UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

DEPARTAMENTO DE ELECTROTECNIA E INFORMÁTICA
INGENIERÍA EN INFORMÁTICA

Desarrollo de Front-end para el Sistema de gestión y confirmación de Citas Médicas en el Hospital Dr. Gustavo Fricke

Ignacio Andrés Tenorio Roa

Ignacio.tenorio@usm.cl

Carlos Alten
Profesor Guía

Hernán Saavedra
Profesor Correferente

Resumen: Este trabajo presenta el desarrollo del front-end de GustaBot, sistema para la gestión y confirmación de citas médicas del Hospital Gustavo Fricke. Se implementa una interfaz web modular y mantenible basada en React y JSX. Esto da como resultado una interfaz alineada con los procesos hospitalarios que reduce la fricción operativa de la gestión diaria, mejorando así también la experiencia de usuario.

Palabras clave: Front-end, React, WhatsApp Bot, Gestión de Citas Médicas, Arquitectura basada en componentes

1 Introducción

1.1 Contexto

El Hospital Gustavo Fricke, ubicado en Viña del Mar, es uno de los principales centros de salud públicos en la Región de Valparaíso y recibe diariamente una alta demanda de pacientes para consultas médicas y procedimientos ambulatorios. La gestión de horas ambulatorias es un proceso crítico, ya que involucra tanto a los pacientes como al personal administrativo y clínico de la institución.

Gran parte de la confirmación de citas se realiza mediante llamadas telefónicas y procesos manuales.

1.2 Definición del problema

Como se explicó en el Contexto, gran parte de la confirmación de citas se realiza mediante llamados telefónicos. Este canal además de ser manual y lento ha perdido efectividad debido a que muchas personas ignoran las llamadas telefónicas por el aumento de estafas telefónicas y "spam". El resultado es una baja tasa de respuesta, dificultad para confirmar o cancelar a tiempo y pérdida de cupos cuando el paciente no asiste ni avisa, situación que se refleja en una tasa de inasistencia del 15% anual en consultas ambulatorias del Hospital Dr. Gustavo Fricke, evidenciando la necesidad de un canal de comunicación más efectivo.

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

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

Título del trabajo: **Desarrollo de Front-end para el Sistema de gestión y confirmación de Citas Médicas en el Hospital Dr. Gustavo Fricke**

Nombre del candidato(a): **Ignacio Andrés Tenorio Roa**

Carrera / Grado: Ingeniería en Informática

Campus: _Viña del Mar **Departamento:** Electrotecnia e Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Carlos Alten López, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

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

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

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

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

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

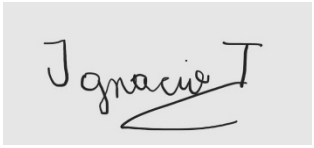
Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

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

Fecha: 20 de abril de 2026 **Firma:** 

Estudiante o Candidato(a):

Fecha: _____17/04/2026_____ **Firma:** 

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

1.2.1 Problemas encontrados

- Baja efectividad de las llamadas telefónicas: por el aumento de estafas/spam, muchas personas ignoran las llamadas telefónicas; el canal de comunicación se percibe invasivo/sospechoso.
- Proceso manual y lento: la posibilidad de contactar a los pacientes depende de llamados y registros manuales, lo que consume tiempo y es propenso a errores.
- Visibilidad limitada del estado actual: no existe una interfaz única donde el operario vea el estado vigente de la cita (confirmada/cancelada).

1.2.2 Síntomas e impacto

Síntomas:

- Inasistencia elevada
- Baja tasa de respuesta a llamados
- Retrabajo (múltiples intentos)
- Demora en reflejar cambio de estado

Impacto: menor productividad operativa, uso ineficiente de horas médicas, mala experiencia para pacientes como para funcionarios y pérdida de recursos hospitalarios.

1.3 Objetivo General

Implementar la interfaz de usuario del sistema GustaBot, desarrollando un front-end modular, escalable y funcional que permita a operarios y administradores interactuar eficientemente con el sistema e integrarse con los servicios de back-end para la confirmación y estado de citas, cubriendo la totalidad de los requerimientos funcionales definidos.

1.4 Objetivos Específicos

- Diseñar la arquitectura del front-end de GustaBot, basada en organización por features y componentes reutilizables en React.

- Implementar las 4 vistas principales del sistema: inicio de sesión, gestión de citas médicas y administración de usuarios y médicos.

- Establecer la comunicación entre el front-end y las APIs del back-end.

- Evaluar el funcionamiento del front-end mediante pruebas funcionales manuales e integración sobre los 7 flujos principales de gestión de citas y envío de recordatorios a pacientes.

- Documentar la solución de front-end desarrollada, asegurando la mantenibilidad y escalabilidad del sistema.

1.5 Justificación del proyecto

La implementación del front-end de GustaBot es un componente esencial para el éxito del sistema, ya que representa el medio de interacción entre los usuarios y las funcionalidades desarrolladas. Desde el punto de vista técnico, este trabajo aporta al desarrollo de soluciones escalables y modulares en React, aplicando buenas prácticas que facilitan la mantenibilidad y futura evolución del sistema.
Desde la perspectiva hospitalaria permitirá reducir la carga operativa de los funcionarios, optimizar la gestión de horas médicas y disminuir las inasistencias, al estandarizar el flujo de confirmación/cancelación y entregar visibilidad oportuna del estado actual de las citas, apoyando una gestión más eficiente y contribuyendo al objetivo institucional de reducir la tasa de inasistencia ambulatoria del 15% al 10% anual.

1.5.1 Beneficios

- Menos carga para quienes se ocupan de confirmar las citas, porque todo se hace desde una sola interfaz sin planillas paralelas.
- Estado de la cita fácil de entender, porque la pantalla muestra de forma clara si está confirmada o cancelada.
- Menos errores en la operación, porque el flujo está guiado con botones y mensajes consistentes.
- Interacción más simple para el paciente, porque puede confirmar o cancelar desde un mensaje de WhatsApp.

1.6 Metodología

El desarrollo del front-end de GustaBot se llevará a cabo mediante un enfoque ágil, específicamente con la metodología Scrum. A diferencia de metodologías tradicionales como cascada o enfoques sin iteraciones estructuradas, Scrum fue seleccionado por su capacidad de adaptar prioridades cada sprint, lo que resultó especialmente adecuado dado que los requerimientos del hospital podían variar a medida que se probaba la solución.
Scrum se basa en un modelo iterativo e incremental, donde el desarrollo se organiza en ciclos cortos denominados Sprints, que permiten entregar incrementos funcionales del producto de manera continua y obtener retroalimentación temprana de los involucrados [1]. Esto resulta especialmente relevante en el contexto hospitalario, donde los requerimientos pueden variar a medida que se prueba la solución.

La elección de Scrum responde a los siguientes factores:

- Adaptabilidad a cambios: a diferencia de metodologías tradicionales, Scrum permite ajustar prioridades en cada sprint, lo que lo hace útil en proyectos donde los requerimientos pueden variar [2].
- Colaboración constante: Scrum fomenta la comunicación de los miembros del equipo mediante reuniones periódicas (daily meetings, sprint reviews), favoreciendo la integración del front-end con otros componentes del sistema [3].
- Validación temprana: al entregar prototipos funcionales al final de cada sprint, es posible validar las vistas y componentes desarrollados con los usuarios involucrados, reduciendo riesgos y asegurando que el producto final sea utilizable [1].

1.7 Breve descripción de la organización del informe en capítulos

En los capítulos siguientes se profundizará en:

Capítulo 1: Introducción. Describe el contexto, el problema específico del front-end, los objetivos, la justificación, la metodología y la organización del trabajo.

Capítulo 2: Marco Teórico. Expone los fundamentos del desarrollo front-end, la arquitectura de software aplicada y las tecnologías utilizadas.

Capítulo 3: Diseño e Implementación. Presenta la arquitectura de componentes, las vistas desarrolladas, el uso de buenas prácticas y patrones, la integración con el back-end y los aspectos de codificación, desarrollo, pruebas y validación del front-end.

Capítulo 4: Conclusiones. Resume los resultados obtenidos, los aportes del trabajo, las limitaciones encontradas y las proyecciones futuras.

Capítulo 5: Bibliografía. Contiene las referencias bibliográficas utilizadas para sustentar el marco conceptual y las decisiones técnicas del trabajo.

2 Marco Teórico

En este capítulo se presentan los fundamentos teóricos que sustentan la implementación del front-end de GustaBot. Se explorarán los principales conceptos relacionados con el desarrollo front-end, su rol dentro de la arquitectura de software y la evolución de las aplicaciones web modernas. Luego se describen las decisiones arquitectónicas aplicadas al proyecto (composición por componentes, manejo de estado, ruteo y consumo de APIs REST). Finalmente, se detallan las tecnologías y herramientas empleadas, estableciendo las bases conceptuales que justifican las elecciones técnicas de los capítulos posteriores.

2.1 Fundamentos del desarrollo front-end

El front-end es la capa de presentación e interacción de un sistema: convierte reglas de negocio y datos en interfaces comprensibles para las personas [4]. Por ello es esencial en la construcción de software, ya que materializa los flujos del negocio en experiencias de uso y posibilita la comunicación con la capa de datos y procesos.

2.1.1 Características principales

El front-end se ocupa de tres responsabilidades principales:

- Interacción con el usuario a través de formularios, botones, validaciones, estados de error/éxito y mensajes claros [4].
- Presentación de la información teniendo en cuenta la jerarquía visual, tipografía y componentes que mantienen la consistencia [5].
- Comunicación con el back-end/servicios como el consumo de APIs y manejo de respuesta/errores para mantener la UI coherente y segura [6].

A continuación, se presenta un diagrama que nos permite ubicar el front-end dentro de una arquitectura cliente-servidor y explicita sus tres responsabilidades (interacción, presentación y comunicación). Véase la Figura 2-1.

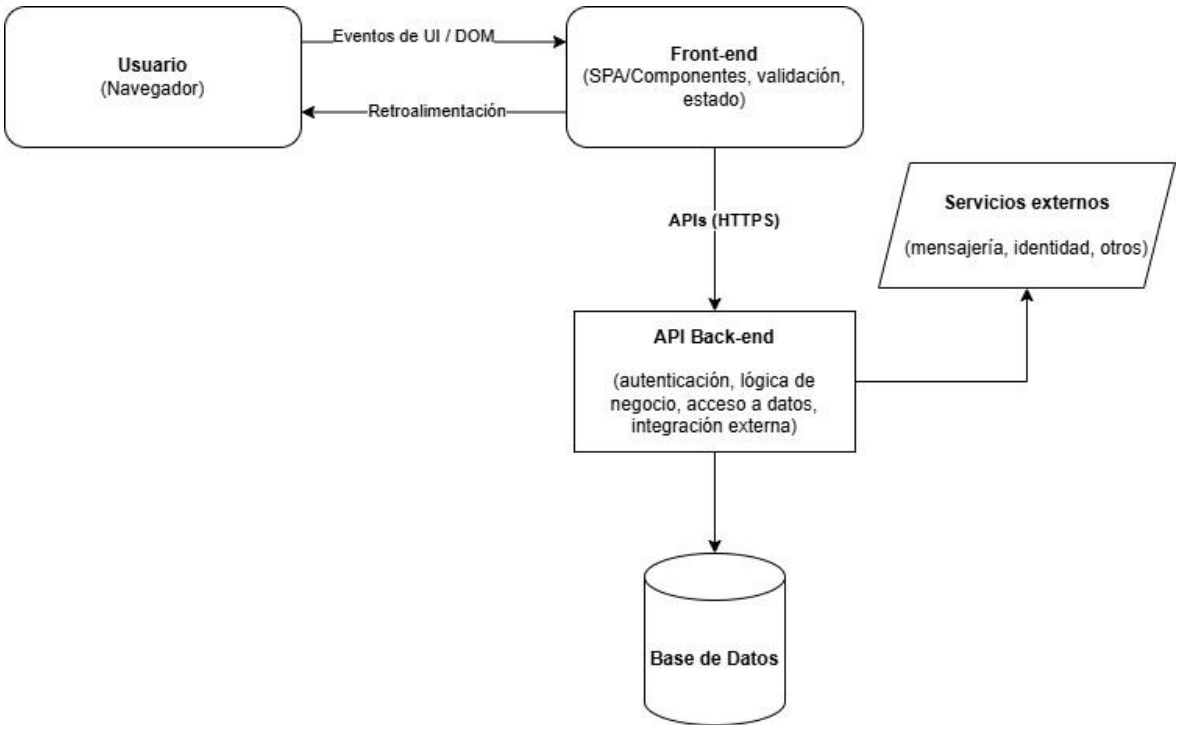


Figura 2-1. Rol del front-end en una arquitectura cliente–servidor

Fuente: Elaboración propia

La lectura del diagrama es de izquierda a derecha donde el usuario interactúa desde el navegador, el front-end captura eventos de UI/DOM y entrega retroalimentación inmediata. A través de APIs (HTTPS) invoca la API del back-end. Este resuelve la lógica del negocio y persiste en la base de datos. Cuando corresponde, se apoya en servicios externos (mensajería, identidad, entre otros) como dependencias complementarias al sistema.

El front-end es esencial porque es el punto de contacto entre las personas y el software, al tener una buena capa de presentación incrementa la eficacia, eficiencia y satisfacción de uso del sistema.

2.1.2 Evolución del front-end

Históricamente, el front-end pasó de documentos estáticos (HTML/CSS) a experiencias dinámicas impulsadas por JavaScript [4]. Con esta evolución surgieron las aplicaciones de una sola página (SPA), que cargan una base inicial y actualizan vistas y estado sin recargar toda la página, mejorando la fluidez percibida [6]. En paralelo, se consolidó la arquitectura basada en componentes, que encapsula la estructura y comportamiento para componer interfaces complejas de forma mantenible [7].

El enfoque tradicional consistía en:

- Páginas generadas en el servidor (MPA), lo que hacía que cada acción de usuario implicara una navegación completa y la reconstrucción de vista.
- Interactividad limitada: validaciones y respuestas post-back.
- Acoplamiento fuerte entre presentación y back-end ya que el HTML era incrustado en plantillas del servidor y el CSS/JS eran globales con bajo reúso.
- Experiencia irregular ante la latencia de la red haciendo que la UI "parpadeara" o quedara en blanco durante las recargas en la página.

Los problemas que impulsaron el cambio fueron la necesidad de mayor interactividad que el modelo página-a-página no resolvía totalmente, la mantenibilidad/escalabilidad del UI, el rendimiento percibido por la latencia visual, la clara separación de UI y servicios (parte del front-end que consume APIs) y la consistencia de múltiples dispositivos para una experiencia homogénea [6].

Para atender estos problemas, la industria adoptó interfaces declarativas y componibles (componentes), navegación en cliente (SPA), gestión de estado y tooling moderno; estas prácticas y enfoques actuales se detallan en la siguiente sección.

2.1.3 Tendencias actuales del front-end

Luego de la transición histórica descrita en el subcapítulo anterior el desarrollo front-end consiste en prácticas y enfoques que buscan mantener la consistencia, rendimiento percibido y la mantenibilidad. A continuación, se sintetizan las tendencias más relevantes:

- Arquitectura por componentes y sistemas de diseño, donde las interfaces se construyen con piezas reutilizables y coherentes, lo que reduce la variabilidad y baja el costo de cambio [5].
- API-first y el desacoplamiento de UI-servicios: la interfaz consume APIs estables y logra evolucionar de forma independiente del back-end, acelerando los ciclos de entrega y facilitando pruebas de integración [6].
- Accesibilidad y responsividad como estándar con semántica HTML, roles/ARIA, contraste y navegación por teclado; diseño adaptable a múltiples tamaños de pantalla [4].
- Aplicaciones web progresivas (PWA) con instalación ligera, operación limitada sin conexión y notificaciones cuando el caso de uso lo amerita [8].
- Tooling moderno y DX que serían herramientas de construcción y recarga rápida como HMR (Hot Module Replacement) las cuales permiten acortar los ciclos de edición-prueba y favorecen la calidad continua (tipado y pruebas de componentes/integración) [9].

2.2 Arquitectura de software aplicada a front-end

En este apartado presentaremos los conceptos teóricos que sustentan el diseño de aplicaciones de interfaz de usuario en la web. Primero se introducen modelos y estilos arquitectónicos, luego se describen patrones transversales de diseño y calidad. A continuación, se ofrece un panorama de frameworks basados en componentes y, finalmente, se explican los criterios que justifican la elección de este.

2.2.1 Modelos y estilos arquitectónicos aplicados al front-end

Antes de presentar los conceptos, recordemos que el objetivo de un front-end moderno busca separar responsabilidades, optimizar la experiencia y acotar el acoplamiento con el servidor.

Modelo cliente-servidor donde el front-end presenta la UI y consume APIs del back-end; la lógica de negocio y la persistencia residen en el servidor [6].

SPA (Single-Page Application) tiene una carga inicial única y actualizaciones parciales de vista/estado sin recargar la página completa, esta mejora el rendimiento percibido y mantiene el contexto del usuario [6].

Arquitectura basada en componentes consiste en que la UI se compone de piezas reutilizables que encapsulan estructura y comportamiento, lo que favorece mantenibilidad y consistencia [7].

Gestión de estado en el cliente mantiene la sincronía entre UI y datos (como formularios, filtros, estados de carga/éxito/error) habilitando interacciones reactivas.

Ruteo en cliente permite la navegación lógica entre vistas sin recarga completa.

Enfoque API-first permite que las interfaces evolucionen desacopladas del servidor mediante contratos de API estables [6].

La Figura 2-2 resume el estilo cliente-servidor aplicado al consumo de servicios desde el front-end.

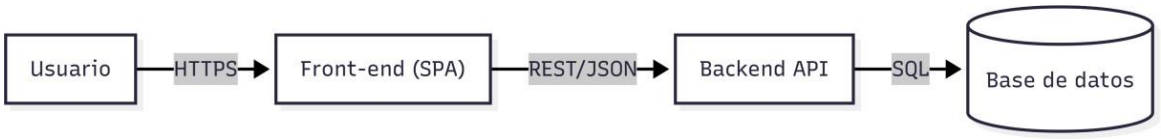


Figura 2-2. Estilo cliente-servidor para consumo de API desde el front-end.

Fuente: Elaboración propia.

2.2.2 Patrones de diseño y buenas prácticas transversales

Estos patrones ayudan a sostener calidad (usabilidad, rendimiento, mantenibilidad) cuando crece la aplicación.

- La composición de componentes permite construir UIs complejas combinando piezas simples, evitando así la duplicación y facilitando pruebas [5].

- El flujo de datos unidireccional hace más predecible el estado, simplificando la depuración.
- Validación y manejo de errores en UI, esto permite prevenir las acciones indebidas y reduce el retrabajo [4].
- El rendimiento percibido con paginación, carga diferida y división de código para evitar bloqueos visibles [8].
- Observabilidad de la UI con mensajes y estados claros como cargando/éxito/error para guiar al usuario.

La Figura 2-3 ilustra el flujo unidireccional de datos, donde las acciones disparan llamadas a servicios y la actualización del estado produce la actualización de la interfaz.

Figura 2-3. Flujo unidireccional de datos en el front-end.

Fuente: Elaboración propia.

2.2.3 Frameworks y librerías basadas en componentes

Antes de justificar una tecnología se presenta el panorama general: React, Angular y Vue comparten la idea de componentes reutilizables y enfoque declarativo de la UI, se integran con ruteo, estado y tooling para soportar SPAs y, cuando aplica SSR/SSG con hidratación. En todos los casos mencionados anteriormente, el denominador común es la componentización y optimización de la entrega [5], [6], [8].

2.2.4 ¿Qué es React?

React es una librería de JavaScript para construir interfaces de usuario a partir de componentes. Permite describir la UI de forma declarativa, es decir, el desarrollador define "qué" se quiere ver según el estado de la aplicación y React se ocupa de actualizar eficientemente la vista cuando el estado cambia [5]. En el contexto de una SPA, React coordina el cambio de vistas sin recargar toda la página, manteniendo la continuidad del trabajo de usuario [6].

2.2.5 Elección de React

Tras el panorama de frameworks basados en componentes, se justifica el uso de React por sus criterios teóricos ligados a la mantenibilidad, experiencia y ecosistema.

- Modelo de componentes + JSX facilita el reúso y legibilidad de vistas, y promueve una jerarquía clara de UI [10].
- Ecosistema amplio y maduro con disponibilidad de ruteo, estado, formularios, pruebas y herramientas de construcción rápida como Vite [9].
- Curva de adopción y comunidad, React cuenta con una documentación extensa y patrones conocidos como la composición y lifting state up que reducen el costo de incorporación.
- Trade-offs y mitigación. Requiere disciplina en rendimiento (evitando renders innecesarios, división de código) y en la accesibilidad. Esto se logra controlar con prácticas de Core Web Vitals y guías de accesibilidad [4], [8].

2.2.6 Riesgos y mitigaciones

Este apartado identifica los riesgos típicos de las arquitecturas de front-end modernas (SPA, componentes, ruteo y estado en cliente) y describe por qué sucede y cómo mitigarlos.

- La complejidad del tooling ocurre debido a cadenas de build, dependencias y configuraciones que se vuelven frágiles al escalar. Para mitigarla se deben ocupar plantillas y convenciones de proyecto, scripts uniformes, bloqueo de versiones, CI con caché de dependencias, linting y formateos automáticos [9].
- La sobrecarga en el cliente ocurre generalmente por paquetes iniciales grandes, listas sin paginar y renderizados innecesarios. Su plan de mitigación sería el code-splitting y carga diferida, paginación/virtualización de tablas, memorizar cálculos/renders, uso prudente de dependencias y análisis de tamaño de bundle [8].
- El manejo deficiente de errores y estados ocurre por omitir estados de carga/vacío/error dejando al usuario sin contexto y genera retrabajo. Para poder mitigarlo se usan estados explícitos (cargando/éxito/error/vacío), toast/alertas consistentes, reintentos controlados y timeouts definidos [4].
- Regresiones y deuda técnica ocurren por los cambios rápidos sin aplicar test ni control de calidad. Para mitigarlas siempre se deben realizar pruebas de componentes e integración para flujos críticos, revisión por pares y feature flags para despliegues graduales.

2.3 Tecnologías y herramientas utilizadas

En esta sección describiremos las tecnologías que se utilizarán en el front-end y su rol dentro de la arquitectura. Para cada tecnología se explicará qué es y por qué se utiliza en este contexto.

2.3.1 Lenguajes base de desarrollo

El desarrollo se apoya en HTML5, CSS y JavaScript. Con HTML5 se define una estructura semántica que facilita la accesibilidad y ordena los formularios y vistas. CSS permite maquetear con Flexbox/Grid y mantener una presentación consistente y adaptable. Sobre esa base, JavaScript controla la interacción en el navegador, coordinando el estado de la interfaz y realiza llamadas asíncronas a los servicios del back-end sin recargar la página.

2.3.2 Framework de UI: React

La interfaz se implementa con React, adoptando un enfoque declarativo y por componentes [5]. Esta decisión favorece la reutilización de piezas de UI, reduce duplicidades y hace más predecible la evolución de pantallas: los componentes describen "qué" debe mostrarse según el estado actual y React se encarga de actualizar el DOM de manera eficiente. El uso de JSX nos facilita leer y mantener la jerarquía visual directamente desde el código [10].

2.3.3 Estado de la UI y datos remotos

Para sincronizar lo que ve la persona usuaria con los datos del servidor se combinan hooks de estado local de React con TanStack React Query para el server state. React Query aporta caché, revalidación automática y estados de carga/errores uniformes, lo que simplifica el consumo de APIs y mejora la experiencia percibida al trabajar con listas y operaciones sobre citas.

2.3.4 Ruteo en cliente

La aplicación navega sin recargas con React Router, mapeando URLs a vistas y conservando historial y contexto. Se define un layout principal y rutas protegidas tras autenticación. Hay redirecciones por sesión expirada/no autorizado, las vistas usan carga diferida por ruta para reducir el paquete inicial [6], [8].

2.3.5 Interacción con APIs

El consumo de servicios se realiza con Fetch API a través de un pequeño cliente propio que estandariza encabezados, manejo de errores y timeouts. La dirección base del back-end se parametriza mediante Vite, lo que facilita cambiar de entorno sin modificar el código [9].

2.3.6 Estilos y sistema visual

El estilo se resuelve con Tailwind CSS, privilegiando utilidades que aceleran la maquetación y ayudan a mantener consistencia entre pantallas. Este enfoque facilita aplicar criterios de accesibilidad (como el contraste y foco visible) directamente en los componentes, y reduce la necesidad de hojas de estilos extensas [4].

2.3.7 Entorno de desarrollo y build

Para el ciclo de edición-prueba se utiliza Vite con el plugin de React. El HMR y el bundling eficiente permiten iterar rápido en desarrollo y generar un paquete de producción optimizado, con soporte para división de código cuando sea necesario [9].

2.3.8 Herramientas de apoyo al desarrollo

Además del stack principal, se utilizan herramientas de apoyo como Git para control de versiones y colaboración. Para depuración e integración se emplean DevTools del navegador y la documentación de la API mediante Swagger/OpenAPI, verificando solicitudes/respuestas en formato JSON. La validación del front-end se realiza mediante pruebas funcionales manuales sobre los flujos principales, comprobando estados de carga y manejo de errores.

3 Diseño e Implementación

Este capítulo describe la arquitectura técnica e implementación del Front-end desarrollado para el sistema de gestión y confirmación de citas médicas del Hospital Gustavo Fricke. Se detalla cómo se organiza el código, qué componentes lo conforman, cómo se integran con la API y qué decisiones técnicas se tomaron para asegurar mantenibilidad, escalabilidad y claridad en la experiencia de uso.

Las decisiones puramente visuales de UI/UX quedan fuera del alcance de esta tesina, ya que forman parte del trabajo de otro integrante del equipo. No obstante, cuando es necesario, se mencionan brevemente aspectos de interfaz que ayudan a entender la estructura técnica del front-end

3.1 Diseño de Componentes

Este apartado presenta la arquitectura del front-end, los componentes principales que la conforman y cómo se integran con el sistema general. El objetivo es que el lector pueda visualizar el proyecto como un conjunto de módulos coherentes y no solo archivos aislados de código.

3.1.1 Arquitectura y estructura de los componentes

El front-end se implementó como una Single Page Application (SPA) basada en React y empacada con Vite. La arquitectura interna sigue un enfoque basado en componentes y organizado en features, donde cada dominio funcional tiene su propio módulo:

- Citas (features/citas): gestión de citas y envío de mensajes de confirmación/cancelación de cita. El front-end solo desencadena estos envíos mediante endpoints del back-end; toda lógica de integración para el envío de mensajes con WhatsApp se maneja en el servidor.
- Médicos (features/medicos): mantenimiento de médicos (crear, habilitar y deshabilitar)
- Usuarios (features/usuarios): gestión local de operadores que usan el sistema.
- Autenticación (features/auth): pantalla de login y manejo de la "sesión" local
- App/Shared (src/app, src/shared): infraestructura común como Router, guards, cliente HTTP, componentes reutilizables, DTOs, etc.

Todo el front-end se comunica con una API REST del proyecto hospitalario, que expone endpoints para citas y médicos. El envío de mensajes hacia WhatsApp (Meta Business) se realiza a través de la misma API mencionada anteriormente, de modo que el front-end nunca interactúa directamente con la plataforma de mensajería.

3.1.1.1 Requerimientos del front-end

En este apartado se presentan los requerimientos funcionales y no funcionales que cubre el front-end desarrollado para el sistema de gestión de citas y mensajería.

Requerimientos funcionales

- RF1. Autenticación y roles en el front-end.
  El front-end debe proporcionar al usuario la capacidad de iniciar sesión como operario o administrador desde una pantalla de login y restringir la navegación

según rol. En este prototipo, la sesión se simula y se almacena localmente en el navegador.

- RF2. Gestión de citas médicas.
  El front-end debe proporcionar al operario la capacidad de listar las citas médicas en una tabla, con filtros por texto, estado y especialidad.
- RF3. Creación de nuevas citas.
  El front-end debe proporcionar al operario la capacidad de registrar nuevas citas mediante un formulario validado y enviarlas al back-end usando el endpoint POST /api/appointments, actualizando luego el listado.
- RF4. Actualización de estados de citas.
  El front-end debe proporcionar al operario la capacidad de seleccionar una o varias citas y cambiar su estado (confirmada, cancelada) mediante el endpoint PATCH /api/appointments.
- RF5. Envío de solicitudes de mensajería.
  El front-end debe proporcionar al operario la capacidad de seleccionar un conjunto de citas y disparar el envío de mensaje de confirmación o cancelación a los pacientes, integrándose con el servicio de mensajería del proyecto.
- RF6. Gestión de médicos.
  El front-end debe proporcionar al administrador la capacidad de listar, crear y actualizar médicos mediante los endpoints /api/doctors, de forma que el catálogo de médicos visibles en la vista de citas se mantenga consistente.
- RF7. Gestión local de usuarios del sistema.
  El front-end debe proporcionar al administrador la capacidad de gestionar un conjunto de usuarios de prueba (operarios) definidos en el código y permite simular operaciones básicas de alta/baja desde la interfaz, sin persistencia real ni integración aún con un servicio de identidad.

Requerimientos no funcionales

- RNF1. Configuración mediante variables de entorno.
  La URL base del back-end y otros parámetros de integración deben configurarse mediante variables de entorno (por ejemplo, VITE_API_BASE_URL), de modo que el despliegue en distintos ambientes no requiera modificar el código fuente.
- RNF2. Usabilidad técnica y retroalimentación de interfaz.
  El front-end debe proveer feedback visual al operario durante la obtención de datos para evitar incertidumbre. Métrica de éxito: Las consultas iniciales de datos (listados de citas y médicos) deben desplegar componentes de estado de carga (Loader o SkeletonTable) durante la espera, y culminar con un mensaje explícito de éxito o error.
- RNF3. Disponibilidad y resiliencia ante fallos de red.
  El sistema debe proteger la navegación del usuario verificando la salud del servicio antes de operar. Métrica de éxito: Al iniciar la aplicación, el componente ApiStatusGate debe validar la conexión con el back-end en un tiempo máximo de 3.5 segundos (3500ms); de fallar, debe bloquear el acceso a vistas dependientes de datos y ofrecer un botón de reintento manual.
- RNF4. Intercambio de datos y estandarización.
  El front-end debe comunicarse con el back-end mediante solicitudes y respuestas en formato JSON, utilizando los encabezados HTTP correspondientes.
- RNF5. Mantenibilidad y desacoplamiento de datos.
  Para ser tolerante a cambios en el back-end, la interfaz no debe consumir las respuestas crudas de la API. Métrica de éxito: Todos los datos entrantes de citas médicas deben ser procesados y formateados (ej. Fechas a zona local) por una

capa de transferencia de datos (dto.js) antes de llegar a los componentes visuales.

3.1.1.2 Diagrama de casos de uso del front-end

Desde la perspectiva del front-end, el usuario principal del sistema es el operador del hospital, que gestiona citas y envía recordatorios, y un administrador, que realiza tareas de mantenimiento sobre médicos y usuarios locales.

Los casos de uso que el front-end soporta incluyen, entre otros:

- Gestionar citas (listar, filtrar y revisar estado).
- Crear una nueva cita.
- Actualizar el estado de una cita (por ejemplo, confirmada y cancelada).
- Enviar recordatorios a un conjunto de citas seleccionadas.
- Gestionar médicos (crear, habilitar o deshabilitar).
- Gestionar usuarios locales y roles de acceso (a nivel de navegador)

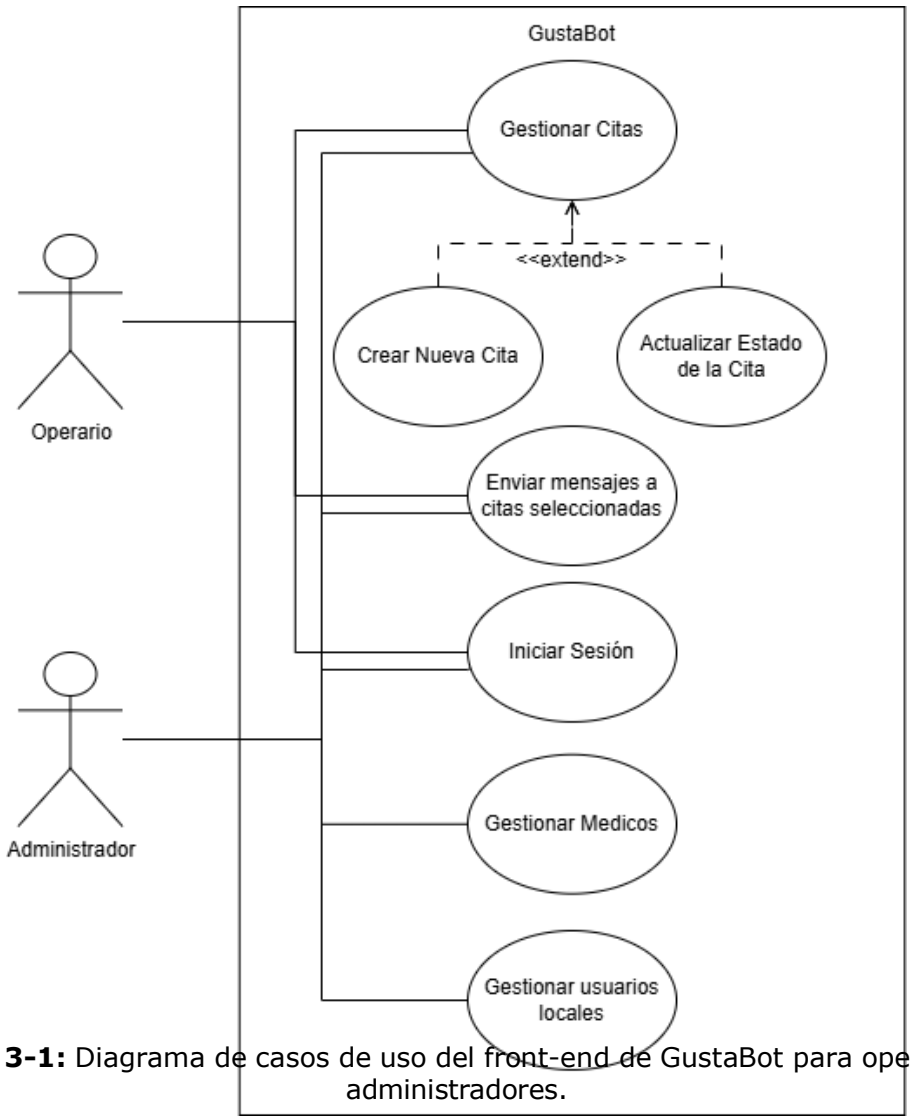


Figura 3-1: Diagrama de casos de uso del front-end de GustaBot para operarios y administradores.

Fuente: Elaboración propia.

Este diagrama permite entender qué funcionalidades ofrece el front-end a los distintos tipos de usuarios del sistema, de forma independiente de la implementación técnica.

3.1.1.3 Arquitectura lógica y módulos principales

A nivel lógico, la aplicación se puede descomponer en varios componentes y paquetes:

- Módulo App (app/)
    - Contiene el punto de entrada (main.jsx), el layout principal (App.jsx), el Router (router.jsx), los guards (guards.jsx), la configuración de React Query (queryClient.js) y el componente ApiStatusGate que verifica el estado de la API antes de mostrar la aplicación.
- Módulos de features (features/)
    - Features/citas: páginas, componentes, hooks y servicios relacionados a citas.
    - Features/medicos: elementos relacionados con la gestión de médicos.
    - Features/usuarios: gestión local de operadores en el navegador.
    - Features/auth: pantalla de login y lógica de autenticación simulada
- Módulos shared (shared/)
    - Shared/components: loader, banners de error, tablas, pills de estado, barras de filtro, etc.
    - Shared/lib: cliente HTTP (apiClient), DTOs (dto.js) constantes (constants.js) y utilidades (fecha, teléfonos).
    - Shared/styles: definición de tokens de diseño (colores, tipografías, etc.).

Hacia el exterior, el front-end se comunica con un componente API Hospitalaria, que expone interfaces REST para citas y médicos. Para las operaciones habituales como listar, crear, actualizar y eliminar el front-end consume endpoints como /api/appointments y /api/doctors.

En el caso del envío de mensajes por WhatsApp, la aplicación no se comunica directamente con la API de Meta, ni recibe respuesta de los mensajes enviados. El front-end solo realiza una petición POST a un endpoint llamado /api/appointments/send-bot enviando el conjunto de identificadores (ID) de citas seleccionadas. A partir de esos IDs, el back-end se encarga de buscar información de las IDs en la base de datos, rellenar la plantilla y enviar el mensaje. Las respuestas de los mensajes enviados actualizan el estado de la cita en la base de datos, por lo que el front-end simplemente vuelve a consultar a la API las citas médicas y refleja los cambios de estados registrados en la base de datos.

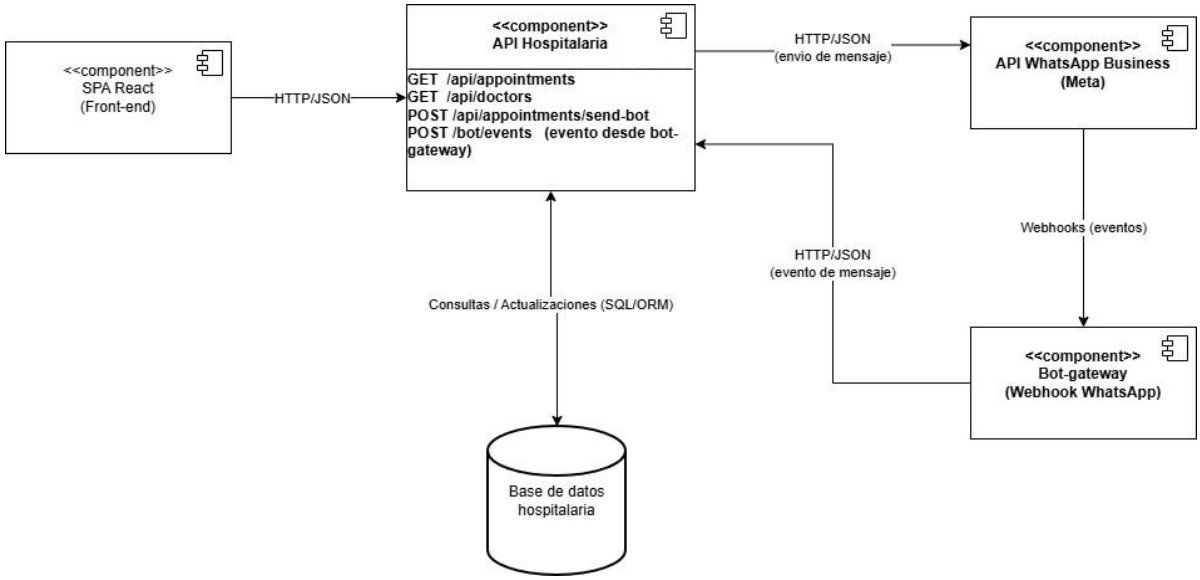


Figura 3-2: Diagrama de componentes de la solución GustaBot y su integración con la API hospitalaria, el Bot Gateway y WhatsApp Business.

Fuente: Elaboración propia.

A nivel de código, esta organización se refleja en la estructura de carpetas del proyecto:

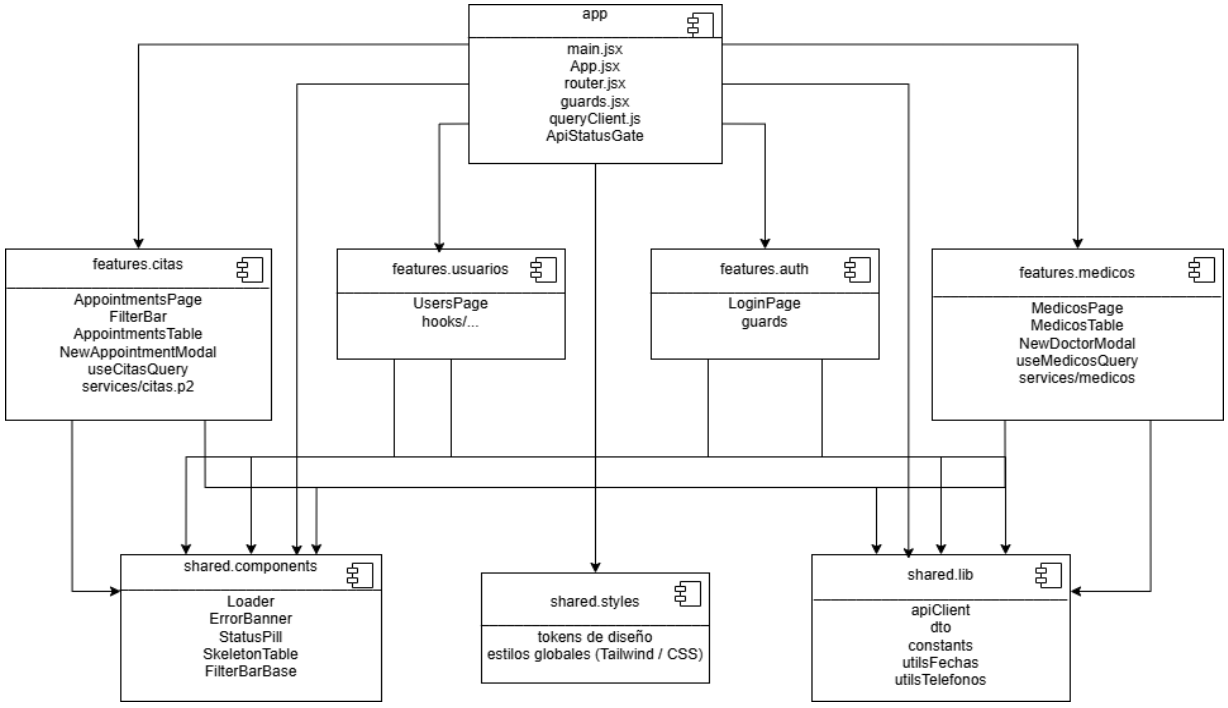


Figura 3-3 : Diagrama de paquetes del front-end de GustaBot que muestra la organización modular del código.

Fuente: Elaboración propia.

En este diagrama de paquetes se aprecia que:

- App depende de los paquetes features.* y shared.*
- Cada features.* depende de shared.components y shared.lib
- Shared.* no depende de ningún módulo de Features, lo que evita ciclos y favorece la reutilización.

3.1.1.4 Vista de componentes React y organización del DOM

Desde la perspectiva del DOM, la aplicación se construye a partir del punto de entrada main.jsx, donde React monta el componente App sobre el elemento . A partir de ahí, la SPA se organiza la interfaz mediante layout y páginas.

En la vista de citas, el árbol de componentes se puede resumir de la siguiente forma:

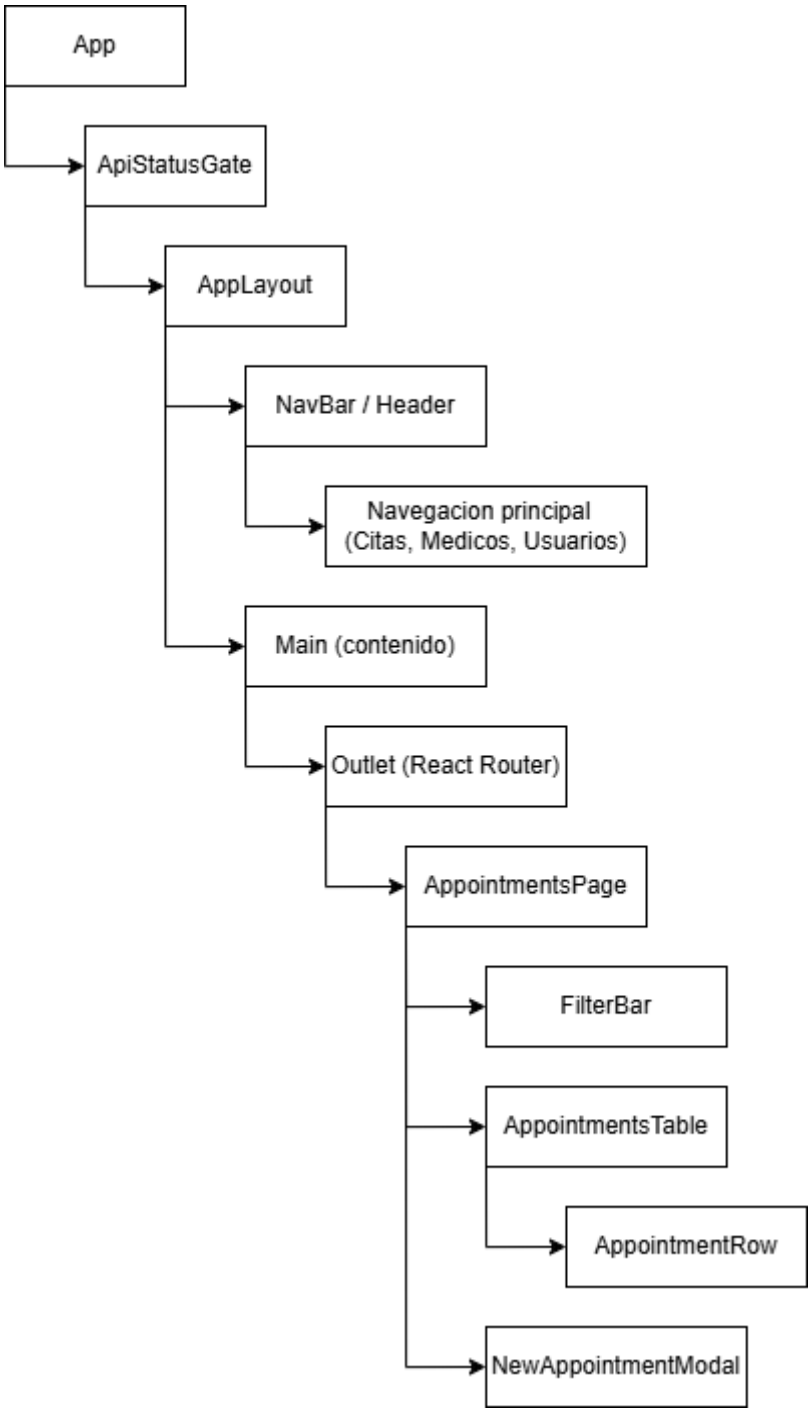


Figura 3-4: Árbol de componentes de la vista de gestión de citas en el front-end de GustaBot.

Fuente: Elaboración propia.

En este diagrama, aunque detalla específicamente la vista de Citas (AppointmentsPage), ilustra el patrón arquitectónico general de la interfaz. El resto de las funcionalidades (médicos y usuarios) siguen una jerarquía similar.

3.1.1.5 Descripción de vistas y componentes React

Para complementar el árbol anterior, a continuación, presentaremos una síntesis de las vistas principales del front-end y de los componentes React que forman parte de la vista. En estos casos el nombre de la clase coincide con el nombre del componente o vista, y su responsabilidad corresponde con lo mostrado en los diagramas UML anteriormente.

AppLayout (layout principal de la aplicación)

- Descripción: componente raíz de la interfaz autenticada. Este componente renderiza la barra superior con el icono de la aplicación, los enlaces de navegación hacia las secciones principales (citas, médicos, usuarios), la información del usuario actual y botón para cerrar sesión. Al centro se encuentra el área de contenido donde se cargan las distintas vistas mediante Outlet (React Router)

AppointmentsPage (vista de gestión de citas)

- Descripción: Vista principal de trabajo del operador. Permite listar las citas, filtrarlas por texto, estado y especialidad, seleccionar múltiples citas, cambiar su estado como confirmada o cancelada, eliminar citas y enviar mensajes a través del back-end. También incluye el formulario para poder crear nuevas citas mediante un modal.

- Componentes principales utilizados:
  - FilterBar: barra de filtros y acciones (barra de búsqueda, selects y botones de "Nueva cita" y "Enviar mensaje")
  - AppointmentsTable: tabla que muestra las citas y permite seleccionar filas.
  - AppointmentRow: componente que se encuentra en AppointmentsTable que se usa para renderizar cada fila de la tabla.
  - NewAppointmentModal: modal con un formulario para poder crear una nueva cita.
  - StatusPill: etiqueta visual que representa el estado actual de la cita.
  - Loader, ErrorBanner, SkeletonTable: componentes de apoyo para estados de carga, error y tabla en modo "esqueleto".

MedicosPage (vista de gestión de médicos)

- Descripción: Vista orientada a la administración del listado de médicos. Esta permite buscar por nombre, crear nuevos médicos y cambiar el estado de los existentes. Los médicos activos son utilizados posteriormente en la creación de citas como opciones disponibles en los formularios.

- Componentes principales utilizados:
  - MedicosTable: tabla que lista los médicos y muestra acciones para habilitar y deshabilitar.
  - NewDoctorModal: modal para crear nuevos médicos.
  - ErrorBanner: componente de feedback en caso de fallas al cargar o actualizar médicos.

UsersPage (vista de usuarios locales)

- Descripción: Vista para gestionar a los operadores locales que se almacenan en el navegador. Permite crear, editar y eliminar usuarios simulados, así como asignarles un rol básico (operario o administrador). Esta vista no se integra con el back-end en esta versión del proyecto, pero sirve para probar la lógica de permisos en el front-end.

LoginPage (pantalla de acceso local)

- Descripción: Pantalla de login que permite iniciar "sesión" local en la aplicación. El operador ingresa con sus credenciales simuladas y una vez autenticado, es redirigido a la vista de citas. El administrador al ingresar con sus credenciales es redirigido a la vista de citas, pero tiene permitido ingresar a la vista de usuarios y médicos, mientras que el operador no puede. Esta pantalla se utiliza principalmente para validar la navegación protegida, aunque no exista un login real contra el back-end.

ApiStatusGate (verificación de disponibilidad de la API)

- Descripción: componente que envuelve la aplicación y realiza comprobaciones periódicas contra la API del back-end. Si la API no está disponible, muestra un mensaje de error global y evita que el usuario interactúe con las vistas que no podrían cargar los datos correctamente.

Componentes compartidos

Además de las vistas anteriores, el proyecto cuenta con una serie de componentes comunes que se reutilizan en distintas secciones para mantener consistencia visual y reducir duplicación:

- FilterBar: barra de filtros, búsqueda y acciones superiores.
- StatusPill: etiqueta visual para mostrar estados de manera consistente.
- Loader: indicador de carga general.
- ErrorBanner: bloque para mostrar errores con opción de reintento.
- SkeletonTable: esqueleto de tabla mostrado mientras cargan los datos reales.

Estos componentes viven en src/shared/components y son consumidos tanto por AppointmentsPage y otras vistas, cumpliendo el rol de "bloques comunes" de la interfaz

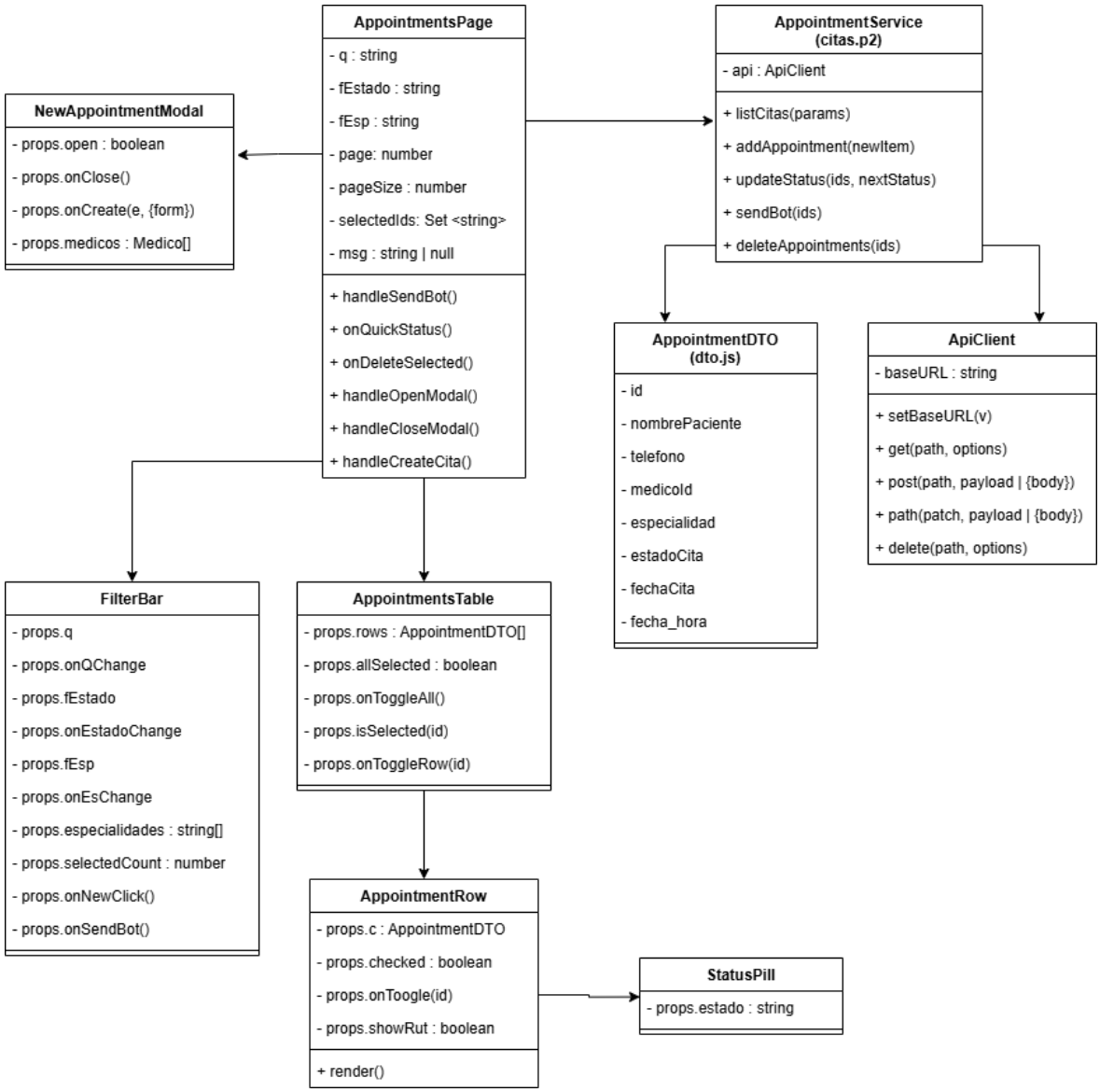


Figura 3-5 : Diagrama de clases y responsabilidades del módulo de gestión de citas en el front-end de GustaBot.

Fuente: Elaboración propia.

Vista de despliegue

Finalmente, la arquitectura física del front-end se puede representar mediante un diagrama de despliegue, en este caso:

- El navegador del operador aloja el artefacto "Aplicación React (SPA)", que se entrega como un conjunto de archivos estáticos generados por Vite.
- El servidor del back-end hospitalario aloja la "API Hospitalaria" que expone los servicios de citas, médicos y enviar mensaje.

- El servidor de base de datos almacena de forma persistente la información clínica relevante y los estados de las citas.

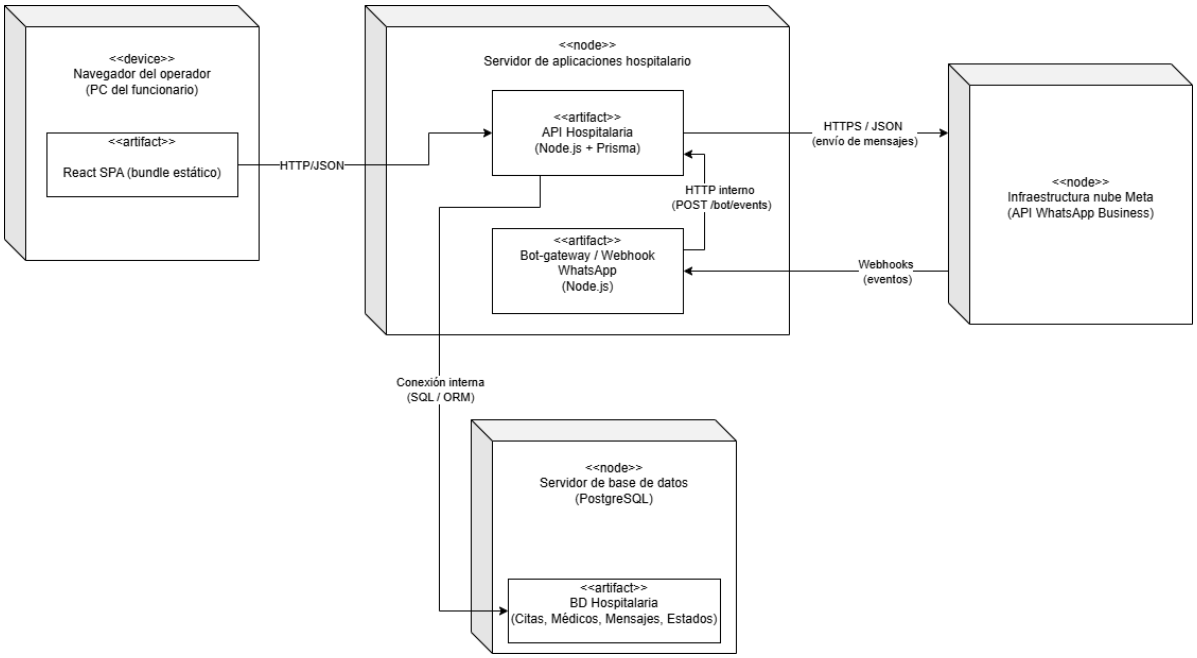


Figura 3-6: Diagrama de despliegue de GustaBot.

Fuente: Elaboración propia

Este diagrama muestra que el front-end se limita a comunicarse con la API hospitalaria a través de HTTP/JSON, y no accede directamente ni a la base de datos ni a la plataforma de mensajería.

3.1.2 Uso de buenas prácticas y patrones de diseño

Durante el desarrollo del front-end se aplicaron diversas buenas prácticas y patrones de diseño con el objetivo de obtener una aplicación modular, mantenible y fácil de extender. Esta sección describe las principales decisiones tomadas desde el punto de vista de arquitectura y organización del código, siempre considerando que la tesina se centra en la capa de presentación desarrollada en React.

3.1.2.1 Arquitectura por Features y separación de responsabilidades

Una de las decisiones más relevantes fue organizar el proyecto siguiendo una arquitectura por Features. En lugar de agrupar el código únicamente por tipo de archivo (todos los componentes, por un lado, todos los servicios por otro), se agruparon los elementos según el dominio funcional:

- Features/citas concentra todo lo relacionado con la gestión de citas.
- Features/medicos agrupa la lógica y la interfaz para el mantenimiento de médicos
- Features/usuarios contiene la gestión local de operadores en el navegador.
- Features/auth se encarga de la pantalla de login y simulación de sesión.

- Shared reúne componentes y utilidades transversales, reutilizadas por todos los módulos

Este enfoque favorece una alta cohesión dentro de cada módulo y un bajo acoplamiento entre los dominios. Por ejemplo, el módulo de citas puede evolucionar (agregar nuevos filtros, estado de citas, etc.) sin necesidad de modificar directamente el código del módulo de médicos, siempre que se respeten las interfaces definidas en shared.

A nivel interno, se aplica además una separación clara entre:

- Componentes contenedores como AppointmentsPage o MedicosPage, que:
  - Coordinan hooks de datos
  - Gestionan el estado de filtros y selección
  - Deciden qué componente presentar en una función del estado (cargando, error, vacío, etc.)

- Componentes de presentación como AppointmentsTable, AppointmentRow, StatusPill, Loader o FilterBar, que:
  - Reciben datos por props.
  - Se concentran en cómo mostrar la información.
  - Evitan contener lógica del negocio compleja.

De esta forma, los componentes de presentación son más fáciles de reutilizar y probar, mientras que los contenedores encapsulan la lógica específica de cada vista.

3.1.2.2 Uso de hooks y React Query para la capa de datos

Para la gestión de datos remotos se utilizó la librería React Query (@tanstack/react-query), sobre la cual se construyeron hooks personalizados por dominio, como, por ejemplo:

- useCitasList, useCreateCita, useUpdateEstadoCita en el módulo citas.
- useMedicosList, useCreateDoctor, useUpdateDoctor en el módulo médicos.

Estos hooks envuelven las llamadas a los servicios (citas.p2.js, médicos.js) y exponen un contrato homogéneo a los componentes:

- data: datos ya normalizados mediante DTOs.
- isLoading: indica si la información está en proceso de carga.
- Error: representa errores de red o del back-end.
- Operaciones como mutate o refetch para ejecutar acciones.

El uso de React Query aporta varias ventajas al proyecto como:

- Manejo automático de caché de datos, evitando solicitudes repetidas innecesarias.
- Posibilidad de revalidar datos en segundo plano, manteniendo la información actualizada.
- Control detallado de estados de carga, error, éxito, lo que facilita mostrar feedback claro al usuario.
- Invalidación selectiva de queries (por ejemplo, al crear o modificar una cita se invalida el listado de citas, manteniendo la coherencia entre operaciones).

De esta forma, la lógica de acceso a datos queda concentrada en los hooks y servicios reutilizables, mientras que los componentes se limitan a solo consumir el estado expuesto por estos hooks.

3.1.2.3 DTOs y cliente HTTP como capa de adaptación

Entre el front-end y la API se definió una capa de adaptación compuesta por:

- El cliente HTTP apiClient.js
- Las funciones de mapeo definidas en dto.js

El cliente HTTP apiClient centraliza la forma en que se realizan las peticiones:

- Define una URL base configurable mediante la variable de entorno VITE_API_BASE_URL
- Estandariza el manejo de respuestas y errores
- Proporciona funciones genéricas (GET, POST, PATCH, DELETE) sobre las que se construyen los servicios de cada módulo.

Por su parte, las funciones DTO se encargan de:

- Normalizar nombres de campos que vienen desde la API (por ejemplo, los distintos formatos de nombre de paciente)
- Convertir fechas desde formatos ISO a formatos adecuados para los componentes de entrada y visualización
- Mapear estados internos de la cita a valores consistentes que la interfaz pueda entender.

Gracias a esta capa, el resto de la aplicación no depende directamente de los detalles internos del back-end. Si la API cambia ligeramente la estructura de los datos, basta con ajustar los DTOs, sin necesidad de modificar todos los componentes.

3.1.2.4 Manejo explícito de estados de carga, error y vacío

Otra buena práctica aplicada en el proyecto fue el manejo explícito de los estados loading, error y empty en las vistas de listado. En lugar de dejar la interfaz "en blanco" mientras se cargan los datos o ante un fallo, se incorporaron componentes reutilizables:

- Loader y SkeletonTable para indicar visualmente que la información se está cargando.
- ErrorBanner para mostrar mensajes claros cuando ocurre un error en la comunicación con la API, con la opción de reintentar la operación.
- Mensaje y layout específicos para el caso en que no existan registros (por ejemplo, ninguna cita para el filtro aplicado).

Esto mejora la experiencia de usuario y facilita el diagnóstico de problemas durante las pruebas.

3.1.2.5 Buenas prácticas específicas para el contexto del proyecto

Existen además algunas decisiones particulares que se consideran buenas prácticas dadas las restricciones del contexto:

- Autenticación simulada y usuarios locales

  Por decisión del Hospital Gustavo Fricke, en esta versión del sistema no se implementó un login real en el back-end. Como solución intermedia, el módulo de autenticación y usuarios se modeló como un stub local:

  - La sesión se almacena en localStorage
  - Los guards (RequireAuth, RequireAdmin) restringen las vistas en la SPA
  - La gestión de operadores se realiza solo en el navegador

  Esto permite probar flujos de navegación con distintos tipos de usuario sin introducir aún la complejidad de integrar un sistema de identidad institucional.

- Uso de variables de entorno para configurar la API

  La URL base del back-end se define en la variable VITE_API_BASE_URL, lo que permite cambiar de entorno (desarrollo, pruebas, integración) sin editar el código fuente. Esta práctica simplifica la configuración y reduce errores al desplegar.

- Consistencia visual mediante componentes compartidos y Tailwind CSS

  Elementos comunes como botones, filtros, tablas, etiquetas de estado y mensajes se basan en componentes reutilizables combinados con clases utilitarias de Tailwind CSS, esto nos permite:

  - Reducir la duplicación de estilos.
  - Acelerar el desarrollo de nuevas vistas.
  - Promover una apariencia coherente a lo largo de la aplicación.

- Control de versiones y estilo de código

  Para el control de versiones del front-end se utilizó Git. Durante gran parte del desarrollo cada integrante trabajó sobre su propia rama o ramas asociadas al sprint, realizando push con mensajes que indicaban las funcionalidades o correcciones incorporadas en cada iteración.

  Si bien el flujo ideal de ramas de características (feature branches) y fusiones ordenadas hacia la rama principal no siempre se cumplió estrictamente, especialmente en etapas finales, donde se generaron algunos conflictos y ajustes de emergencia el uso de Git permitió:

  - Mantener un historial de cambios asociados a cada sprint.
  - Recuperar versiones anteriores del proyecto cuando fue necesario.
  - Coordinar el trabajo en paralelo de los integrantes del equipo.

En conjunto, estas buenas prácticas y patrones de diseño permiten que el front-end sea más robusto ante cambios en los requisitos, más sencillo de mantener y más fácil de extender en futuras versiones del sistema, manteniendo siempre una separación clara entre la lógica de presentación, la lógica de datos y la integración con la API hospitalaria.

3.1.3 Integración con el sistema general

La integración del front-end con el resto del sistema se realiza exclusivamente a través de la API del proyecto hospitalario, utilizando peticiones HTTP y mensajes en formato JSON. El front-end no accede directamente a la base de datos ni a la API de WhatsApp Business de Meta, todas esas responsabilidades se concentran en el back-end.

En particular el front-end consume:

Módulo	Método	Ruta	Descripción
Citas	GET	/api/appointments	Listar citas con filtros y paginación.
Citas	POST	/api/appointments	Crear una nueva cita a partir del formulario del front-end.
Citas	PATCH	/api/appointments	Actualizar el estado de un conjunto de citas (operación bulk).
Citas	PATCH	/api/appointments/{id}	Actualizar el estado de una cita específica cuando no hay endpoint bulk.
Citas	DELETE	/api/appointments/{id}	Eliminar una cita seleccionada desde la interfaz.
Citas	POST	/api/appointments/send-bot	Enviar al back-end los IDs de citas seleccionadas para disparar mensajes.
Médicos	GET	/api/doctors	Listar médicos con filtros básicos y estado (habilitado/deshabilitado).
Médicos	POST	/api/doctors	Crear un nuevo médico en el catálogo.
Médicos	PATCH	/api/doctors/{id}	Actualizar el estado de un médico.
Médicos	DELETE	/api/doctors/{id}	Deshabilitar un médico del catálogo.

La URL base del back-end se configura mediante una variable de entorno (VITE_API_BASE_URL) y se utiliza en el cliente HTTP (apiClient) para construir todas las llamadas desde los distintos módulos del front-end.

3.1.3.1 Llamadas principales del módulo citas

1. Listar Citas

- Método y ruta: GET /api/appointments
- Parámetros (Query) pueden incluir:
    - Search: texto libre (nombre, RUT, teléfono)
    - Estado: estado de la cita
    - MedicoID: identificador del médico
    - from, to: rango de fechas
    - parámetros de paginación (page, pageSize,etc.)

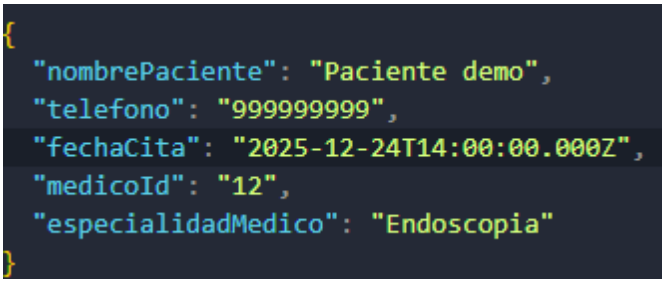
- Respuesta esperada (éxito):

```
{
  "data": [
    {
      "id": "34",
      "nombrePaciente": "Paciente demo",
      "rut": "11111111-1",
      "telefono": "999999999",
      "fechaCita": "2025-11-24T14:00:00.000Z",
      "medicoId": "12",
      "nombreMedico": "Carlos",
      "especialidadMedico": "Endoscopia",
      "estadoCita": "pendiente",
      "createdAt": "2025-11-23T03:32:31.955Z",
      "updatedAt": "2025-11-23T03:32:31.955Z"
    },
```


- Respuesta esperada (error):
  Código 4xx o 5xx con un JSON que incluye un mensaje de error, el front-end muestra un banner de error y permite reintentar la carga.

2. Crear una nueva cita

- Método y ruta: POST /api/appointments
- Cuerpo (request) enviado por el front-end:



```
{
  "nombrePaciente": "Paciente demo",
  "telefono": "999999999",
  "fechaCita": "2025-12-24T14:00:00.000Z",
  "medicoId": "12",
  "especialidadMedico": "Endoscopia"
}
```


- Respuesta esperada (éxito)
  Código 201 Created con la cita creada:

```
{
  "id": "35",
  "nombrePaciente": "Paciente demo",
  "telefono": "999999999",
  "fechaCita": "2025-12-24T14:00:00.000Z",
  "medicoId": "12",
  "nombreMedico": "Carlos",
  "especialidadMedico": "Endoscopia",
  "estadoCita": "pendiente",
  "createdAt": "2025-11-23T18:38:44.922Z",
  "updatedAt": "2025-11-23T18:38:44.922Z"
}
```


Tras recibir esta respuesta, el front-end invalida el listado de citas y lo vuelve a consultar para reflejar la nueva cita en la tabla.

3. Actualizar estado de una o varias citas

- Método y ruta: PATCH /api/appointments
- Cuerpo (request):


```
{
  "estadoCita": "cancelada"
}
```


- Respuesta esperada (Éxito):


```
{
  "id": "35",
  "nombrePaciente": "Paciente demo",
  "telefono": "999999999",
  "fechaCita": "2025-12-24T14:00:00.000Z",
  "medicoId": "12",
  "nombreMedico": "Carlos",
  "especialidadMedico": "Endoscopia",
  "estadoCita": "cancelada",
  "createdAt": "2025-11-23T18:38:44.922Z",
  "updatedAt": "2025-11-23T18:43:39.078Z"
}
```


El front-end puede mostrar un mensaje (“3 citas actualizadas”) y actualizar el listado de citas.

4. Eliminar una cita

- Método y ruta: DELETE /api/appointments/{id}
- Respuesta esperada (éxito):
  Código 204 No Content (sin cuerpo)
  El front-end retira la cita correspondiente de la tabla

3.1.3.2 Flujo de envío de mensajes por WhatsApp

Para el envío de mensajes de confirmación/cancelación a través de WhatsApp, la integración también se hace mediante HTTP y JSON, pero con una responsabilidad muy acotada en el front-end.

5. Disparar el envío de mensajes

- Método y ruta: POST /api/appointments/send-bot
- Cuerpo (request) enviado por el front-end
  Es decir, el front-end solo envía los identificadores de las citas seleccionadas en la tabla.
- Respuesta esperada (éxito)
  Un código 200 OK o 202 Accepted indicando que la solicitud fue recibida. El cuerpo puede incluir un pequeño resumen de cuántas citas se procesaron, pero desde la perspectiva del front-end lo importante es saber si la operación fue aceptada o no.
- A partir de aquí el back-end se encarga con las IDs proporcionadas de buscar los datos de las citas en la base de datos, invocar la API de WhatsApp Business de Meta para enviar los mensajes, recibir los webhooks de respuesta y actualizar el estado de las citas en la base de datos.
- El front-end no recibe directamente las respuestas de WhatsApp lo único que hace es limpiar la selección de citas después de un envío exitoso y volver a consultar el GET /api/appointments para ver los nuevos estados de las citas, ya modificados por el back-end.

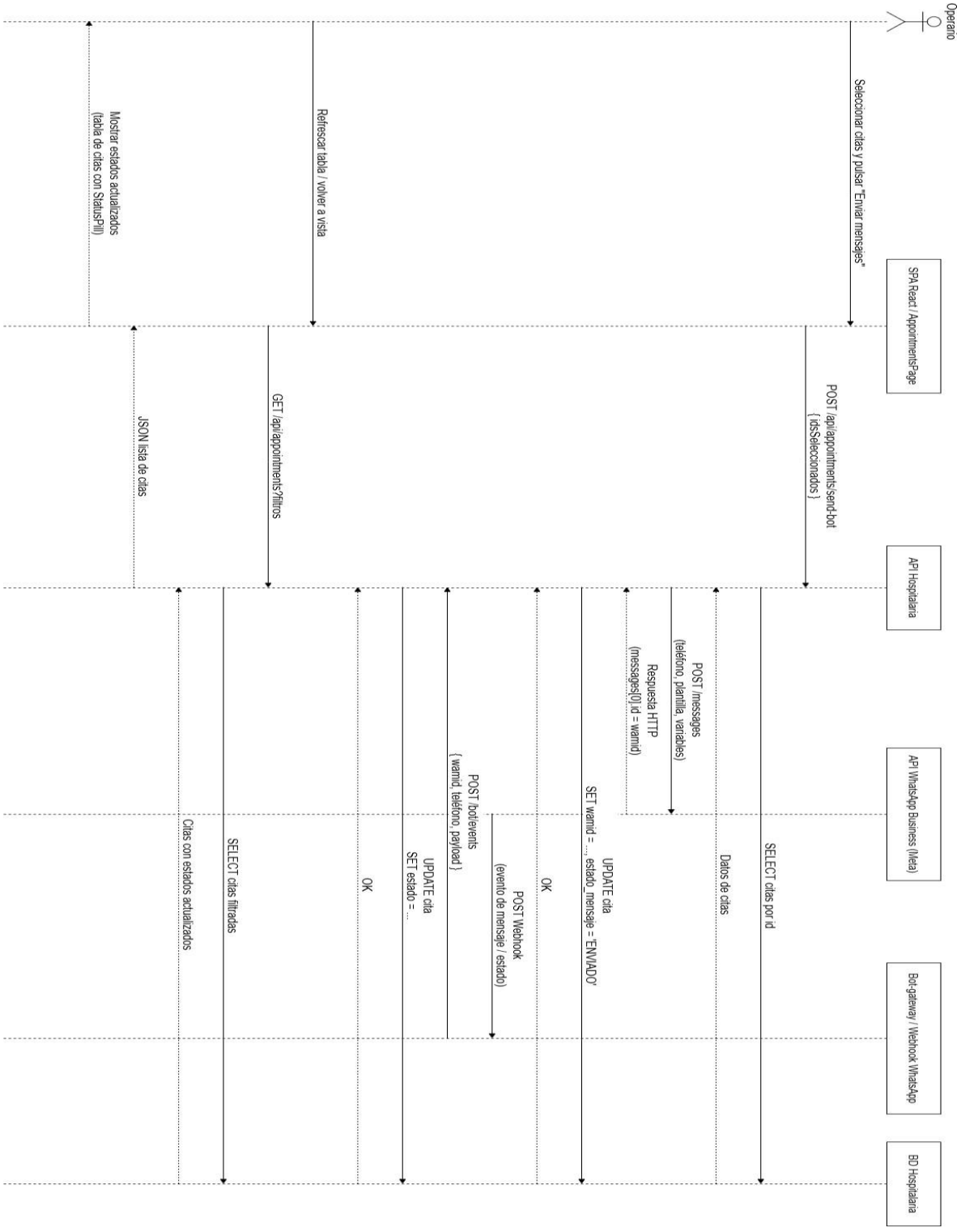


Figura 3-7: Diagrama de secuencia del flujo de envío y actualización de mensajes de confirmación de citas entre el front-end, la API hospitalaria, el Bot Gateway y WhatsApp.

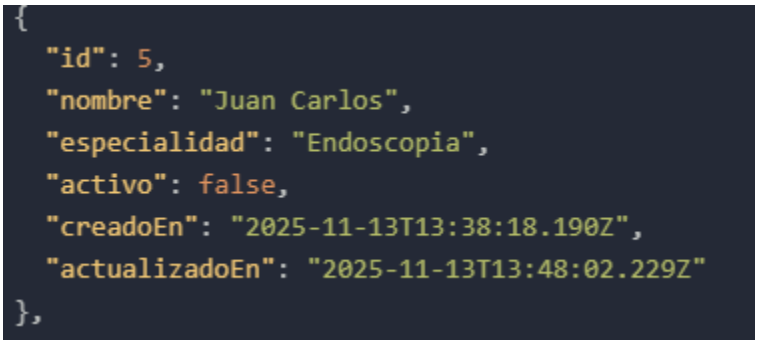
Fuente: Elaboración propia.

3.1.3.3 Llamadas principales del módulo de médicos

El módulo de médicos (features/medicos) se integra con la API a través de endpoints del tipo /api/doctors, también por HTTP y JSON.

6. Listar médicos

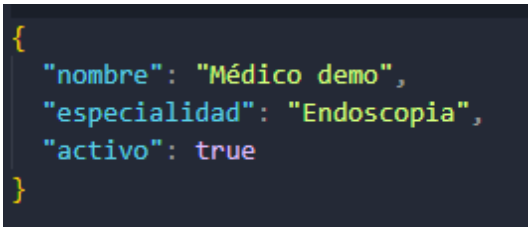
- Método y ruta: GET /api/doctors
- Parámetros (Query) típicos:
  - Search: texto para buscar por nombre
  - Parámetros de paginación o filtros de estado (activo), según la configuración del back-end.
- Respuesta esperada (éxito)



- El front-end usa esta lista para el administrador de médicos (MedicosPage) y el selector de médicos al momento de crear una cita (solo médicos activos).

7. Crear o actualizar médicos

- Método y ruta: POST /api/doctors y PATCH /api/doctors/{id}
- Cuerpo (request) enviado por el front-end para crear un médico:



- Respuesta esperada (éxito): 201 Created con el médico creado en JSON


```
{
  "id": 14,
  "nombre": "Médico demo",
  "especialidad": "Endoscopia",
  "activo": true,
  "creadoEn": "2025-11-23T18:50:38.558Z",
  "actualizadoEn": "2025-11-23T18:50:38.558Z"
}
```


- Cuerpo (request) enviado por el front-end para modificar estado de un médico:


```
{
  "activo": false
}
```


- Respuesta esperada (éxito): 200 OK o 204 No Content.
- El front-end vuelve a consultar la lista de médicos y refleja el cambio.

3.1.3.4 Alcance de la integración de usuarios y autenticación

En esta versión del sistema, por decisión del Hospital Gustavo Fricke, no existe integración HTTP para usuarios y autenticación:

- No se llama a un endpoint de login
- No se consumen endpoints de /api/users ni similares

El módulo de usuarios y autenticación en el front-end funciona como un stub local:

- La información de la "sesión" y los operadores se guarda en localStorage
- Los guards de React Router (RequireAuth y RequireAdmin) usan estos datos para decidir qué vistas mostrar
- No hay intercambio de credenciales ni sesiones con el back-end

En resumen, el front-end se integra con el sistema general a través de un pequeño conjunto y bien definido de endpoints HTTP con respuestas JSON para:

- Gestionar citas
- Gestionar médicos
- Disparar el proceso de envío de mensajes a través de la API de WhatsApp Business de Meta, dejando toda la lógica de negocio y mensajería avanzada al lado del servidor

3.2 Diseño de la interfaz

En esta sección se presenta la interfaz visual de usuario del front-end desde una perspectiva principalmente visual, mostrando cómo se materializan en pantalla los

componentes descritos en la sección 3.1. El diseño UI/UX detallado fue realizado por otro integrante del equipo, en esta sección se documenta cómo quedaron implementadas las vistas principales y componentes en React.

El foco está en la vista de gestión de citas y en la vista de gestión de médicos, que corresponden a los flujos más relevantes para los operadores del Hospital Gustavo Fricke.

3.2.1 Layout general y navegación

Toda la aplicación se construye a partir del componente AppLayout (definido en src/app/App.jsx), que actúa como layout principal. Su estructura visual es:

Barra superior fija implementada con un <header> que contiene:

- El icono de la aplicación a la izquierda
- Un conjunto de botones de navegación al centro, basados en el componente NavItem (Citas, Médicos, Usuarios)
- A la derecha la información del usuario actual y el botón de cierre de sesión.

El área de contenido se encuentra debajo de la barra superior, el layout renderiza el <Outlet /> de React Router. En este espacio se muestran las páginas:

- AppointmentsPage (citas)
- MedicosPage (médicos)
- UsersPage (usuarios)
- LoginPage cuando no hay sesión simulada

3.2.2 Vista de gestión de citas

La vista de citas está implementada en src/features/citas/pages/AppointmentsPage.jsx y es la pantalla principal para el operador.

En la Figura 3-8 se muestra la interfaz de la vista de gestión de citas, con la barra de filtros, la tabla de citas y las etiquetas de estado.

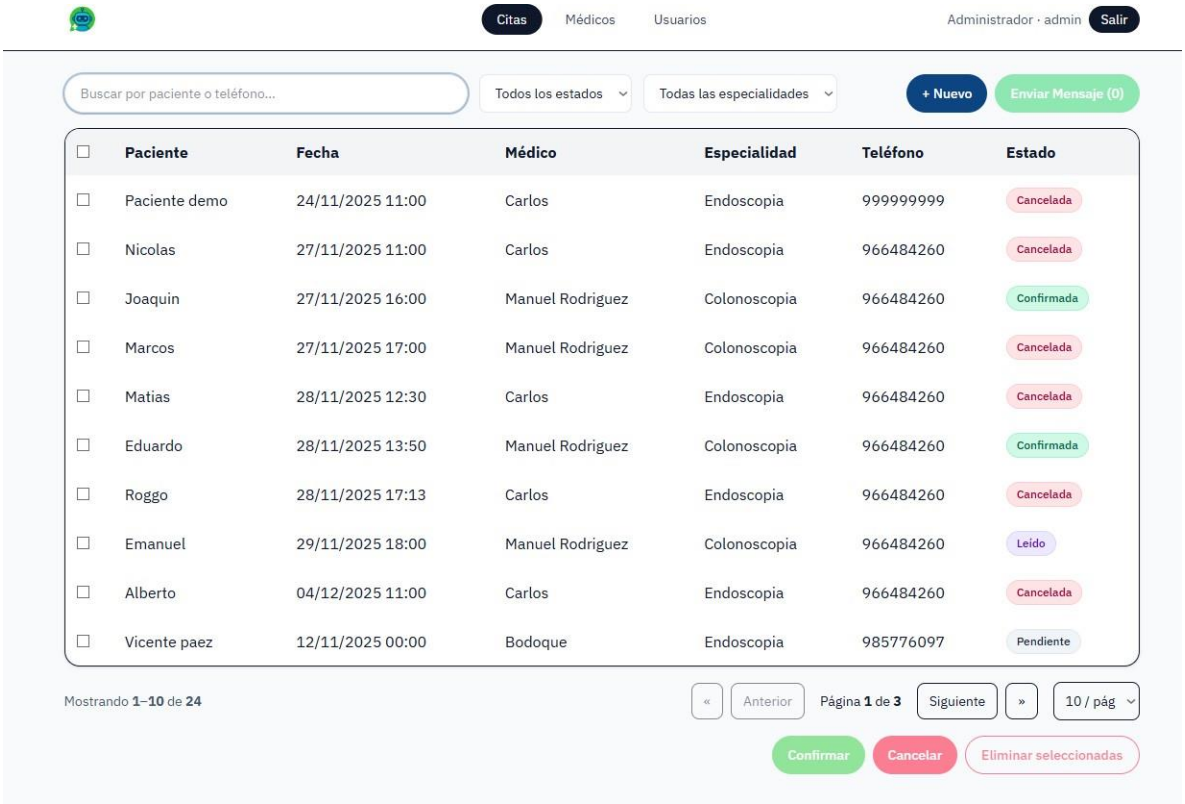


Figura 3-8: Vista de gestión de citas en el front-end de GustaBot, con barra de filtros, tabla de citas y etiquetas de estado.

Fuente: Elaboración propia.

Visualmente se compone de:

Barra de filtros y acciones implementada con el componente FilterBar, en esta barra se encuentran:

- El campo de búsqueda por texto (nombre, RUT, Teléfono)
- Filtro por especialidad
- Botón para "Nueva cita"
- Botón para "Enviar mensaje" a las citas seleccionadas

Tabla de citas implementada por el componente AppointmentsTable (TablaCita.jsx), que a su vez utiliza AppointmentRow (FilaCita.jsx) para representar cada fila mostrando:

- Nombre del paciente
- Teléfono
- Fecha y hora de la cita
- Médico
- Especialidad
- Estado de la cita (con StatusPill)
- Checkbox de selección para operaciones masivas

Modal de nueva cita implementado en NuevaCitaModal.jsx, se superpone a la vista para permitir crear una cita sin abandonar la pantalla principal.

3.2.3 Vista de gestión de médicos

La vista de médicos se encuentra en src/features/medicos/pages/MedicosPage.jsx y está orientada a la administración del catálogo de los profesionales

La Figura 3-9 ilustra la vista de administración de médicos, con el buscador, el listado y las acciones de habilitar y deshabilitar.

Figura 3-9: Vista de gestión de médicos en el front-end de GustaBot, con buscador, listado de profesionales y acciones de habilitar o deshabilitar.

Fuente: Elaboración propia.

Controles y acciones en el área superior con:

- Campo de búsqueda por nombre
- Botón para registrar un nuevo médico

Tabla de médicos implementada mediante el componente MedicosTable (MedicosTable.jsx) que muestra:

- Nombre del médico
- Especialidad
- Botones para habilitar o inhabilitar actividad

Modal de médico NewDoctorModal.jsx se utiliza para crear nuevos médicos.

3.2.4 Pantalla de login y usuarios locales

El módulo de autenticación y usuarios locales se representa en:

- LoginPage (src/features/auth/pages/LoginPage.jsx)
- UsersPage (src/features/usuarios/pages/UsersPage.jsx)

Aunque en esta versión no existe un back-end real de autenticación, las pantallas permiten simular el inicio de sesión de un operador o administrador y gestionar usuarios locales almacenados en el navegador.

La interfaz de LoginPage se compone de un formulario sencillo:

- Campo para ingresar usuario y contraseña
- Botón para iniciar sesión
- Botón de auto rellenado para ingresar más rápido (Demo)

La interfaz de UsersPage se compone de una tabla con los usuarios almacenados y su estado actual

3.2.5 Resumen visual

En resumen, la interfaz del front-end se organiza en torno a:

- Un layout principal (AppLayout) con barra superior y navegación por secciones
- Vistas específicas (AppointmentsPage, MedicosPage, UsersPage, LoginPage)
- Componentes clave como FilterBar, AppointmentsTable, MedicosTable, NuevaCitaModal, NewDoctorModal que estructuran el trabajo cotidiano del operador.

Las figuras incluidas en esta sección permiten apreciar cómo los componentes descritos anteriormente se concretan en pantallas reales facilitando la comprensión de la experiencia de usuario del sistema.

3.3 Detalles de la codificación y desarrollo

Esta sección describe cómo se organizó el trabajo de desarrollo front-end a lo largo de distintas iteraciones del proyecto, así como el entorno de trabajo, el uso de herramientas y el flujo de control de versiones.

3.3.1 Organización del backlog e iteraciones

El proyecto se organizó en varias iteraciones (Sprints), cada una con un conjunto de objetivos específicos. Desde el punto de vista del front-end, la evolución puede resumirse en cuatro hitos principales.

Iteración 1: vista general de citas

En la primera iteración, el foco del front-end fue construir una vista de administración de pacientes por operario del sistema, que permitiera:

- Visualizar las citas en una tabla central
- Aplicar filtros mediante un "collapse de filtro"
- Marcar pacientes para seleccionar la acción del operario (confirmar, cancelar, enviar mensaje)
- Cambiar el estado de una cita de pendiente a confirmada o cancelada
- Contar con un botón para agregar citas

Esta vista inicial constituyó el MVP del front-end, ya que demostraba el flujo básico de trabajo del operador sobre las citas en una única vista.

Iteración 2: médicos, login, usuarios y conexión con el back-end

En la segunda iteración, el esfuerzo fue orientado a las nuevas vistas necesarias para el sistema que consisten en:

- Vista de médicos (listado y mantenimiento básico de los médicos)
- Vista de login
- Vista de usuarios, donde se simularían distintos operadores dentro del SPA

Al mismo tiempo, la vista de citas dejó de trabajar con datos totalmente "mock" y comenzó a integrar datos reales a través de la API construida en esta iteración, permitiendo que la tabla de citas mostrara información persistida en la base de datos y que el flujo general del sistema pudiese ser demostrado de principio a fin.

Iteración 3: nuevo apartado visual y reorganización del front-end

En esta iteración, el equipo adaptó el trabajo de UI/UX y se redefinió el aspecto visual del sistema. Las tareas asignadas para el desarrollo front-end fueron:

- Implementación del nuevo apartado visual.
- Modularización y reorganización de componentes.

Esto supuso un cambio importante a la "cara" de la aplicación, manteniendo la lógica funcional, pero mejorando su claridad y consistencia visual.

Iteración 4:

En la etapa previa a la presentación en la Feria de Software, el trabajo de front-end se concentró en:

- Incorporar nuevos estados en las "pills" de las citas (enviado, recibido, leído), que nos permiten entregarle al operario más información del estado del mensaje.
- Ajustar estilos y detalles visuales para mejorar la apariencia general del sistema.
- Alinear la interfaz con el flujo completo de mensajería implementado en el back-end.

Esta última iteración no introdujo grandes cambios estructurales en la arquitectura, pero sí aportó un nivel adicional de pulido visual y coherencia entre el estado interno de la cita y lo que observa el operador en la pantalla.

3.3.2 Entorno y flujo de trabajo de desarrollo

El desarrollo del front-end se realizó utilizando un conjunto de herramientas estándar del ecosistema JavaScript:

- React como biblioteca principal para la construcción de la interfaz basada en componentes
- Vite como la herramienta de building y servidor de desarrollo, lo que permitió tiempos de recarga rápidos durante la programación.
- React Router para la navegación interna de la SPA entre las vistas de citas, médicos, usuarios, y login.

- React Query para la obtención y caché de datos remotos, facilitando la integración con la API descrita anteriormente.
- Tailwind CSS (combinado con estilos propios) para la definición de layout y estilos

El flujo de trabajo típico era:

1. Levantar el entorno en modo desarrollo
2. Tomar una tarea del sprint
3. Implementar las modificaciones en los módulos correspondientes
4. Probar manualmente el flujo completo en el navegador, generalmente con el back-end.
5. Ajustar mensajes, estilos y pequeños detalles según el feedback del equipo o el encargado de UI/UX
6. Registrar los cambios en Git y subirlos a la rama correspondiente.

Este ciclo se repitió a lo largo de las iteraciones, permitiendo avanzar de forma incremental sobre una base ya funcional.

3.3.3 Control de versiones y gestión de tareas

Para el control de versiones se utilizó Git y para la gestión de tareas y retrospectivas se apoyaron en herramientas como Jira y Confluence, coordinadas por el Scrum Master del equipo.

En cuanto al repositorio del código:

- Se mantuvo una rama principal asociada a la versión estable del proyecto
- Se trabajó con ramas asociadas a sprint y/o funcionalidades específicas, donde cada integrante realizaba sus commits y pushes.

En la práctica:

- Durante las primeras iteraciones, el uso de ramas permitió aislar el desarrollo de la vista de citas y luego de las vistas de médicos, login y usuarios
- En las fases finales, la integración de cambios de todos los integrantes generó algunos conflictos y ajustes de emergencia, lo que hizo que el flujo de ramas no fuese siempre ideal.

Aún así, el uso de Git resultó fundamental para:

- Mantener un historial de cambios asociado a cada iteración
- Recuperar versiones anteriores cuando fue necesario
- Coordinar el trabajo paralelo entre front-end, back-end y Bot.

Desde el punto de vista de estilo de código, se procuró mantener:

- Nombres coherentes de componentes y archivos
- La organización por Features y shared.
- El uso consistente de hooks y servicios para acceder a la API.

No se llegó a definir una configuración formal de linting y formateo automatizado (ESlint, Prettier) ni un conjunto de pruebas automatizadas aplicadas por pipeline, lo que se identifica como una mejora prioritaria para futuras evoluciones del proyecto.

3.3.4 Construcción y despliegue del front-end

El proceso de construcción del front-end se apoya en las capacidades de Vite:

- Compilación para producción:
  Se realiza mediante el comando de construcción definido en el package.json. Este proceso genera una carpeta con los artefactos estáticos optimizados (HTML, CSS y JavaScript optimizado) listos para ser servidos por un servidor Web o integrarse con el back-end del proyecto.

- Despliegue en entornos de prueba y demostración:
  En el contexto de este trabajo, el despliegue se realizó de forma manual, sirviendo los archivos generados por Vite junto al back-end para poder probar el sistema de extremo a extremo y mostrarlo en instancias como la Feria de Software.
  No se implementó un pipeline formal de integración y despliegue continuo (CI/CD) exclusiva para el front-end; las nuevas versiones se construían y desplegaban bajo supervisión del equipo de desarrollo.

En conjunto, el desarrollo del front-end combinó iteraciones centradas en funcionalidades como las vistas, iteraciones orientadas a la integración y refinamiento visual, y un flujo de trabajo apoyado en herramientas modernas de JavaScript, Git y plataformas de gestión de proyectos. A pesar de las limitaciones y ajustes sobre la marcha, se consiguió una versión funcional visualmente coherente, alineada con los requerimientos planteados por el Hospital Gustavo Fricke.

3.4 Pruebas y validación

La validación del front-end de GustaBot se realizó de forma iterativa a lo largo de los sprints del proyecto, priorizando que los flujos principales funcionaran de extremo a extremo en conjunto con la API del proyecto. Se aplicaron pruebas funcionales manuales sobre la propia interfaz y el entorno de integración con el back-end.

3.4.1 Estrategias de testing aplicadas a los componentes

Para comprobar el buen funcionamiento de GustaBot, en cada iteración se probó directamente en el navegador verificando que:

- El listado de citas se cargó correctamente y permitió filtrar.
- Se pudieron crear y actualizar estado de las citas, observando el cambio en la tabla y píldoras de estado.
- La acción de "enviar mensaje" solo se habilitó con citas seleccionadas y completó el flujo sin errores visibles.
- Las vistas de médicos, usuarios y login permitieron realizar las operaciones básicas definidas (listar, crear, editar, etc).

Estas pruebas se apoyaron con las herramientas de desarrollo del navegador (consola y pestaña de Network) y revisiones con el integrante encargado de UI/UX, para ajustar detalles de textos, botones y distribución de los elementos.

3.4.2 Resolución de problemas y limitaciones

Cuando se detectaban incidentes (por ejemplo, selecciones no se limpiaban luego de enviar un mensaje, estados no se actualizaban correctamente, etc.) se revisaba el manejo de estado de componentes, se corregía el código afectado y se repetía el flujo completo hasta confirmar el resultado esperado. Una de las estrategias más importantes al momento de encontrar errores fue tener el código documentado y la organización por Features del proyecto, ya que nos permitía agilizar la búsqueda del origen del problema y su resolución.

A pesar de este trabajo iterativo, el proyecto no cuenta aún con un conjunto de pruebas automatizadas ni con pruebas de usabilidad formales con personal del hospital, por lo que se propone como trabajo futuro complementar esta validación manual con test unitarios, end-to-end y sesiones estructuradas con usuarios finales.

4 Conclusiones

El desarrollo del front-end de GustaBot permitió materializar en una interfaz web el flujo de gestión y confirmación de citas médicas definido para el Hospital Gustavo Fricke. A través de una SPA de React se implementaron las vistas principales para administrar citas, médicos y usuarios, permitiendo también enviar mensajes a los pacientes. De esta manera se cumplió la mayor parte de las funcionalidades planificadas para esta versión.

El front-end tuvo un rol clave dentro del proyecto, ya que fue la parte visible para los interesados y la forma más concreta de cómo funcionaría el sistema en la operación diaria. Entre los aportes técnicos concretos destacan la capa de adaptación compuesta por apiClient.js y dto.js que desacopla la interfaz del back-end, el componente ApiStatusGate para proteger la navegación ante fallos de conectividad, y el uso de React Query para el manejo de estado remoto con caché e invalidación selectiva. A pesar de los problemas de coordinación y tiempos con el back-end, se logró integrar la interfaz con la API del proyecto y demostrar el flujo completo de trabajo desde la perspectiva de la persona operaria.

Este trabajo significó el primer acercamiento del autor a React como tecnología, pasando de no haber trabajado antes con este ecosistema a construir una SPA completa conectada a un back-end real. En el proceso aprendió sobre componentes, manejo de estado, consumo de APIs desde el navegador y organización del código por Features, además de poner en práctica el flujo de trabajo colaborativo con otros roles del proyecto como back-end y UI/UX.

Como trabajo futuro se propone complementar la validación manual con pruebas automatizadas unitarias con Vitest y end-to-end con Playwright, realizar pruebas de usabilidad formales con el personal del hospital, ampliar las funcionalidades disponibles en la interfaz (como, por ejemplo, más filtros, reportes, métricas) y su posible implementación en el hospital. Dado que el front-end se integra mediante APIs, también existe la posibilidad de adaptar esta solución a otros centros de salud u organizaciones con necesidades similares, ajustando solo los contratos del back-end y reglas del negocio.

5 Bibliografía

[1] P. Diebold, A. Schmitt, S. Theobald y M. Gaedke, «Scrum in practice: an overview of Scrum adaptations,» de *Proceedings of the 2015 International Conference on Software and System Process*, Tallinn, Estonia, 2015.

[2] K. Schwaber y M. Beedle, Agile Software Development with Scrum, Upper Saddle River, NJ: Prentice Hall, 2002.

[3] J. Sutherland, Scrum: The Art of Doing Twice the Work in Half the Time, New York, NY: Crown Business, 2014.

[4] IONOS Digital Guide, «¿Qué es el Frontend? Definición y explicación,» IONOS, 07 06 2024. [En línea]. Available: https://www.ionos.com/es-us/digitalguide/paginas-web/creacion-de-paginas-web/que-es-el-frontend/. [Último acceso: 26 09 2025].

[5] React Team, «Learn React,» Meta Open Source, [En línea]. Available: https://react.dev/learn. [Último acceso: 28 09 2025].

[6] MDN Web Docs, «Single-page application,» Mozilla Foundation, 11 07 2025. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Glossary/SPA. [Último acceso: 26 09 2025].

[7] M. Kothapalli, «The Evolution of Component-Based Architecture in Front-End Development,» *Journal of Scientific and Engineering Research,* vol. 8, nº 7, pp. 261-264, 2021.

[8] HTTP Archive, «The Web Almanac 2024: The State of the Web,» HTTP Archive, 2024. [En línea]. Available: https://almanac.httparchive.org/en/2024/. [Último acceso: 27 09 2025].

[9] Vite, «Guide,» Vite, [En línea]. Available: https://vitejs.dev/guide/. [Último acceso: 28 09 2025].

[10] React Team, «Writing Markup with JSX,» Meta Open Source, [En línea]. Available: https://react.dev/learn/writing-markup-with-jsx. [Último acceso: 28 09 2025].