# KÓMODO

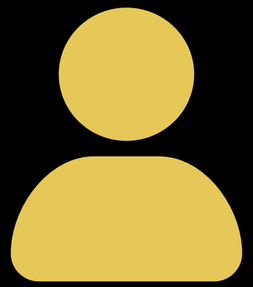
## PROYECTO APT — INGENIERÍA EN INFORMÁTICA 2025

**Integrantes**:
- Tamara Lecaros
- Enghel Cerpa
- Fabián Riquelme

**Institución**: DUOC UC

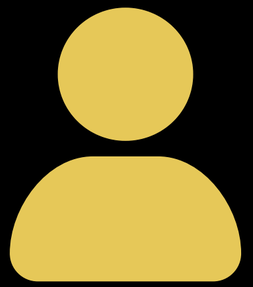
**Año**: 2025

# ROLES DEL EQUIPO



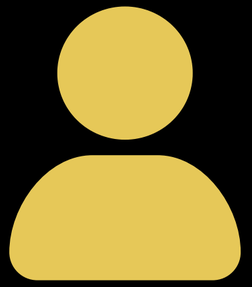
**Tamara Lecaros**

- Product Owner
- UX/UI Designer
- Analista Funcional
- QA Funcional



**Enghel Cerpa**

- Backend Developer
- Arquitecto de Software
- DBA



**Fabián Riquelme**

- QA Técnico
- Frontend Support
- Analista Funcional

**“Si bien los roles fueron definidos, el desarrollo se llevó a cabo de manera colaborativa, con la participación de todos los integrantes en cada fase del proyecto.”**

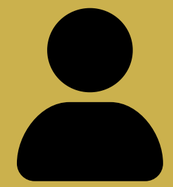
# ¿QUÉ ES KÓMODO?

KÓMODO es una plataforma web integral que digitaliza y automatiza los procesos principales de una peluquería, permitiendo una gestión eficiente, ordenada y totalmente centralizada.

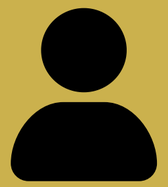
El sistema automatiza tareas que antes eran manuales, como:

- Cita
- Pago (abono)
- Servicios
- Inventario
- Solicitudes

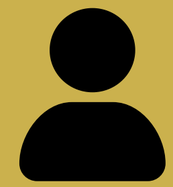
## Roles del sistema



**Cliente**



**Trabajador**



**Administrador**

*"Su objetivo es digitalizar la operación, mejorar el orden y optimizar la atención."*

# ARQUITECTURA GENERAL DEL SISTEMA

Características: - **Arquitectura API REST** - **Capas desacopladas** - **Comunicación en formato JSON**

## FRONTEND

- HTML
- CSS
- JavaScript (Fetch API)
- Integración con EmailJS

## BACKEND

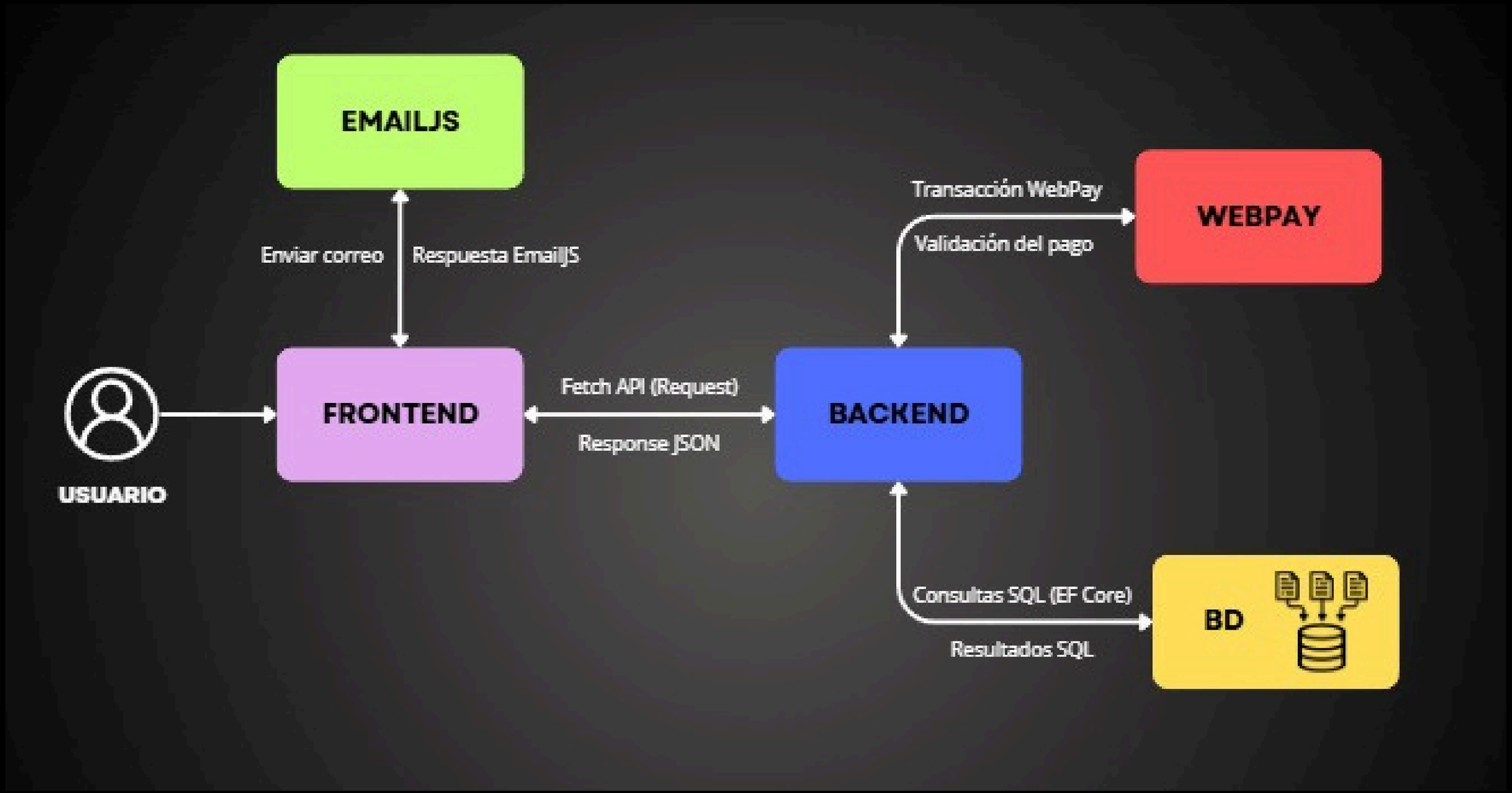
- ASP.NET Core Web API
- Controladores
- Models
- Validaciones
- Respuestas en JSON
- Integración con WebPay Sandbox

## BASE DE DATOS

- SQL Server
- Entity Framework Core (ORM)

**Modelo normalizado (3FN)**

# FLUJO GENERAL DEL SISTEMA



# PRUEBAS DEL SISTEMA

## POSTMAN — PRUEBAS FUNCIONALES

**Qué probamos**:

- 27 pruebas funcionales en total.
- Endpoints de login, registro, recuperación, usuarios, reservas, pagos, inventario y solicitudes.
- Validaciones de datos, roles, disponibilidad y estados de reserva.
- Estructura y consistencia de las respuestas JSON.

**Resultados**:

- 100% de las rutas esenciales respondieron correctamente.
- Códigos HTTP adecuados (200, 201, 400, 404).
- Validaciones funcionando en todos los módulos.
- Cero errores 500 en todas las pruebas.
- Flujo completo de reserva + abono WebPay verificado correctamente.

## ZAP (OWASP) — PRUEBAS DE SEGURIDAD

**Qué probamos**:

- Escaneo automático del backend y el frontend.
- Revisión de encabezados HTTP, cookies, formularios, rutas y parámetros.
- Detección de posibles vulnerabilidades comunes (XSS, CSRF, headers faltantes).

**Resultados**:

- 0 vulnerabilidades críticas.
- 0 vulnerabilidades de riesgo alto.
- Alertas bajas típicas de entornos locales:
  - Falta de HTTPS (localhost).
  - Cabeceras de seguridad no configuradas por entorno académico.
- Sin hallazgos que afecten la estabilidad o integridad del sistema.

## JMETER — PRUEBAS DE CARGA

**Qué probamos**:

- Simulación de 100 usuarios concurrentes.
- Rendimiento del login, reservas y solicitudes críticas.
- Estabilidad del backend bajo estrés.

**Resultados**:

- Sin errores 500 ni caídas del backend.
- El sistema mantuvo tiempos de respuesta estables bajo carga.
- Ninguna pérdida significativa de rendimiento.
- Backend apto para un volumen de usuarios superior al promedio del rubro (peluquería).

# BACKEND DEL SISTEMA

## ESTRUCTURA

- Desarrollado en ASP.NET Core Web API.
- Arquitectura modular basada en controladores.
- +40 endpoints REST entre GET, POST, PUT y DELETE.
- Uso de Entity Framework Core (ORM) para consultas.
- Respuestas estandarizadas en JSON.
- Integración con servicios externos (WebPay).

## CONTROLADORES PRINCIPALES

- AuthController (login y seguridad)
- UsuarioController
- ServiciosController
- CitasController
- ReservaController
- PagoController
- InventarioController
- Solicitud_CompraController
- IngresosController
- HomeController

## FLUJO INTERNO DEL BACKEND

1. Frontend envía petición HTTP (GET/POST/PUT/DELETE) vía fetch.
2. Backend valida formatos, campos y rol del usuario (si no corresponde → bloquea).
3. Ejecuta la lógica del negocio: disponibilidad, evitar duplicados, validar estado y calcular diferencias.
4. EF Core genera y ejecuta consultas en SQL Server.
5. Backend procesa la respuesta de la BD.
6. Si aplica, se comunica con WebPay o EmailJS.
7. Retorna al frontend un JSON limpio y estandarizado.

# BASE DE DATOS – SQL SERVER

## CARACTERÍSTICAS PRINCIPALES

- Base creada en SQL Server (SSMS).
- Modelo relacional normalizado hasta 3FN
  - ➔ evita duplicidad de datos y mantiene integridad entre módulos.
- Integridad garantizada mediante claves primarias y foráneas.
- Contraseñas almacenadas con hash.
- Validaciones y mapeo gestionados por Entity Framework Core.

## TABLAS PRINCIPALES

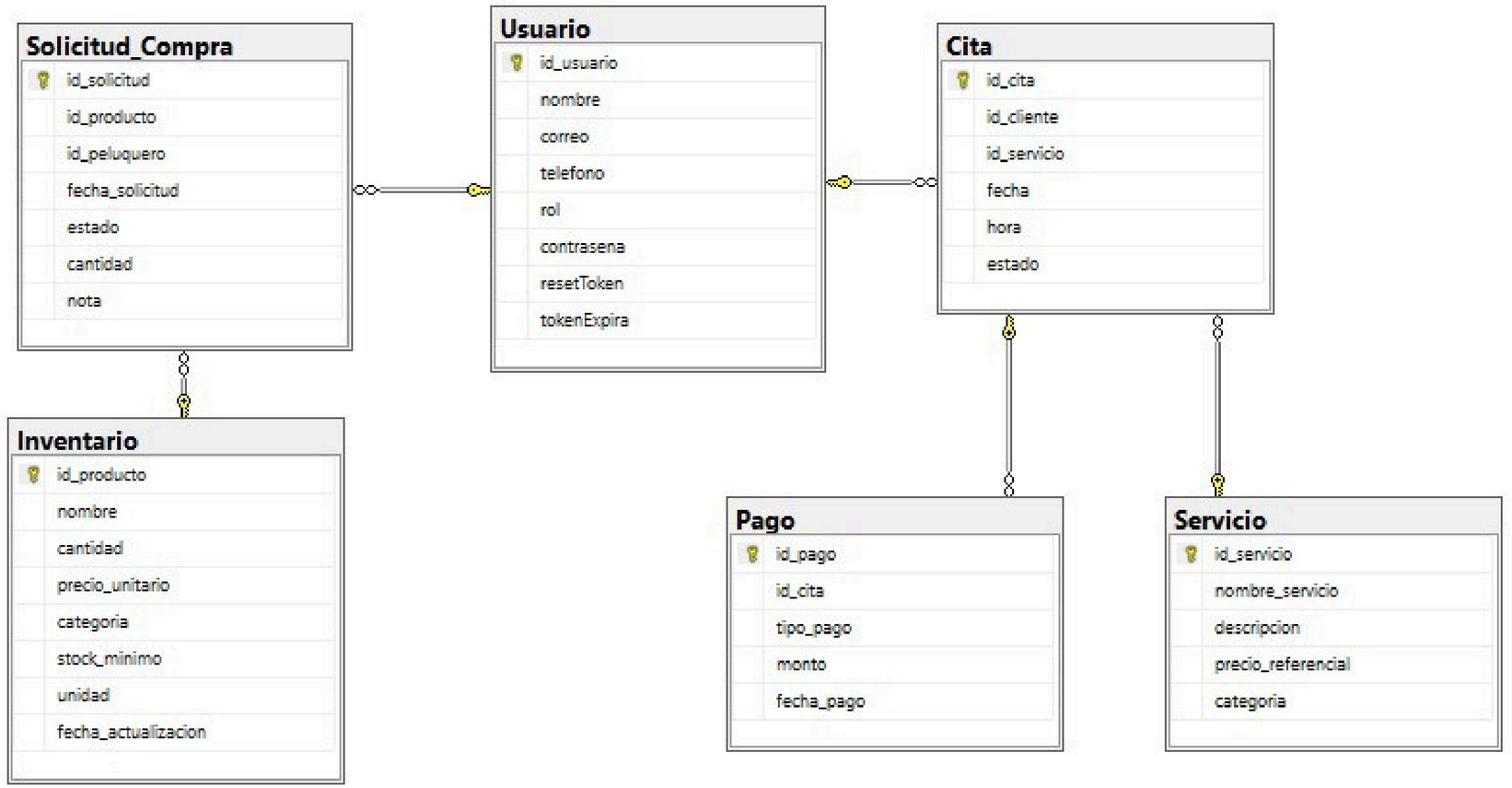
- Usuarios
- Servicio
- Cita
- Pago
- Inventario
- Solicitud_Compra

Total: 6 tablas normalizadas y relacionadas.

## RELACIONES CLAVE (1 ➔ N)

- Usuario ➔ Cita
  - ➔ historial completo de reservas.
- Servicio ➔ Cita
  - ➔ define qué servicio se agenda.
- Cita ➔ Pago
  - ➔ registra abono WebPay y pago final.
- Inventario ➔ Solicitud_Compra
  - ➔ trazabilidad de insumos solicitados.

Todas las relaciones son administradas por EF Core para asegurar integridad referencial.




# DIAGRAMA ENTIDAD RELACIÓN

# COMUNICACIÓN BACKEND → BASE DE DATOS

## ENTITY FRAMEWORK CORE (ORM)

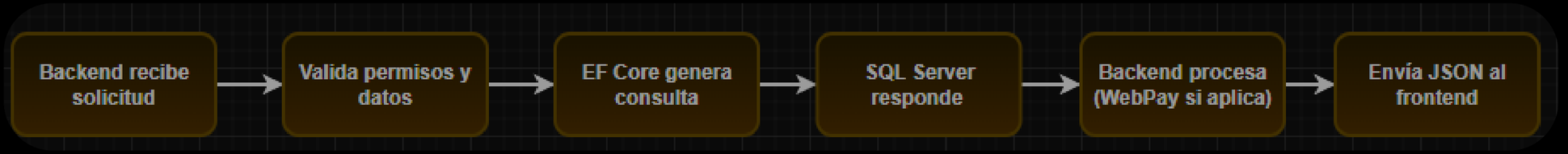
- Traduce instrucciones C# a consultas SQL.
- Maneja modelos, relaciones y claves foráneas.
- Controla integridad de datos.
- Evita escribir SQL manual.
- Automatiza operaciones CRUD.
- Reduce riesgos como inyección SQL.

## PROCESO DE CONSULTA

1. Backend recibe la solicitud y valida permisos y reglas del negocio.
2. EF Core convierte esa instrucción en una consulta SQL.
3. SQL Server ejecuta la consulta y devuelve los datos.
4. EF Core transforma esos datos en objetos y los entrega al backend.
5. Si corresponde, el backend también se comunica con WebPay.
6. El backend envía al frontend un JSON limpio y estandarizado.

## BENEFICIOS

- Código más limpio y mantenible.
- Mayor seguridad e integridad de datos.
- Menos errores en consultas.
- Relaciones administradas automáticamente.
- Escalabilidad: fácil agregar nuevas tablas o lógica.



# LÓGICA DE NEGOCIO – RESERVAS Y PAGOS

## RESERVAS

- Validación de disponibilidad (día, hora y servicio)
- Registro del cliente y servicio asociado
- Verificación de horarios duplicados
- Restricción según estado de la cita
- Relación directa con pagos (cada cita genera un pago)
- Cambio de estado según avance: creada ➔ pagada ➔ finalizada

## PAGOS

- Abono obligatorio de $1.000 para confirmar reserva
- Integración WebPay Sandbox (token y validación de estado)
- Registro del abono como primer pago
- Cálculo automático de la diferencia al finalizar el servicio
- Registro del monto final como “caja”

## PROCESOS CLAVE

- Backend controla toda la lógica (no el frontend)
- Validaciones se ejecutan antes de guardar en BD
- Reglas de negocio centralizadas en controladores + servicios
- Manejo de estados para evitar inconsistencias
- Lógica que asegura que solo citas pagadas pueden confirmarse
- Prevención de reservas sin servicio o sin cliente válido
- Flujo consistente con la BD (PK – FK siempre respetadas)

# METODOLOGÍA ÁGIL (SCRUM)

**¿Por qué elegimos con Scrum?**

- Nos permitió dividir el proyecto en módulos funcionales y manejables (backend, BD, frontend, pruebas).
- Facilitó el control semanal del avance y la integración entre componentes.
- Permitió ajustar prioridades al descubrir dependencias técnicas (por ejemplo: WebPay, EmailJS, lógica de reservas).
- Aceleró la toma de decisiones técnicas.

**¿Cómo lo aplicamos?**

- Dividimos el proyecto en 7 sprints reales, basados en dependencias técnicas.
- Priorizamos funcionalidades críticas: autenticación, reservas, pagos, inventario, solicitudes.
- Integramos servicios externos (WebPay y EmailJS) siguiendo la planificación del Sprint 5 y 6.
- Ajustamos el backlog cuando aparecieron problemas de validación, disponibilidad y seguridad.
- Cada sprint cerró con un módulo completamente funcional, probado con Postman y validado por el equipo.
- Al final del Sprint 6 se alcanzó la integración total backend → BD → frontend.

**Artefactos utilizados dentro del proyecto**

- **Product Backlog**: 31 HU. (priorizadas según dependencia técnica y complejidad.)
- **Sprint Backlog**: Tareas distribuidas en capas (BD → backend → frontend → QA).
- **Incrementos por Sprint**:
  - **S1**: Configuración inicial
  - **S2**: Autenticación
  - **S3**: Servicios
  - **S4**: Reservas
  - **S5**: Pagos WebPay
  - **S6**: Integración total
  - **S7**: Correcciones + Pruebas

**Eventos aplicados durante el desarrollo**

- **Sprints**: 7 en total → Sprint 1-6 (2 semanas), Sprint 7 (1 semana).
- **Daily**: Comunicación continua para revisar bloqueos (especialmente BD, pagos, disponibilidad).
- **Sprint Review**: Validación del incremento ante el equipo (ej.: reserva funcionando con abono WebPay).
- **Sprint Retrospective**: Ajustes en la planificación según resultados de pruebas y fallos de integración.

# DOCUMENTACIÓN DEL PROYECTO

## PRODUCT BACKLOG

- **31 HU** priorizadas por dependencia técnica.
- Base para planificar los **7 sprints.**
- Organizado por módulos del sistema.

## MAPA DE ACTORES

- **3 roles del sistema**: Cliente, Trabajador, Administrador.
- Define accesos, restricciones y permisos en backend.

## HISTORIAS DE USUARIO (HU)

- Describen funcionalidades por rol.
- Incluyen criterios de aceptación.
- Aseguran trazabilidad HU → sprint → pruebas.

## DIAGRAMAS DE FLUJO

- **14** diagramas en total.
- **Procesos clave**: Reserva + Pago WebPay, Autenticación y Roles.
- Modelan la lógica crítica del sistema y guiaron la implementación del backend.

## ÉPICAS

- **6 épicas** que agrupan todo el sistema: Usuarios, Servicios, Reservas, Pagos, Citas, Inventario.

## MOCKUPS Y WIREFRAMES

- **6 pantallas diseñadas**: inicio, login, cotización, citas, registro e inventario.
- Definieron la estructura del frontend y guiaron la construcción de las vistas reales.

# CONCLUSIÓN FINAL

- Sistema construido con arquitectura API REST robusta e integrada con SQL Server.
- Conexión coherente entre frontend, backend y BD usando Entity Framework Core.
- Validaciones, reservas, WebPay, caja y roles funcionando correctamente.
- 27 pruebas Postman, 0 vulnerabilidades críticas en ZAP y prueba de carga con 100 usuarios confirmaron estabilidad.
- Scrum permitió avances progresivos mediante 7 sprints y backlog de 31 HU.
- Proyecto desarrollado con organización, comunicación constante y compromiso del equipo.

**Cortes**

Dama, caballero y niños. Tendencias, capas, fades y más.

Leer más | Cotizar

**Alisados**

Control de frizz y liso según tu objetivo.

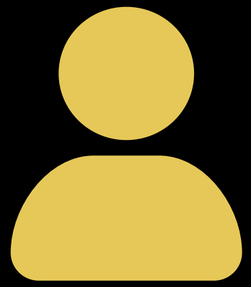
Leer más | Cotizar

**Coloración y Decoloración**

Tinte, balayage, reflejos. Diagnóstico y cuidados.

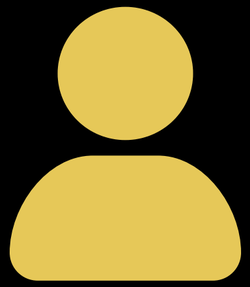
Leer más | Cotizar

# ¡ GRACIAS POR SU ATENCIÓN !



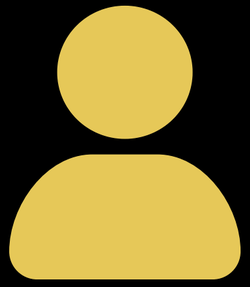
**Tamara Lecaros**

- Product Owner
- UX/UI Designer
- Analista Funcional
- QA Funcional



**Enghel Cerpa**

- Backend Developer
- Arquitecto de Software
- DBA



**Fabián Riquelme**

- QA Técnico
- Frontend Support
- Analista Funcional

**“Si bien los roles fueron definidos, el desarrollo se llevó a cabo de manera colaborativa, con la participación de todos los integrantes en cada fase del proyecto.”**