



## Entrega final Visicontrol

Integrantes:

- Kevin Adrián Gil Soto (@kevingS0712)
  - Boris Guillermo Briones Dupla
    - Fernando Moreno Mora
  - Rafael Alejandro Ajila Gallegos

Carrera: Ingeniería de Software

Fecha: [03/12/2025]

# 1. Cobertura de pruebas

Durante el desarrollo se aplicaron distintos tipos de pruebas.

Primero se revisó la accesibilidad con Lighthouse en modo móvil sobre las pantallas de login, dashboard, creación de visitas e historial. En la mayoría de vistas se obtuvieron puntajes cercanos a 100 en accesibilidad; el dashboard y el historial quedaron un poco por debajo, principalmente por temas de contraste y nombres accesibles en algunos elementos.

En paralelo, se hicieron pruebas de usabilidad con usuarios reales. Se les pidió completar tareas concretas: registrarse, iniciar sesión, revisar y editar su perfil, cambiar la contraseña, crear una visita, ver sus próximas visitas y consultar el historial. A partir de estas pruebas se ajustaron algunos mensajes de error y se confirmó que, en general, la aplicación se entiende sin necesidad de explicaciones adicionales.

A nivel de código, se implementaron pruebas unitarias con Jest centradas en la lógica de formato de fechas y horas, etiquetas de estado y construcción del objeto “meta” de las visitas. Estos helpers son la base para mostrar información coherente en notificaciones y pantallas, y todas las pruebas definidas pasan correctamente.

También se construyeron pruebas de integración usando Supertest. Aquí se validaron los flujos completos de autenticación (`/auth/login` y `/auth/me`) y la creación y consulta de visitas contra una base PostgreSQL real. En estas suites se verifica que con credenciales correctas el sistema entrega el token JWT esperado, que con credenciales erróneas responde 401, y que las visitas creadas se recuperan luego desde los endpoints correspondientes.

Sobre la API ya desplegada se hicieron pruebas funcionales “caja negra” con Postman, cubriendo registro, login, panel de visitas de usuario, panel de administración, perfil, cambio de contraseña (casos positivo y negativo) y cancelación de visitas. Todo esto confirma que los flujos de negocio más importantes están operativos de punta a punta.

Finalmente se realizaron pruebas de rendimiento con JMeter simulando alrededor de diez usuarios concurrentes. Se ejercitaron tres endpoints clave: login (POST `/api/auth/login`), listado de visitas y listado de historial. Los tiempos promedio se mantuvieron alrededor de 60–70 ms para el login y en torno a 2–7 ms para los listados, siempre con 0 % de errores.

Como complemento, se hizo un set de pruebas de seguridad muy básicas: intentos de inyección SQL en login y filtros, acceso a rutas protegidas sin token, token inválido y accesos de un usuario normal a rutas de administrador. En todos los casos el backend respondió con códigos 401/403 o errores genéricos, sin exponer detalles de la base de datos ni stacktraces.

En resumen, la cobertura de pruebas incluye accesibilidad, usabilidad, lógica interna, integración, rendimiento y seguridad básica.

## 2. Resultados de las pruebas

De accesibilidad, las pantallas de login y creación de visita alcanzan el máximo de Lighthouse en este aspecto. El dashboard y el historial están ligeramente por debajo por pequeños problemas de contraste y por algunos controles que no tienen label accesible, pero no se detectaron barreras graves de uso.

Las pruebas de usabilidad mostraron que las personas pueden registrarse y autenticarse sin dificultad, ubicar la sección de perfil, actualizar sus datos y entender el flujo de cambio de contraseña. A raíz de los tests se mejoraron los mensajes cuando la contraseña actual es incorrecta o cuando las nuevas no coinciden. La creación de visitas resultó intuitiva y los usuarios pudieron diferenciar claramente entre “próximas visitas” e “historial”.

En las pruebas unitarias e integración todas las suites pasan sin fallos. Esto indica que la lógica principal de formato de datos y los flujos de autenticación y gestión de visitas están cubiertos y se comportan como se espera, al menos para los casos más representativos definidos en el plan.

En rendimiento, incluso con varios usuarios simultáneos los tiempos de respuesta se mantienen muy por debajo de los umbrales que se suelen considerar aceptables para una aplicación web de este tipo. No se observaron errores ni caídas durante las pruebas, por lo que el sistema se ve estable bajo carga ligera.

En seguridad, los intentos de saltarse el login mediante inyección SQL fueron bloqueados y la API devolvió simplemente “credenciales inválidas”. También se confirmó que las rutas protegidas exigen token, que un token manipulado no pasa las validaciones y que un usuario normal no puede acceder a los endpoints de administrador. Desde la perspectiva del proyecto, estos controles básicos están bien cubiertos.

## 3. Deuda técnica y puntos pendientes

Aunque la app no presenta errores críticos, sí quedaron algunos pendientes que se consideran deuda técnica:

- Mejorar algunos detalles de accesibilidad: añadir un texto o aria-label al botón de cerrar sesión, revisar contraste de ciertos textos del dashboard y etiquetar mejor los filtros del historial.
- Ajustar la validación de parámetros en los listados de visitas para que, cuando se envía una fecha con formato totalmente roto (por ejemplo intentando inyección), el backend responda con un 400 controlado en lugar de un 500, manteniendo un mensaje genérico de error.

- Ampliar la cobertura de pruebas automatizadas hacia los módulos de notificaciones (SSE y correos), el flujo completo de recuperación de contraseña y más casos límite de solapamientos y reprogramaciones de visitas.
- Hacer, en el futuro, pruebas de carga con más usuarios concurrentes y complementar con métricas de monitorización (tiempos promedio, picos, errores por minuto) para acercarse a un escenario de producción real.

## 4. Bugs detectados

Durante las pruebas aparecieron algunos problemas que se fueron corrigiendo:

- En el cambio de contraseña, el manejo de errores resultaba confuso cuando la contraseña actual era incorrecta o las nuevas no coincidían. Tras la retroalimentación, se mejoraron los mensajes y el flujo quedó más claro.
- Lighthouse detectó controles sin nombre accesible y pequeños problemas de contraste. Estos no bloquean el uso, pero quedaron registrados como mejoras de interfaz.
- El escenario de inyección SQL en el filtro de visitas provoca un error 500 genérico. Aunque desde seguridad es aceptable (no filtra información sensible), se dejó anotado para devolver en el futuro un error 4xx más amigable para el usuario.

En esta versión no quedaron abiertos bugs que rompan los flujos principales (registro, login, gestión de visitas, panel de administración, perfil, cambio de contraseña o cancelación).

## 5. Manual de usuario

[https://www.canva.com/design/DAG7n\\_Zllb4/r8vzm6NyMzVC142XWOvMMA/edit?utm\\_content=DAG7n\\_Zllb4&utm\\_campaign=designshare&utm\\_medium=link2&utm\\_source=sharebutton](https://www.canva.com/design/DAG7n_Zllb4/r8vzm6NyMzVC142XWOvMMA/edit?utm_content=DAG7n_Zllb4&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton)

## 6. Reporte de defectos: lista priorizada y resuelta

Nº	Defecto	Tipo	Prioridad	Estado	Solución
1	Falta de nombre accesible en botón de cerrar sesión	Accesibilidad	Media	Resuelto	El botón de logout en el dashboard solo tenía ícono. Se añadió un nombre accesible (por ejemplo aria-label="Cerrar sesión" o un texto visible) para que los lectores de pantalla lo identifiquen correctamente.
2	Filtros de historial sin etiquetas asociadas	Accesibilidad	Media	Resuelto	En la vista de historial el campo de fecha y el select de estado no tenían label. Se agregaron etiquetas visibles o atributos aria-label / aria-labelledby para que los lectores de pantalla reconozcan el propósito de cada control.
3	Falta de validación de duración y rango horario de visitas	Funcional / Reglas de negocio	Alta	Resuelto	Al inicio no se validaban bien las duraciones ni el horario de atención. En createVisit y updateVisit se forzó que duration_minutes solo pueda ser 30, 60, 90 o 120, y que la visita esté siempre entre las 08:00 y las 17:00, verificando hora de inicio y fin.
4	Solapamiento de visitas para el mismo interno	Funcional / Integridad de agenda	Alta	Resuelto	Podían generarse dos visitas que se crucen para el mismo interno el mismo día. Se implementó validación de solapamiento en createVisit y updateVisit, calculando minutos totales y comparando los intervalos [inicio, fin) con visitas existentes. Si hay cruce, se devuelve error 409 indicando que el horario está ocupado.
5	Reprogramación de visitas sin control claro de dueño y estado	Seguridad / Autorización	Alta	Resuelto	Un usuario podía editar visitas que no eran suyas o cambiar estados sin restricciones claras. En el controlador putVisit se agregó lógica para que el usuario normal solo modifique visitas que creó, reprogramando únicamente las que están en PENDING, y dejando cambios de estado y datos sensibles solo al rol administrador.
6	Cancelación de visitas no soportada para usuarios finales	Funcionalidad faltante	Media	Resuelto	Solo el administrador podía eliminar visitas. Se creó cancelVisit(userId, visitId) en visits.service.ts, que marca la visita como CANCELLED si pertenece al usuario, está en PENDING o APPROVED y la fecha no ha pasado. Se añadió la acción de cancelar en el historial del frontend.
7	Notificaciones internas incompletas para	Usabilidad / Comunicación con usuario	Media	Resuelto	Al inicio solo había notificaciones básicas. En updateVisit se añadió la creación de VISIT_APPROVED al aprobar,

	cambios de estado y horario				VISIT_CANCELED al rechazar y VISIT_UPDATED al reprogramar sin cambiar estado, incluyendo en meta los valores antiguos y nuevos para apoyar la auditoría.
8	Recordatorios por correo no se disparaban automáticamente	Integración / Jobs	Media	Resuelto	Los recordatorios para visitas del día siguiente no se ejecutaban o no enviaban correos. Se implementó upsertTomorrowReminders en notifications.service.ts para buscar visitas de mañana, evitar duplicados y crear notificaciones VISIT_REMINDER, integrando sendVisitReminderEmail y el job startReminderJob.
9	Problemas de configuración SMTP al pasar de Mailtrap a Gmail	Configuración	Media	Resuelto	Al cambiar de Mailtrap a Gmail aparecieron errores de autenticación y bloqueo. Se ajustaron las credenciales SMTP, se habilitó el acceso desde la cuenta configurada y se actualizó el .env del backend con SMTP_HOST, SMTP_PORT, SMTP_USER, SMTP_PASS y SMTP_FROM correctos.

# MANUAL TÉCNICO – APLICACIÓN VISICONTROL

---

## 1. Introducción y objetivos

VisiControl es una aplicación tipo PWA pensada para gestionar las visitas de familiares a personas privadas de libertad. El sistema permite que un usuario se registre, inicie sesión, asocie a sus internos autorizados y agende visitas con fecha, hora, duración y notas. El administrador del centro puede revisar todas las solicitudes, aprobarlas o rechazarlas, ver el historial y gestionar la información general.

Este manual técnico tiene como objetivo dejar documentado cómo está construida la aplicación por dentro: tecnologías, arquitectura, base de datos, módulos, seguridad, procesos de instalación y puntos clave para que otro desarrollador pueda mantener, corregir o ampliar el sistema sin depender del equipo original.

---

## 2. Requerimientos técnicos

### **Backend**

- Node.js (versión 18+ recomendada).
- TypeScript.
- Gestor de paquetes: npm.
- Base de datos PostgreSQL.
- Variables de entorno para conexión a BD y correo SMTP.
- Entorno para ejecutar scripts de línea de comandos (terminal).

### **Frontend**

- React con Vite.
- TypeScript.
- Navegador moderno compatible con PWA (Chrome, Edge, etc.).
- Soporte para Service Worker y almacenamiento local (para el modo PWA).

### **Base de datos**

- Servidor PostgreSQL accesible desde el backend.
- Un esquema lógico llamado, por ejemplo, visicontrol (o la base por defecto configurada en PGDATABASE).

### **Correo electrónico**

- Servidor SMTP válido (por ejemplo Gmail u otro proveedor).
- Credenciales SMTP configuradas en el archivo .env del backend.

### 3. Arquitectura del sistema

La aplicación está diseñada con una arquitectura clásica cliente–servidor:

- **Cliente (PWA)**  
Aplicación React que se ejecuta en el navegador y también puede instalarse como app PWA. Desde aquí se realizan todas las operaciones de usuario: login, registro, agendamiento de visitas, consulta de historial y visualización de notificaciones en tiempo real.
- **API Backend (Node + Express + TypeScript)**  
Servidor HTTP que expone endpoints REST bajo rutas como /api/auth, /api/visits, /api/notifications, /api/users, etc.  
La lógica se organiza en:
  - Rutas (routes/\*.ts)
  - Controladores (controllers/\*.ts)
  - Servicios (services/\*.ts)
  - Utilidades comunes (utils/\*.ts)
  - Configuración de BD (config/db.ts)
  - Jobs de background (jobs/\*.ts)
- **Base de datos PostgreSQL**  
Contiene las tablas de usuarios, internos, visitas, relación usuario–interno, notificaciones y otras tablas de soporte. Toda la lógica de negocio persiste su estado en esta BD.
- **Canal de notificaciones en tiempo real (SSE)**  
Se usa Server-Sent Events para enviar notificaciones en vivo al frontend. El endpoint /api/notifications/stream abre una conexión unidireccional desde el servidor hacia el cliente.
- **Correo SMTP**  
El módulo mailer.ts se encarga de enviar correos de recuperación de contraseña y recordatorios de visita. Se integra dentro de los servicios de negocio para disparar correos cuando corresponde.

El patrón general es similar a una arquitectura por capas: capa de presentación (React), capa de API (Express), capa de servicios de dominio y capa de acceso a datos (PostgreSQL).

#### 4. Especificaciones del software utilizado

- Lenguaje backend: TypeScript sobre Node.js.
  - Framework backend: Express.
  - ORM / acceso a datos: uso directo de pg con un Pool configurado en config/db.ts.
  - Lenguaje frontend: TypeScript.
  - Framework frontend: React + Vite.
  - Pruebas unitarias: Jest + ts-jest.
  - Pruebas de integración: Jest + Supertest (levantando la app Express en modo prueba).
  - Pruebas de rendimiento: Apache JMeter (endpoint de login y, opcionalmente, otros endpoints).
  - Pruebas de accesibilidad: Lighthouse (modo Mobile, pestaña Accessibility).
  - Envío de correos: Nodemailer, configurado en src/backend/src/utils/mailer.ts.
  - Notificaciones internas: Implementadas en src/backend/src/services/notifications.service.ts, usando SSE para el envío al frontend.
- 

#### 5. Estructura de la base de datos

A nivel lógico, la base de datos PostgreSQL incluye las siguientes tablas relevantes:

- **users**  
Campos principales:
  - id (PK, uuid)
  - name, email, password\_hash
  - role (por ejemplo: ADMIN, USER)
  - notify\_email (booleano para indicar si el usuario desea recibir correos)
  - Timestamps de auditoría (created\_at, updated\_at)
- **inmates**  
Contiene los internos disponibles para agendamiento:
  - id (PK)
  - first\_name, last\_name, full\_name (opcional)
  - Datos de identificación propios del centro
  - Campos de estado (activo, pabellón, etc., según el diseño)
- **user\_inmates**  
Tabla de relación que define qué usuarios pueden agendar visitas a qué internos:
  - user\_id (FK a users)
  - inmate\_id (FK a inmates)
  - PK compuesta (user\_id, inmate\_id)
- **visits**  
Tabla central del sistema:
  - id (PK, uuid)
  - visitor\_name
  - inmate\_name

- inmate\_id (FK opcional a inmates)
- visit\_date (date)
- visit\_hour (time)
- duration\_minutes (int, valores permitidos: 30, 60, 90, 120)
- status (PENDING, APPROVED, REJECTED, CANCELLED)
- notes
- created\_by (FK a users, dueño de la visita)
- created\_at, updated\_at
- **notifications**  
Registra las notificaciones internas del sistema:
  - id (PK, uuid)
  - user\_id (FK a users)
  - visit\_id (FK opcional a visits)
  - kind (VISIT\_CREATED, VISIT\_APPROVED, VISIT\_UPDATED, VISIT\_REMINDER, VISIT\_CANCELED, SYSTEM)
  - title, body
  - meta (jsonb con información adicional de la visita)
  - is\_read (booleano)
  - created\_at, read\_at

Existen índices sobre las claves foráneas y campos usados en filtros frecuentes (por ejemplo, visit\_date, status, created\_by) para mejorar el rendimiento en las consultas.

---

## 6. Procedimientos de instalación y ejecución

### **6.1. Preparar entorno backend**

1. Clonar el repositorio del proyecto.
2. Entrar en la carpeta src/backend.
3. Instalar dependencias con npm install.
4. Configurar el archivo .env del backend con las variables:
  - PGHOST, PGPORT, PGDATABASE, PGUSER, PGPASSWORD
  - DB\_SSL si se requiere SSL en la conexión (por ejemplo "false" en local).
  - JWT\_SECRET para la firma de tokens.
  - SMTP\_HOST, SMTP\_PORT, SMTP\_USER, SMTP\_PASS, SMTP\_FROM para el envío de correos.
  - NODE\_ENV=development para el entorno local.
5. Ejecutar los scripts de creación de tablas y datos iniciales en PostgreSQL (según el archivo SQL del proyecto).

Para levantar el backend:

- npm run dev para modo desarrollo (por ejemplo usando ts-node-dev).
- npm run build + npm start si se desea compilar y ejecutar la versión transpiled.

## 6.2. Preparar entorno frontend

1. Entrar en la carpeta src/frontend.
2. Instalar dependencias con npm install.
3. Configurar el archivo .env o .env.local con la URL del backend, por ejemplo:
  - VITE\_API\_BASE\_URL=http://127.0.0.1:3000/api
4. Levantar el frontend con npm run dev.

La aplicación se abre en el navegador (Vite normalmente usa el puerto 5173). Desde ahí se puede usar también la opción de instalar como PWA.

## 6.3. Ejecución de pruebas

- Pruebas unitarias e integración (backend): desde src/backend:
  - npm test para ejecutar Jest.
- Pruebas de rendimiento: cargar el plan de pruebas en JMeter (Thread Group, HTTP Request a /api/auth/login, Summary Report).
- Pruebas de accesibilidad: abrir la URL correspondiente en el navegador y ejecutar Lighthouse en modo Mobile, pestaña Accessibility.

## 6.4. Notificaciones de recordatorio

- Asegurarse de tener el job startReminderJob registrado al arrancar el servidor (por ejemplo en server.ts).
- Este job ejecuta periódicamente upsertTomorrowReminders, que inserta notificaciones internas y envía correos de recordatorio a los usuarios que tienen visitas para el día siguiente y notify\_email = true.

## 7. Módulos y funcionalidades internas

### **Auth**

- Endpoints /api/auth/login, /register, /forgot-password, /reset-password.
- Genera tokens JWT que se remiten al frontend y se usan en el header Authorization: Bearer ....
- En /forgot-password se genera un token de recuperación y se envía un correo usando sendResetEmail.

### **Visitas (visits.service.ts y visits.controller.ts)**

- createVisit:
  - Valida duración permitida (30, 60, 90, 120 minutos).
  - Valida que la visita esté entre las 08:00 y las 17:00.
  - Verifica solapamientos para el mismo interno en la misma fecha.
  - Verifica que el usuario esté autorizado para ese interno, salvo que skip\_auth sea verdadero (caso administrador).
  - Inserta la visita y crea una notificación VISIT\_CREATED para el usuario.
- updateVisit:
  - Lee el estado anterior de la visita.
  - Mezcla los datos nuevos con los existentes.
  - Vuelve a validar duración, rango horario y solapamientos.
  - Verifica autorización si se cambia el interno.
  - Actualiza la visita.
  - Genera notificaciones:
    - VISIT\_APPROVED si el estado cambia a aprobado.
    - VISIT\_CANCELED si el estado pasa a rechazado (o cancelado).
    - VISIT\_UPDATED si se modifica fecha, hora o interno sin cambiar estado.
- cancelVisit:
  - Permite a un usuario cancelar su propia visita si aún está en estado PENDING o APPROVED y la fecha es igual o posterior al día actual.
  - Marca la visita como CANCELLED y actualiza la fecha de modificación.
- listVisits:
  - Devuelve las visitas filtrando por fecha, estado, usuario creador o interno.
  - Se usa tanto para el listado del usuario como para el panel del administrador.

### **Notificaciones (notifications.service.ts y notifications.routes.ts)**

- createNotification:
  - Inserta una fila en la tabla notifications.
  - Llama a ssePush para enviar la notificación en tiempo real al usuario conectado.
- listNotifications:

- Permite obtener la lista de notificaciones de un usuario, opcionalmente solo las no leídas y con límite.
- markAsRead:
  - Marca un conjunto de notificaciones como leídas y registra read\_at.
- countUnread:
  - Retorna el número de notificaciones pendientes para mostrar el badge en el frontend.
- upsertTomorrowReminders:
  - Busca visitas del día siguiente con estado PENDING o APPROVED.
  - Filtra solo usuarios que tienen activo notify\_email.
  - Verifica que no exista ya un recordatorio del tipo VISIT\_REMINDER para esa visita.
  - Crea la notificación interna y envía el correo correspondiente.
- Rutas principales:
  - GET /api/notifications/stream: abre el canal SSE.
  - GET /api/notifications: lista notificaciones del usuario autenticado.
  - GET /api/notifications/unread-count: devuelve el conteo de no leídas.
  - POST /api/notifications/mark-read: marca notificaciones como leídas.
  - POST /api/notifications/run-reminders: endpoint de desarrollo para disparar manualmente upsertTomorrowReminders (bloqueado en producción).

### **Mailer (utils/mailer.ts)**

- Configura nodemailer con los datos del SMTP.
- sendResetEmail: envía el correo de recuperación de contraseña.
- sendVisitReminderEmail: envía el recordatorio de visita para mañana, incluyendo datos de interno, fecha y hora.

---

## 8. Seguridad y perfiles de usuario

- Autenticación basada en JWT:
  - El backend emite un token en /api/auth/login.
  - El frontend lo almacena (por ejemplo en localStorage) y lo envía en cada petición protegida.
  - Middleware requireAuth valida el token y adjunta el usuario al objeto req.
- Autorización por rol:
  - Rol ADMIN: puede ver y gestionar todas las visitas, aprobar, rechazar y eliminar.
  - Rol USER: solo puede ver y manipular sus propias visitas (las que él creó).
  - En el backend se hacen verificaciones directas de user.id y role antes de permitir operaciones críticas.
- Autorización por relación usuario–interno:
  - La tabla user\_inmates define qué internos puede visitar un usuario.
  - Antes de crear o modificar una visita (cuando no se usa skip\_auth), el sistema valida que exista la relación user\_id–inmate\_id.

- Validaciones contra ataques:
    - Uso de consultas parametrizadas con pg para evitar inyecciones SQL.
    - Validación de rangos, formatos de fecha/hora y estados permitidos en las funciones de servicio.
    - Pruebas específicas de seguridad: inyección SQL en login y filtros, acceso sin token, token inválido y accesos a endpoints de administrador con usuario normal (estos casos se documentan en el reporte de defectos y de pruebas).
- 

## 9. Solución de problemas (Troubleshooting)

Algunos problemas frecuentes y cómo abordarlos:

- **No conecta con la base de datos**  
Verificar que el servicio PostgreSQL esté levantado, que las credenciales en el .env sean correctas y que PGPORT coincida con el puerto real (por ejemplo 5432 o 5433). Revisar también si se requiere DB\_SSL=true o false según el entorno.
- **Error al enviar correos**  
Revisar que las variables SMTP\_HOST, SMTP\_USER, SMTP\_PASS y SMTP\_PORT sean válidas. Probar el envío mediante una herramienta como Postman llamando al endpoint de prueba de correo (por ejemplo el de “forgot password”) para confirmar que Nodemailer se conecta bien.
- **No se muestran notificaciones en el frontend**  
Confirmar que el cliente está llamando a /api/notifications/stream con el token correcto y que el usuario tiene notificaciones en la tabla. Revisar también la consola del navegador por errores de SSE o CORS.
- **No permite agendar en ciertos horarios**  
Verificar que la hora y la duración cumplan con las restricciones (entre 08:00 y 17:00, duración máxima de 120 minutos) y que no exista solapamiento con otra visita para el mismo interno.
- **Usuario no puede agendar a un interno específico**  
Comprobar que existe la relación en user\_inmates entre ese usuario y ese interno. Si no existe, debe crearla el administrador.

## 10. Glosario

- **PWA:** Aplicación web progresiva. Sitio web que se puede instalar como aplicación en el dispositivo.
- **SSE (Server-Sent Events):** Mecanismo para que el servidor envíe eventos al navegador de forma continua usando una conexión HTTP.
- **JWT:** Json Web Token, formato de token usado para autenticación.
- **Endpoint:** Ruta específica de la API que expone una funcionalidad (por ejemplo /api/visits).
- **Thread** (en JMeter): Hilo que simula un usuario concurrente en las pruebas de carga.
- **Ramp-up:** Tiempo que tarda JMeter en levantar todos los usuarios simulados.
- **API REST:** Conjunto de endpoints HTTP que permiten a otras aplicaciones comunicarse con el sistema usando operaciones estándar como GET, POST, PATCH y DELETE.
- **JWT (JSON Web Token):** Token firmado que se utiliza para identificar al usuario en cada petición sin guardar sesión en el servidor.
- **SSE (Server-Sent Events):** Mecanismo en el que el servidor mantiene una conexión abierta y envía eventos al navegador en tiempo real.
- **Endpoint:** URL específica de la API que atiende una operación concreta (por ejemplo, /api/auth/login).
- **Middleware:** Función de Express que se ejecuta entre la llegada de la petición y la lógica final, usada para tareas como autenticación o logging.
- **SMTP:** Protocolo para envío de correos electrónicos desde la aplicación hacia Internet.
- **Sandbox de correo (Mailtrap):** Servicio que permite recibir correos de prueba sin enviarlos a direcciones reales, útil en desarrollo y pruebas.

# RETROSPECTIVA DEL EQUIPO

Durante el desarrollo de VisiControl hubo varias cosas que funcionaron bien y otras que costaron más de lo esperado.

Lo que funcionó:

- La separación entre frontend y backend ayudó a organizar el trabajo. Fue posible avanzar en las pantallas React (login, dashboard, visitas, historial) mientras se afinaba la API.
- Introducir pruebas automatizadas con Jest y Supertest permitió detectar errores de lógica en las funciones de visitas y autenticación antes de las pruebas manuales.
- El uso de PostgreSQL con un diseño relativamente normalizado hizo más fácil agregar reglas como evitar solapamiento de horarios y manejar la relación usuario–interno.
- Tener un sistema de notificaciones interno con SSE mejoró la experiencia del usuario, porque no tiene que recargar la página para ver cambios importantes.

Lo que no funcionó tan bien:

- La planificación del despliegue se dejó para muy tarde. Se trabajó mucho tiempo en entorno local y la parte de hosting y acceso externo (por ejemplo Render) se complicó al final, sobre todo por temas de costos y configuración.
- La configuración de correo real (Gmail) también se resolvió sobre la marcha, lo que generó errores de conexión y bloqueos que tomaron tiempo.
- Algunos flujos clave (como cancelar visitas, reprogramar con reglas claras y notificaciones completas) fueron apareciendo después de avanzar en otras partes, así que hubo que modificar varias veces el servicio de visitas y la interfaz de historial.
- El trabajo con variables de entorno y diferentes IPs en el enlace de recuperación de contraseña y en las pruebas desde el celular fue más confuso de lo esperado.

Aprendizajes:

- Es importante planificar desde el inicio el entorno donde se va a desplegar la app y probar las integraciones externas (SMTP, hosting, etc.) antes de la entrega final.
- Definir bien los casos de uso y las reglas de negocio al principio evita reescrituras grandes en servicios como `visits.service.ts`.
- Las pruebas automatizadas (unitarias, integración y de rendimiento) ahorran tiempo cuando se realizan de forma iterativa y no solo al final del proyecto.
- La accesibilidad no se debe dejar como “extra”, porque pequeños detalles como `aria-label` o `label` en formularios se pueden resolver fácilmente si se piensa en eso desde el diseño.
- Documentar las decisiones técnicas y los cambios importantes ayuda mucho cuando se arma el manual técnico y cuando alguien más tiene que entender el proyecto.

# PLAN DE EVOLUCIÓN FUTURA

Para seguir mejorando VisiControl después de la entrega académica se plantean varias líneas de evolución:

1. Despliegue estable en la nube
  - Migrar el backend y la base de datos a un proveedor en la nube (Render, Railway, VPS, etc.) con dominio público estable.
  - Ajustar correctamente DATABASE\_URL y SMTP para que la app sea accesible desde cualquier red sin tener que cambiar la IP en los enlaces.
2. Notificaciones push reales en la PWA
  - Implementar Service Worker para manejar push notifications.
  - Integrar un servicio de envío (por ejemplo, Firebase Cloud Messaging u otro) para que el usuario reciba avisos aunque no tenga abierta la app, complementando las notificaciones internas y los correos.
3. Mejoras en el panel del administrador
  - Filtros más avanzados por rango de fecha, pabellón, tipo de interno y estado.
  - Estadísticas básicas: número de visitas por día, internos más visitados, horas pico, etc.
  - Exportación de datos a CSV o PDF para reportes.
4. Gestión más completa de usuarios e internos
  - Pantallas para que el administrador gestione la relación usuario–interno de manera visual.
  - Manejo de estados del interno (traslado, baja, suspensión de visitas) y reflejo automático en el módulo de agendamiento.
5. Seguridad y auditoría
  - Registro de logs más detallados de acciones sensibles: quién aprobó, quién rechazó, quién reprogramó, desde qué IP.
  - Posible integración con 2FA (doble factor) para cuentas de administrador.
  - Endpoints adicionales para exportar logs y revisar actividad sospechosa.
6. Optimización de rendimiento
  - Ampliar las pruebas de JMeter a otros endpoints críticos (por ejemplo, listados de visitas y panel de administrador) con más usuarios simulados.
  - Añadir índices adicionales en la base de datos si las consultas de filtrado se vuelven pesadas con muchos registros.
7. Experiencia de usuario
  - Mejorar el diseño responsive y la usabilidad en móviles, aprovechando que la app funciona como PWA.
  - Añadir mensajes más claros en la interfaz cuando se produzcan errores de validación o solapamiento de horarios.
  - Incluir un pequeño tour o guía para usuarios nuevos.