a cómo usas la nube en tu empresa y el nivel de gestión del que eres responsable en tus entornos en la nube.

Además de estas tres categorías generales, es posible que encuentres otros tipos de servicios en la nube que incorporan otras tecnologías, como contenedores. Por ejemplo, la creciente

adopción de contenedores y arquitecturas de microservicios ha derivado en la aparición de CaaS (contenedores como servicio).

"Como servicio" suele indicar que el modelo de servicio lo ofrece un tercero en la nube. En otras palabras, no tienes que comprar, gestionar ni usar hardware, software, herramientas ni aplicaciones desde un centro de datos on‑premise. En su lugar, puedes pagar una suscripción o pagar en función del consumo (pago por uso) para acceder a lo que necesites bajo demanda a través de una conexión a Internet.

Para este proyecto, el modelo PaaS es el más adecuado, ya que permite desplegar el backend desarrollado en Spring Boot sin la necesidad de configurar servidores manualmente, reduciendo costos y simplificando la administración.

3.8. Transformación Digital y Sistemas de Información en PYMES

La transformación digital implica integrar tecnologías digitales en todos los procesos de una organización para mejorar su eficiencia, experiencia del cliente y toma de decisiones. Espina-Romero et al. (2024) afirman que las PYMES en Lima que adoptan herramientas digitales logran mayor sostenibilidad y competitividad.

Los sistemas de información son el núcleo de esta transformación, pues permiten automatizar operaciones, centralizar datos y ofrecer servicios más personalizados. En el caso de Diamond Barbershop, la digitalización mediante un sistema de reservas en línea y gestión de clientes representa un paso clave hacia su profesionalización y expansión.

3.9. Atributos de Calidad del Software

Los atributos de calidad determinan cómo de bien cumple un sistema con sus requisitos no funcionales. Según Ford y Richards (2020), los principales atributos que guían el diseño arquitectónico son:

Disponibilidad: Garantizar que el sistema esté accesible el mayor tiempo posible.

Escalabilidad: Soportar un aumento en la cantidad de usuarios o servicios.

Rendimiento: Minimizar el tiempo de respuesta y optimizar el uso de recursos.

Mantenibilidad: Facilitar la corrección de errores y la evolución del software.

Seguridad: Proteger los datos del cliente y del negocio contra accesos no autorizados.

Usabilidad: Ofrecer una experiencia intuitiva al usuario final.

El cumplimiento de estos atributos en el sistema de Diamond Barbershop se logra mediante una arquitectura modular, el uso de Spring Boot y servicios REST, garantizando una base técnica sólida, confiable y preparada para el crecimiento futuro.

4. DESARROLLO

4.1. Identificación de los requerimientos arquitectónicos

El sistema de gestión para la barbería requiere una arquitectura que responda tanto a la operación diaria del negocio como a su futura expansión. Dado que la plataforma necesita dos aplicaciones móviles (una para clientes y otra para administradores) junto con un backend centralizado, el primer requerimiento arquitectónico fundamental es la **separación Cliente–Servidor**. Esta estructura permite que ambas aplicaciones consuman servicios desde un único backend desarrollado en Spring Boot, garantizando coherencia en la información, sincronización en tiempo real y facilidad para introducir mejoras sin afectar a los clientes finales.

En segundo lugar, se requiere una **base de datos centralizada**, estructurada en MySQL y diseñada para manejar información crítica como usuarios, barberos, horarios, reservas, servicios y comprobantes de pago. Esta base de datos debe asegurar integridad, consistencia y trazabilidad, especialmente en procesos sensibles como la asignación de horarios y la confirmación de reservas. Al concentrar todos los datos en un solo repositorio, el sistema garantiza que tanto clientes como administradores interactúen con información actualizada y confiable.

Un tercer requerimiento arquitectónico es la **modularidad del backend**, organizada en capas bien definidas (controladores, servicios, repositorios y entidades). Esta estructura permite ampliar o modificar funcionalidades sin afectar otros módulos. Por ejemplo, el sistema de reservas, el módulo de horarios y la gestión de barberos pueden evolucionar de manera independiente, facilitando la mantenibilidad y la incorporación de nuevas capacidades, como la futura implementación de notificaciones push o la integración con otros medios de pago.

Asimismo, debido a la naturaleza visual del negocio y a la necesidad de gestionar imágenes de barberos y comprobantes de pago, la arquitectura debe integrar un servicio externo como **Cloudinary**. Esta integración es un requerimiento clave, ya que permite almacenar imágenes de forma segura, optimizada y sin sobrecargar el servidor de la aplicación, garantizando tiempos de respuesta adecuados y un manejo eficiente de recursos.

Otro requerimiento fundamental es la implementación de un esquema de **seguridad basado en autenticación JWT**, que permita diferenciar y controlar el acceso según los dos grandes roles del sistema: cliente y administrador. Este mecanismo garantiza que solo usuarios autorizados accedan a funcionalidades críticas, como la aprobación de reservas o la modificación de la información de la barbería. Junto con el cifrado de contraseñas y las validaciones de entrada, este enfoque protege la información sensible de la empresa y de sus clientes.

4.2. Requisitos Funcionales y No Funcionales

4.2.1. Requisitos Funcionales (RF)

Módulo	Requerimiento Funcional
Autenticación	RF01 - El sistema deberá contar con un login específico para las 2 aplicaciones (usuario administrador y usuario cliente)
	RF02 - El sistema deberá permitir la recuperación de contraseña de los usuarios con ayuda del correo electrónico usado para la creación de este.
	RF03 - El sistema deberá realizar la autorización debida a ciertos endpoints según rol.
Servicios	RF04 - El sistema deberá permitir gestionar al usuario administrador gestionar el modelo servicios, es decir, poder listar, agregar, editar y eliminar.
	RF05 - El sistema deberá permitir al usuario cliente poder visualizar los servicios en el aplicativo cliente.
Barberos	RF06 - El sistema deberá permitir gestionar al usuario administrador gestionar el modelo barberos, es decir, poder listar, agregar, editar y eliminar.
	RF07 - El sistema deberá permitir al usuario administrador poder gestionar la asignación de horarios semanales (mañana/tarde/noche) de los barberos disponibles.
Reservas	RF08 - El sistema deberá permitir al usuario cliente poder realizar reservas de los servicios brindados en los horarios disponibles.
	RF09 - El sistema deberá calcular el tiempo estimado del servicio para reservar dicho horario al barbero asignado.
	RF10 - El sistema deberá permitir al usuario poder visualizar y elegir las fechas y horarios disponibles al momento de la elección.
	RF11 - El sistema deberá permitir al usuario administrador poder aprobar o cancelar las reservas.
Pagos	RF12 - El sistema deberá permitir al usuario cliente subir la captura del comprobante en caso desee.
	RF13 - El sistema deberá permitir al usuario administrador poder visualizar la imagen subida por el usuario cliente para revisar la veracidad de este.
Notificaciones	RF14 - El sistema deberá notificar por whatsapp al usuario cliente cuando se cree su usuario.

	RF15 - El sistema deberá notificar por whatsapp al usuario cliente cuando se confirme su reserva.
Historial y Recompensas	RF16 - El sistema deberá permitir visualizar su historial de citas al cliente.
	RF17 - El sistema deberá calcular la cantidad de citas realizadas para poder cobrar su recompensa.
Reportes	RF18 - El sistema deberá permitir visualizar al cliente administrador un reporte de todos los ingresos generados por cliente.
	RF19 - El sistema deberá permitir al administrador visualizar los reportes usando distintos filtros.
Landing Page	RF20 - El sistema deberá contar además con una landing page aparte, donde el usuario final podrá descargar la apk para usar la aplicación.
	RF21 - El sistema deberá contar dentro de la landing page con las instrucciones claras y precisas para la instalación y uso.

4.2.2. Requisitos No Funcionales (RNF)

Categoría	**Requerimiento Arquitectónico**
Interoperabilidad	El sistema deberá estar abierto a integraciones, en este caso debe poder permitir realizar la integración de aplicaciones móviles y aplicaciones web para un futuro.
Portabilidad	El software debe ser capaz de migrar a distintos entornos tecnológicos, como servidores locales o la nube
Seguridad	Se requiere control de accesos basados en roles, cifrado de datos sensibles y auditoría de acciones del usuario.
Visibilidad en tiempo real	Se requiere control de accesos basados en roles, cifrado de datos sensibles como contraseñas.
Modularidad	La arquitectura debe permitir añadir o modificar funcionalidades sin afectar el sistema completo.

4.3. Requisitos de calidad

Los requisitos de calidad son determinantes para garantizar la fiabilidad, el rendimiento y la sostenibilidad del sistema en el largo plazo.

- **Seguridad**. El sistema debe implementar protocolos de autenticación robustos, cifrado de datos sensibles y control de roles. Esto protege la información de cliente, asegurando confianza en el uso del sistema.
- **Rendimiento**. El sistema debe ofrecer tiempos de respuesta estables y rápidos, incluso en momentos de alta concurrencia. Este requisito impacta directamente en la productividad de los empleados y en la satisfacción de los usuarios finales.
- **Mantenibilidad**. Un diseño modular y bien documentado permitirá realizar cambios, correcciones o ampliaciones sin interrumpir el funcionamiento general. La mantenibilidad es clave para asegurar que el sistema no quede obsoleto y pueda evolucionar con la empresa.
- **Portabilidad**. El software debe ser capaz de migrar a distintos entornos tecnológicos, como servidores locales o la nube, según las necesidades futuras de la organización.
- **Confiabilidad**. La estabilidad operativa del sistema se garantiza mediante pruebas exhaustivas y monitoreo continuo, reduciendo la probabilidad de fallos críticos que interrumpan la operación diaria.

4.4. Selección de estilos y tácticas de arquitectura

4.4.1. Estilos de arquitectura cliente-servidor

El sistema propuesto adopta también el estilo de arquitectura cliente-servidor, en el cual los clientes consumen servicios expuestos por un servidor centralizado. Bajo este enfoque, las aplicaciones móviles actúan como clientes ligeros que envían solicitudes a un backend común desplegado en la nube, responsable de procesar la lógica de negocio y acceder a la base de datos.

- Cliente: Incluye la aplicación móvil para administradores, la aplicación móvil para clientes y la página web (landing y formulario de recuperación de contraseña). Estos clientes se encargan de la interacción con el usuario, el manejo de formularios y la presentación de la información, comunicándose con el servidor a través de peticiones HTTPS a una API REST.

- Servidor: Corresponde al backend desarrollado en Spring Boot, desplegado en AWS Elastic Beanstalk. Este componente procesa las solicitudes de los clientes, aplica las reglas de negocio (gestión de citas, barberos, servicios, horarios y reportes), gestiona la autenticación y autorización mediante JWT y coordina el acceso a la base de datos MySQL alojada en AWS RDS, así como a servicios externos como el proveedor de correo y Cloudinary.

Este estilo arquitectónico permite centralizar la lógica del sistema y los datos en un único servidor, facilitando la seguridad, el control de acceso, el mantenimiento y la evolución de la solución. Al mismo tiempo, posibilita que múltiples tipos de clientes (aplicaciones móviles y web) consuman los mismos servicios de forma consistente, favoreciendo la escalabilidad y la reutilización de la funcionalidad existente.

4.4.2. Estilos de arquitectura en capas

El backend de nuestro sistema se fundamenta en una arquitectura en capas, que organiza el sistema en niveles con responsabilidades claramente delimitadas. Este estilo permite mantenibilidad, seguridad y trazabilidad, asegurando que cada capa pueda evolucionar sin afectar las demás.

- Capa de presentación: Gestiona las solicitudes HTTP, middlewares de seguridad (CORS). Esta capa será la puerta a la conexión con nuestros aplicativos móviles tanto para administrador como para cliente.
- Capa de controladores: Encargada de recibir las peticiones, validar datos iniciales y coordinar la respuesta hacia los servicios.
- Capa de servicios: Contiene la lógica de negocio (barberos, horarios, usuarios, valoraciones), reglas de validación y procesos críticos.
- Capa de repositorios: Abstracción del acceso a datos mediante JPA, siguiendo el patrón Repository para garantizar independencia de la base de datos.
- Capa de datos: Base de datos MySQL, con índices y constraints que aseguran integridad, rendimiento y consistencia.

Este estilo asegura que el sistema sea modular, fácil de mantener y seguro, con responsabilidades bien definidas que facilitan pruebas y documentación.

4.5. Justificación del estilo elegido según necesidades del caso

Para el sistema de gestión y reservas de la barbería *Diamond Barbershop* se ha elegido una arquitectura en capas bajo el estilo cliente-servidor, ya que se ajusta de manera adecuada al contexto del negocio, al uso de aplicaciones móviles y web, y a los requisitos de escalabilidad y mantenimiento a futuro.

En este enfoque, las aplicaciones cliente (app móvil para administradores, app móvil para clientes y páginas desplegadas en Vercel) consumen una API REST centralizada desarrollada en Spring Boot y desplegada en AWS Elastic Beanstalk. El backend organiza su lógica interna en capas bien definidas (presentación, controladores, servicios, repositorios y acceso a datos), lo que permite que la complejidad del sistema se gestione de forma gradual y estructurada.

Principales razones de esta elección:

- Soporte a múltiples clientes: El estilo cliente-servidor facilita que distintas interfaces (aplicación móvil de cliente, aplicación de administrador y landing web) consuman los mismos servicios backend. Esto garantiza una experiencia consistente independientemente del dispositivo y simplifica la incorporación de nuevos clientes en el futuro (por ejemplo, una futura web administrativa).

- Modularidad y mantenibilidad: La arquitectura en capas separa claramente responsabilidades: controladores para la recepción de peticiones, servicios para la

lógica de negocio (gestión de citas, barberos, horarios, servicios), repositorios para el acceso a datos y una capa de datos centralizada en MySQL (AWS RDS). Esta organización reduce el acoplamiento, facilita la lectura del código y permite realizar cambios o correcciones en una capa sin afectar al resto del sistema.

- Escalabilidad y rendimiento: Al centralizar la lógica de negocio en el servidor y desplegarla en la nube (AWS Elastic Beanstalk), es posible escalar vertical u horizontalmente el backend según la demanda, manteniendo tiempos de respuesta aceptables en horarios de alta concurrencia. La separación en capas también permite optimizar de forma específica la capa de acceso a datos (índices, consultas, pooling de conexiones) sin alterar la lógica de presentación o negocio.

- Seguridad y control de acceso: Concentrar las reglas de autenticación y autorización en el servidor, mediante Spring Security y JWT, facilita aplicar políticas de seguridad coherentes para todas las aplicaciones cliente. Además, el despliegue dentro de una VPC y el uso de HTTPS en las comunicaciones minimizan riesgos de exposición de datos sensibles como información de clientes y reservas.

- Interoperabilidad con servicios externos: La API backend actúa como punto central de integración con servicios de terceros, como el proveedor de correo electrónico para notificaciones y Cloudinary para el almacenamiento de imágenes. El uso de contratos REST y formato JSON estándar facilita la incorporación o sustitución de estos servicios sin impactar en las aplicaciones cliente.

- Evolución futura del sistema: Esta combinación de arquitectura en capas y cliente-servidor permite que la solución crezca de forma progresiva. En caso de que el negocio lo requiera, módulos específicos (por ejemplo, reportes o notificaciones) podrían desacoplarse del monolito y evolucionar hacia servicios más especializados sin necesidad de rediseñar completamente la arquitectura.

En conjunto, estos argumentos muestran que la arquitectura seleccionada responde tanto a las necesidades actuales de *Diamond Barbershop* (automatización de reservas, acceso móvil y web, seguridad de datos) como a los posibles escenarios de crecimiento y mantenimiento a largo plazo del sistema.

4.6. Aplicación de tácticas para alcanzar los atributos de calidad

Atributo de calidad	Tácticas aplicadas en el sistema	Impacto esperado
Seguridad	Autenticación JWT en cookies HTTP-only, hashing de contraseñas con bcrypt, control de roles RBAC y CORS seguro	Protección de datos sensibles, reducción de accesos no autorizados y confianza en el sistema

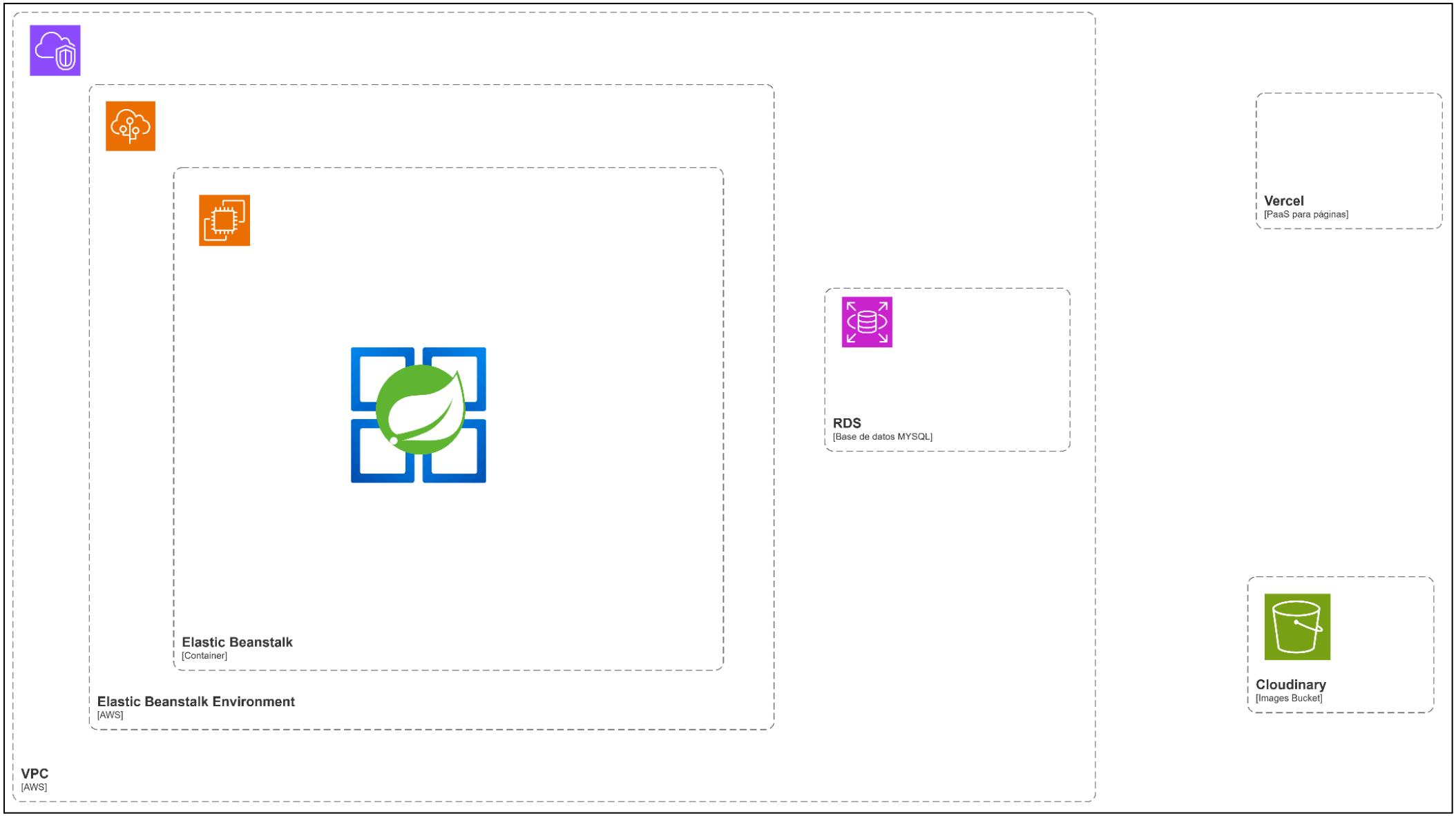
Rendimiento	Índices en BD, consultas optimizadas, paginación en listados grandes,	Respuestas rápidas y estables incluso en escenarios de alta concurrencia
Disponibilidad	Health checks en /health, manejo centralizado de errores, logging estructurado, backups automáticos de la base de datos	Operatividad continua, detección temprana de fallos y recuperación eficiente
Confiabilidad	Pruebas exhaustivas de endpoints, rollback en transacciones críticas, monitoreo de métricas de rendimiento	Reducción de fallos críticos, estabilidad operativa y menor riesgo de interrupciones
Mantenibilidad	Arquitectura modular en capas, Repository Pattern, documentación Swagger/OpenAPI.	Facilidad para realizar cambios, correcciones o ampliaciones sin afectar el sistema
Interoperabilidad	APIs RESTful para conexión con el frontend, formato JSON estándar	Integración fluida con servicios externos y reutilización de APIs en distintos contextos
Escalabilidad	Servicios stateless, despliegue en contenedores Docker, preparación para balanceo de carga y microservicios futuros	Crecimiento progresivo del sistema sin degradar el rendimiento
Portabilidad	Configuración mediante variables de entorno, despliegue flexible en nube o servidores locales	Migración sencilla entre entornos tecnológicos según necesidades futuras

4.7. Diseño de la arquitectura

Para esta oportunidad se optó por tener una infraestructura cloud que sea de fácil acceso tanto para los desarrolladores actuales como para futuros desarrolladores, se planteó debido al contexto actual (no se cuenta con servidores actualmente) una arquitectura basada en cloud en términos de infraestructura.

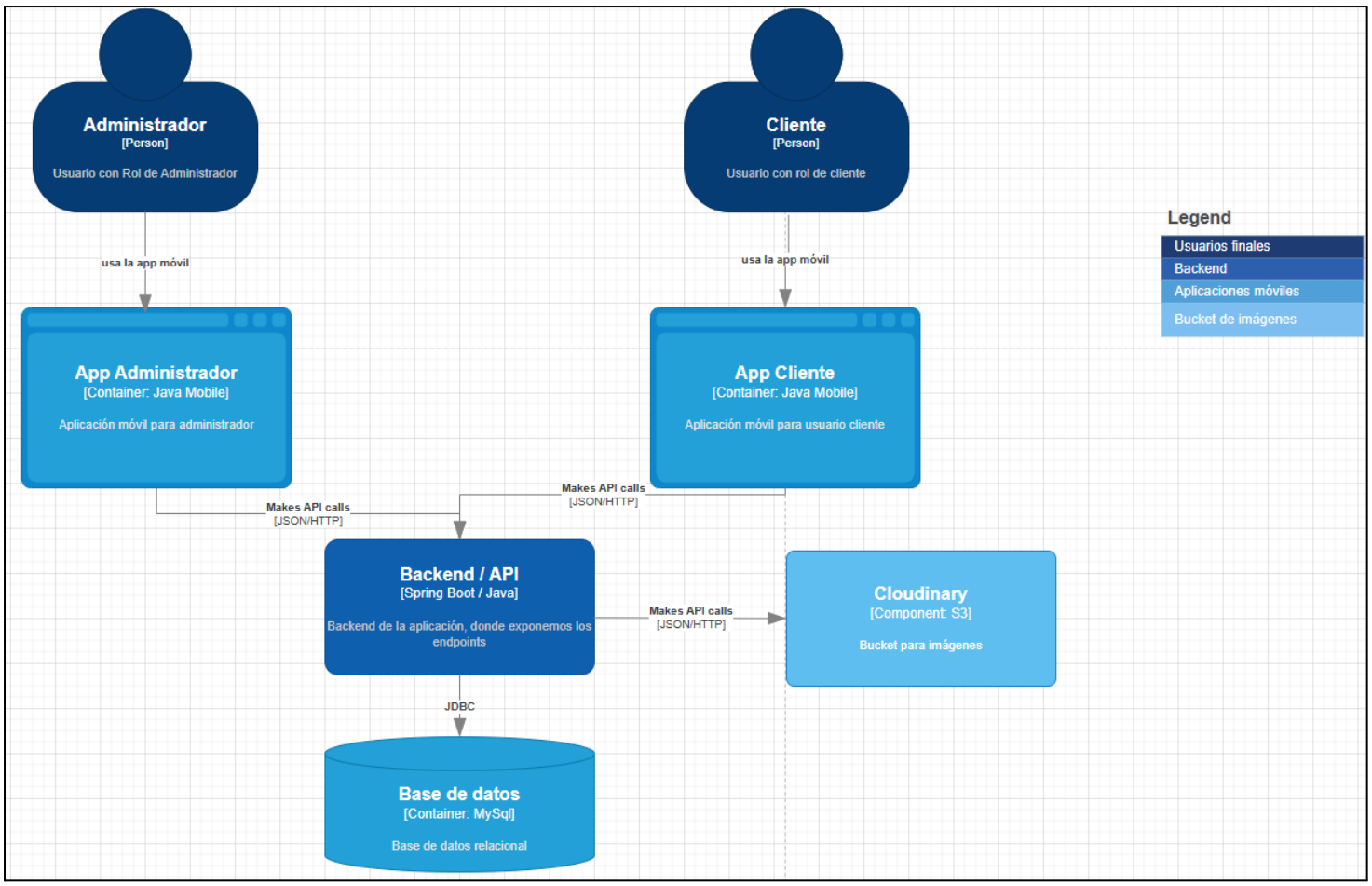
4.7.1. Diagrama General Cloud

El diagrama general cloud nos muestra los distintos recursos utilizados para lograr que funcione el sistema, teniendo en cuenta la utilización de “Elastic Beanstalk” por su facilidad para despliegue de archivos *build,* en este caso un archivo “.jar”, este necesita de una Virtual Private Cloud para vivir, además de también aprovechar los recursos de AWS y utilizar RDS para la base de datos. Considerar también la utilización de Cloudinary, sistema de almacenamiento basado en S3, y Vercel, PaaS basado también en AWS que nos facilita el despliegue de páginas web sencillas.

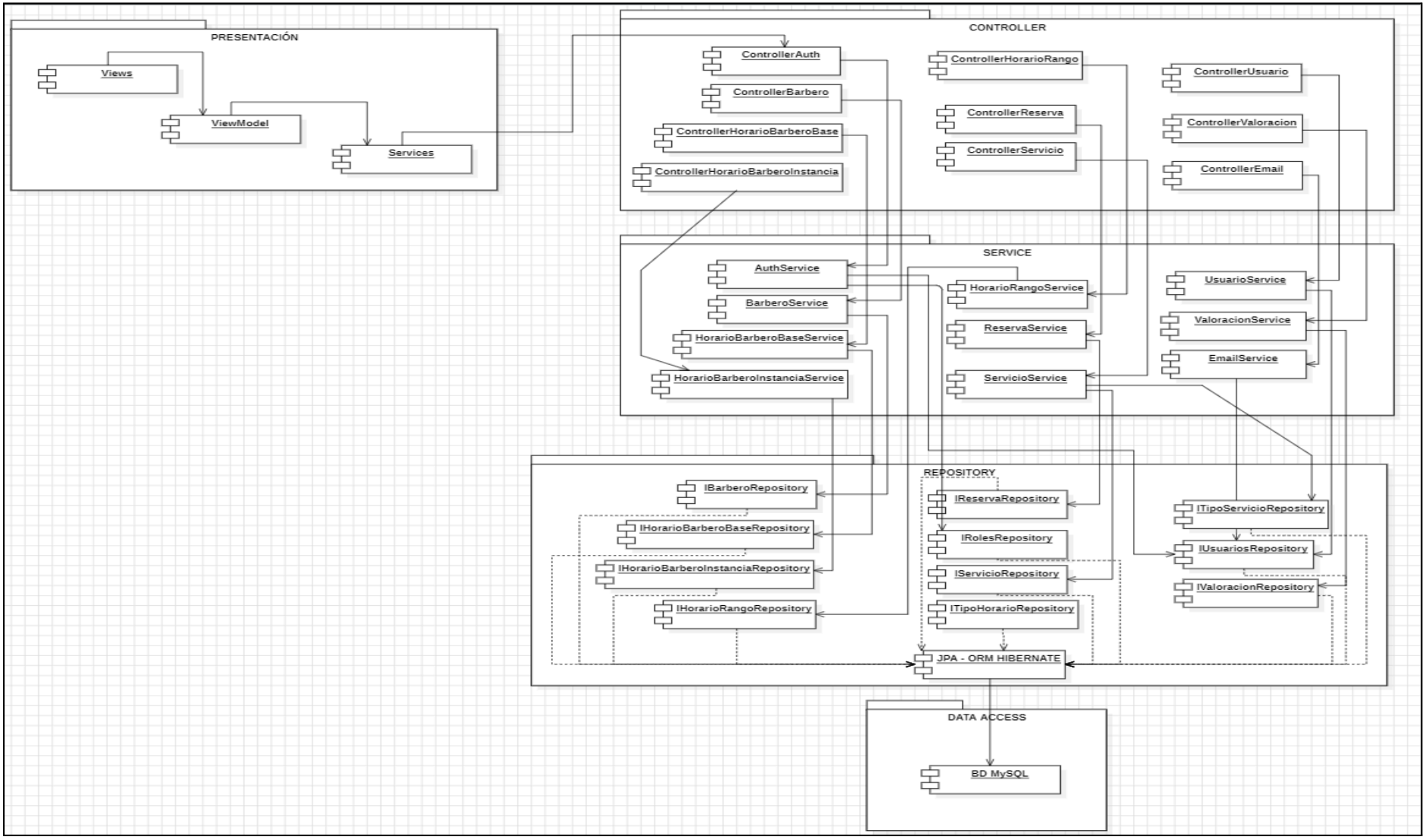

Vercel
[PaaS para páginas]
RDS
[Base de datos MYSQL]
Elastic Beanstalk
[Container]
Elastic Beanstalk Environment
[AWS]
Cloudinary
[Images Bucket]
VPC
[AWS]

4.7.2. Diagrama C4

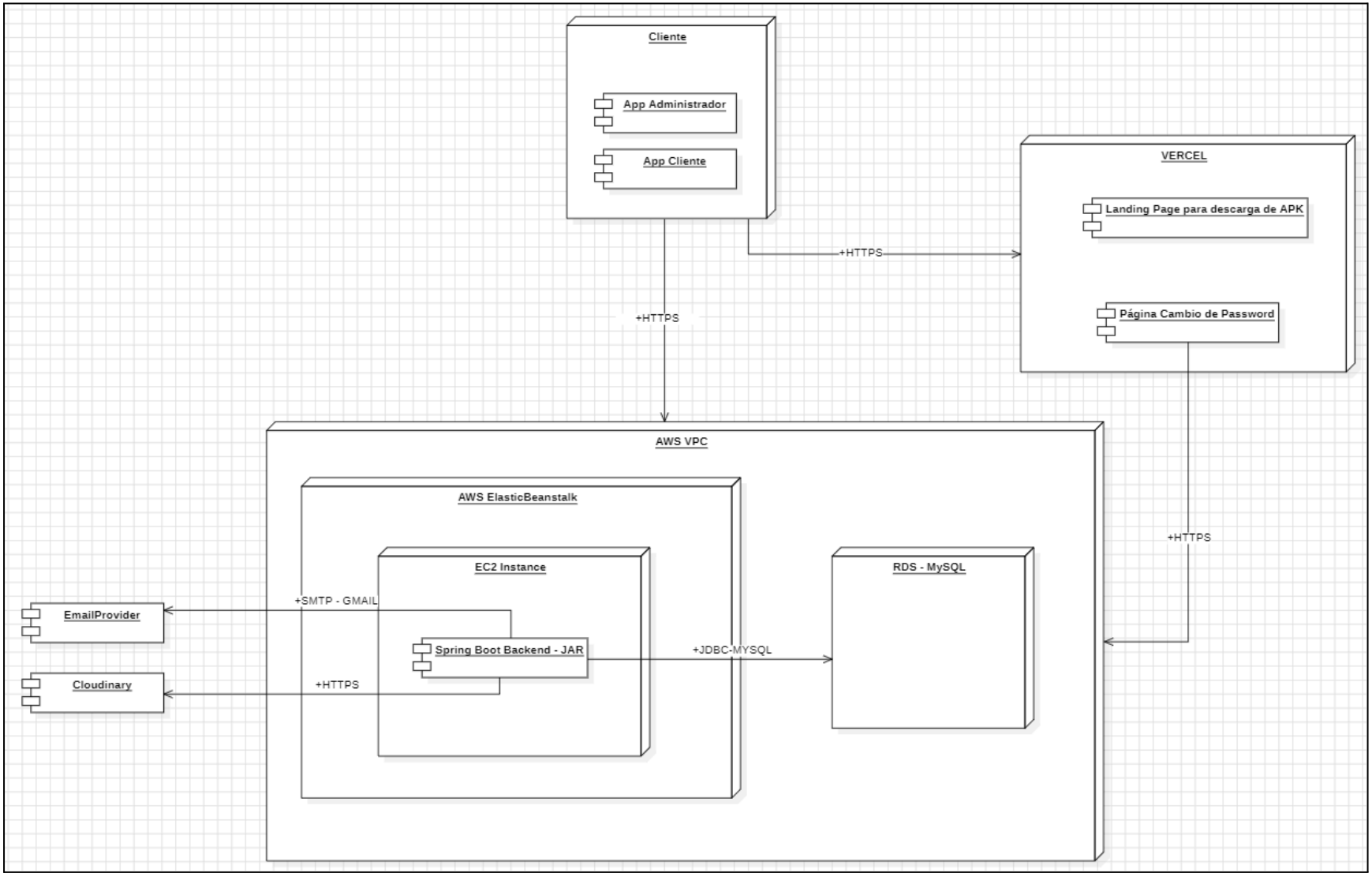
En el diagrama C4 se observa de manera general el sistema completo sin infraestructura, para visualizar a modo general el flujo de las 2 aplicaciones móviles hasta llegar al backend.



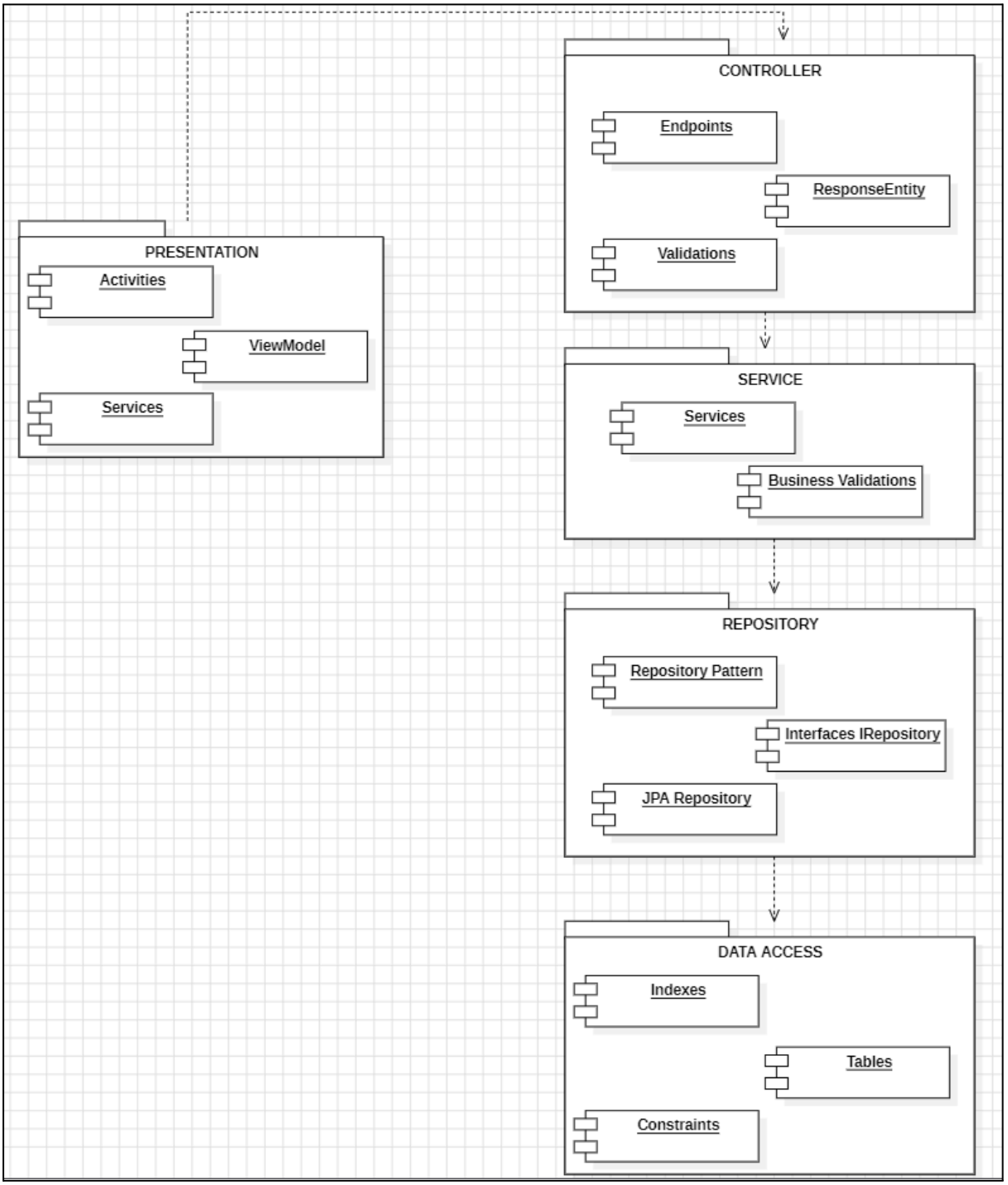
4.8. Diagrama de componentes



4.9. Diagrama de despliegue

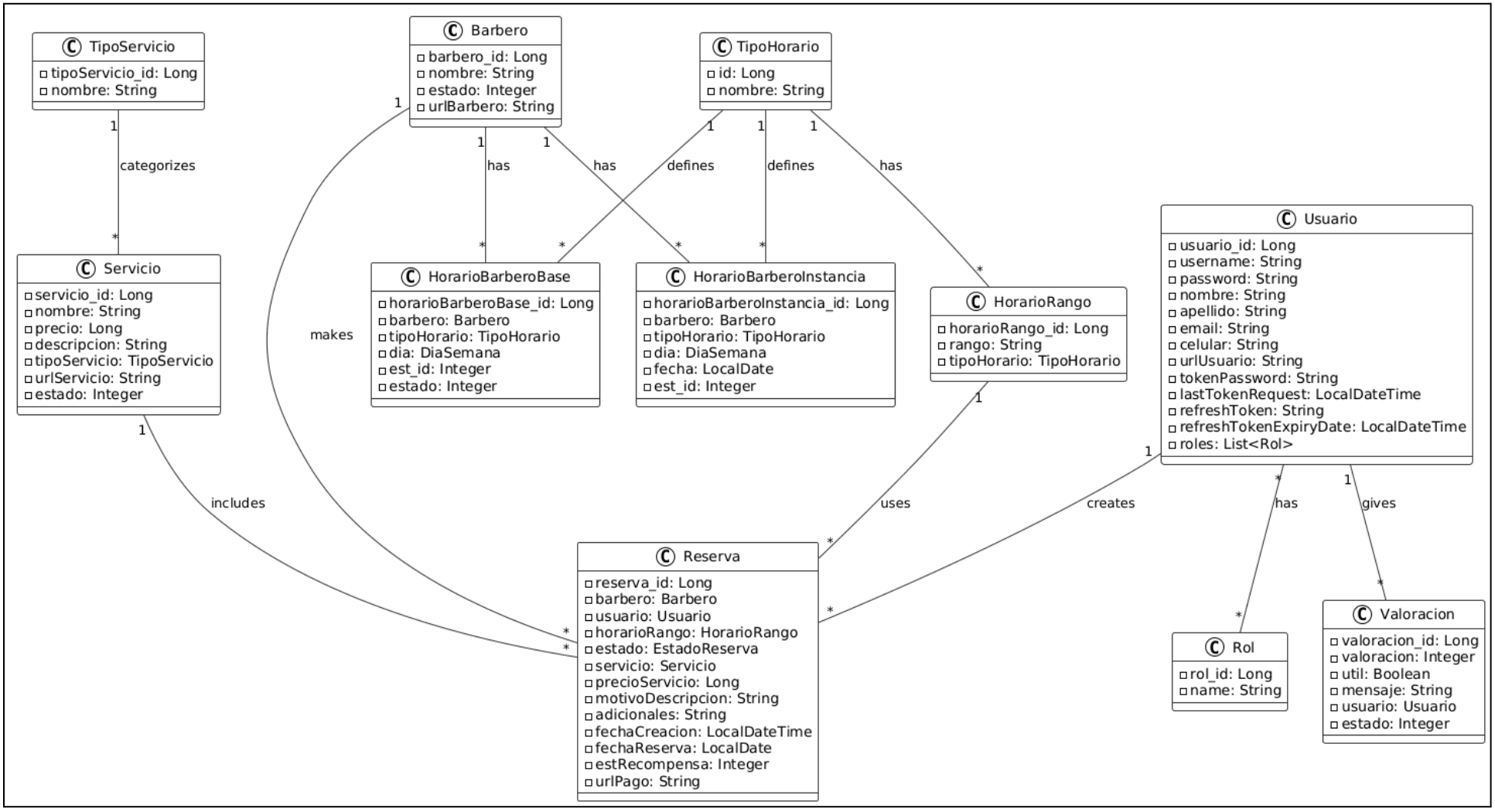


4.10. Diagrama de capas



4.11. Vistas arquitectónicas

4.11.1 Diagrama de Clases



4.12. Aplicación de patrones arquitectónicos

4.12.1. Arquitectura en capas

En el backend del sistema de gestión y reservas para Diamond Barbershop se aplica de forma explícita el patrón de arquitectura en capas, organizando el código en niveles lógicos con responsabilidades bien delimitadas. Este patrón se materializa en el proyecto Spring Boot mediante la separación en paquetes de controladores, servicios, repositorios, modelos/entidades, DTOs y configuración de seguridad, lo que refuerza la mantenibilidad y el desacoplamiento entre componentes.

- Capa de presentación (API REST / Controladores):
  Implementada a través de los controladores de Spring (@RestController), expone los endpoints HTTP consumidos por las aplicaciones móviles y las páginas desplegadas en Vercel. Esta capa se encarga de recibir las solicitudes, mapear parámetros, validar datos básicos y devolver respuestas estandarizadas (ResponseEntity, códigos de estado, mensajes de error), sin contener lógica de negocio compleja.

- Capa de servicios (lógica de negocio):
  Incluye las clases de servicio que concentran los procesos centrales del dominio: gestión de usuarios, barberos, servicios, horarios, reservas, pagos y valoraciones. Aquí se aplican reglas de negocio, validaciones específicas, cálculo de recompensas, transición de estados de la reserva y coordinación con servicios externos (Cloudinary, proveedor de correo), manteniendo esta lógica aislada de los detalles de transporte HTTP o de persistencia.

- Capa de repositorios (acceso a datos):
  Implementada mediante interfaces de Spring Data JPA, siguiendo el Repository Pattern. Esta capa abstrae el acceso a la base de datos MySQL en AWS RDS, encapsulando operaciones CRUD, consultas personalizadas y uso de paginación. De esta forma, la lógica de negocio no depende de consultas SQL concretas ni de la tecnología de persistencia, lo que facilita cambios futuros en el motor de base de datos o en la estrategia de acceso.

- Capa de datos (modelo persistente):
  Corresponde al esquema relacional en MySQL (tablas, índices, constraints y relaciones) y a las entidades JPA que lo representan. Esta capa garantiza integridad referencial, consistencia de la información y buen rendimiento en operaciones frecuentes como consultas de horarios disponibles, reservas por usuario o reportes básicos de negocio.

La aplicación de este patrón de arquitectura en capas permite que el sistema sea modular, fácil de probar y seguro, ya que cada cambio se limita a la capa correspondiente, reduciendo

el riesgo de introducir errores en otras partes del sistema y facilitando la documentación y evolución de la solución.

4.12.2. Modularización por dominios (arquitectura modular)

Además de la separación en capas, el backend adopta una arquitectura modular por dominios, donde el código se organiza en paquetes que agrupan componentes relacionados con un mismo contexto funcional. Este enfoque complementa la arquitectura en capas al añadir una vista horizontal basada en las áreas de negocio de la barbería.

A nivel de estructura de paquetes se distinguen, entre otros, los siguientes módulos:

- Módulo de Autenticación y Seguridad:
  Agrupa controladores, servicios, configuraciones y filtros relacionados con el login, emisión y validación de tokens JWT, recuperación de contraseña y control de accesos por rol (cliente, administrador). Centralizar estas responsabilidades facilita la aplicación de políticas de seguridad de forma coherente en todo el sistema.

- Módulo de Usuarios:
  Gestiona el ciclo de vida de los usuarios (registro, actualización de perfil, subida de imagen de perfil). Incluye DTOs específicos, servicios de mapeo y reglas de validación para asegurar la consistencia de la información personal.

- Módulo de Barberos y Horarios:
  Contiene las clases encargadas de administrar barberos, horarios base, rangos horarios e instancias de disponibilidad. Este módulo concentra la lógica de generación de horarios semanales, confirmación de turnos y exposición de la disponibilidad hacia el módulo de reservas, manteniendo una alta cohesión interna.

- Módulo de Servicios y Pagos:
  Reúne la gestión de los servicios que ofrece la barbería (tipos, duración, precio, estado) y las operaciones relacionadas con comprobantes de pago. Aquí se orquesta también la integración con Cloudinary para el almacenamiento de imágenes asociadas a servicios o comprobantes.

- Módulo de Reservas y Recompensas:
  Implementa la lógica central del sistema: creación, actualización y transición de estados de la reserva (creada, confirmada, realizada, cancelada), cálculo de recompensas por número de citas completadas y validación de disponibilidad de barberos. La combinación con patrones de diseño como State y Decorator permite gestionar de forma clara las reglas asociadas al ciclo de vida de las reservas.

- Módulo de Valoraciones y Reportes:
  Administra las valoraciones que realizan los clientes y la generación de reportes básicos para el administrador (reservas realizadas en un rango de fechas, ingresos por

cliente, horarios de mayor demanda). Este módulo se apoya en consultas agregadas y en la base de datos centralizada para ofrecer visibilidad sobre el desempeño del negocio.

Esta modularización, combinada con la arquitectura en capas, ofrece varias ventajas arquitectónicas:

- Aumenta la cohesión dentro de cada módulo y reduce el acoplamiento entre áreas funcionales.
- Facilita el trabajo en equipo, permitiendo que distintos desarrolladores se enfoquen en módulos específicos sin interferir con los demás.
- Prepara el sistema para una evolución futura hacia servicios más independientes (por ejemplo, convertir el módulo de reportes o de notificaciones en servicios separados), sin necesidad de rediseñar desde cero la arquitectura.

En conjunto, la aplicación de estos patrones arquitectónicos garantiza que la solución para Diamond Barbershop sea organizada, extensible y preparada para acompañar el crecimiento del negocio en los próximos años.

4.13. Descripción de patrones utilizados

4.13.1. Builder Pattern

El Builder Pattern es un patrón creacional que permite construir objetos complejos paso a paso, separando la lógica de construcción de la representación final. Esto mejora la legibilidad, evita constructores con demasiados parámetros y facilita la creación de variantes del mismo objeto sin duplicar código.

Ubicación en el sistema:

Se aplica en 8 DTOs: DtoReservaResponse, DtoReserva, DtoRegistro, DtoBarberoResponse, DtoUsuarioResponse, DtoServicioResponse, DtoValoracion, DtoValoracionResponse.

Responsabilidad:

Simplificar y estandarizar la construcción de los DTOs, permitiendo crear objetos inmutables y expresivos, especialmente cuando se manejan muchos atributos opcionales o combinaciones de campos.

4.13.2. Factory Pattern

El Factory Pattern es un patrón creacional que encapsula la lógica de creación de objetos en una clase dedicada (la fábrica). De esta manera, el código cliente no necesita conocer los detalles de cómo se instancian las entidades, lo que reduce el acoplamiento y centraliza las reglas de inicialización.

Ubicación en el sistema:

Clases BarberoFactory y ReservaFactory.

Responsabilidad:

Encapsular la lógica compleja de creación de entidades de dominio (por ejemplo, inicialización de valores por defecto, transformación desde DTOs, asignación de estados iniciales), evitando que esta lógica se repita en controladores o servicios.

4.13.3. State Pattern

El State Pattern es un patrón de comportamiento que permite que un objeto cambie su comportamiento cuando cambia su estado interno. Cada estado se modela como una clase independiente, lo que facilita manejar reglas de transición y operaciones válidas según el estado actual.

Ubicación en el sistema:

Interface ReservaState y cuatro implementaciones concretas: Creada, Confirmada, Realizada, Cancelada.

Responsabilidad:

Modelar la máquina de estados de una reserva y controlar de forma explícita las transiciones entre estados. Esto permite aplicar validaciones específicas (por ejemplo, qué acciones se permiten en cada estado) y facilita la evolución de la lógica de negocio asociada al ciclo de vida de una reserva.

4.13.4. Decorator Pattern

El Decorator Pattern es un patrón estructural que permite añadir responsabilidades adicionales a un objeto de forma dinámica, envolviéndolo en otros objetos decoradores sin modificar su código original. Es una alternativa flexible a la herencia para extender funcionalidades.

Ubicación en el sistema:

Interface ReservaDecorador y tres decoradores concretos: ValidacionDecorador, RecompensaDecorador, DisponibilidadDecorador.

Responsabilidad:

Enriquecer el procesamiento de reservas y sus DTOs añadiendo, de forma independiente y combinable, funcionalidades de validación, cálculo de recompensas y verificación de disponibilidad. Cada decorador añade una preocupación concreta, manteniendo el código más limpio y desacoplado.

4.13.5. Repository Pattern

El Repository Pattern es un patrón de acceso a datos que actúa como una capa intermedia entre la lógica de negocio y la fuente de datos. Su objetivo es abstraer los detalles de persistencia (consultas SQL, ORM, etc.) y ofrecer una interfaz orientada al dominio (por ejemplo, guardarReserva, buscarBarberoPorId).

Ubicación en el sistema:

Nueve interfaces de repositorio: IBarberoRepository, IReservaRepository, IUsuariosRepository, entre otras.

Responsabilidad:

Abstraer el acceso a datos de la lógica de negocio, encapsulando las operaciones CRUD sobre las entidades.

Tecnología: Implementadas con Spring Data JPA, lo que permite generar implementaciones automáticamente a partir de las interfaces y aprovechar consultas derivadas por nombre de método.

4.13.6. Dependency Injection Pattern

El Dependency Injection Pattern es un patrón estructural que forma parte del principio de inversión de control (IoC). Consiste en delegar en un contenedor (en este caso, Spring) la responsabilidad de crear e inyectar las dependencias que necesitan las clases, en lugar de que ellas mismas las instancien. Esto reduce el acoplamiento y facilita las pruebas unitarias.

Ubicación en el sistema:

Uso de @RequiredArgsConstructor y inyección por constructor en la mayoría de las clases de servicios, controladores y componentes.

Responsabilidad:

Permitir que Spring resuelva e inyecte automáticamente las dependencias necesarias (repositories, servicios, factories, etc.), evitando acoplamiento fuerte entre clases y creando un código más testeable y mantenible.

Uso concreto:

Las clases declaran sus dependencias en el constructor; Spring se encarga de proporcionar las instancias adecuadas en tiempo de ejecución, en lugar de utilizar new de forma directa.

4.14. Propuesta de servicios REST

La propuesta de servicios se organiza bajo el paradigma REST, por su simplicidad, compatibilidad con aplicaciones modernas y facilidad de consumo desde el frontend, en este

caso, las aplicaciones móviles.REST permite trabajar con recursos claros, verbos HTTP estándar y respuestas en formato JSON, lo que asegura interoperabilidad y estandarización.

Endpoint	Responsabilidad
POST /api/auth/register	Registrar nuevo usuario (CLIENTE) con validación de rol ADMIN
POST /api/auth/v1/registerAdm	Registrar nuevo administrador con validación de rol ADMIN
POST /api/auth/login	Autenticación de usuario y generación de tokens JWT
POST /api/auth/refreshToken	Renovar token JWT usando refresh token válido
POST /api/auth/resetPassword	Cambiar contraseña con validación de token temporal
POST /api/auth/logout	Invalidar refresh token del usuario autenticado
GET /api/usuario/listar	Listar todos los usuarios con rol USER
GET /api/usuario/listarId/{id}	Obtener usuario específico por ID
GET /api/usuario/listarme	Obtener datos del usuario autenticado
PUT /api/usuario/actualizar/{id}	Actualizar datos de usuairo y subir imagen
PUT /api/usuario/actualizar-mi-perfil	Actualizar perfil del usuario autenticado
POST /api/barbero/crear	Crear barbero con imagen y generar horarios base
GET /api/barbero/listar	Listar todos los barberos activos
GET /api/barbero/listarId/{id}	Obtener barbero específico
PUT /api/barbero/actualizar/{id}	Actualizar datos del barbero e imagen
DELETE /api/barbero/eliminar/{id}	Deshabilitar barbero (soft delete)
GET /api/servicio/listar	Listar todos los servicios activos
GET /api/servicio/listarId/{id}	Obtener servicio específico
POST /api/servicio/crear	Crear servicio con imagen y tipo
PUT /api/servicio/actualizar/{id}	Actualizar servicio e imagen
DELETE /api/servicio/eliminar/{id}	Deshabilitar servicio (soft delete)
GET /api/rango/listar	Listar todos los rangos horarios
GET /api/rango/listarId/{id}	Obtener rango específico
PUT /api/horarioBarberoBase/actualizarTurnosDia	Actualizar turnos de barberos para un día específico
PUT /api/horarioBarberoBase/confirmarHorario	Confirmar horarios base y generar instancias para próxima semana
GET /api/horarioInstancia/actual	Obtener instancias de la semana actual agrupadas por día
GET /api/reserva/barberos-disponibles	Listar barberos disponibles para fecha/horario

	específico
POST /api/reserva/crear	Crear reserva de usuario con validación de disponibilidad
POST /api/reserva/crearReservaRecompensa	Crear reserva gratis si usuario tiene 7+ reservas completadas
GET /api/reserva/consultarRecompensa	Verificar si usuario califica para reserva de recompensa
POST /api/reserva/subir-comprobante/{reservaId}	Subir comprobante de pago de reserva
GET /api/reserva/mis-reservas	Listar reservas del usuario autenticado
GET /api/reserva/admin/listar	Listar reservas con filtros opcionales (fecha, estado, usuario)
PUT /api/reserva/admin/cambiar-estado/{reservaId}	Cambiar estado de reserva (CREADA → CONFIRMADA → REALIZADA)
GET /api/reserva/obtenerReportes	Generar reportes de reservas realizadas en rango de fechas
GET /api/reportes/horario	Exportar PDF con horarios de barberos (Jasper Reports)
POST /api/valoracion/crear	Crear valoración de usuario sobre servicio
GET /api/valoracion/listar	Listar todas las valoraciones activas
PUT /api/valoracion/responder/{valoracionId}	Marcar valoración como respondida (cambiar estado a 0)

4.15. Gestión de configuración (básico)

La gestión de configuración en un sistema, es un proceso esencial que garantiza la coherencia, trazabilidad y estabilidad de todos los artefactos del proyecto. Desde la perspectiva arquitectónica, este proceso asegura que cada cambio introducido en el sistema esté documentado, validado y pueda replicarse en distintos entornos sin pérdida de consistencia.

En este proyecto, la gestión de configuración abarca:

- Código fuente del backend: desarrollado en Spring Boot (Java 17), organizado en capas (controladores, servicios, repositorios, models, dto, exceptions).
- Archivos de configuración del entorno: variables en “aplication.properties”, definiciones de puertos, credenciales y parámetros de seguridad.
- Esquemas y migraciones de base de datos: gestionados con JPA (Java Persistence API), incluyendo scripts SQL para replicar el estado de la base en desarrollo, pruebas y producción.
- Documentación técnica: manuales de usuario, diagramas UML, especificaciones de APIs (Swagger/OpenAPI) y README del sistema.

Para reforzar la trazabilidad, se recomienda vincular cada requisito funcional con commits y ramas específicas, de modo que sea posible auditar qué parte del sistema responde a cada necesidad. Asimismo, la gestión de configuración se complementa con Docker y docker-compose, que permiten levantar entornos homogéneos y reproducibles. Finalmente, se sugiere integrar GitHub Actions para automatizar pipelines de CI/CD, de manera que cada commit relevante ejecute pruebas unitarias, validaciones de estilo y despliegues controlados en staging o producción.

4.16. Recomendaciones sobre uso de herramientas como Git, Jenkins u otras

El uso de Git como herramienta de control de versiones es fundamental en proyectos CRM, donde la confiabilidad y trazabilidad son requisitos críticos. Git permite mantener un historial detallado de cambios, evitando conflictos y asegurando que cada modificación esté vinculada a un requerimiento funcional. Desde la perspectiva arquitectónica, esto garantiza que el sistema pueda auditarse en cualquier momento y que las decisiones técnicas queden registradas.

La estructura de ramas debe ser clara y controlada:

- Main: contiene la versión estable del sistema.
- Develop: integra nuevas funcionalidades en proceso de validación.
- Feature/nombre-funcionalidad: permite el desarrollo modular de cada componente (ejemplo: gestión de barberos, servicios).
- Hotfix: corrige errores urgentes en producción.

Los commits deben ser frecuentes y descriptivos, reflejando el avance del proyecto y facilitando la revisión de código mediante pull requests.

Complementariamente, se recomienda integrar herramientas como:

- Docker y docker-compose: para contenerizar la base de datos y asegurar entornos homogéneos.
- Prettier: para mantener un código uniforme y libre de errores de estilo.
- Postman: para validar endpoints REST y asegurar la correcta comunicación entre servicios.
- GitHub Actions: para CI/CD, automatizando pruebas, validaciones y despliegues. Por ejemplo, un push en develop puede ejecutar pruebas unitarias y desplegar en staging, mientras que un merge en main puede desencadenar el despliegue en producción.

En conjunto, Git y estas herramientas fortalecen la trazabilidad y la calidad del sistema, asegurando que cada cambio esté documentado, probado y alineado con los objetivos del proyecto. En un entorno empresarial, donde los errores pueden impactar directamente en la operación y en la confianza de los clientes, esta disciplina técnica se convierte en un pilar esencial para el éxito.

4.17. Buenas prácticas para la trazabilidad y seguridad

4.17.1. Trazabilidad

La trazabilidad en un sistema hospitalario implica vincular cada requisito funcional con su implementación técnica, pruebas y documentación. Para lograrlo, se recomienda:

- Asignar identificadores únicos a cada requisito (ejemplo: RF01, RF02) y utilizarlos en commits, ramas y tareas.
- Implementar una matriz de trazabilidad que muestre cómo cada requisito se refleja en el código, pruebas unitarias y endpoints REST.
- Vincular commits y pull requests con issues en plataformas como GitHub o Jira, asegurando que cada cambio esté asociado a una necesidad concreta.
- Generar documentación automática de APIs (Swagger/OpenAPI) para mantener contratos actualizados y verificables.

4.17.2. Seguridad

La seguridad debe abordarse desde múltiples capas:

- Control de acceso por roles (RBAC): definir perfiles de usuario (administrador, técnico, vendedor, consulta) y restringir funcionalidades según privilegios.
- Autenticación robusta: uso de JWT con expiración y renovación, y posibilidad de autenticación multifactor para usuarios críticos.
- Cifrado de datos: en tránsito mediante HTTPS y en reposo para información sensible en la base de datos.

Prácticas adicionales de modelos de datos: además de las medidas anteriores, se recomienda implementar campos de auditoría y borrado lógico en los modelos de datos:

- created_at y updated_at: permiten registrar la fecha de creación y última modificación de cada registro, asegurando trazabilidad temporal y facilitando auditorías.
- Soft deletes (deleted_at): en lugar de eliminar físicamente un registro, se marca con una fecha de eliminación. Esto evita la pérdida irreversible de información y permite recuperar datos en caso de errores o auditorías.
- Consistencia en naming conventions: todos los modelos deben seguir un estándar uniforme para estos campos, lo que facilita consultas y reportes.

Estas prácticas fortalecen la integridad histórica del sistema, permitiendo reconstruir el estado de la información en cualquier momento y garantizando que los datos sensibles se gestionen con responsabilidad y seguridad.

5. CONCLUSIONES

En conclusión, el trabajo logró diseñar una arquitectura de software escalable, robusta y mantenible para el sistema integral de gestión y reservas de Diamond Barbershop,

respondiendo a los problemas identificados en la gestión manual de citas, la falta de visibilidad del negocio y la ausencia de un historial centralizado de clientes. La propuesta articula una solución que combina aplicación web y móvil sobre un backend único, alineada con el objetivo principal de automatizar los procesos operativos clave y mejorar de forma significativa la experiencia del cliente.

La elección de un estilo combinado cliente–servidor más arquitectura en capas se demostró adecuada para el contexto del caso de estudio. Por un lado, permitió centralizar la lógica de negocio y los datos en un backend Spring Boot desplegado en AWS Elastic Beanstalk, consumido por múltiples clientes (apps móviles y páginas en Vercel); por otro, la división en capas de presentación, servicios, repositorios y datos favorece la separación de responsabilidades y reduce el acoplamiento interno. Esta estructura facilita la mantenibilidad, el control de accesos y la incorporación futura de nuevas interfaces o canales digitales sin romper la arquitectura existente.

Asimismo, se cumplió con los objetivos de definir una infraestructura de despliegue en la nube y de integrar servicios externos clave. El uso de AWS (VPC, Elastic Beanstalk y RDS MySQL) proporciona una base escalable y segura, mientras que la integración con Cloudinary para almacenamiento de imágenes y con un proveedor de correo para notificaciones soporta los requerimientos de experiencia de usuario y visibilidad del negocio. Sobre esta base se implementan servicios REST que automatizan la gestión de citas, horarios, pagos, recompensas y reportes, reduciendo errores de agendamiento y la carga administrativa del personal.

En términos de calidad, la arquitectura propuesta incorpora tácticas específicas para seguridad (JWT, control de roles, cifrado), rendimiento (paginación), disponibilidad y confiabilidad (health checks, logging, backups), así como mantenibilidad, interoperabilidad y portabilidad mediante modularización por dominios, uso de Repository Pattern, APIs RESTful y configuración basada en variables de entorno. Estas decisiones aseguran que el sistema no solo atienda las necesidades actuales de Diamond Barbershop, sino que también esté preparado para escenarios de crecimiento, como la apertura de nuevas sedes o la evolución hacia servicios más especializados, manteniendo una base tecnológica sólida para la transformación digital continua del negocio.

6. REFERENCIAS

- Espina-Romero, L., Parra, D., Hurtado, H., Rodriguez, E., Arias-Montoya, F., Noroño-Sánchez, J., Talavera-Aguirre, R., Corzo, J., & Pírela, R. (2024). The Role of Digital Transformation and Digital Competencies in Organizational Sustainability: A Study of SMEs in Lima, Peru. Sustainability. https://doi.org/10.3390/su16166993.

- Nugroho, D. (2025). Analysis of the Role of Customer Relationship Management (CRM) in Maintaining Customer Loyalty. Eduvest - Journal of Universal Studies. https://doi.org/10.59188/eduvest.v5i4.49939.

- Ford, M., & Richards, N. (2020). *Fundamentals of Software Architecture*. O'Reilly Media.

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

- Espina-Romero, L. et al. (2024). *The Role of Digital Transformation and Digital Competencies in Organizational Sustainability: A Study of SMEs in Lima, Peru*. *Sustainability*.

- Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing (NIST SP 800-145).* National Institute of Standards and Technology.

- Amazon Web Services. (s. f.). *What is cloud computing?* Amazon Web Services. https://aws.amazon.com/what-is-cloud-computing/

- Google Cloud. (s.f). *PaaS, IaaS, SaaS y CaaS: ¿en qué se diferencian?* Google Cloud. https://cloud.google.com/learn/paas-vs-iaas-vs-saas?hl=es

7. ANEXOS

6:31

Coloración

S/ 20.0

Coloración del color a tu eleccion

SEDE: VILLA MARIA

Elegir Barbero

Diego

Andre

Arian

Reservar

Cancelar

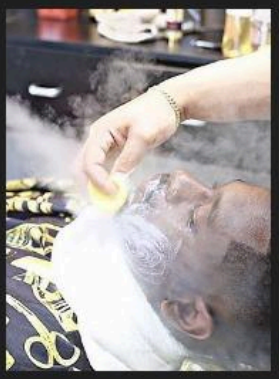
6:30

Corte High fade

S/ 20.0

Degradado alto

Reservar

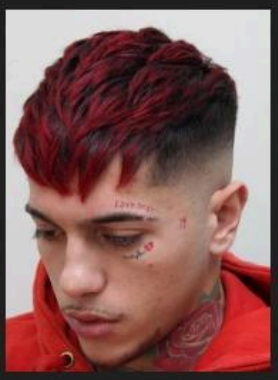


Skincare completo

S/ 20.0

Skincare completo

Reservar



Coloración

S/ 20.0

Coloración del color a tu eleccion

Reservar

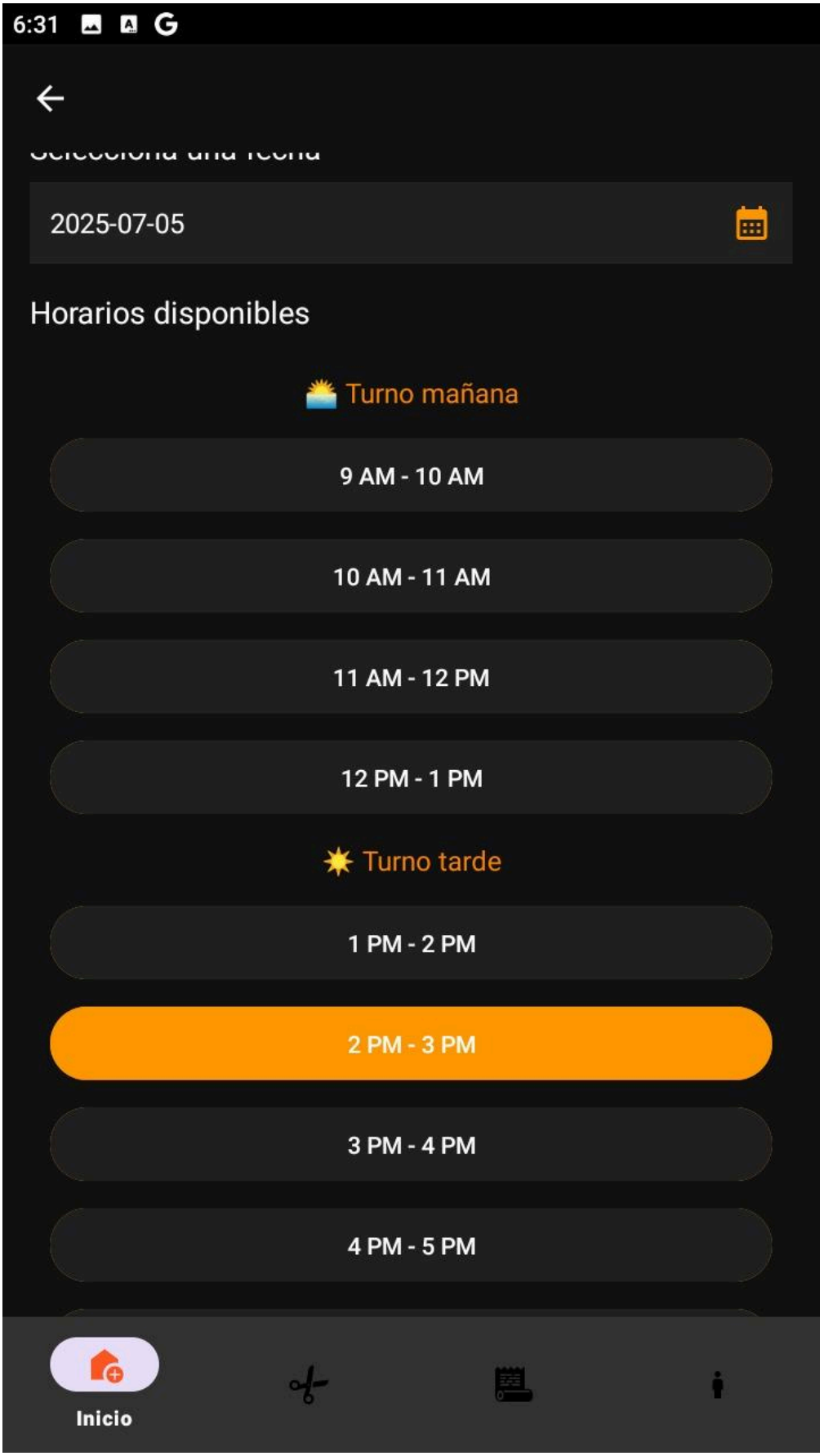
Afeitado de Barba

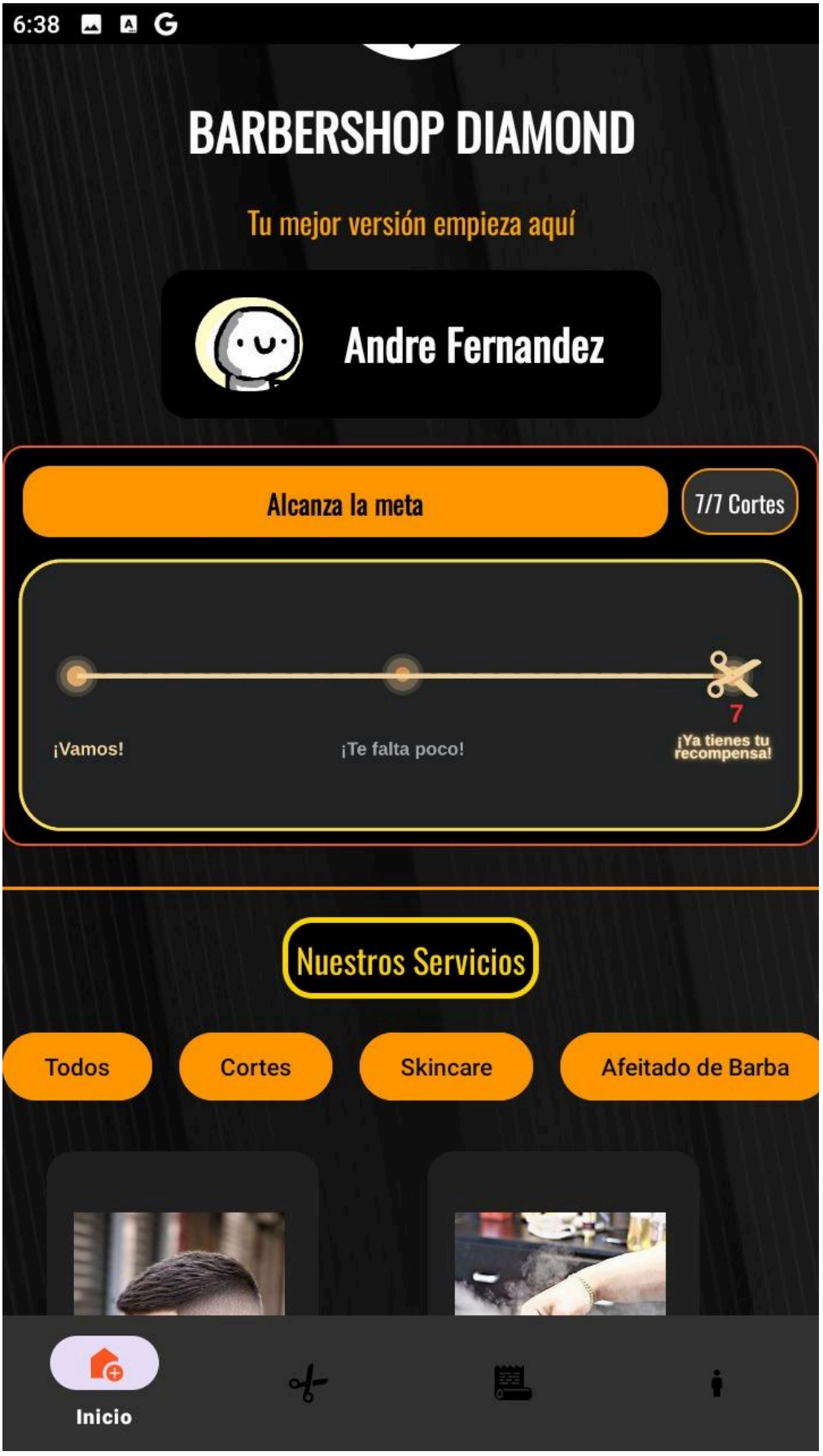
S/ 20.0

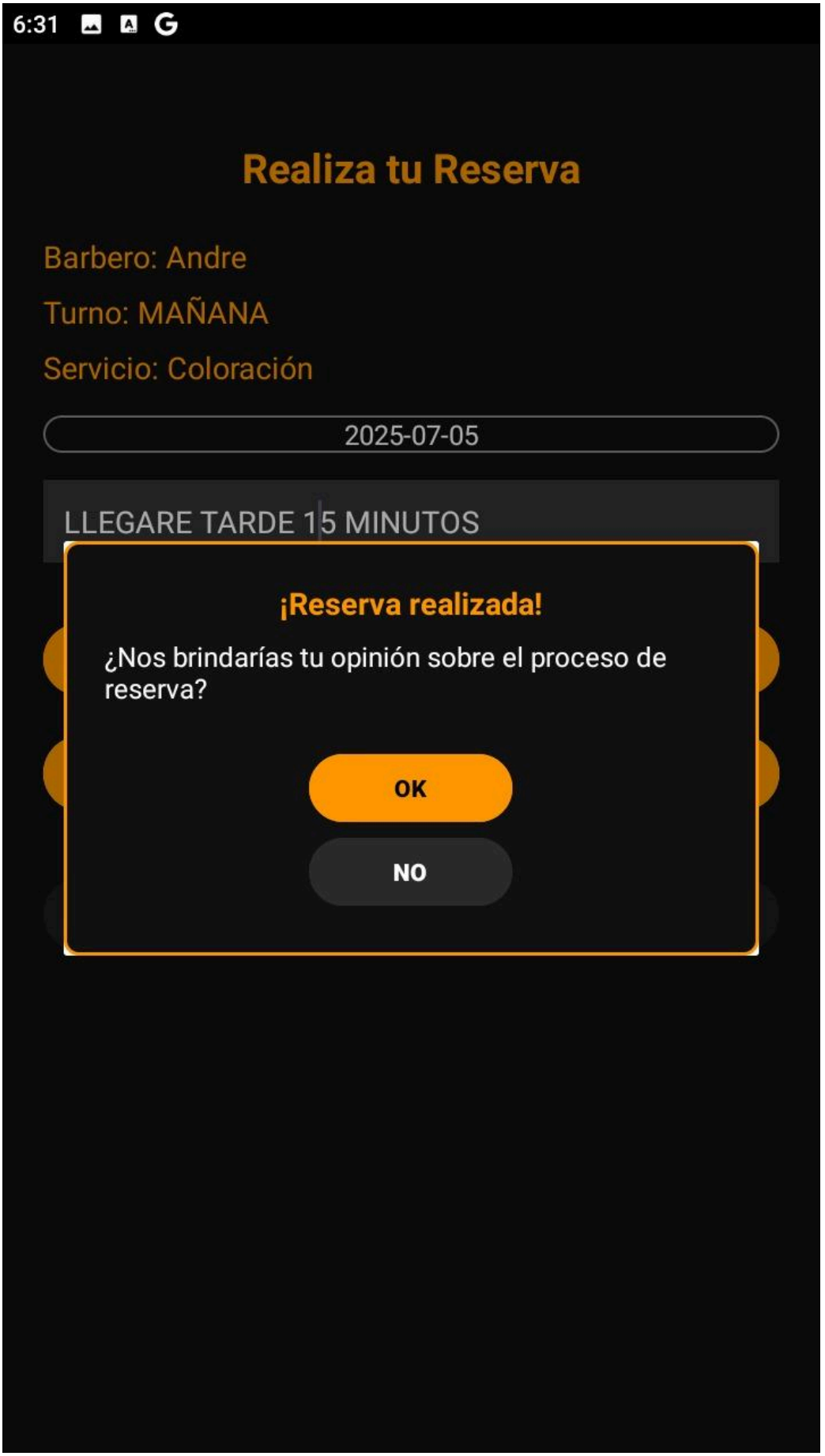
Perfilado de barba

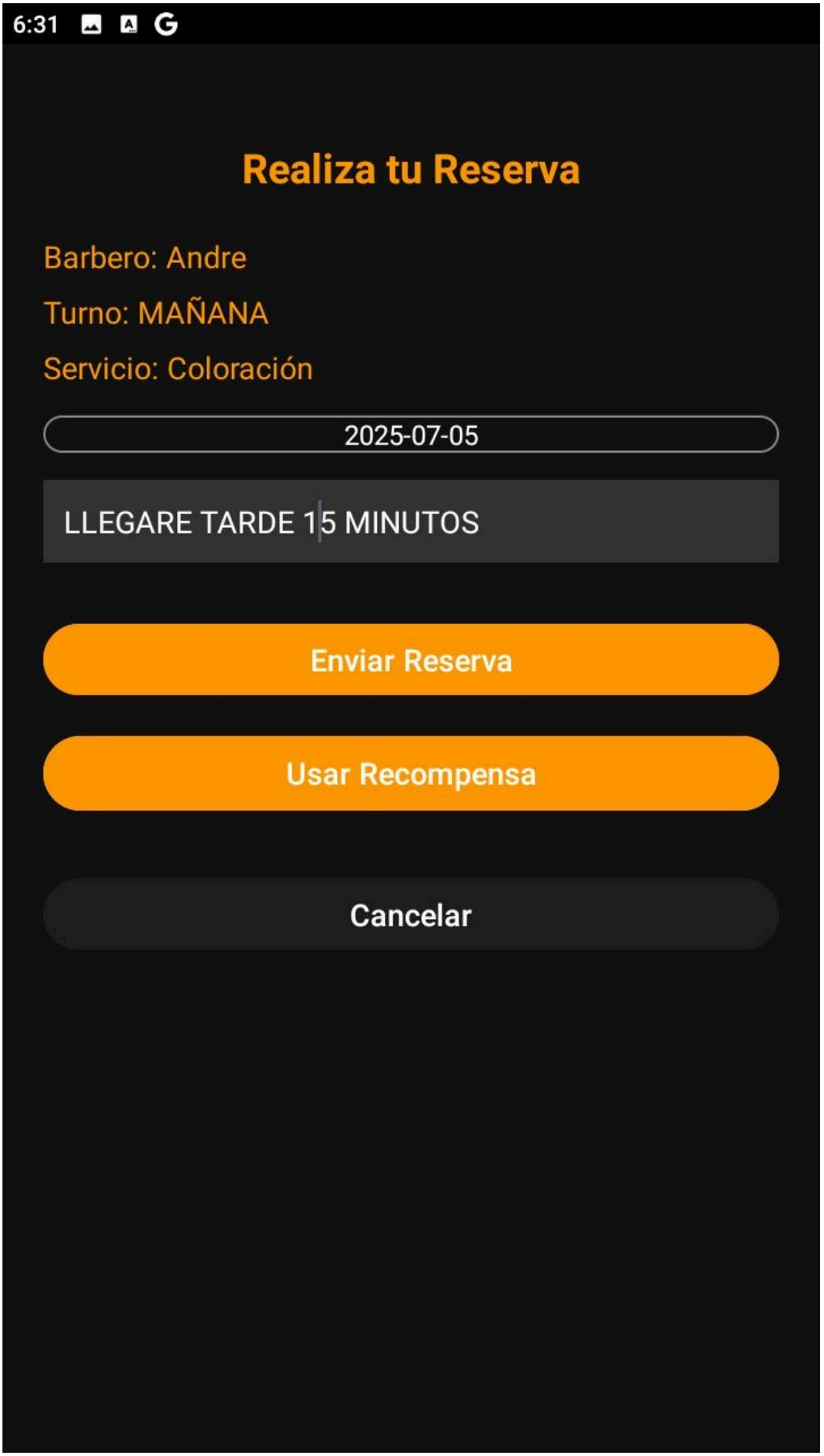
Reservar

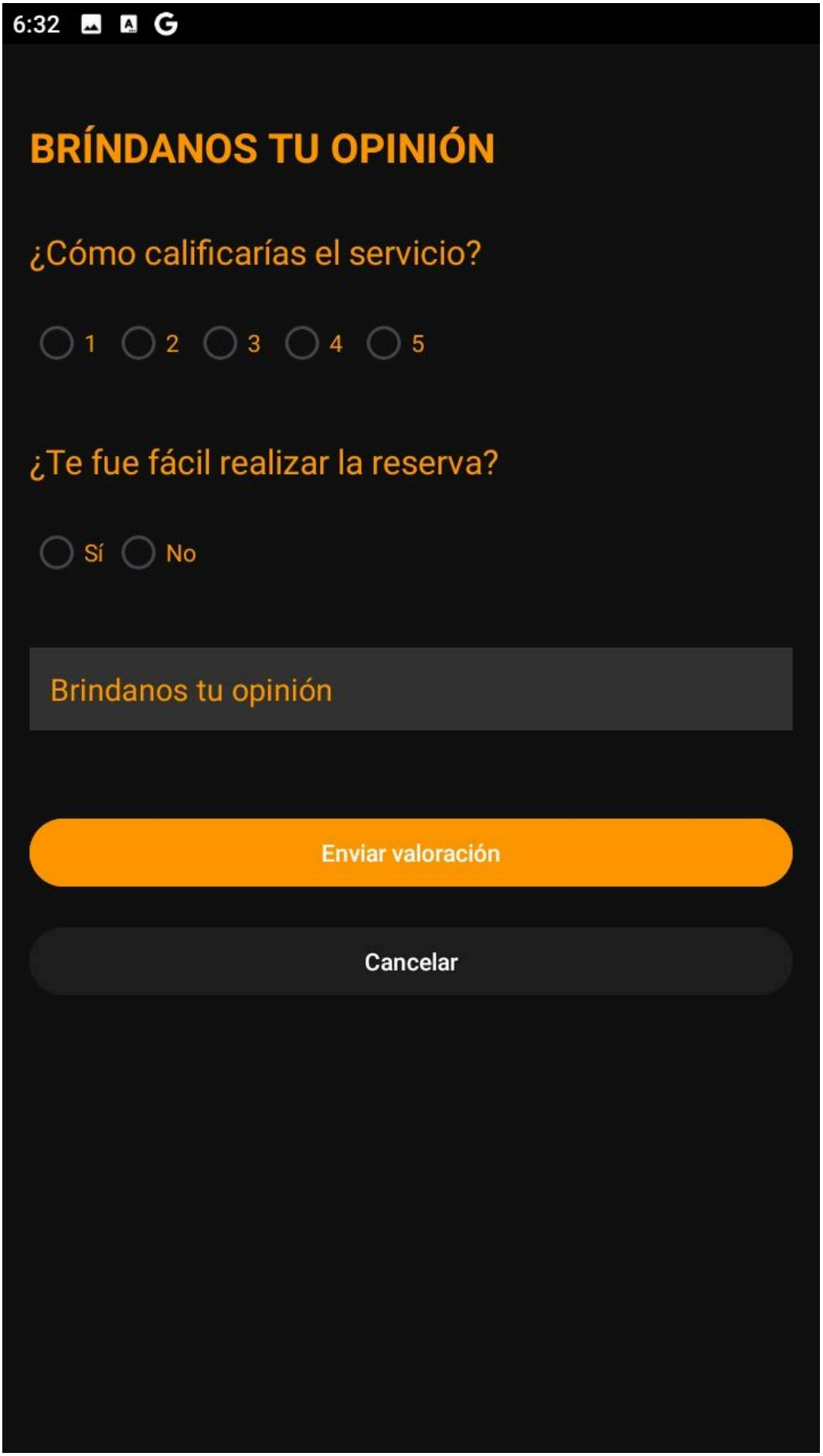
Inicio


6:31
2025-07-05
Horarios disponibles
Turno mañana
9 AM - 10 AM
10 AM - 11 AM
11 AM - 12 PM
12 PM - 1 PM
Turno tarde
1 PM - 2 PM
2 PM - 3 PM
3 PM - 4 PM
4 PM - 5 PM
Inicio


6:38
BARBERSHOP DIAMOND
Tu mejor versión empieza aquí
Andre Fernandez
Alcanza la meta
7/7 Cortes
¡Vamos!
¡Te falta poco!
7
¡Ya tienes tu recompensa!
Nuestros Servicios
Todos
Cortes
Skincare
Afeitado de Barba
Inicio


6:31
Realiza tu Reserva
Barbero: Andre
Turno: MAÑANA
Servicio: Coloración
2025-07-05
LLEGARE TARDE 15 MINUTOS
¡Reserva realizada!
¿Nos brindarías tu opinión sobre el proceso de reserva?
OK
NO


6:31
Realiza tu Reserva
Barbero: Andre
Turno: MAÑANA
Servicio: Coloración
2025-07-05
LLEGARE TARDE 15 MINUTOS
Enviar Reserva
Usar Recompensa
Cancelar


6:32
BRÍNDANOS TU OPINIÓN
¿Cómo calificarías el servicio?
1
2
3
4
5
¿Te fue fácil realizar la reserva?
Sí
No
Brindanos tu opinión
Enviar valoración
Cancelar